PermNet-RM: Eliminating Side-Channel Leakage in HQC Reed-Muller Encoding via the GF(2) Zeta Transform

Bader Alissaei
VaultBytes Innovations Ltd
b@vaultbytes.com

April 19, 2026

Abstract

We show that encoding with the first-order Reed–Muller code $\mathrm{RM}(1, m)$ is algebraically identical to applying the GF(2) zeta transform to a fixed indicator vector. We exploit this equivalence to construct PermNet-RM, a Reed–Muller encoder for HQC that removes all message-dependent control flow and data access at the algorithmic level. (Concurrent work by Chen et al. (TCHES 2026) also targets constant-time RM encoding via an additive-FFT approach; the two constructions differ in mechanism and are compared in Section 1.4.) Our encoder computes the zeta transform using a fixed-topology butterfly network made only of straight-line XOR and shift operations: no message bit influences any branch, conditional select, or memory address. The construction is ABI-compatible with `reed_muller_encode()` from the HQC reference implementation.

Our work is motivated by two recent attacks on HQC. Jeon et al. (ePrint 2026/071) recover the full 128-bit encapsulation message from a single decapsulation trace with up to 96.9% success, using a total of 5,000 power traces for profiling and evaluation on an STM32F303 (ARM Cortex-M4), by exploiting leakage in the Reed–Muller encoding routine during the FO re-encryption step. Lai et al. (ePrint 2025/2162, YODO) show that ciphertext-independent timing leakages from sparse-vector processing in the HQC implementation enable passive, single-trace secret-key recovery, even in implementations believed to be constant-time. The Jeon attack directly targets the RM encoder's `BITOMASK` idiom; the YODO attack targets a broader class of sparse-vector timing leakages. PermNet-RM is designed to close the specific encoder leakage that Jeon exploit.

We establish a structural result on the register after the first butterfly stage (Theorem 4.2): exactly $2m - 1$ nonzero cells, with precisely one depending on two message bits and the remaining $2m - 2$ depending on single bits, and per-bit integer Hamming-weight residuals $\Delta_i \in \{0, 1, 2\}$.

This is a structural property, not a zero-correlation probing-model claim; the per-bit $\Delta_i > 0$ for all $i \neq 1$. A zero-correlation guarantee at any intermediate state requires the Boolean masking composition of Section 4.5, which is implemented and exhaustively correct. On x86-64, we empirically verify binary-level constant-time by disassembling the encoder under six GCC optimisation levels, with a per-stage compiler barrier that prevents the optimiser from folding the butterfly on isolated-bit registers into a single `neg` instruction. ELMO power simulation on ARM Cortex-M0 shows that PermNet-RM reduces the mean per-bit signal amplitude by a factor of 9.1 and shrinks the leaking attack surface by a factor of 1.8 compared to `BITOMASK`, with a further reduction under Boolean masking composition. Addressing the Jeon attack on its original target platform (ARM Cortex-M4) additionally requires real-silicon evaluation, which remains future work. On 64-bit and SIMD targets, the encoder exhibits zero timing spread across all 256 inputs, with an overhead of 1.3 cycles per encode.

Keywords: Reed-Muller codes, HQC, side-channel attacks, constant-time cryptography, power analysis, post-quantum cryptography, GF(2) zeta transform

1 Introduction

1.1 HQC Standardisation

HQC was selected by NIST for standardisation in March 2025 [7]. Its security relies on the hardness of decoding random quasi-cyclic codes. Internally, HQC first wraps messages in a first-order Reed–Muller code $\mathrm{RM}(1, m)$ before embedding them into the quasi-cyclic structure.

The reference C implementation [6] has been copied verbatim into PQ-Clean [8], liboqs [9], and the NIST submission package. All of these codebases use the same Reed–Muller encoder. That encoder leaks.

1.2 The Encoding Vulnerability

Every current HQC implementation uses a Reed–Muller encoder that follows the same pattern:

Listing 1: BIT0MASK idiom in the HQC reference encoder (all public implementations).

```
for (int i = 0; i < PARAM_M; i++) {
    uint64_t mask = -((uint64_t)((m >> i) & 1));
    cw_low  ^= mask & G[i].low;
    cw_high ^= mask & G[i].high;
}
```

The expression `-((uint64_t)((m >> i) & 1))` produces either `0x0000...0000` or `0xFFFF...FFFF` depending on whether bit i of the message `m` is 0 or 1.

The idea is to select or skip a generator row using only bitwise operations, avoiding an explicit branch in the source code. This idiom is widespread in constant-time programming, and at a quick glance it looks safe.

It is not.

1.3 Two Published Attacks

Jeon et al., IACR ePrint 2026/071 [1]. Jeon, Kim, Lee, and Kim demonstrate single-trace message recovery on a ChipWhisperer CW308 UFO board with an STM32F303 target (ARM Cortex-M4). They identify a leakage source in the Reed–Muller encoding routine that appears in the NIST submission, the HQC team's official codebase, and PQClean. Crucially, their attack targets the *decapsulation* path: during the Fujisaki–Okamoto re-encryption check, the encoder is called on the recovered message, and the attacker observes the encoder's power trace. Instead of recovering each RM input byte independently, they exploit Reed–Solomon post-correction on the full 46-byte RS codeword. Approximate recovery of RM-input symbols is enough, and RS decoding cleans up the remaining errors. They report up to 98.9% full 128-bit message recovery from a single decapsulation trace with as few as 20 profiling traces. On a noisier STM32F415 platform, RS-corrected recovery reaches 99.5% with 60 profiling traces.

Lai et al., IACR ePrint 2025/2162 (YODO) [2]. Lai et al. take a broader view. They show that ciphertext-independent, passive single-trace side-channel attacks can recover the HQC secret key, without touching the CCA-insecure decryption path and without chosen ciphertexts. They uncover timing leakages from sparse-vector processing that persist in the current "constant-time" HQC implementation, and they demonstrate that these leakages remain exploitable even under AMD SEV. On embedded devices, they further show corresponding power leakages. Their analysis applies to all code-based KEMs in NIST Round 4, including BIKE and Classic McEliece. Among several leakage sources, the RM encoder's `BITOMASK` idiom is one piece of a larger picture: the whole sparse-vector processing pipeline leaks timing information correlated with the secret key, even when the source code looks constant-time.

1.4 Why Existing Countermeasures Do Not Fix This

Three approaches have been explored so far. None of them fix the encoder.

Boolean masking of the decoder. Goy et al. [3] and Spyropoulos et al. [4] propose Boolean masking for the Berlekamp–Massey decoder and the Bernoulli sampler. However, they do not mask the encoder. Jeon's attack hits the encoder call inside the FO re-encryption check, which runs during

decapsulation on a value derived from the ciphertext and secret key. Masking the decoder does not protect this step.

Operation shuffling. One might try to randomise the order in which generator rows are XORed. This does not help. For any fixed message byte, the encoder must still XOR exactly the same set of generator rows, just in a different order. The multiset of intermediate Hamming weights is therefore unchanged. An attacker who sees the full sequence of register Hamming weights, rather than a single point, can still distinguish messages.

Additive-FFT acceleration. Chen et al. (TCHES 2026 [5]) speed up HQC's polynomial multiplication using an additive FFT combined with the Chinese Remainder Theorem, and also present time-constant optimised encoding and decoding of the RM and RS codes used in HQC. Their work targets both throughput and side-channel mitigation across the full HQC pipeline.

The two approaches are complementary in scope. Chen et al. optimise the complete HQC pipeline, including polynomial multiplication, whereas PermNet-RM focuses specifically on removing the `BIT0MASK` idiom from the RM encoder via the zeta-transform equivalence. That said, a direct comparison of the constant-time properties of the two RM encoding approaches would be valuable.

On throughput, Chen et al. report encoder performance only as part of their full-pipeline benchmark; a standalone encoder-only cycle count on comparable hardware is not available from their paper. On constant-time properties, both approaches aim to avoid message-dependent control flow in the encoder, but they do so via different mechanisms (additive-FFT decomposition versus zeta-transform butterfly), and with differently structured security arguments. Chen et al. argue primarily at the source-code level, while PermNet-RM additionally provides binary-level disassembly inspection and ELMO-based power simulation. On 32-bit targets, neither approach currently has published ELMO results sufficient to compare residual leakage amplitudes directly. A rigorous side-by-side evaluation—using the same compiler flags, hardware, and ELMO setup—would therefore be a useful piece of follow-on work.

The core problem is not the loop structure or the order of operations. It is that the standard encoding algorithm must test each message bit to decide whether to include a generator row. Any implementation that does this will leak under the Jeon attack model.

1.5 This Paper

The key observation behind PermNet-RM is that $\mathrm{RM}(1, m)$ encoding is equal to the GF(2) zeta transform of a fixed indicator vector. The zeta transform admits a butterfly decomposition in which each selector depends only on the register index, never on the message. Message bits appear just once, when

they are injected into fixed positions of the initial register state. From that point on, every operation is unconditional.

To avoid any ambiguity about what we actually prove, what we measure, and what remains open, we summarise our claims and the type of evidence for each in Table 1.

Table 1: Summary of claims and evidence.

Claim	Type	Scope / limitations
RM$(1, m)$ encoding = GF(2) zeta transform of indicator vector	Proof	Exact algebraic equivalence (Thm. 3.2)
Stage-1 register structure: exactly $2m - 1$ nonzero cells; one depends on two message bits, the remaining $2m - 2$ each depend on a single bit	Proof	Per-bit HW residuals $\Delta_i \in \{0, 1, 2\}$ enumerated (Thm. 4.2); not a zero-correlation claim on $\mathrm{wt}(R^{(1)})$
Zero-correlation probing-model security at all intermediate states	Proof	Requires Boolean masking composition at $d = 1$; cost $\sim 2\times$ encoder + one fresh share byte per call (§4.5)
No conditional branches in compiled binary (x86-64)	Empirical	GCC 15.2.0, six optimisation levels; not a formal proof (Prop. 4.3)
Zero timing spread on x86-64	Measured	Intel Core Ultra 9 275HX at `-O3`; single platform
$9.1\times$ mean signal reduction on ARM (ELMO)	Measured	Noise-free simulation on Cortex-M0 with per-stage compiler barriers; further $\sim 1.4\times$ reduction under the shared-output masked $d = 1$ variant (§5.5, §4.5)

We highlight two limitations up front. First, Theorem 4.2 gives a *structural* stage-1 statement: exactly $2m - 1$ nonzero cells, one of which depends on two message bits and the rest on single bits, with per-bit integer Hamming-weight residuals $\Delta_i \in \{0, 1, 2\}$. It does not claim zero correlation between $\mathrm{wt}(R^{(1)})$ and any single a_i; in fact $\Delta_i > 0$ for all $i \neq 1$. The zero-correlation probing-model guarantee is recovered by the Boolean masking composition of Section 4.5. Second, the butterfly's residual single-bit cells manifest physically as isolated 32-bit-word Hamming-weight signals on 32-bit ARM targets (Section 5.5); 64-bit and SIMD targets avoid this because the full logical register sits in a single physical register. The 32-bit residue is addressed by masking.

2 Background

2.1 Reed–Muller Codes

The first-order Reed–Muller code $\mathrm{RM}(1, m)$ is a binary linear code with parameters $[n, k, d] = [2^m, m + 1, 2^{m-1}]$, meaning it encodes $k = m + 1$ message bits into codewords of length $n = 2^m$ and has minimum distance $d = 2^{m-1}$.

One convenient way to describe $\mathrm{RM}(1, m)$ is via evaluations of linear Boolean functions. Let $v_0, v_1, \ldots, v_{m-1}$ be the m coordinate functions on $\mathbb{F}_2^m$, where $v_i(x) = x_i$ for $x = (x_0, \ldots, x_{m-1}) \in \mathbb{F}_2^m$, and let $v_m \equiv 1$ be the all-ones function.

Definition 2.1 (RM(1,m) via evaluation)**.** *Let $(a_0, a_1, \ldots, a_m) \in \mathbb{F}_2^{m+1}$ be a message. Define the Boolean function*

$$f(x) = a_0 v_m(x) + a_1 v_0(x) + a_2 v_1(x) + \cdots + a_m v_{m-1}(x),$$

where the addition is over $\mathbb{F}_2$. The Reed–Muller code $\mathrm{RM}(1, m)$ consists of all codewords

$$c = \mathrm{ev}(f) = (f(x))_{x \in \mathbb{F}_2^m}.$$

Equivalently, $\mathrm{RM}(1, m)$ has a generator matrix $G \in \mathbb{F}_2^{(m+1)\times 2^m}$ whose rows are v_m (the all-ones vector) and $v_0, \ldots, v_{m-1}$ (the coordinate-evaluation vectors). Encoding a message $\mathbf{a} = (a_0, \ldots, a_m)$ is then simply $c = \mathbf{a}G$ over $\mathbb{F}_2$.

In the context of HQC, the relevant instances are:

- **HQC-128**: uses $\mathrm{RM}(1, 7)$ with $n = 128$ and $k = 8$ (7 message bits + 1 constant bit).
- **HQC-192 and HQC-256**: use $\mathrm{RM}(1, 8)$ with $n = 256$ and $k = 9$ (8 message bits + 1 constant bit).

The minimum distance $d = 2^{m-1}$ gives the code the list-decoding properties that HQC relies on in its decapsulation algorithm.

2.2 HQC Encapsulation and Decapsulation

At a high level (see the specification [6] for full details), HQC is a code-based KEM that works over a quasi-cyclic code and internally wraps messages with an $\mathrm{RM}(1, m)$ code.

Encapsulation. Given a public key and a random message $m \in \mathbb{F}_2^k$:

1. **Reed–Muller encoding.** Compute $c_m \leftarrow \text{RM-encode}(m)$ to obtain a 128- or 256-bit codeword.

2. **Error sampling.** Sample quasi-cyclic error vectors e, r_1, r_2 from a Bernoulli distribution, using a seed derived from a hash of m.

3. **Ciphertext computation.** Let h, s be the public-key components in the quasi-cyclic ring. Compute $(u, v) = (r_1 + h \cdot r_2,\ m \cdot r_2 + s \cdot r_1 + e + c_m)$.

Decapsulation. Given the secret key and a ciphertext (u, v):

1. **Noisy codeword decoding.** Using the secret key element y, form $v' = v - u \cdot y$ and apply the Reed–Muller decoder to obtain $m' \leftarrow$ RM-decode(v').

2. **Re-encryption (FO check).** Call RM-encode(m') to reconstruct a codeword and perform the Fujisaki–Okamoto consistency check against the received ciphertext.

3. **Key derivation.** If the check passes, derive and output the session key; otherwise, output a fixed fallback key.

The re-encryption step in line 2 is exactly where the Jeon attack is mounted: the encoder is invoked on m', which depends jointly on the ciphertext and the secret key, while the attacker observes the power trace of this encoder call.

2.3 The BIT0MASK Encoder and Its Leakage

The HQC reference implementation, and all public derivatives, implement Reed–Muller encoding via a loop over the generator rows, using a bitmask idiom that is intended to be constant-time. In simplified form:

Listing 2: HQC reference encoder (simplified).

```
1  void reed_muller_encode(uint8_t *cdw, const uint8_t *msg) {
2      uint64_t cw_low = 0, cw_high = 0;
3      for (int i = 0; i < PARAM_M; i++) {
4          uint64_t mask = -((uint64_t)((msg[0] >> i) & 1));
5          cw_low  ^= mask & G[i].low;
6          cw_high ^= mask & G[i].high;
7      }
8      memcpy(cdw, &cw_low, 8);
9      memcpy(cdw + 8, &cw_high, 8);
10 }
```

Here, the expression `-((uint64_t)((msg[0] >> i) & 1))` evaluates to either `0x0000...0000` or `0xFFFF...FFFF` depending on whether bit i of the message is 0 or 1. The intention is to use this mask to include or exclude the i-th generator row using only bitwise operations, avoiding any explicit branch. This pattern is widely used in "constant-time" programming and, on a superficial reading, appears safe.

Under the standard Hamming-weight power model, however, it leaks directly. The Hamming weight of `mask` is 0 when the bit is 0 and 64 when the bit is 1. As a result, the instantaneous power consumption at the instruction that writes to `mask` is tightly correlated with bit i of the message. Jeon et al. [1] exploit exactly this effect during the FO re-encryption step to recover RM-input symbols, and then rely on Reed–Solomon post-correction to reconstruct the full message.

A second issue arises at the compiler level. Since `(uint64_t)((msg[0] >> i) & 1)` takes values only in $\{0, 1\}$, optimising compilers recognise that its negation yields a mask in $\{0, \texttt{0xFFFFFFFFFFFFFFFF}\}$ and that the subsequent XOR-with-AND pattern implements a conditional select. At optimisation levels `-O2` and above, GCC and Clang may lower this into a data-dependent instruction sequence (for example, using a `CMOV` or a `TEST`/`JE` pair), reintroducing data-dependent control flow even though the original C code contains no explicit branches.

This combination of Hamming-weight leakage and compiler rewrites makes the `BITOMASK`-style encoder vulnerable in exactly the way the recent attacks demonstrate.

3 The PermNet-RM Encoder

3.1 Mathematical Foundation: RM Encoding as a GF(2) Zeta Transform

Our starting point is that Reed–Muller encoding can be expressed in a way that completely avoids per-bit decisions. Concretely, encoding for $\mathrm{RM}(1, m)$ is algebraically identical to applying the GF(2) zeta transform (also known as the subset-sum or Möbius transform over the Boolean lattice) to a carefully chosen indicator vector.

Definition 3.1 (GF(2) Zeta Transform)**.** *Let $f : 2^{[m]} \to \mathbb{F}_2$ be a function on subsets of $[m] = \{0, 1, \ldots, m-1\}$, equivalently a vector $f \in \mathbb{F}_2^{2^m}$ indexed by subsets $S \subseteq [m]$. The GF(2) zeta transform $\hat{f} : 2^{[m]} \to \mathbb{F}_2$ is defined by*

$$\hat{f}(S) = \bigoplus_{T \subseteq S} f(T), \quad S \in 2^{[m]}.$$

We now show that $\mathrm{RM}(1, m)$ encoding is exactly this transform applied to an indicator vector that contains the message bits.

Theorem 3.2 (RM Encoding as a Zeta Transform)**.** *Let $(a_0, a_1, \ldots, a_m) \in \mathbb{F}_2^{m+1}$ be a message, where a_0 is the constant term and $a_1, \ldots, a_m$ are the linear coefficients. Define an indicator vector $\delta \in \mathbb{F}_2^{2^m}$ by*

$$\delta_S = \begin{cases} a_i & \text{if } S = \{i-1\} \text{ for some } i \in \{1, \ldots, m\}, \\ a_0 & \text{if } S = \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

In other words, a_i is placed at the singleton subset $\{i-1\}$ for $i \geq 1$, and a_0 is placed at the empty set. Then the $\mathrm{RM}(1,m)$ *codeword corresponding to $(a_0, \ldots, a_m)$ is exactly the GF(2) zeta transform of δ:*

$$c = \hat{\delta}.$$

Proof. Each position of the codeword is indexed by a point $x \in \mathbb{F}_2^m$. The usual evaluation definition of $\mathrm{RM}(1,m)$ gives

$$c_x = a_0 + \sum_{i=1}^{m} a_i x_{i-1} \quad (\text{over } \mathbb{F}_2).$$

Associate to each x the subset $S_x = \{j : x_j = 1\} \subseteq [m]$. The zeta transform at S_x is

$$\hat{\delta}(S_x) = \bigoplus_{T \subseteq S_x} \delta(T).$$

By construction, the only nonzero entries of δ are at $T = \emptyset$ (contributing a_0) and $T = \{j\}$ for $j \in [m]$ (contributing a_{j+1}). Among these, only the singletons with $j \in S_x$ appear in the sum. Therefore

$$\hat{\delta}(S_x) = a_0 \oplus \bigoplus_{j \in S_x} a_{j+1} = a_0 + \sum_{i=1}^{m} a_i x_{i-1},$$

which is exactly c_x. □

This equivalence is the core algebraic observation behind PermNet-RM. Instead of looping over generator rows and conditionally XORing them depending on message bits, we: (1) inject each message bit once at a fixed position in the initial register state, and then (2) apply a fixed sequence of linear operations (the butterfly decomposition of the zeta transform) that does not depend on the message. The message is never tested during this propagation phase.

3.2 Butterfly Decomposition of the Zeta Transform

The zeta transform on $\mathbb{F}_2^{2^m}$ can be implemented in a structure strongly reminiscent of an FFT: an m-stage butterfly network. Each stage propagates information along one coordinate direction.

Proposition 3.3 (Butterfly Decomposition)**.** *The GF(2) zeta transform on $\mathbb{F}_2^{2^m}$ can be computed in m stages. Initialising $f^{(0)} = \delta$, define for each stage $\ell = 0, 1, \ldots, m-1$:*

$$f_S^{(\ell+1)} = f_S^{(\ell)} \oplus \begin{cases} f_{S \setminus \{\ell\}}^{(\ell)} & \text{if } \ell \in S, \\ 0 & \text{otherwise,} \end{cases}$$

for all $S \subseteq [m]$. After m stages, $f^{(m)} = \hat{\delta}$.

Each stage can be seen as: for all subsets whose ℓ-th coordinate is 1, add (XOR) the value from the subset with that coordinate cleared. When the state is stored as a bit vector indexed by the subsets, this becomes a pattern of XORs between bits separated by a fixed stride. Packed into machine words, each stage is implemented by XORing the register with a mask-and-shifted version of itself. Crucially, the selector ("if $\ell \in S$") depends only on the position index S, not on any message bit.

3.3 Construction Overview

The PermNet-RM encoder uses this butterfly structure to compute $\mathrm{RM}(1, m)$ codewords without any message-dependent control flow or data access. Conceptually, the encoder has three steps:

1. **Injection.** Each of the $m + 1$ message bits is written into a predetermined bit position in an n-bit register, where $n = 2^m$. These positions are compile-time constants and are the same for all messages. No conditionals or bit tests are used: the encoder simply shifts each message bit by a fixed amount and XORs it into the register state.

2. **Butterfly propagation.** The encoder applies m stages of the GF(2) zeta transform as in Proposition 3.3. Each stage is a straight-line sequence of XORs and shifts by fixed distances, masked with fixed constants. All masks, shifts, and operation counts are fixed once m is fixed.

3. **Output.** After the final stage, the register contains the full $\mathrm{RM}(1, m)$ codeword, which is written to the output buffer.

By design, the encoder contains:

- no instances of the `BITOMASK` pattern `-((uint64_t)(bit & 1))`;
- no conditional branches (no `if`, `?:`, `switch` in the encoder body);
- no data-dependent memory accesses (no array lookups indexed by message bits);
- no loops whose iteration count depends on the message.

All data-dependent behaviour is confined to the initial register contents (which naturally depend on the message) and then propagated linearly through a fixed circuit.

3.4 Reference Implementation for RM(1,7) (HQC-128)

We now give a complete scalar C implementation of PermNet-RM for RM$(1, 7)$, the case used by HQC-128. This function is intended as a drop-in replacement for `reed_muller_encode()` in the HQC reference implementation: it uses the same signature and produces the same codewords.

Listing 3: PermNet-RM scalar encoder for HQC-128 (RM(1,7)). No conditional branches or data-dependent operations; all masks and shifts are compile-time constants.

```c
#include <stdint.h>
#include <string.h>
void permnet_rm7_encode(uint8_t *cdw, const uint8_t *msg)
{
    uint8_t  m = msg[0];
    uint64_t lo, hi;
    /* Step 1: inject each message bit at its singleton-set position.
     * Bit i of m is placed at register position 2^i.  No bit is
     * compared or branched on; every line has the same form.       */
    lo  = (uint64_t)((m >> 0) & 1);          /* a_0 → pos  0 (empty set ) */
    lo ^= (uint64_t)((m >> 1) & 1) <<  1;    /* a_1 → pos  1 = {0}   */
    lo ^= (uint64_t)((m >> 2) & 1) <<  2;    /* a_2 → pos  2 = {1}   */
    lo ^= (uint64_t)((m >> 3) & 1) <<  4;    /* a_3 → pos  4 = {2}   */
    lo ^= (uint64_t)((m >> 4) & 1) <<  8;    /* a_4 → pos  8 = {3}   */
    lo ^= (uint64_t)((m >> 5) & 1) << 16;    /* a_5 → pos 16 = {4}   */
    lo ^= (uint64_t)((m >> 6) & 1) << 32;    /* a_6 → pos 32 = {5}   */
    hi  = (uint64_t)((m >> 7) & 1);          /* a_7 → pos 64 = {6}   */

    /* Step 2: seven butterfly stages of the GF(2) zeta transform.
     * Stage l: reg ^= (reg & mask_l) << 2^l, where mask_l zeroes out
     * all positions whose l-th index bit is 1.  Topology is fixed;
     * message bits are never read again after Step 1.              */
    lo ^= (lo & 0x5555555555555555ULL) <<  1; /* stage 0: stride  1 */
    hi ^= (hi & 0x5555555555555555ULL) <<  1;

    lo ^= (lo & 0x3333333333333333ULL) <<  2; /* stage 1: stride  2 */
    hi ^= (hi & 0x3333333333333333ULL) <<  2;

    lo ^= (lo & 0x0F0F0F0F0F0F0F0FULL) <<  4; /* stage 2: stride  4 */
    hi ^= (hi & 0x0F0F0F0F0F0F0F0FULL) <<  4;

    lo ^= (lo & 0x00FF00FF00FF00FFULL) <<  8; /* stage 3: stride  8 */
    hi ^= (hi & 0x00FF00FF00FF00FFULL) <<  8;

    lo ^= (lo & 0x0000FFFF0000FFFFULL) << 16; /* stage 4: stride 16 */
```

```
36      hi ^= (hi & 0x0000FFFF0000FFFFULL) << 16;
37
38      lo ^= (lo & 0x00000000FFFFFFFFULL) << 32; /* stage 5: stride 32 */
39      hi ^= (hi & 0x00000000FFFFFFFFULL) << 32;
40
41      hi ^= lo;                                 /* stage 6: stride 64 */
42
43      /* Step 3: write 128-bit codeword. */
44      memcpy(cdw,     &lo, 8);
45      memcpy(cdw + 8, &hi, 8);
46  }
```

Sanity check on generator rows. For $m =$ `0x01` (only the constant term a_0 set), injection yields `lo = 1`, `hi = 0`. After the six intra-word butterfly stages, `lo` becomes `0xFFFFFFFFFFFFFFFF`. The final cross-word stage copies `lo` into `hi`, so the output is the all-ones codeword. This matches the constant generator row $v_m \equiv 1$. For $m =$ `0x02` (only a_1 nonzero, corresponding to v_0), the output is the alternating-bit pattern `0xAAAAAAAAAAAAAAAA` in both words, which encodes the function $x \mapsto x_0$ over all $x \in \mathbb{F}_2^7$. Analogous checks for $m =$ `0x04`, `0x08`, ... reproduce the remaining coordinate functions. These checks, together with exhaustive comparison against a reference implementation (Section 5.7), validate that the butterfly encoder computes the intended RM$(1, 7)$ code.

3.5 Embodiments

The same design scales across the HQC parameter sets and across scalar and SIMD implementations.

RM(1,7) for HQC-128. The 128-bit codeword fits in two `uint64_t` variables. The encoder injects 8 message bits into predetermined bit positions, then applies 7 butterfly stages. Each stage uses a small fixed sequence of register operations. The total instruction count is low and does not depend on the message.

RM(1,8) for HQC-192 and HQC-256. The 256-bit codeword occupies four `uint64_t` words or a single 256-bit SIMD register. The encoder injects 9 message bits and applies 8 butterfly stages. As before, each stage is defined entirely by compile-time masks and shift amounts.

x86-64 SSE2/AVX2 SIMD. On x86-64, the RM(1,7) codeword fits in a single XMM register. Each butterfly stage is a small sequence of SIMD instructions. The entire encoder fits into on the order of a dozen SIMD instructions with no branches.

ARM NEON / AArch64. On ARM architectures with NEON, the structure is analogous. NEON's `VEOR` corresponds to `PXOR`, and `VSHL`/`VSHR` correspond to `PSLLQ`/`PSRLQ`.

ABI compatibility. The PermNet-RM encoder presents the same function signature as `reed_muller_encode()` in the HQC reference code. Integrating PermNet-RM therefore requires only swapping the encoder implementation; no changes are needed at call sites.

4 Security Analysis

4.1 Hamming-Weight Power-Leakage Model

We analyse PermNet-RM in the standard Hamming-weight leakage model, following Jeon et al. [1]. In this model, the power consumed at each clock cycle is assumed to be approximately proportional to the Hamming weight of the value written to a register at that cycle. An attacker observes a single power trace and tries to recover information about the message bits from it.

Definition 4.1 (Single-Trace Hamming-Weight Distinguisher)**.** *A distinguisher $\mathcal{D}$ for a message bit b receives a power trace $(p_0, p_1, \ldots, p_T)$ where $p_t = \mathrm{wt}(R_t) + \epsilon_t$, R_t is the register value written at cycle t, and ϵ_t models measurement noise. $\mathcal{D}$ succeeds if it outputs b with probability noticeably greater than $1/2$.*

4.2 Structural Properties of the Register After Stage 1

We first give a precise structural statement about the logical n-bit register state after the first butterfly stage. It is weaker than a full probing-security claim—such a claim requires the Boolean masking composition of Section 4.5—but it isolates exactly what the butterfly alone achieves.

Theorem 4.2 (Stage-1 bit structure and per-bit Hamming-weight residues)**.** *Let $R^{(1)}$ denote the logical n-bit register state after the first butterfly stage of PermNet-RM, with $n = 2^m$. Under the injection pattern of Theorem 3.2 (a_0 at $\emptyset$, a_i at the singleton $\{i-1\}$ for $i = 1, \ldots, m$) and the stage-0 rule of Proposition 3.3, the nonzero entries of $R^{(1)}$ are exactly:*

$$R^{(1)}_{\emptyset} = a_0, \qquad R^{(1)}_{\{0\}} = a_0 \oplus a_1,$$
$$R^{(1)}_{\{k\}} = a_{k+1} \text{ for } k = 1, \ldots, m-1, \qquad R^{(1)}_{\{0,k\}} = a_{k+1} \text{ for } k = 1, \ldots, m-1.$$

All other cells of $R^{(1)}$ are zero. Consequently:

(i) $R^{(1)}$ has exactly $2m-1$ nonzero cells. Exactly one of them ($R^{(1)}_{\{0\}}$) depends on two distinct message bits; the remaining $2m-2$ cells each depend on a single message bit.

(ii) The integer Hamming weight satisfies

$$\mathrm{wt}(R^{(1)}) = [a_0] + [a_0 \oplus a_1] + 2\sum_{k=1}^{m-1}[a_{k+1}],$$

where $[\cdot]$ *denotes the* $\{0,1\}$*-valued indicator.*

(iii) When the remaining bits $\mathbf{a}_{\neq i}$ *are uniformly random, the per-bit conditional-expectation gap* $\Delta_i := \mathbb{E}[\mathrm{wt}(R^{(1)}) \mid a_i = 1] - \mathbb{E}[\mathrm{wt}(R^{(1)}) \mid a_i = 0]$ *equals* 1 *for* $i = 0$*,* 0 *for* $i = 1$*, and* 2 *for every* $i \in \{2, \ldots, m\}$*.*

Proof. The initial register $R^{(0)} = \delta$ has exactly $m + 1$ nonzero positions: $R^{(0)}_{\emptyset} = a_0$ and $R^{(0)}_{\{i-1\}} = a_i$ for $i = 1, \ldots, m$. Stage $\ell = 0$ of Proposition 3.3 updates $R^{(1)}_S = R^{(0)}_S \oplus R^{(0)}_{S\setminus\{0\}}$ when $0 \in S$, leaving $R^{(1)}_S = R^{(0)}_S$ otherwise. Case-splitting on $|S|$:

- $S = \emptyset$: $0 \notin S$, so $R^{(1)}_{\emptyset} = R^{(0)}_{\emptyset} = a_0$.
- $S = \{0\}$: $R^{(1)}_{\{0\}} = R^{(0)}_{\{0\}} \oplus R^{(0)}_{\emptyset} = a_1 \oplus a_0$.
- $S = \{k\}$ for $k \geq 1$: $0 \notin S$, so $R^{(1)}_{\{k\}} = R^{(0)}_{\{k\}} = a_{k+1}$.
- $S = \{0, k\}$ for $k \geq 1$: $R^{(1)}_S = R^{(0)}_S \oplus R^{(0)}_{\{k\}} = 0 \oplus a_{k+1} = a_{k+1}$.
- $|S| \geq 2$ and $0 \notin S$: $R^{(1)}_S = R^{(0)}_S = 0$.
- $|S| \geq 3$ and $0 \in S$: $R^{(1)}_S = R^{(0)}_S \oplus R^{(0)}_{S\setminus\{0\}} = 0$, since $|S \setminus \{0\}| \geq 2$.

This establishes the enumeration and part (i). Summing the indicators of these nonzero cells over the integer domain yields part (ii).

For part (iii), compute the conditional expectations term by term under uniform $\mathbf{a}_{\neq i}$. For $i = 0$: $\mathbb{E}[[a_0] \mid a_0 = 0] = 0$ and $= 1$ for $a_0 = 1$; $\mathbb{E}[[a_0 \oplus a_1] \mid a_0] = 1/2$ in either case; the final sum contributes $2(m-1)/2 = m - 1$ regardless of a_0. So $\Delta_0 = 1$. For $i = 1$: only $[a_0 \oplus a_1]$ depends on a_1, and its conditional expectation is $1/2$ under either value of a_1. So $\Delta_1 = 0$. For $i \in \{2, \ldots, m\}$: a_i appears as $2[a_i]$ in the sum (unmasked at both $\{i-1\}$ and $\{0, i-1\}$); setting $a_i = 0$ contributes 0 and $a_i = 1$ contributes 2, giving $\Delta_i = 2$. □

What the stage-1 result does and does not guarantee. Theorem 4.2 captures the structural property of the first butterfly stage precisely: it eliminates the direct single-bit → 32-wide Hamming-weight mask that `BITOMASK` exposes, replacing it with a pattern in which every nonzero cell depends on at most two message bits and one cell depends on two. It does *not*

deliver a zero-correlation probing-model guarantee at the level of the integer Hamming weight $\mathrm{wt}(R^{(1)})$: as part (iii) shows, $\mathrm{wt}(R^{(1)})$ has per-bit gaps $(\Delta_0, \ldots, \Delta_m) = (1, 0, 2, 2, \ldots, 2)$. An attacker observing $\mathrm{wt}(R^{(1)})$ as a single integer under uniform $\mathbf{a}_{\neq i}$ therefore retains nonzero correlation with a_i for $i \neq 1$.

The gap is a structural consequence of the fact that the GF(2) Möbius inverse of the RM(1,m) generator's constant column is the indicator of $\emptyset$, so a_0 must be injected there, and of the fact that the stage-0 butterfly XORs only the pair $(\{0\}, \emptyset)$ across axis 0, leaving a_i for $i \geq 2$ unmixed at stage 1. No rearrangement of the zeta butterfly can reduce all per-bit gaps to zero simultaneously while preserving the RM(1,m) output.

What does close the gap. A zero-correlation probing-model guarantee at the level of $R^{(1)}$ (or any other intermediate) is recovered by the Boolean masking composition of Section 4.5: under $d = 1$ masking, each share's register state is a function of a uniformly random share alone, and per-bit correlation with any a_i is identically zero in the probing model. Section 5.5 additionally reports the empirical Cortex-M0 ELMO behaviour of both the unmasked and masked encoders.

Applicability to the Jeon model. In the Jeon attack model [1], the attacker observes a decapsulation trace on a fixed secret key and a chosen ciphertext. The attack that ePrint 2026/071 actually executes exploits the `BITOMASK` idiom's 0-vs-0xFFFFFFFF pattern in the mask register; PermNet-RM removes that specific pattern entirely (Section 4.4) and is verified in disassembly at six GCC optimisation levels not to reintroduce it. The per-bit residual described in Theorem 4.2(iii) is a distinct, much smaller effect that manifests in the 32-bit decomposition discussed in Section 5.5.

4.3 Bit-Level Residues in Early Stages

The previous result considers only the Hamming weight of the full logical state. If we move to a finer-grained, bit-level view, some dependencies remain.

Theorem 4.2(i) already describes these positions explicitly: under the canonical injection, $R^{(1)}_{\emptyset} = a_0$ and $R^{(1)}_{\{k\}} = a_{k+1}$ for $k = 1, \ldots, m-1$ depend on a single message bit. The stage-0 XOR also produces single-bit cells at $R^{(1)}_{\{0,k\}} = a_{k+1}$ for $k = 1, \ldots, m-1$. The total number of single-bit cells is $2m - 2$, determined by the public injection positions and independent of the message.

This bounded set is progressively covered by subsequent butterfly stages. By stage j, every bit position S with $|S| \leq j$ has received at least one contribution from a higher-order XOR term. After all m stages, the codeword corresponds to the full zeta transform, and (except for the all-zero message) no single position remains a function of a single message bit.

From a countermeasure perspective, this structure is friendly to masking. Since the encoder is GF(2)-linear, composing PermNet-RM with a Boolean masking scheme is straightforward: one can encode $d+1$ Boolean shares of the message independently and XOR the resulting codeword shares. The ISW composition theorem then provides t-probing security with $t = d$.

4.4 Binary-Level Constant-Time (Empirical)

We next examine the control-flow behaviour of PermNet-RM at the binary level on x86-64.

Proposition 4.3 (Binary-Level Constant-Time — Empirical)**.** *For the reference C implementation of PermNet-RM, compiled on x86-64 with* `gcc 15.2.0` *at each of the optimisation levels* `-O0, -O1, -O2, -O3, -Os, -Ofast`*, disassembly shows that the encoder contains no conditional branch instruction whose condition register depends on any message bit.*

This claim is empirical rather than formal: it holds for the tested compiler version, target architecture, and flags. However, there are structural reasons to expect this behaviour to be robust.

The PermNet-RM source does not contain any conditional operations at all: no `if`, no ternary operator, no `switch`, and no data-dependent loop bounds. All operations are straight-line arithmetic and bitwise instructions on the register state. As a result, the compiler has no conditional-select idiom to optimise into a branch or a `CMOV`.

Disassembly of the compiled encoder at all six optimisation levels confirms this: the encoder body consists solely of register-to-register moves, XORs, shifts, and ANDs. There are no `Jcc` (conditional jump) instructions whose condition can be influenced by the message, and no memory accesses whose address is derived from message bits.

By contrast, disassembly of the `BITOMASK` encoder exhibits exactly the pattern that Jeon et al. [1] exploit: an instruction sequence that materialises a $\{0, 0\text{xFFFFFFFF}\}$ mask register from a $\{0,1\}$-valued message-bit extraction. The exact mnemonic depends on target and optimisation level. On x86-64 GCC emits `negq` on a register that holds a $\{0,1\}$ value; on ARM Cortex-M0 at `-Os` GCC lowers the same idiom to an `ands` followed by `muls` against the generator row. Both sequences produce the direct 0-vs-32 (or 0-vs-64) Hamming-weight signature of the input bit.

4.5 Composability with Boolean Masking

An attractive property of PermNet-RM is that it is purely linear over $\mathbb{F}_2$. This makes it compatible with standard Boolean masking techniques. To achieve t-probing security with $t = d$:

1. Split each message bit a_i into $d+1$ shares: $a_i = a_i^{(0)} \oplus a_i^{(1)} \oplus \cdots \oplus a_i^{(d)}$.

2. Run PermNet-RM independently on each share vector $\mathbf{a}^{(j)}$ to obtain codeword shares $c^{(j)}$.

3. XOR the resulting $d+1$ codeword shares: $c = c^{(0)} \oplus c^{(1)} \oplus \cdots \oplus c^{(d)}$.

Because $\mathrm{RM}(1, m)$ encoding is linear over $\mathbb{F}_2$, the shares propagate correctly. No non-linear gadgets are required. Under the standard ISW framework, this gives t-probing security at $t = d$.

5 Experimental Results

5.1 Experimental Setup

All x86-64 benchmarks were run on an Intel Core Ultra 9 275HX (Meteor Lake) at stock frequency. Hyperthreading was disabled on the measurement core, and the CPU frequency governor was set to `performance`. Timing measurements used the `RDTSC` instruction with serialising `CPUID` fences. Reported values are the median over 10^7 iterations, preceded by a warm-up phase of 10^5 iterations.

We compare three implementations:

- **PermNet-RM**: the proposed butterfly-network encoder.
- `BITOMASK`: the HQC reference encoder using the `BITOMASK` idiom.
- **Branched**: a naive encoder with explicit conditionals `if (bit) { cw ^= G[i]; }`.

Unless otherwise stated, code was compiled with `gcc 15.2.0`.

5.2 Throughput at `-O3 -march=native`

Table 2: Encoder throughput on Intel Core Ultra 9 275HX, `gcc 15.2.0 -O3 -march=native`.

Encoder	Throughput (Mops/s)	Latency (ns)	Cycles/encode
PermNet-RM	479.62	2.08	6.41
`BITOMASK`	605.04	1.65	5.08
Branched	644.29	1.55	4.77

PermNet-RM runs at 479.62 million encodes per second, about 20% slower than `BITOMASK` and about 26% slower than the Branched encoder. In absolute terms, this corresponds to an overhead of roughly 1.33 cycles per encode compared to `BITOMASK`.

In a full HQC-128 encapsulation, the encoder is invoked roughly 16 times. The total additional cost of using PermNet-RM instead of `BITOMASK` is therefore about $16 \times (6.41 - 5.08) \approx 21$ cycles, or approximately 7 ns at 3 GHz. A complete HQC-128 encapsulation takes on the order of hundreds of microseconds, so this overhead is negligible in practice.

5.3 Per-Input Timing at `-O3` (Constant-Time Behaviour)

Table 3: Per-input timing distribution across all 256 possible input bytes at `-O3 -march=native`. Spread = max − min.

Encoder	min	median	max	spread
PermNet-RM	6	6	6	**0**
`BITOMASK`	4	4	5	1
Branched	4	4	5	1

PermNet-RM exhibits zero timing spread: all 256 inputs take exactly 6 cycles. Both `BITOMASK` and Branched exhibit a 1-cycle spread, indicating data-dependent timing at this optimisation level.

5.4 Per-Input Timing at `-O0` (Unoptimised Code)

Table 4: Per-input timing at `-O0` across all 256 input bytes. HW bucket means are average cycle counts grouped by Hamming weight of the input (0–8).

Encoder	min	median	max	spread	HW bucket means
PermNet-RM	15	16	18	3	flat: 16.0–17.0 cycles
`BITOMASK`	17	18	18	1	flat: 18.0–18.1 cycles
Branched	6	9	22	**16**	monotone from HW 4: 8.0–16.0 cycles

The Branched encoder shows clear Hamming-weight dependence: inputs with higher Hamming weight take more cycles, with bucket means rising from about 8 cycles to about 16 cycles. PermNet-RM's bucket means are flat within the noise range (16.0–17.0 cycles), with no systematic dependence on input Hamming weight.

The `BITOMASK` encoder also shows flat bucket means at `-O0`, but this does not contradict its vulnerability: Jeon et al. [1] exploit Hamming-weight leakage in the mask register (power side channel), not timing. That leakage is present regardless of timing uniformity.

We additionally repeated the timing experiments on the same CPU under Windows 11 with MSVC; the behaviour of PermNet-RM was consistent across

toolchains.

5.5 ELMO Power Simulation on ARM Cortex-M0

The Jeon attack targets an ARM Cortex-M4 platform. To study PermNet-RM in a similar embedded setting without requiring physical hardware, we used ELMO [10], a statistical power simulator for ARM Cortex-M0 that models Hamming-weight-style leakage. ELMO takes a bare-metal Thumb binary and produces noise-free simulated power traces. While it is not a perfect substitute for real measurements, it is well-suited for relative comparisons between two implementations.

Setup. We compiled both PermNet-RM and `BITOMASK` for ARM Cortex-M0 using `arm-none-eabi-gcc 15.2.0 -Os -mcpu=cortex-m0 -mthumb`. On this 32-bit architecture, the 128-bit codeword is stored as four `uint32_t` words (`lo0, lo1, hi0, hi1`). Each encoder was linked into an ELMO harness that iterates over all 256 possible 8-bit messages and records a trace per input. For each message bit i and cycle c, we compute the signal amplitude

$$\Delta_i(c) = \big|E[\text{power}(c) \mid m_i = 1] - E[\text{power}(c) \mid m_i = 0]\big|.$$

Table 5: ELMO simulation on ARM Cortex-M0: per-bit signal amplitude and leaking surface. PermNet-RM column is the unmasked encoder with per-stage compiler barriers; PermNet-RM masked is the shared-output Boolean masked $d = 1$ variant of Section 4.5. "Leaking cycles" counts cycles where $|r| > 0.3$.

Metric	PermNet-RM	PermNet-RM masked*	`BITOMASK`
Trace length (cycles)	144	284	293
Max single-bit signal	1 757.7	405.6	4 493.4
Mean single-bit signal	294.97	229.58	2 687.0
Bit-6 single-bit signal	1 757.7	120.9	3 778.4
Bit-7 single-bit signal	221.2	99.5	3 778.4
Leaking cycles ($\lvert r\rvert > 0.3$)	55 / 144	3 / 284	199 / 293
Mask-idiom instructions in body†	0	0	5

* shared-output Boolean masking at $d = 1$; see Section 4.5. † Count of mask-producing instructions whose operand is directly derived from a message bit (`negq` on x86-64, `ands`+`muls` on Cortex-M0 at `-Os`). Zero in both PermNet-RM variants at every GCC optimisation level.

Results. The `BITOMASK` encoder shows very strong leakage: for all 8 message bits, there are cycles with $|r| = 1.0$ and signal amplitude around 4,493, corresponding to the mask-generation instructions. PermNet-RM significantly reduces leakage. The maximum per-bit signal (1,757.7) is dominated by bit 6, the mean per-bit signal (294.97) is 9.1× lower than for `BITOMASK`, and

bit 7's signal drops 17× relative to `BITOMASK`. The shared-output masked $d = 1$ variant brings the peak signal down a further 4.3× to 405.6, the bit-6 signal to 120.9 (a 14.5× reduction relative to the unmasked PermNet-RM and a 31× reduction relative to `BITOMASK`), and the leaking-cycle fraction to 1.06% (3/284).

32-bit decomposition effect. The residual signal on bit 6 in the unmasked PermNet-RM has a specific cause. On a 32-bit processor, message bit m_6 ends up occupying one 32-bit word (`lo1`) essentially in isolation. During early butterfly stages (up to stage 4), this word propagates without mixing with other message bits, and its Hamming weight doubles at each stage (from 1 to 32). This behaviour is structurally analogous to the `BITOMASK` mask but now distributed across six cycles rather than concentrated in a single `neg` instruction.

On a 64-bit processor, the logical 64-bit word `lo` contains all message bits $m_0, \ldots, m_6$, and the first butterfly stage already mixes adjacent bits. The stage-1 structural result of Theorem 4.2 applies to the full 64-bit state, and the single-bit isolation effect that appears in the 32-bit decomposition does not arise.

Compiler barriers against `neg`-fold. A subtle implementation issue on both x86-64 and Cortex-M0 is that GCC at `-O1` and above recognises that the full butterfly applied to a register that starts as a single 1-bit message value equals $-m_i$ (all zeros or all ones), and folds the six-stage propagation into a single `neg` instruction. The resulting instruction is syntactically the `BITOMASK` idiom: a 0-vs-0xFFFFFFFF mask register that directly leaks the message bit. To block this, the reference C implementation applies an empty inline-asm barrier after every butterfly stage, `__asm__ volatile ("" : "+r"(x))`, which forces the compiler to materialise each intermediate. The compiled binary then contains zero `neg` instructions in the encoder body at every GCC optimisation level. Table 5 reports post-barrier numbers; without barriers, bit 6 and bit 7 reach 3,779 each, essentially identical to `BITOMASK`.

Mitigation of the remaining 32-bit residual. After the compiler-barrier fix, bit 6 still reaches 1,757.7 because m_6 propagates through six stages of progressively doubling Hamming weight in the isolated 32-bit word `lo1`. Closing this residual requires Boolean masking composition ($d = 1$), implemented in shared-output form in Section 4.5. Under masking, the operand of every data-dependent instruction is a uniformly random share, so the per-stage `lo1` Hamming weight is decorrelated from m. Empirically (Table 5) the masked variant reduces peak per-bit signal by a further 4.3×, the bit-6 signal by a further 14.5×, and the leaking-cycle fraction from 38%

to 1.06%.

Even for the unmasked encoder, the 9.1× reduction in mean signal amplitude implies that, in noisy real measurements, an attacker would need roughly $(9.1)^2 \approx 83$ times more traces to reach the same confidence as with the `BITOMASK` encoder.

5.6 Limitations on 32-bit Targets and Path to Mitigation

The ELMO results in Section 5.5 show that the unmasked PermNet-RM encoder reduces leakage substantially but does not fully eliminate it on 32-bit targets. After the compiler-barrier fix, message bit m_6, which resides in the word `lo1` on its own during butterfly stages 0–4, reaches a peak per-bit signal amplitude of 1,757.7—about 39% of the `BITOMASK` peak of 4,493. This is a direct consequence of the 32-bit register decomposition and will occur on any 32-bit architecture where the 128-bit codeword state is split across several physical registers.

The residual is closed by Boolean masking composition ($d = 1$), implemented in shared-output form in Section 4.5. Under masking, bit 6 drops to 120.9 (a 14.5× reduction versus the unmasked PermNet-RM and a 31× reduction versus `BITOMASK`) and the leaking-cycle fraction falls from 38% to 1.06%.

Interleaved injection. An alternative that was explored in this work is interleaving injections across the four 32-bit words so that no word ever holds a single message bit in isolation. We did not find a concrete, correctness-preserving placement that keeps the RM(1,m) output unchanged: placing a_i at non-standard positions requires a replacement linear network for the zeta butterfly, and designing an efficient straight-line XOR network for that map is open work. A proof-of-concept stage-reordering variant is included in the public implementation as a documented negative result.

Boolean masking ($d = 1$). The shared-output Boolean masking variant (Section 4.5) is the mitigation we recommend. It takes two shares (s_0, s_1) with $m = s_0 \oplus s_1$, runs two independent baseline encodes, and returns the codeword in shared form (c_0, c_1) with $c_0 \oplus c_1 = E(m)$. No unmasking occurs inside the encoder. Cost: roughly 2× encoder runtime, one fresh byte of cryptographic randomness per call, and an API change so that downstream HQC code consumes both shares until it reaches a region where reconstruction is safe. In exchange, ELMO measures peak per-bit signal 405.6, mean 229.58, leaking-cycle fraction 1.06%, and bit 6 signal 120.9—a 14.5× reduction on bit 6 relative to the unmasked encoder and an 11.1× reduction in peak signal relative to `BITOMASK` (a 31× reduction on bit 6 relative to `BITOMASK`).

On 64-bit and SIMD targets, where the logical state fits into one or a small number of wide registers and message bits are mixed earlier, the

32-bit residual does not arise; the unmasked encoder is already effectively branch-free at the binary level.

A further limitation of our evaluation is that ELMO models an ARM Cortex-M0 core, whereas the Jeon attack [1] was demonstrated on an ARM Cortex-M4 (STM32F303, ChipWhisperer platform). These cores differ architecturally: the Cortex-M0 uses a simple two-stage in-order pipeline with no multiply–accumulate unit and limited operand forwarding, while the Cortex-M4 has a three-stage pipeline, DSP extensions, and a floating-point unit. As a result, pipeline-level leakage behaviour—such as operand transitions and data-dependent bus activity—can differ between the two.

We nonetheless chose Cortex-M0 for two reasons. First, ELMO is, to our knowledge, the only publicly available noise-free power simulator for an ARM core with a validated statistical leakage model; there is no comparable tool for Cortex-M4. Second, the `BITOMASK` leakage exploited by Jeon et al. occurs at the register-assignment level (the Hamming weight of the mask variable) and is present on all ARM Cortex-M variants regardless of pipeline depth. The core leakage mechanism we target is therefore not specific to M0.

Even so, the amplitudes reported in Table 5 should be read as indicative rather than predictive for Cortex-M4. They support a relative comparison between `BITOMASK` and PermNet-RM under the ELMO model, but they do not substitute for measurements on the actual Jeon platform. Physical ChipWhisperer experiments on a Cortex-M4 running PermNet-RM would provide the definitive assessment and are a natural next step for future work.

5.7 Functional Correctness

We verified that PermNet-RM produces exactly the same codewords as the textbook generator-matrix encoder:

- **RM(1,7)** (HQC-128): All 256 possible 8-bit messages matched.
- **RM(1,8)** (HQC-192/256): All 512 possible 9-bit messages matched.

Because the message spaces are small, these exhaustive checks leave no room for undetected correctness errors.

6 Conclusion

HQC is now on a clear path to standardisation, with a FIPS draft expected in 2026. Once the current reference implementation is cemented into standards, libraries, and hardware, changing even small components such as the Reed–Muller encoder will become significantly more difficult. This makes the present moment the right time to revisit the encoder design.

The core message of this work is simple: there is no compelling reason to ship the known-leaky encoder.

We showed that encoding for RM$(1, m)$ can be viewed as applying the GF(2) zeta transform to a fixed indicator vector. This perspective leads directly to PermNet-RM, a butterfly-network encoder with: no message-dependent branches or data-dependent memory accesses; a fixed topology of XOR-and-shift operations; and ABI compatibility with the existing `reed_muller_encode()` in the HQC reference code.

On x86-64, PermNet-RM adds about 1.3 cycles per encode compared to the `BITOMASK` encoder. At the KEM level, this overhead is negligible (around 21 extra cycles, or roughly 7 ns, per HQC-128 encapsulation). Disassembly under multiple GCC optimisation levels shows no message-dependent control flow.

On the side-channel side, our results are mixed:

- After the first butterfly stage, the logical register has exactly $2m - 1$ nonzero cells, one of which depends on two message bits (Theorem 4.2). This eliminates the direct single-bit-to-32-wide-mask pattern that `BITOMASK` exposes and replaces it with bounded per-bit integer HW residuals $\Delta_i \in \{0, 1, 2\}$. A zero-correlation probing-model guarantee at intermediate states requires Boolean masking composition (Section 4.5).

- ELMO simulations on ARM Cortex-M0 indicate that the unmasked PermNet-RM encoder (with per-stage compiler barriers) reduces the mean per-bit signal amplitude by approximately 9.1× and bit 7's signal by 17× compared to the `BITOMASK` encoder. The shared-output Boolean masked $d = 1$ variant reduces peak signal a further 4.3× and the leaking-cycle fraction to 1.06%.

- After the compiler-barrier fix, bit 6 in the unmasked encoder reaches 1,757.7, about 39% of the `BITOMASK` peak and half the pre-fix value. The shared-output Boolean masked $d = 1$ variant drives bit 6 down to 120.9, a 31× reduction relative to `BITOMASK`.

This residual leakage does not appear on 64-bit or SIMD targets. On 32-bit ARM platforms, composition with the shared-output Boolean masked $d = 1$ variant of Section 4.5 closes the residual: it has been implemented, exhaustively verified, benchmarked, and measured under ELMO, with 11.1× peak-signal reduction versus `BITOMASK` and 14.5× reduction on bit 6 versus the unmasked PermNet-RM. Real-silicon evaluation on the Cortex-M4 platform where the Jeon attack was originally demonstrated remains future work.

Several open directions remain:

1. **Generalisation beyond HQC.** The same additive-structure idea may apply to other code-based KEMs, such as BIKE's QC-MDPC encoder. We have not analysed this case in detail.

2. **More compact masked variants.** Our masking composition requires evaluating $d + 1$ encoder instances. Designing shared-state or partially fused masked versions of the butterfly network would be particularly interesting for resource-constrained devices.

3. **Optimised embedded implementations.** The 32-bit decomposition effect highlighted by ELMO suggests that dedicated ARM implementations that avoid isolated single-bit words would be valuable.

From a standardisation perspective, the bar for changing a long-lived primitive is high. However, the change proposed here is narrow and well-scoped: PermNet-RM is a drop-in replacement for a single, small function with essentially no impact on API, code structure, or performance at the KEM level, and with materially better side-channel behaviour. The vulnerable encoder can be replaced at negligible cost. We hope this construction makes doing so straightforward.

Code availability. A complete C reference implementation, benchmark harness, and x86-64 disassembly listings are available at `https://github.com/BAder82t/PermNet-RM`.

Acknowledgements

Thanks to the authors of ePrint 2026/071 and 2025/2162 for publicly documenting the attack surface in enough detail to diagnose the root cause. Correspondence regarding the security analysis of HQC implementations is welcome via the contact address on the title page.

References

[1] J. Jeon, D. Kim, S. Lee, and Y.-S. Kim. Single-Trace Message Recovery in HQC via RS Post-Decoding and FO Re-Encryption. Cryptology ePrint Archive, Report 2026/071, 2026. `https://eprint.iacr.org/2026/071`

[2] Z. Lai, R. Zhang, Z. Zhang, J. Hermelink, M. Schwarz, V.-T. Pham, and U. Parampalli. YODO: You Only Decapsulate Once — Ciphertext-Independent Single-Trace Passive Side-Channel Attacks on HQC. Cryptology ePrint Archive, Report 2025/2162, 2025. `https://eprint.iacr.org/2025/2162`

[3] G. Goy, M. Spyropoulos, N. Aragon, P. Gaborit, R. Pacalet, F. Perion, L. Sauvage, and D. Vigilant. Side-Channel Sensitivity Analysis on HQC: Towards a Fully Masked Implementation. Cryptology ePrint Archive, Report 2025/1344, 2025. `https://eprint.iacr.org/2025/1344`

[4] M. Spyropoulos, D. Vigilant, F. Perion, R. Pacalet, and L. Sauvage. Masked Vector Sampling for HQC. Cryptology ePrint Archive, Report 2024/1106, 2024. https://eprint.iacr.org/2024/1106

[5] M.-S. Chen, C.-M. Chiu, C.-T. Peng, and B.-Y. Yang. Accelerating HQC with Additive FFT. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2026. https://eprint.iacr.org/2026/014

[6] C. Aguilar-Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Hauteville, G. Zemor, and X. Carpent. HQC: Hamming Quasi-Cyclic – NIST Post-Quantum Cryptography Standardisation, Round 4 Submission. Technical specification, NIST, 2023. https://pqc-hqc.org/

[7] National Institute of Standards and Technology. NIST Announces Additional Post-Quantum Cryptography Standards: HQC Selected for Standardisation. NIST News, March 2025. https://csrc.nist.gov/News/2025/hqc-announced-as-a-4th-round-selection

[8] PQClean Contributors. PQClean: A Collection of Clean, Portable, and Tested Implementations of Post-Quantum Cryptographic Algorithms. https://github.com/PQClean/PQClean

[9] Open Quantum Safe Project. *liboqs*: An Open Source C Library for Quantum-Safe Cryptographic Algorithms. https://openquantumsafe.org/liboqs/

[10] D. McCann, E. Oswald, and C. Whitnall. ELMO: Emulating Leakage for the ARM Cortex-M0 without Access to a Side Channel Lab. In *26th USENIX Security Symposium*, pp. 399–416, 2017. https://github.com/sca-research/ELMO