Breaking Optimized HQC: The First Cache-Timing Full Decryption Oracle Key-Recovery Attack in Post-Quantum Cryptography

Haiyue Dong[0009−0007−0349−7998]✉ and Qian Guo[0000−0003−0930−3174]

Dept. of Electrical and Information Technology, Lund University, Lund, Sweden
{haiyue.dong,qian.guo}@eit.lth.se

Abstract. Hamming Quasi-Cyclic (HQC) has been selected by NIST for standardization in the post-quantum landscape. As deployment approaches, implementation security becomes as critical as mathematical hardness. In this work, we demonstrate that source-level constant-time coding is not a standalone guarantee: the compiled binary must inherently preserve this behavior.

We identify a severe compiler-induced vulnerability within the official `AVX2-optimized` implementation of HQC, despite its claims of being constant-time. Although the C source code relies on secure, mask-based conditional selection, certain compiler optimizations rewrite this logic systematically. This transformation silently introduces secret-dependent control flow into the inner Reed-Muller decoding process, resulting in secret-dependent cache access patterns.

Exploiting this vulnerability, we mount, to the best of our knowledge, the first cache-timing Full-Decryption-style oracle attack against a post-quantum cryptosystem. Using Flush+Reload on shared libraries, an unprivileged co-located adversary can extract fine-grained predicates of the decoder's internal state. To achieve full key recovery, we develop a novel, reliability-aware Soft Information Set Decoding (Soft-ISD) post-processing framework. Leveraging a GPU-accelerated meet-in-the-middle strategy optimized for heterogeneous platforms (including Apple Silicon), we demonstrate end-to-end secret key recovery for hqc-1 with less than 10 seconds of online trace collection.

1 Introduction

The transition to Post-Quantum Cryptography (PQC) represents a fundamental reconstruction of the global cryptographic infrastructure. While the National Institute of Standards and Technology (NIST) has standardized the lattice-based ML-KEM (Kyber) [48], the necessity for cryptographic diversity remains paramount to mitigate the risk of hidden algorithmic vulnerabilities. Consequently, Hamming Quasi-Cyclic (HQC) [15] has been selected for standardization by NIST, serving as the critical code-based alternative to lattice schemes.

Built upon the hardness of decoding random quasi-cyclic codes, HQC provides a mature security foundation distinct from lattice assumptions.

As these primitives transition from theoretical specifications to standardized protocols, the locus of security analysis has shifted toward implementation robustness. Within this domain, Side-Channel Attacks (SCA) have surfaced as a paramount concern, requiring exhaustive analysis to guarantee secure deployment. By leveraging physical leakages—ranging from execution timing and power consumption to electromagnetic emissions—these vectors allow adversaries to recover sensitive key information from otherwise mathematically sound primitives. A substantial body of literature has scrutinized HQC's implementation security, e.g., [4, 9–11, 16–19, 22, 24, 26–28, 30, 37, 38, 46, 47, 51, 53, 54, 57], reflecting this priority.

The Evolution of PC Oracle SCA on HQC. The Plaintext Checking (PC) oracle model has historically constituted a major branch of side-channel analysis (SCA) on HQC. This abstraction has proven universally effective at capturing a diverse spectrum of vulnerabilities, ranging from timing leakages [19, 22] and power and electromagnetic (EM) leakages [10, 44, 46, 53] to complex microarchitectural exploits [16, 47] and cache-timing channels [26]. Fundamentally, a standard PC oracle determines whether a specific ciphertext c decrypts to a target message m, providing a binary validity output. However, the SCA landscape is shifting. Mirroring the maturity of lattice-based cryptanalysis—where attacks against schemes like Kyber rapidly evolved from simple failure oracles to extracting more internal states [23, 24, 33–35, 42, 43, 49, 51, 58]—recent research on HQC has extended the PC oracle paradigm toward Multi-Value PC (MV-PC) and Full Decryption (FD) oracles [9, 11]. In the context of HQC, an FD oracle transcends simple validity checks by recovering the intermediate outputs of the n_1 independent Reed-Muller decoding blocks. This granular leakage is devastating, theoretically enabling single-trace key recovery [11]. Yet, a critical gap persists: while established in power and EM analysis, FD-style oracles remain absent in microarchitectural timing attacks, as constant-time implementations are presumed to mask the necessary granular data dependencies.

Fragility of Source-Level Constant-Time Security. The constant-time paradigm stands as the primary defense against microarchitectural timing leakage, necessitating that modern PQC implementations rigorously avoid secret-dependent branches and memory access patterns. Despite this industry-standard objective, the history of HQC security is characterized by a recurring cycle of timing vulnerabilities and subsequent patches [16, 19, 22, 26, 38, 47], highlighting the difficulty of maintaining this property in practice. A critical, yet often overlooked, factor in this fragility is the gap between source-level hygiene and binary-level behavior; aggressive compiler optimizations can silently nullify the security guarantees of mathematically sound code. This discrepancy has been demonstrated in lattice-based schemes, where compilers introduced variable-time division instructions (KyberSlash [6]) or secret-dependent execution flows (CVE-2024-37880 [40]). In the context of HQC, however, this threat surface remains under-explored.

While Lai et al. [28] recently identified compiler-induced leakage in HQC, their analysis was restricted to reference implementations and necessitated a highly privileged threat model—specifically a controlled-channel attack on an AMD SEV-SNP platform (utilizing single-stepping and performance monitoring counters) or physical access for power analysis. In contrast, the resilience of the high-performance `AVX2-optimized` implementation against standard, unprivileged adversaries remains an open question.

Research Questions. This context raises a fundamental question: does the current `AVX2-optimized` HQC implementation—the August 2025 version—truly preserve its constant-time semantics after the compiler has finished its work? Specifically, can aggressive compiler heuristics be exploited to derive a powerful FD-style oracle via microarchitectural timing? Furthermore, deriving this oracle is only half the challenge; robust post-processing plays a critical role in translating these noisy observations into a full secret key recovery, yet a significant gap remains in the literature. While previous works (e.g. [9–11]) have largely relied on theoretical estimations or simulations to evaluate the complexity of key recovery, there is a lack of efficient, GPU-accelerated software capable of exploiting probabilistic soft information from noisy side channels to perform an end-to-end attack on code-based schemes.

Technical Overview. In this work, we bridge the gap between theoretical side-channel oracles and practical exploitation by demonstrating that FD-style attacks on code-based cryptography are feasible via microarchitectural timing—even against implementations presumed to be secure. Our attack exploits a critical discrepancy between source code and binary behavior in the official `AVX2-optimized` HQC implementation. While the C source code strictly employs mask-based constant-time logic to determine the peak correlation in the Reed-Muller decoder, we discover that the Clang compiler (specifically `clang-14` or `clang-15` at `-O3` optimization) aggressively rewrites this logic. To optimize for register usage, the compiler transforms the bitwise masking into a chain of conditional branches (e.g., `cmp rdi, i; je ...`). This optimization silently introduces a secret-dependent control flow where the execution path—and consequently the memory access pattern—depends directly on the lower 4 bits of the intermediate message $\hat{m}$.

We exploit this leakage using a Flush+Reload attack [59,60] on the shared library. We identify a specific cache line that is accessed if and only if the decoded message fragment satisfies a specific modular condition, specifically $\hat{m} \pmod{16} \in \{3, 4, 5, 6\}$. This allows us to construct an FD-style oracle: rather than recovering the exact message as in power analysis, we recover a binary predicate indicating whether the message lies within this *hit* set. We call it FD-style because it leaks knowledge of $\hat{m}$ per Reed-Muller block, though each block yields only a 1-bit predicate rather than the full byte.

A major challenge in exploiting this leakage is that the CPU processes Reed-Muller blocks significantly faster than the resolution limit of the Flush+Reload

channel. To overcome this resolution gap, we introduce a novel ciphertext construction technique. By injecting structured error patterns that are sparse—consisting mostly of zeros with isolated optimized errors—we effectively separate the processing of target blocks in time. This approach allows us to reliably extract leakage for specific key blocks without signal interference, despite the limited temporal resolution of the side channel.

Ultimately, the cache attack yields soft information (log-likelihood ratios) about the secret key bits rather than determining them with certainty. To recover the key from this noisy data, we formulate it as a syndrome decoding problem with soft information and develop a high-performance Soft Information Set Decoding (Soft-ISD) software tailored for heterogeneous CPU-GPU architectures. By utilizing a 2-list meet-in-the-middle strategy optimized for the Unified Memory of Apple Silicon, we eliminate the PCIe bottleneck usually associated with such large list-merging operations, enabling the recovery of the full secret key in seconds or minutes.

Summary of Contributions. Our core contributions are:

- We uncover a critical vulnerability in the official `AVX2-optimized` implementation of HQC, demonstrating that aggressive compiler optimizations (specifically `clang-14/15 -O3/-Os/-Oz`) systematically refactor the developers' mask-based, constant-time logic into secret-dependent conditional branches. This silent regression effectively nullifies the protocol's implementation security guarantees despite the correctness of the source code.
- We construct the first FD-style oracle attack on a post-quantum cryptosystem via cache timing. Unlike traditional PC oracle attacks that yield only a single validity bit, our approach exploits the identified leakage to recover fine-grained predicates of the internal decoder states. We introduce a ciphertext construction method that overcomes the resolution limits of the Flush+Reload channel, enabling the extraction of information from multiple Reed-Muller blocks within a single trace.
- We develop a novel, reliability-aware Soft-ISD decoder software to bridge the gap between noisy microarchitectural observations and full key recovery. Our implementation utilizes a two-list meet-in-the-middle strategy optimized for heterogeneous CPU-GPU architectures, exploiting Unified Memory to eliminate data transfer bottlenecks during the list-merging phase. This efficiency gain establishes a new benchmark: under a 30-minute post-processing constraint, our solver requires only 276 queries to achieve a 50% success rate in the perfect PC oracle model for hqc-1, representing a 40% reduction in query complexity compared to the previous state-of-the-art [10].
- We demonstrate the attack's practicality through an end-to-end exploit on a standard, unprivileged system without administrative access, strictly controlled laboratory conditions, or disabled frequency scaling. In our experiments using a commodity CPU, we achieve full key recovery for hqc-1 with less than 10 seconds of online trace collection and under 30 minutes of offline post-processing.

Responsible Disclosure. We formally notified both the HQC design team and the National Institute of Standards and Technology (NIST) of the compiler-induced timing vulnerabilities in February 2026. The HQC team acknowledged our report, confirming that they are working to improve the implementation against compiler-induced vulnerabilities and that our findings will assist in these mitigation efforts.

Outline. The remainder of this paper is organized as follows. Section 2 introduces the necessary background on the HQC KEM, the FD attack on HQC, and cache timing attacks, followed by the threat model and attack framework in Section 3. In Section 4, we detail the discovery of the compiler-induced vulnerability and our proposed oracle. This lays the groundwork for Section 5, which presents practical cache timing attacks using Flush+Reload. Section 6 then describes our optimized Soft-ISD decoder implementation tailored for Apple Silicon GPUs. Finally, Section 7 provides a discussion of our findings, and Section 8 concludes the paper.

The code is available at `https://github.com/ot-pca/cache-timing`.

2 Preliminary

We begin by introducing the notations and the specifications of the HQC KEM.

Notations. We denote the finite field with two elements by $\mathbb{F}_2$ and the field with 2^8 elements by $\mathbb{F}_{2^8}$. Since arithmetic is performed in a field of characteristic two, addition and subtraction are equivalent to the bitwise XOR operation; consequently, we use the operators $+$, $-$, and $\oplus$ interchangeably.

Throughout this paper, polynomial operations are performed within the ambient ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$, where n is a prime integer. We identify polynomials in $\mathcal{R}$ with vectors in $\mathbb{F}_2^n$ via a natural isomorphism, using their representations interchangeably. For a vector $v \in \mathbb{F}_2^n$ (or its corresponding polynomial), we define the Hamming weight $\omega(v)$ as the number of its non-zero entries. The multiplication of two polynomials $x, y \in \mathcal{R}$ is denoted by $x \cdot y$.

We use the notation $x \leftarrow\!\!\$\ S$ to represent the uniform sampling of an element x from a finite set S. If S is a distribution, $x \leftarrow\!\!\$\ S$ denotes sampling according to that distribution. Concatenation of bit strings or vectors is denoted by $\|$. Matrices are denoted by bold uppercase letters (e.g., $\mathbf{G}, \mathbf{H}$), and vectors generally by lowercase letters. Finally, for integers a and b, the notation $[a, b]$ represents the integer set $\{a, a + 1, \ldots, b\}$.

2.1 HQC KEM

HQC (Hamming Quasi-Cyclic) [15] is a code-based Key Encapsulation Mechanism (KEM) operating in the Hamming metric. The scheme relies on the Decisional Quasi-Cyclic Syndrome Decoding (DQCSD) problem, basing its security on the hardness of decoding a random quasi-cyclic code.

Table 1: Detailed HQC parameter sets [15]. The inner code utilizes a duplicated Reed-Muller code derived from a base $[128, 8, 64]$ code: a duplication factor of 3 results in parameters $[384, 8, 192]$, while a factor of 5 yields $[640, 8, 320]$. The outer code is a shortened Reed-Solomon code.

	Outer RS Code			Inner RM Code		Global Parameters		
Instance	n_1	k_1	d_{RS}	n_2	d_{RM}	n	ω	$\omega_r = \omega_e$
hqc-1	46	16	31	384	192	17 669	66	75
hqc-3	56	24	33	640	320	35 851	100	114
hqc-5	90	32	59	640	320	57 637	131	149

Parameters and Codes HQC utilizes a linear code $\mathcal{C}$ of dimension k_m and length $n_1 n_2$ over $\mathbb{F}_2$. This code $\mathcal{C}$ is a concatenated code composed of two distinct layers:

- **Outer Code:** A shortened Reed-Solomon (RS) code $[n_1, k_1, d_{RS}]$ defined over $\mathbb{F}_{2^8}$.
- **Inner Code:** A duplicated Reed-Muller (RM) code $[n_2, 8, d_{RM}]$, constructed by repeating a base $[128, 8, 64]$ RM code (e.g., 3 or 5 times).

The scheme is parameterized by the tuple $(n, k_m, \omega, \omega_r, \omega_e)$, where n is the prime defining the ambient ring $\mathcal{R}$, k_m is the message length (k_1 bytes), ω is the Hamming weight of the secret key vectors (x, y), and (ω_r, ω_e) is the weight of the vectors r_1, r_2, and e used during encryption. Table 1 details these system parameters alongside the specific configurations for the concatenated error-correcting codes.

Encryption Scheme The underlying Public Key Encryption (PKE) scheme works as follows:

- KeyGen: The secret key is $sk = (x, y)$ where $x, y \leftarrow\!\!\$\ \mathcal{R}$ are sampled uniformly at random with weight ω. The public key is $pk = (h, s)$ where $h \leftarrow\!\!\$\ \mathcal{R}$ is uniformly random and $s = x + h \cdot y$.
- Encrypt(pk, m): To encrypt a message $m \in \{0,1\}^{k_m}$, the sender generates error vectors $e, r_1, r_2 \leftarrow\!\!\$\ \mathcal{R}$ with weights $\omega(e) = \omega_e$ and $\omega(r_1) = \omega(r_2) = \omega_r$. The ciphertext is $c = (u, v)$ where $u = r_1 + h \cdot r_2$ and $v = m\mathbf{G} + s \cdot r_2 + e$. Note that the term $s \cdot r_2 + e$ is truncated to match the length of the code $\mathcal{C}$.
- Decrypt(sk, c): The receiver computes $V = v - u \cdot y = m\mathbf{G} + (x \cdot r_2 - r_1 \cdot y + e)$. The term $e' = x \cdot r_2 - r_1 \cdot y + e$ acts as an error vector. Decoding succeeds if $\omega(e') \leq \delta$.

Decoding HQC utilizes a concatenated coding scheme $\mathcal{C}$ composed of an outer Reed-Solomon code and an inner duplicated Reed-Muller code. As illustrated in Figure 1, the decoding process is hierarchical and highly parallel. The decoder

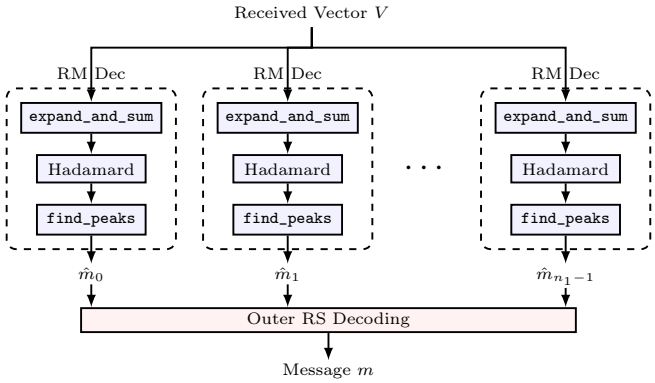


Fig. 1: HQC decoding process. The received vector is partitioned into blocks, each processed by a RM decoder instance involving: (1) `expand_and_sum`, (2) Hadamard transform, and (3) `find_peaks`. The recovered bytes are then passed to the outer RS decoder.

initially partitions the received vector into n_1 separate blocks. Each block is passed to an inner RM decoder instance. Within each instance, the decoder first aggregates the repeated segments (e.g., via `expand_and_sum`) and applies a transform. The critical operation is *finding the peak* in the transformed vector (via `find_peaks`), which corresponds to the recovered message byte. Finally, the sequence of recovered bytes is passed to the outer RS decoder for final error correction.

KEM Construction The HQC KEM is derived from the PKE scheme using a variant of the Fujisaki-Okamoto transformation [14, 25] to achieve IND-CCA2 security. Specifically, it follows the HHK framework with implicit rejection [25]. The KEM algorithms are defined as follows:

- KeyGen: The encapsulation key pk_{KEM} is identical to the PKE encryption key pk. The decapsulation key sk_{KEM} contains sk, pk, a randomness σ, and the seed used to generate them.
- $\mathsf{Encaps}(pk_{KEM})$: To encapsulate a shared secret, the sender samples a message $m \leftarrow\!\!\$\ \{0,1\}^{k_m}$ and a random salt $salt$. It computes the shared secret K and encryption randomness θ as $(K, \theta) \leftarrow G(H(pk_{KEM})\|m\|salt)$. The ciphertext is $c_{KEM} = (c_{PKE}, salt)$, where $c_{PKE} \leftarrow \mathsf{PKE.Encrypt}(pk_{KEM}, m; \theta)$. The output is (K, c_{KEM}).
- $\mathsf{Decaps}(sk_{KEM}, c_{KEM})$: The receiver parses c_{KEM} as $(c_{PKE}, salt)$ and decrypts $m' \leftarrow \mathsf{PKE.Decrypt}(sk, c_{PKE})$. It then re-computes (K', θ'), where $(K', \theta') \leftarrow G(H(pk_{KEM})\|m'\|salt)$, and re-encrypts c'_{PKE}, where $c'_{PKE} \leftarrow \mathsf{PKE.Encrypt}(pk_{KEM}, m'; \theta')$. If $c'_{PKE} = c_{PKE}$ and $m' \neq \perp$, the shared key is K'. Otherwise, the function outputs a pseudorandom rejection key $\bar{K}$ derived from sk_{KEM} and c_{KEM}.

2.2 Overview on FD-Oracle-Based Key Recovery Attack on HQC

This subsection provides an overview of the standard FD attack in [9]. We refer interested readers to the original paper for further details. Dong and Guo [9] define the FD oracle for HQC as one that returns the n_1 inner RM decoding results in one query:

$$\mathcal{O}_{\mathrm{FD}}(c) = (o_0, \ldots, o_{n_1-1}), \qquad o_i \in \mathbb{F}_2^8. \tag{1}$$

For an oracle with perfect accuracy, $o_i = \hat{m}_i$, where $\hat{m}_i \in \mathbb{F}_2^8$ is the RM decoding result for block i.

Ciphertext Construction. To isolate the secret key parts x and y, specific r_1 and r_2 are used: setting $(r_1, r_2) = (0, 1)$ yields a received vector $V = m\mathbf{G} + x + e$, and setting $(r_1, r_2) = (1, 0)$ yields $V = m\mathbf{G} + e - y$. For both, the target secret key part effectively becomes an addition to the error vector e given message m.

The error vector e is constructed to exploit the parallel structure of the inner RM decoding. The received vector V is partitioned into n_1 disjoint blocks each processed by an independent RM decoder. This implies a corresponding partitioning of the secret key vector and the error vector. The vector e is thus configured by repeating a single optimized error pattern $\epsilon \in \mathbb{F}_2^{n_2}$ across all n_1 blocks (i.e., $e = (\epsilon || \ldots || \epsilon)$). Since the secret key blocks are identically distributed, this makes the per-block decoding comparable and the entire n_1 inner RM decoding can be reduced to analyzing a single RM decoding.

Offline Error Pattern Optimization and Template Construction. The error patterns ϵ are selected offline via a genetic-style algorithm with the objective of maximizing the empirical entropy of the RM decoding outputs (evaluated by repeatedly decoding $\epsilon + \xi$ for ξ sampled from the secret key block distribution).

For an optimized pattern ϵ and a given message m, a template $\mathcal{T}_{\epsilon,m}$ is constructed to link each attainable RM decoding output $\hat{m}$ to the reliabilities of the secret key bits. Concretely, n_2-length vectors ξ following the secret key block distribution are repeatedly sampled, added to the pattern ϵ (i.e., $\epsilon + \xi$), fed into the RM decoder, and classified based on the decoding outputs. For each observed output, the empirical probability of being 1 can thus be estimated for each position in a secret key block. This template serves as a lookup table that maps oracle outputs to reliabilities of the secret key bits.

Online Attack and Post-Processing. In the online phase, the chosen ciphertexts constructed as above are submitted to the target device for decapsulation, enabling the construction of an FD oracle through observed side-channel leakages. Given the oracle outputs, the offline templates are consulted to retrieve the reliabilities of the secret key bits. These reliability scores are accumulated across multiple queries with different error patterns, thereby reducing the uncertainty of the secret key bits. The online phase of the standard FD attack [9] is summarized in Algorithm 2 in Appendix A. The final reliability scores serve as the input to an ISD algorithm for key recovery.

2.3 Cache Side-Channels and Flush+Reload

Microarchitectural side-channels exploit the timing differences inherent in the memory hierarchy, allowing adversaries to infer execution patterns by monitoring access latencies. Among the various classes of cache attacks, techniques are generally categorized based on their requirement for shared memory. Methods such as Prime+Probe [29, 36] rely on cache set contention; they require no shared memory but suffer from lower resolution and higher noise. Conversely, when the attacker and victim share physical memory pages—a common occurrence with shared libraries in modern operating systems—Flush+Reload [60] becomes the optimal choice due to its high signal-to-noise ratio and fine-grained spatial resolution.

In this work, we leverage Flush+Reload to monitor specific instruction flow. This attack targets the Last-Level Cache (LLC) and operates in three distinct phases:

1. **Flush:** The attacker actively evicts a specific memory line corresponding to a target instruction or data address from the entire cache hierarchy using the unprivileged `clflush` instruction.
2. **Wait:** The attacker idles for a predetermined interval. During this window, the victim process continues execution and may or may not access the monitored memory line.
3. **Reload:** The attacker accesses the memory line again and carefully measures the reload time.

The interpretation of the signal is binary: a short reload time indicates a cache *hit*, implying the victim accessed the line during the wait phase and pulled it back into the cache. Conversely, a long latency suggests the data was fetched from the main memory, indicating the victim did not trigger the monitored event.

3 Threat Model and Attack Overview

3.1 Threat Model

We consider a standard unprivileged attacker co-located with the victim on the same physical machine. The adversary does not require administrative privileges or physical access to the device; they only require the ability to execute user-space code. We assume the victim and the attacker share the Last Level Cache (LLC), and we do not require Simultaneous Multithreading (SMT/Hyper-Threading). Our attack is effective across physical cores.

The target HQC implementation is assumed to be deployed as a dynamically linked shared library. This is a standard configuration in modern operating systems where cryptographic libraries are installed via package managers and shared among processes. Because the library is shared, the attacker can verify the presence of the vulnerability and determine the exact memory offsets of the

target functions (e.g., `find_peaks`) by loading and profiling the library. By default, our analysis focuses on `hqc-1`; however, the methodology is generic and extends to other parameter sets with minimal adaptation.

We employ the Flush+Reload technique [60] to monitor memory access patterns with high temporal resolution. In the cryptographic setting, we model a Chosen-Ciphertext Attack (CCA). The adversary (Alice) crafts specific, mathematically structured ciphertexts and submits them to the victim (Bob) for decapsulation. By observing the cache *hits* and *misses* on the vulnerable code paths during Bob's decapsulation, the adversary constructs a Full Decryption (FD) style oracle to recover the secret key.

3.2 General Idea of New FD-Style Attack via Cache Timing

We identify a compiler-induced vulnerability that introduces conditional branching in the inner RM decoding, causing a specific cache line to be accessed only for certain decoding outcomes. Using Flush+Reload and chosen ciphertexts to probe that cache line, we obtain *hit*/*miss* signals to build an FD-style oracle.

Let $\hat{m}_i \in \mathbb{F}_2^8$ denote the decoded RM message byte for block i. Unlike the standard FD oracle defined in Equation (1), our side-channel oracle with perfect accuracy does not reveal $\hat{m}_i$ directly; it instead reveals a single bit:

$$o_i = \mathbf{1}[\hat{m}_i \in \mathcal{H}], \qquad o_i \in \mathbb{F}_2,$$

where $\mathcal{H} \subsetneq \mathbb{F}_2^8$ is the set of values that trigger a cache *hit*. Section 4.2 provides the precise definition of $\mathcal{H}$.

Unlike in power leakage settings on microcontrollers which provide richer per-trace decoding information (e.g., [9,37]), the RM decoding on modern processors runs too fast to reliably resolve every block with a single query. Therefore, we design ciphertexts to target only a few blocks per query. Figure B.1 in Appendix B summarizes our attack; its framework follows the standard FD attack in [9].

4 New Vulnerability and Cache-Timing Side Channel

In this section, we first demonstrate the compiler-induced vulnerability within the inner Reed-Muller decoding and how it can be exploited to build a cache line *hit*/*miss* oracle via Flush+Reload. We then show the practical constraints in building a standard FD oracle and propose two ciphertext construction methods that enable an FD-style attack.

4.1 Vulnerability Discovery

Our analysis identifies a critical discrepancy between the code designers' intent and the resulting `AVX2-optimized x86_64` binary execution within the `find_peaks` function of the inner RM decoder. This function is responsible for processing the sensitive decoding result of each RM block. As illustrated in Listing 1,

Listing 1: Excerpt of `find_peaks` in `reed_muller.c` for `avx256`

```
152 inline int32_t find_peaks(rm_expanded_cdw128_t *transform) {
        ...
229     uint16_t result = 0;
230     for (uint32_t i = 0; i < 16; i++) {
231         uint16_t *ptr = (uint16_t *)&tmp;
232         int32_t message_mask = (-(int32_t)(i == message % 16)) >>
      ↪ (sizeof(int32_t) * 8 - 1);
233         result |= message_mask & ptr[i];
234     }
235     message |= (0x8000 & ~result) >> 8;
236     return message;
237 }
```

the C source code is explicitly written to adhere to constant-time programming principles: to determine the peak correlation, it employs bit-wise masking and logical operations. Under standard assumptions, this approach is intended to keep the instruction pointer's path independent of the secret message bits.

However, we observe that this source-level guarantee is effectively nullified by the compiler during the optimization phase. Specifically, we analyze the official HQC optimized implementation compiled using `clang-14` and `clang-15` with the `-mavx2` target. When compiling with high-level optimization flags, i.e., `-O3`, `-Os`, or `-Oz`, the compiler aggressively refactors bit-wise logic into conditional control-flow instructions. This creates a secret-dependent execution path.

The following section demonstrates this vulnerability on the `clang-14` build with `-O3`.[1] We note that while our analysis utilizes debugging symbols to map source lines to memory addresses, the vulnerability persists in stripped binaries and requires only standard reverse engineering to exploit. As these symbols are not loaded at runtime, their inclusion does not affect the victim's run.

Binary Control Flow Analysis The compiled `AVX2-optimized x86_64` binary implements the final selection step (Line 232–233 in Listing 1) as a chain of secret-dependent `cmp` and `je`/`jne` branches. As shown in the disassembly generated with Rizin [45] in Listing 2, the control flow is decided by the register `rdi`, which holds the lower 4 bits of the decoded message, i.e., `rdi` $= \hat{m} \bmod 16$.

The execution sequentially compares `rdi` against each $i \in [0, 15]$. It begins by testing `rdi` against 0 and depending on the comparison result, branches to one of two initial blocks. Both initial blocks proceed to compare `rdi` against 1. If this comparison evaluates to true, regardless of the previous branch, execution moves to a specific block (hereafter referred to as a *compare-true block*). Conversely, if the comparison evaluates to false, execution moves to a different specific block (hereafter referred to as a *compare-false block*). Both destination blocks then proceed to compare `rdi` against 2.

[1] Specifically, we target the `next-release` branch at commit `296425bd` of the repository `https://gitlab.com/pqc-hqc/hqc`. The analysis uses the build produced with `clang-14` version 14.0.0-4ubuntu1, the default compiler for Ubuntu 22.04 LTS, and is verified across several subsequent releases.

Listing 2: Conditional branching in disassembly

```
...
b5b9 mov      edi, ecx
b5bb and      edi, 0xf
b5be test     rdi, rdi
b5c1 je       b760
b5c7 mov      dword [var_270h], 0
b5d2 cmp      rdi, 1
b5d6 jne      b778
b5dc vpextrw  edx, xmm0, 1
b5e1 mov      dword [var_290h], edx
b5e5 cmp      rdi, 2
b5e9 jne      b78a
...
b6b6 vpextrw  r11d, xmm0, 6
b6bb cmp      rdi, 0xf
b6bf jne      adc0
...
b760 vmovd    edx, xmm0
b764 movzx    edx, dx
b767 mov      dword [var_270h], edx
b76e cmp      rdi, 1
b772 je       b5dc
b778 mov      dword [var_290h], 0
b780 cmp      rdi, 2
b784 je       b5ef
...
b838 xor      r11d, r11d
b83b cmp      rdi, 0xf
b83f jne      adc0
```

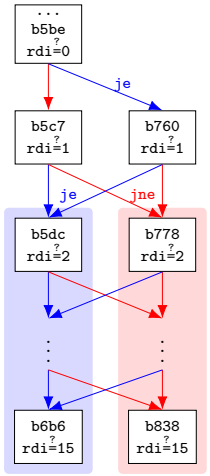


Fig. 2: Illustration of conditional branching

In summary, for all $i \in [1, 15]$, a comparison (`cmp rdi`, i) is executed, followed by a conditional jump (`je`/`jne`). If `rdi` equals i, the execution moves to a unique *compare-true block*; otherwise, it moves to a unique *compare-false block*, regardless of the path taken. Both blocks then proceed to compare `rdi` against $i + 1$. Crucially, the compiler arranges *compare-true* and *compare-false blocks* in separate memory regions, as indicated by their offsets. This conditional branching is illustrated in Figure 2. The *compare-true blocks* are shown on the left panel with a blue background, and the *compare-false blocks* are shown on the right panel with a red background.

For a given decoded message $\hat{m}$ where $\hat{m} \not\equiv 0 \pmod{16}$, `rdi`$= i$ evaluates to true exactly once. Consequently, the execution passes through exactly one *compare-true block*, while visiting the *compare-false blocks* for all other i.

Conditional Cache Line Access Pattern Each *compare-true* and *compare-false block* effectively partitions the message space $\hat{m} \in \mathbb{F}_2^8$ into two disjoint subsets. For example, block `0xb5dc` is the *compare-true block* for `rdi` $= 1$. If it is executed, it implies $\hat{m} \equiv 1 \pmod{16}$. If, instead, the *compare-false block* at `0xb778` is executed, it implies $\hat{m} \not\equiv 1 \pmod{16}$.

Such partitioning creates an exploitable side channel for attackers monitoring cache access patterns. Specifically, if a cache line contains blocks that are executed only by a subset of messages, a cache *hit* on that line reveals that the message lies within that subset; conversely, a *miss* indicates it lies in the complement. In this way, the cache line partitions the messages into disjoint *hit* and

miss subsets. We are able to identify several cache lines satisfying this property. For example, the cache line at `0xb600` contains instructions from the blocks at offsets `0xb5fe`, `0xb611`, `0xb629`, and `0xb638`, which are the *compare-true blocks* executed for `rdi` $\in \{3, 4, 5, 6\}$.[2] We use the cache line `0xb600` as our target for the analysis and attacks presented in the following sections.

4.2 From Cache Hit/Miss Oracle to Key Block Structure

Cache Hit/Miss as an Oracle As a result of this secret-dependent branching, the cache line `0xb600` is accessed if and only if the lower 4 bits of the decoded RM message $\hat{m}$ lies in $\{3, 4, 5, 6\}$, i.e., $\hat{m} \bmod 16 \in \{3, 4, 5, 6\}$. Equivalently, the *hit* set is:

$$\mathcal{H} = \{x \in \mathbb{F}_2^8 \ : \ x \bmod 16 \in \{3, 4, 5, 6\}\},$$

and therefore $\varnothing \neq \mathcal{H} \subsetneq \mathbb{F}_2^8$; specifically, $|\mathcal{H}| = 64$.

The FD-style oracle with perfect accuracy, which returns n_1 *hit*/*miss* results simultaneously given the query c, is defined as:

$$\mathcal{O}_{\mathrm{FD}}(c) = (o_0, \ldots, o_{n_1-1}), \quad o_i = \mathbf{1}[\hat{m}_i \in \mathcal{H}]. \tag{2}$$

Compared with the standard FD oracle from [9] shown in Equation (1), where o_i reveals the full decoded byte value for block i ($o_i \in \mathbb{F}_2^8$), this variant provides only a binary cache *hit*/*miss* indicator ($o_i \in \mathbb{F}_2$).

Linking Cache Hit/Miss to Key Block Structure We employ the offline optimization framework in the standard FD attack [9] shown in Section 2.2 to find error patterns ϵ that maximize the information leakage of our specific binary side channel. Unlike the standard FD attack where entropy is calculated over the attainable decoding outcomes, here we seek ϵ such that the decoding output of $\epsilon+\xi$ (ξ is a random vector sampled according to the secret key block distribution) lands in the *hit* set $\mathcal{H}$ with approximately 50% probability, thereby maximizing the entropy of the binary *hit*/*miss* outcome.

We construct the offline template $\mathcal{T}_{\epsilon,m}$ for each of the optimized patterns ϵ and a given m following the same methodology as the standard FD attack [9]: by repeatedly sampling ξ and decoding $\epsilon + \xi$, we estimate the empirical probability of a secret key bit being 1, conditioned on whether the decoded output $\hat{m}$ lands in the *hit* set $\mathcal{H}$ or its complement. The offline template thus links the observable binary outcome, *hit* or *miss*, with the reliabilities of the secret key bits. As illustrated in Figure C.1 in Appendix C, the resulting reliabilities exhibit distinct deviations from the a priori probability $p = \frac{\omega}{n}$, confirming that a cache *hit*/*miss* leaks the structure of the secret key block.

[2] These offsets correspond to the build with `clang-14` v14.0.0-4ubuntu1. While offsets vary across `clang-14/15` versions, the underlying branching structure and thus the vulnerability and attack methods remain consistent.

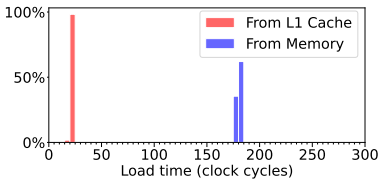


Fig. 3: Distribution of load time from 100 000 runs.

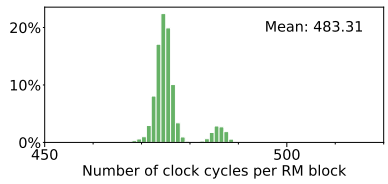


Fig. 4: Distribution of the estimated number of cycles per RM block decoding from 10 000 runs.

4.3 Cache-Timing Side Channel Calibration and Constraints

To exploit the identified vulnerability, we first calibrate a timing threshold to distinguish cache *hit* from *miss* and determine if the temporal resolution of the Flush+Reload channel is sufficient to capture the leakage from each of the RM blocks.

We measure the load time for data residing in the L1 cache and in main memory on our platform, a 2019 MacBookPro16,1 with an Intel i7-9750H processor running Ubuntu 22.04 LTS. For data from the L1 cache, the loading latency is between 18–24 clock cycles, including memory fences and measurement overhead, as shown in Figure 3. In contrast, fetching data from main memory requires about 170–190 cycles. Very few extremely large values are not plotted. Since the LLC latency falls between these two extremes, we select a threshold of 110 clock cycles. Load times below this value are classified as a cache *hit*, indicating the victim has accessed the line, while those above are classified as a *miss*.

The main practical challenge lies in associating *hit*/*miss* observations with the corresponding iterations of the RM decoding loop. In an ideal setting, a single decapsulation trace returns the *hit*/*miss* signals for each of the n_1 RM blocks simultaneously. However, programs running on high-performance CPUs typically execute fast, while the Flush+Reload probe cannot be too frequent. To quantify this constraint, we instrument the shared HQC library using dynamic symbol interposition and wrap the `reed_muller_decode` function to capture timestamps immediately at the entry and exit points of this function. By dividing the total execution time by n_1 (46 for `hqc-1`), we obtain an estimate of the processing time for a single RM block. As shown in Figure 4, a single RM block decoding takes 483 clock cycles on average. The plotted range covers 98.44% of the samples; the rest take more cycles.

In contrast, a reliable Flush+Reload probe typically requires a longer interval, because the adversary must busy-wait long enough to allow the victim to access (or not access) the monitored cache line, while also accounting for the overhead of the `clflush` instruction, memory fences, and measurement instructions. Probing at a high frequency risks interfering with the victim's access [60], making the signal noisy. Furthermore, temporal alignment is another challenge: since the blocks are processed sequentially, it is difficult to distinguish signals

from adjacent iterations (i.e., identifying which signal belongs to which RM block) especially in the presence of prefetching and system jitter.

To overcome this, we design the ciphertext, precisely the error vector e, to isolate or space out the target blocks, effectively reducing the signal frequency to match our sampling capability. We propose two specific ciphertext construction methods, detailed in the following subsection, to extract signals only for one or a few target blocks with each ciphertext.

4.4 Ciphertext Construction

Following [9], we construct chosen ciphertexts with $m = 0$ by default and select $(r_1,\ r_2)$ to isolate specific key parts: to probe the secret key part x, we set $(r_1,\ r_2) = (0,\ 1)$ yielding $(u,\ v) = (h,\ s + e)$; to probe the part y, we set $(r_1,\ r_2) = (1,\ 0)$, yielding $(u,\ v) = (1,\ e)$. These are the same as in the standard FD attack.

Regarding the error vector e, due to the Flush+Reload frequency constraints discussed, we use the following two methods to construct e such that each ciphertext only targets one or three blocks of the secret key.

Method 1: Targeting One Block For each ciphertext we target exactly one block b of the secret key by setting the corresponding block in the error vector e to the optimized error pattern ϵ, and the remaining blocks to a cache-*miss* pattern (e.g., 0^{n_2}):

$$e = (e_0 \| \cdots \| e_{n_1-1}), \quad e_i = \begin{cases} \epsilon, & i = b, \\ 0^{n_2}, & i \neq b. \end{cases} \tag{3}$$

The cache measurement on this ciphertext yields a single oracle output $o_b \in \mathbb{F}_2$ associated with the target block only. Repeating this construction for every $b \in [0,\ n_1 - 1]$ gives the oracle outputs for all the n_1 blocks. The total required number of ciphertexts for one key is $2 \times n_1 \times \tau$, where τ is the number of error patterns used.

Method 2: Targeting Three Blocks To exploit the parallelism while maintaining reliable temporal separation between blocks, we construct e such that the error pattern ϵ is placed in regularly spaced blocks, while the remaining blocks are set to the cache-*miss* pattern 0^{n_2}. In this way, each ciphertext is used to extract information only about the corresponding spaced blocks in the secret key. This spacing ensures that the cache signal from a target block is separated by several non-target cache-*miss* blocks, thereby reducing inter-block interference and improving classification accuracy.

Concretely, we set the first block of e to an always-*hit* pattern ϵ_H to serve as a synchronization marker. Next, we insert a fixed *miss* region consisting of six blocks of 0^{n_2}. For the remaining blocks with indices $i \in [7,\ n_1 - 1]$, we distribute the error pattern ϵ using residue classes modulo 13. Specifically, for

each residue $\hat{r} \in [0,\ 12]$, we generate one ciphertext by placing the error pattern ϵ in blocks with indices $i \equiv \hat{r} \pmod{13}$, while setting all other blocks to 0^{n_2}. This configuration is summarized as follows:

$$e = (e_0 \| \cdots \| e_{n_1-1}), \quad e_i = \begin{cases} \epsilon_H, & i = 0, \\ 0^{n_2}, & 1 \le i \le 6, \\ \epsilon, & 7 \le i \le n_1 - 1 \ \wedge \ i \equiv \hat{r} \pmod{13}, \\ 0^{n_2}, & 7 \le i \le n_1 - 1 \ \wedge \ i \not\equiv \hat{r} \pmod{13}, \end{cases} \tag{4}$$

where $\hat{r} \in [0,\ 12]$ and one ciphertext is generated for each $\hat{r}$. For `hqc-1`, this means each ciphertext targets three secret key blocks, instead of one as in Method 1, albeit at the cost of losing information about the first 7 blocks entirely.

5 Real-World Attack

We executed the attack on 30 randomly generated key pairs on a 2019 MacBookPro16,1 with an Intel i7-9750H processor running Ubuntu 22.04 LTS with T2 hardware support patches.[3] The CPU features 6 physical cores, each with 2 logical threads (SMT). L1 and L2 are private per physical core. The Last Level Cache (LLC) is shared across all cores and inclusive of the L1 and L2 caches.

We targeted the official `AVX2-optimized` HQC implementation, compiled using `clang-14` v14.0.0-4ubuntu1 with the `-O3` optimization flag and the `-mavx2` target. For this specific build, the monitored instruction is located at offset `0xb611` within the cache line starting at `0xb600`. We utilized the Mastik toolkit [59] to launch the Flush+Reload attack. In the experiment, the victim and adversary programs were pinned to separate physical cores.

We also successfully validated the attack on a desktop workstation with an Intel i7-10700K processor running Ubuntu 20.04 LTS, accessed and monitored remotely via Chrome Remote Desktop. Results on this target are provided in Appendix H.

In the following sections, we present the specific attack settings for the two ciphertext construction methods detailed in Section 4.4.

5.1 Method 1

For a ciphertext c constructed following Method 1 in Section 4.4, the adversary triggers R_1 executions of $\mathsf{Decaps}(sk, c)$ by the victim program and records the reload times while repeatedly probing the monitored address. A similar strategy to that in [60] is adopted to sample at a fixed frequency. Specifically, the adversary program partitions time into fixed 10 000-cycle intervals. In each interval, it reloads the target cache line, measures the access latency, and then immediately

[3] These patches are needed because standard Linux kernels lack native support for the Apple T2 Security Chip on this device, which manages internal peripherals such as the trackpad, keyboard, and SSD.

flushes the line. It then busy-waits until the interval ends before proceeding to the next.

For each key, the number of traces collected is $2 \times n_1 \times \tau \times R_1$, where τ is the number of error patterns used and R_1 is the number of runs for each ciphertext. With $\tau = 5$ and $R_1 = 50$, the execution time of this attack on one key of hqc-1 is approximately 4 seconds, and the number of traces collected is 23 000.

Constructing Oracle from Traces For each ciphertext, we aggregate measurements from R_1 repetitions. In an ideal, noise-free environment, a cache line access would yield exactly R_1 *hits* and the oracle output for that target block is then a *hit*; while an unaccessed line would yield zero *hits*, and the oracle output is then a *miss*. However, to account for microarchitectural noise (e.g., prefetching, interrupts), we define the oracle output as a *hit* only if the total hit count exceeds 80% of R_1. Figure D.1 in Appendix D illustrates the *hit* counts across $n_1 = 46$ blocks based on ciphertext traces corresponding to one error pattern and one key part.

Given these oracle outputs, we consult the offline templates to calculate the reliability score for each secret key position. Appendix A shows how this attack adapts the online stage of the standard FD attack in [9].

5.2 Method 2

For a ciphertext c constructed following Method 2 in Section 4.4, the adversary triggers R_2 runs of the victim's $\mathsf{Decaps}(sk, c)$ and records one timing record per run by repeatedly probing the monitored address at a fixed frequency (once every 2 000-cycle interval).

For each key, the total number of traces collected is $2 \times 13 \times \tau \times R_2$, where τ is the number of error patterns used and R_2 is the number of runs per ciphertext. The factor 13 represents the number of ciphertexts required to cover the 39 target blocks of hqc-1 ($n_1 = 46$, first 7 blocks are excluded). With $\tau = 30$ and $R_2 = 11$, the execution time of this attack on one key of hqc-1 is approximately 3 seconds, and the number of traces collected is 8 580.

Constructing Oracle from Traces To recover the oracle outputs, we exploit the known structure of the ciphertext where block 0 is designed to trigger a cache *hit*. We align the R_2 traces for each ciphertext relative to block 0 by treating the first observed hit as block 0. Using the average RM decoding duration per block in Section 4.3, we calculate the expected time slots for the three target blocks relative to this anchor and delineate their respective ranges using midpoints between their expected time slots. A *hit* within a specific block's range is attributed to that block. We observe R_2 *hit*/*miss* signals for each target block in a ciphertext, and the oracle output for each target block is decided by a majority voting over the R_2 signals.

Figure 5 visualizes the aligned $R_2 = 11$ traces for one ciphertext. The vertical gray lines delineate the computed ranges for the three target blocks. As shown,

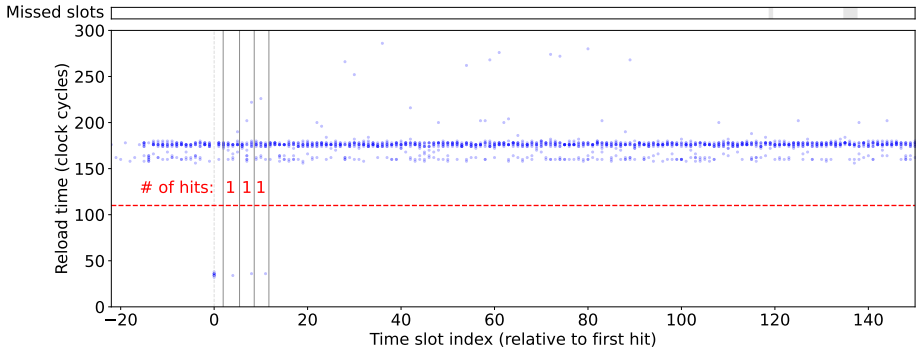


Fig. 5: Reload times across $R_2 = 11$ runs of one ciphertext, aligned to the first-*hit* time slot. Shaded areas in the missed slots panel indicate at least one run does not have samples in the corresponding time slot. The ranges for the three target blocks are delineated by solid gray vertical lines. With only 1 *hit* (out of $R_2 = 11$ signals) observed in each range, the oracle outputs for all three blocks are *miss*.

hits (reload time $<$ 110 clock cycles) are scarce within these boundaries; the oracle outputs for all three blocks are determined to be *miss* via majority voting.

We collect oracle outputs for blocks 7 through 45, excluding the first 7. These predictions are used to consult the offline template to get reliability scores, which represent the probability that a corresponding secret key position is 1. For the key positions in the first 7 blocks, we assign an a priori probability of $p = \frac{\omega}{n}$. Appendix A summarizes this attack's modifications to the standard FD attack [9].

5.3 Results

We reorder the secret key vector positions based on the probability of each position being 1. Specifically, positions are sorted such that those most likely to be 0 are ranked first. Figure 6 illustrates the distribution of Hamming weights within the first n positions of the reordered secret key vectors across 30 keys. For Method 1, fixing the repetition at $R_1 = 50$ while increasing the number of error patterns τ from 4 to 7 results in a reduction in the average Hamming weight from 8.2 to 3.3. For Method 2, the repetition is fixed at $R_2 = 11$ and the number of error patterns increases from 20 to 35 in steps of 5. Similarly, the average Hamming weight decreases from 7.4 to 4.8. However, significant outliers with high Hamming weights persist. While increasing the number of error patterns reduces the average Hamming weight, these outliers remain challenging for the subsequent ISD algorithm. These outliers primarily stem from trace alignment inaccuracies. For instance, if the first observed hit does not actually correspond to the first block, or if variations in the RM decoding duration cause misalignments of subsequent blocks, the oracle predictions become erroneous. For Method 1,

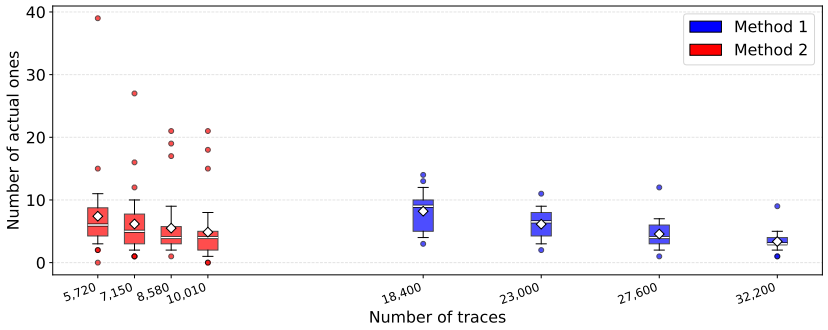


Fig. 6: Distribution of the Hamming weight within the first n positions of the reordered secret key vector across 30 distinct keys. The data is plotted against the number of traces per key on the x-axis. Whiskers denote the 10th–90th percentiles. The white bar and diamond denote the median and mean, respectively.

while it is possible to reduce the number of repetitions per ciphertext, we expect that the oracle accuracy will drop, meaning more error patterns will be needed.

Finally, the reliability scores for the secret key positions serve as the input to the new Soft-ISD implementation detailed in the next section.

6 High-Performance Soft-ISD Post-Processing

Recovering the full secret vector requires solving the decoding problem for random linear codes, a challenge that constitutes the security foundation of code-based cryptography. Information Set Decoding (ISD) remains the primary algorithmic toolkit for this task. While the conceptual basis traces back to Prange [41], it was Stern's seminal work [50] that established the collision-based, meet-in-the-middle paradigm which underpins most modern ISD algorithms [5, 31, 32]. Recently, this landscape has expanded to include distinct approaches such as sieving-based ISD [12, 21], further refining the efficiency of generic decoding.

In the context of side-channel analysis, the standard ISD approach must be adapted to exploit soft information—specifically, the reliability vector derived from a cache-timing oracle that quantifies the likelihood of specific key bits [20, 39]. However, prior research on Soft-ISD, including [11], has predominantly focused on theoretical evaluations. These works typically provide complexity estimations based on the Random Access Machine (RAM) model or simplified simulations, often neglecting the practical bottlenecks of modern hardware, such as memory latency and PCIe transfer overheads. Consequently, a significant gap remains between theoretical leakage models and operational, high-performance key recovery.

We bridge this gap by presenting a novel, open-source 2-List Soft-ISD implementation designed for heterogeneous Unified Memory architectures. Unlike the more theoretical approaches (e.g., [11] detailed in Appendix F), our solver

leverages the Apple Silicon architecture to eliminate the data transfer overheads inherent in traditional CPU-GPU setups. We introduce several algorithmic refinements to maximize throughput: a preprocessing step utilizing the Method of the Four Russians (M4RI) [1] to amortize the cost of Gaussian Elimination; a multi-round strategy that re-randomizes the search within a single decoding attempt to avoid costly restarts; and a partial weight-3 enumeration to optimize list density for the least reliable indices. By highly parallelizing this workflow across independent workers (utilizing 28 threads in our experimental platform), we achieve a balance between memory usage and computational speed that significantly outperforms theoretical baselines.

6.1 Problem Formulation: Soft Syndrome Decoding

We formulate the key recovery as a Soft Syndrome Decoding problem. Given the public key and the soft reliability information derived from the cache-timing oracle, the objective is to recover the sparse secret vectors x and y. The fundamental constraint is the syndrome decoding equation, which we express in matrix form as:

$$\left[\mathbf{I}_n \ \mathrm{cir}(h)\right] \begin{bmatrix} x \\ y \end{bmatrix} = s, \tag{5}$$

where $\mathrm{cir}(h)$ denotes the circulant matrix defined by the public polynomial h, and $s \in \mathbb{F}_2^n$ is the public key component (acting as the syndrome in this formulation). When targeting the HQC cryptosystem using a generic ISD solver with code length n_c, dimension k, and redundancy r, the parameters map as follows: $n_c = 2n$, $k = n$, and $r = n$. The input to our solver includes the reliability vector $L \in \mathbb{R}^{2n}$ derived from the FLUSH+RELOAD traces, where each entry L_j represents the log-likelihood ratio (LLR) of the j-th bit of the concatenated vector $(x||y)$.

6.2 Algorithm Design for Heterogeneous Unified Memory

We propose a high-performance implementation of the Soft-ISD framework, designed specifically to exploit the Unified Memory architecture of Apple Silicon. To overcome the traditional bottleneck of PCIe data transfer between host and device, our solver realizes the core collision search as a *two-list meet-in-the-middle* (MITM) join where CPU and GPU threads operate on shared memory buffers.

Our algorithmic approach adapts the Finiasz–Sendrier (FS) framework [13] to the soft-information setting. In each decoding attempt, we select an information set S of size $k + \ell$ based on the provided LLRs and permute the parity-check matrix columns to satisfy a quasi-systematic form. We introduce two critical optimizations—*fixed-set preprocessing* and a *multi-round strategy*—to minimize the computational cost of Gaussian elimination and maximize the throughput of the GPU-accelerated collision search. Figure E.1 summarizes the architecture; Algorithm 1 provides the pseudocode.

Algorithm 1 Two-List CPU–GPU Soft ISD Decoder (implementation)

Input: $\mathbf{H} \in \mathbb{F}_2^{r \times n_c}$, $s \in \mathbb{F}_2^r$, LLRs $L \in \mathbb{R}^{n_c}$, parameters (k, ℓ, w, α), N_{round}, optional `UN_L`, `UN_R`

Output: error vector $e \in \{0,1\}^{n_c}$ with $\mathbf{H}e^T = s$ and $wt(e) \le w$ (or $\perp$)

(Optional, once) Fix S^c_{fixed} as lowest-LLR columns; M4RI-reduce $[\mathbf{H} \mid s]$ with S^c_{fixed} leading; cache reduced $(\mathbf{H}, s)$ and column maps.
while not solved **do**
  Form candidate pool C from the highest-LLR columns (size $\max(k+\ell, \lceil \alpha k \rceil)$) among the remaining columns.
  $S \leftarrow \textsc{WeightedSampling}(C, k+\ell;\ W_j = 1 + \exp(L_j))$
  $S^c \leftarrow [0, n_c - 1] \setminus S$ (with fixed part S^c_{fixed} if enabled)
  Obtain the systematic split $(\mathbf{H}^U, \mathbf{H}^D, s^U, s^D)$ for the current (S, S^c)
  for $t = 1$ **to** N_{round} **do** ▷ *rerandomize the Mod-4 split only*
    Sort S by decreasing LLR to get ranks $p_0, \ldots, p_{k+\ell-1}$
    Base split: $I_L = \{p_i : i \bmod 4 \in \{0,3\}\}$, $I_R = \{p_i : i \bmod 4 \in \{1,2\}\}$
    Randomly permute assignment within each block
    Kernel A (build): enumerate e_L on I_L with weights 0/1/2 and optional weight-3 subset over the `UN_L` least-reliable indices in I_L; insert keys $k_L = \mathbf{H}^U e_L$ into a hash table
    Kernel B (probe): enumerate e_R on I_R with weights 0/1/2 and optional weight-3 subset over the `UN_R` least-reliable indices in I_R on-the-fly; for each e_R, compute $k_R = \mathbf{H}^U e_R$ and lookup $k_L = s^U \oplus k_R$; emit candidate matches
    for each candidate (e_L, e_R) **do**
      $e_S \leftarrow e_L \oplus e_R$
      $e_{S^c} \leftarrow s^D \oplus (\mathbf{H}^D e_S)$
      if $wt(e_S) + wt(e_{S^c}) \le w$ **and** $\mathbf{H}e^T = s$ **then**
        Recover e in original column order; **return** e
      end if
    end for
  end for
end while
return $\perp$

Preprocessing and Systematization A central component of the FS framework is the transformation of the parity-check matrix $\mathbf{H}$ into a systematic form. This operation, typically cubic in the row dimension, dominates the cost of each decoding attempt. To mitigate this, we employ the Method of the Four Russians (M4RI) [1], which utilizes packed bit-slicing and cache-friendly lookups to accelerate matrix inversion over $\mathbb{F}_2$.

To further amortize the linear algebra overhead, we introduce a *fixed-set prediagonalization*. We observe that error positions are statistically concentrated in columns with low LLRs. Consequently, we partition the column indices into a fixed low-reliability set S^c_{fixed} (consisting of the $n_c - \lceil \alpha k \rceil$ least reliable positions) and a variable high-reliability pool. In a one-time preprocessing step, we prediagonalize the submatrix of $[\mathbf{H} \mid s]$ corresponding to S^c_{fixed}. Subsequent attempts strictly sample the information set from the variable pool. This allows us to reuse

the pre-computed basis, requiring M4RI reduction only on the small subset of swapped columns.

Step 0 (CPU): Weighted Sampling and Split The decoding process proceeds in iterative *attempts*. Let $L_j = \log(\Pr[e_j = 0]/\Pr[e_j = 1])$ be the LLR of position j. We construct a seed pool C comprising the $\min(n_c, \max(k+\ell, \lceil \alpha k \rceil))$ columns with the highest LLRs, where α is set to 1.1 by default. From this pool, we select an information set $S \subset C$ of size $k+\ell$ using weighted random sampling without replacement, where the weight of index j is given by $W_j = 1 + \exp(L_j)$. Following the Finiasz–Sendrier framework, we apply a permutation π that maps the complementary set S^c to the leftmost columns. We then perform Gaussian elimination to reduce the permuted matrix $[\pi(\mathbf{H}) \mid s]$ to the form:

$$\widetilde{\mathbf{H}} = \begin{bmatrix} \mathbf{I}_{r-\ell} & \mathbf{H}^D \\ \mathbf{0} & \mathbf{H}^U \end{bmatrix}, \quad \widetilde{s} = \begin{bmatrix} s^D \\ s^U \end{bmatrix}. \tag{6}$$

This transformation yields the parity constraint $\mathbf{H}^U e_S = s^U$ on the information set, and the recovery equation $e_{S^c} = s^D \oplus (\mathbf{H}^D e_S)$ for the redundancy set.

Step 1 (CPU): Multi-Round Mod-4 Partitioning To fully utilize the GPU's hashing power, we perform N_{round} distinct search *rounds* for every systematic split computed in Step 0. In each round, we re-randomize the MITM partition without repeating the costly Gaussian elimination. Specifically, we sort the indices in S by descending LLR to obtain ranks $p_0, \ldots, p_{k+\ell-1}$ and apply a Mod-4 partitioning rule:

$$I_L = \{p_i : i \bmod 4 \in \{0, 3\}\}, \quad I_R = \{p_i : i \bmod 4 \in \{1, 2\}\}.$$

We further diversify the search space by randomly swapping assignments within blocks of four ranks. This strategy effectively decouples the CPU-bound linear algebra from the GPU-bound collision search, allowing us to test a wider variety of error patterns per matrix reduction.

Step 2 (GPU): Hash-Join Collision Search The collision search is implemented as a parallel hash-join on the GPU. In the build phase (Kernel A), we enumerate error patterns e_L supported on I_L with weights $w \in \{0, 1, 2\}$. For each pattern, we compute the partial syndrome $k_L = \mathbf{H}^U e_L$ and insert the tuple (k_L, e_L) into a hash table in unified memory. In the probe phase (Kernel B), we enumerate error patterns e_R supported on I_R (weights $0, 1, 2$) on-the-fly. For each e_R, we compute $k_R = \mathbf{H}^U e_R$ and query the hash table for a collision satisfying $k_L = s^U \oplus k_R$. To improve success rates for noisy keys, we optionally extend the search space on either the left or right side (or both) by including a subset of weight-3 patterns restricted to the least reliable indices (`UN_L` or `UN_R`) within the respective partition.

Step 3 (CPU): Candidate Validation Matches identified by the GPU are written to a shared candidate queue. The CPU retrieves each candidate tuple (e_L, e_R), reconstructs the full error vector on the information set $e_S = e_L \oplus e_R$, and computes the implied redundancy error e_{S^c}. The candidate is accepted as a valid key if the total Hamming weight satisfies the system parameters and $\mathbf{H}e^\top = s$. If all N_{round} rounds are exhausted without success, the algorithm restarts with a new weighted sampling of S in Step 0.

6.3 Implementation Optimizations

We evaluate our implementation on an Apple Mac Studio equipped with an M3 Ultra processor (24 performance cores, 8 efficiency cores, an 80-core GPU, and 512 GB of unified memory). Profiling reveals that the two-list approach is strictly dominated by the initial Gaussian elimination (Step 0) and the GPU hash-join (Step 2). By setting the matching parameter $l = 59$, candidate validation (Step 3) incurs negligible overhead. Baseline Gaussian elimination via M4RI requires 15–16 seconds per core, and internal multithreading (e.g., OpenMP) yields only marginal improvements.

The hash-join complexity scales linearly with list size. Our measurements indicate that a list size of 2^{33} consumes roughly 400 GB of RAM and requires 80 seconds to complete. However, expanding the list size yields diminishing returns for soft-information syndrome decoding, as highly probable error patterns overwhelmingly dominate. This behavior aligns with Torres and Sendrier [52], who demonstrated that massive RAM or list size offers limited practical improvements for modern ISD [5, 31, 32] applied to extremely sparse error distributions.

Consequently, we prioritize hardware concurrency over expanding list sizes. Executing multiple lock-free algorithmic passes simultaneously significantly increases GPU utilization. Although this parallelization naturally multiplies RAM usage, the unified memory architecture comfortably accommodates the overhead. To maximize throughput, we deploy an optimized configuration integrating three key refinements. First, we apply a fixed-S^c pre-diagonalization to amortize matrix operations, which drastically reduces Step 0 execution time from 15–16 seconds to 5–7 seconds. Second, we balance time and space complexity by setting $\texttt{UN_L} = 1\,024$, $\texttt{UN_R} = 0$, and $N_{\text{round}} = 3$; this restricts the list size to approximately $2^{27.7}$ and limits the per-thread memory footprint to roughly 7 GB. Finally, by aggressively scaling to 28 parallel threads, we achieve a sustained throughput of 2.3 to 2.5 attempts per second while consuming a manageable aggregate of about 200 GB of unified memory.

6.4 Performance Evaluation and Benchmarks

Algorithmic Efficiency (The Perfect Oracle) To isolate our algorithmic improvements from measurement noise, we first benchmark against a perfect Plaintext Checking (PC) oracle. This generic model serves as a universal theoretical baseline in HQC side-channel analysis, effectively abstracting a wide

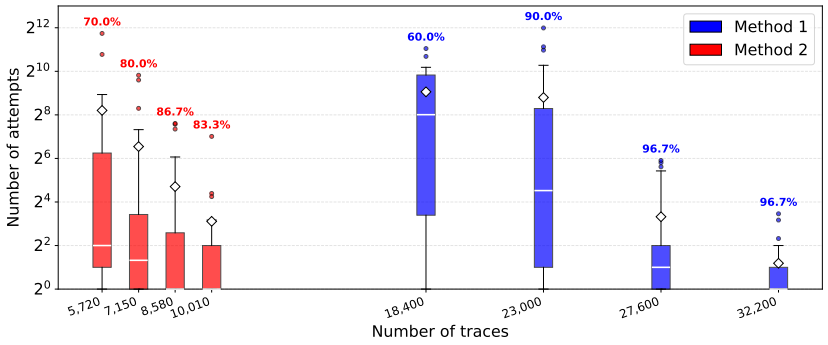


Fig. 7: Success rates and decoding costs (log_2 scale). The visualization displays the distribution of attempts for successful recoveries, capped at a maximum of 2^{12}. Boxplot markers include the median (white line), mean (diamond), and the 10th–90th percentile range (whiskers). The percentages above each bar indicate the success rate derived from 30 keys per configuration.

spectrum of vulnerabilities. These range from timing leakages [19, 22] and power/EM emanations [10,44,46,53], to complex micro-architectural exploits [16,47] and cache-timing channels [26]. While early cryptanalytic efforts required over 800,000 queries [19], recent state-of-the-art techniques have reduced this bound to approximately 460 queries [10].

We demonstrate that integrating Soft-ISD post-processing pushes this lower bound significantly further. By simulating a noise-free PC oracle in [10], we can rigorously evaluate the impact of our multi-pass Soft-ISD decoder—a technique notably absent in the methodology of [10]. We conducted experiments on 20 randomly generated secret keys, capping post-processing at 30 minutes. With a budget of merely 276 oracle calls (3×92), our approach successfully recovers the secret key in 55% of cases (11/20). Expanding the budget to 368 calls (4×92) increases the success rate to 75% (15/20), and 460 calls (5×92) yields a 100% recovery rate (20/20). Consequently, our multi-pass Soft-ISD software reduces the query complexity for a majority of keys to just 60% of the previous best [10], establishing 276 queries as the new baseline for effective key recovery.

Real-world Attack Performance We evaluate the end-to-end attack by feeding the soft reliability information (log-likelihood ratios) derived from the FLUSH+RELOAD cache-timing leakage (specifically the data detailed in Figure 6) into our GPU-accelerated Soft-ISD solver. Our evaluation consists of two phases: a micro-benchmark to validate algorithmic behavior against theoretical models, and a macro-benchmark to assess practicality in a realistic environment.

We first validate the solver using two fixed keys representing distinct difficulty levels, determined by the Hamming weight of the secret error vector within the reliability-ordered information set. Figure G.1 in Appendix G illustrates the cumulative success probability over 100 trials, where our GPU-accelerated

implementation (solid blue) aligns closely with theoretical estimations (dashed red). When benchmarked against the state-of-the-art Soft-ISD simulation by Dong et al. [11], our solver outperforms their $w_0 = 2$ configuration. While their $w_0 = 3$ parameter set theoretically offers higher per-attempt success, it imposes prohibitive overheads. Specifically, $w_0 = 3$ requires a list size of $2^{36.5}$, demanding terabytes of RAM for hash table implementations—far exceeding the 512 GB unified memory capacity of our platform. Furthermore, we estimated the time per attempt for $w_0 = 3$ to be approximately 2 400 times slower than our optimized 28-thread configuration. Consequently, our approach prioritizes high throughput over per-attempt success rate, yielding a significantly superior time-to-solution.

Finally, we assess the attack's practicality across 30 randomly generated keys with a computational timeout of approximately 27 minutes ($\approx$ 4 000 attempts). As summarized in Figure 7, the post-processing overhead is generally negligible; for Method 2, the median recovery required only a single attempt across the best trace configurations (medians: 4, 3, 1, 1). We note that success rates did not reach 100% due to the realistic threat model: we deliberately eschewed laboratory controls—such as disabling Dynamic Voltage and Frequency Scaling (DVFS) or isolating cores—resulting in measurement noise that can render specific keys more difficult to recover. Nevertheless, the attack demonstrates a high overall success rate on commodity hardware without requiring administrative privileges.

7 Discussions

Affected Compiler Versions The exploitability of the reported timing leakage is tied to the compiler's optimization strategies. In our experiments with `clang-14` and `clang-15` (using flags `-O3`, `-Os`, and `-Oz`), the resulting binaries contain secret-dependent branches that expose the four least significant bits (indices 0–3) of the Reed-Muller decoding output. While subsequent versions—specifically `clang-16` through `clang-18`—eliminate this specific leakage path, they nevertheless retain analogous control-flow dependencies regarding bits 4–6. Although extracting an FD-style oracle from this specific leakage requires methodological adjustments, the underlying vulnerability persists. It confirms that using even recent versions of the Clang compiler is highly susceptible to microarchitectural timing attacks.

Enhancing Oracle Resolution Our current implementation demonstrates a fully functional, end-to-end attack utilizing the standard Flush+Reload side-channel on the Last-Level Cache (LLC). It is crucial to emphasize that these experimental results were derived under realistic, noisy system conditions where the adversary possesses no administrative control over the target machine. Operating alongside standard background user activity, we achieved a reliable exploration rate of approximately one out of every 13 decoding blocks using Method 2. This resolution should be viewed as a lower bound on the attack's potential rather than an intrinsic limit. The primary constraint on higher resolution is

the temporal granularity mismatch between the victim's operation and the side-channel measurement; in our setup, a single Reed-Muller decoding completes in roughly 483 clock cycles, whereas we set an interval of 2 000 cycles for a stable Flush+Reload measurement.

This resolution limit is further compounded by unpredictable CPU behavior, such as speculative execution and jitter, which varies significantly across different hardware platforms. However, the current extraction rate is not a theoretical ceiling but a consequence of our conservative measurement intervals. The recovery of secret key information could be significantly improved without altering the attack vector by employing sophisticated signal processing methodologies. In particular, the application of advanced trace alignment algorithms and spectral filtering techniques would allow for better isolation of the leakage signal from environmental noise, thereby increasing the density of usable blocks.

Beyond post-processing improvements, an attacker could manipulate the system environment to overcome the bandwidth limitations of the Flush+Reload channel. Stricter laboratory controls, such as disabling hardware prefetchers or DVFS, would immediately reduce measurement jitter. Notably, such configurations could be adopted in server environments to mitigate other microarchitectural threats, such as Data Memory-Dependent Prefetcher (DMP) attacks [8] or HertzBleed [55], inadvertently rendering the system more susceptible to our high-precision timing analysis. More aggressively, the effective resolution can be artificially amplified by degrading the victim's execution speed [2,3]. By inducing resource contention, an attacker can dilate the execution time of the Reed-Muller decoding by orders of magnitude. Such techniques would stretch the target operation sufficiently to accommodate multiple Flush+Reload measurements within a single block, effectively widening the side-channel window and enabling the recovery of a significantly larger portion of the internal state.

Finally, the granularity of the oracle could be maximized within Trusted Execution Environments (TEEs) by leveraging controlled channel attacks. Frameworks capable of single-stepping the victim, such as SGX-Step [7] or SEV-Step [56], allow an adversary to pause execution at the instruction boundary, effectively removing all temporal constraints imposed by the decoder's execution speed. This level of precise control would expose the complete sequence of secret-dependent branches, allowing the full exploitation of the FD oracle for every block. In these isolated environments, where the shared library required for Flush+Reload is typically absent, the attack vector would necessarily shift to contention-based techniques such as Prime+Probe or page-fault analysis; nevertheless, the underlying vulnerability would remain fully exploitable.

Countermeasures Mitigating the compiler-induced vulnerabilities identified in this work requires a multi-layered approach. Since the Flush+Reload attack demonstrated relies on shared code pages from a dynamically linked library, static linking is a useful deployment hardening step, as it removes the particular page-sharing condition. However, static linking should not be interpreted as a complete countermeasure. If code pages are shared through other mechanisms, or

if the adversary switches to contention-based channels such as Prime+Probe, the secret-dependent control flow in the emitted binary remains a vulnerability.

At the source level, developers can attempt to rewrite the C logic to make the intended constant-time behavior more robust against optimizer transformations. Through additional testing, we found that source-level hardenings based on compiler barriers, `volatile` casts, and Clang-specific attributes (e.g., `optnone`) removed the specific secret-dependent branching pattern for the tested Clang versions and optimization levels. These results should be viewed as engineering evidence for possible short-term mitigations rather than as a portable guarantee. Such techniques are fragile under toolchain evolution and may introduce non-trivial runtime overhead. Long-term use of this approach therefore requires toolchain pinning, binary-level inspection of the generated machine code, and constant-time testing such as `ctgrind`-style validation, among other rigorous verification practices.

A more robust long-term mitigation is to replace the vulnerable selection logic with hand-audited assembly for the target instruction set architecture and toolchain. This gives direct control over the emitted instruction sequence and prevents the compiler from reintroducing the branch pattern studied in this paper. This remedy is still platform-specific: the resulting object code must be audited for the target microarchitecture and build configuration, including variable-latency instructions and secret-dependent memory accesses. Under such an audited target, assembly can preserve the intended constant-time behavior.

8 Conclusion and Future Work

In this work, we have presented the first cache-timing FD-style oracle attack against the NIST-selected HQC KEM. Specifically, we target the recent high-performance `AVX2-optimized` implementation. While the source code is designed to be constant-time, we demonstrate that aggressive compiler optimizations (such as Clang `-O3`) can introduce secret-dependent control flow. This silent transformation creates a critical discrepancy between the programmer's intent and the resulting executable binary. By exploiting this vulnerability via a Flush+Reload side-channel on shared libraries, we show that an unprivileged attacker can reliably recover the full secret key. Our attack is highly practical. Using our novel, GPU-accelerated Soft-ISD decoder software, we achieve a high success rate requiring less than 10 seconds of online trace collection and under 30 minutes of offline post-processing.

These findings have immediate implications for the standardization and deployment of HQC. As a selected standard destined for widespread use, ensuring the robustness of its reference and optimized implementations is paramount. Our results serve as a stark reminder that algorithmic hardness and source-level hygiene are insufficient without rigorous binary-level verification. Consequently, we recommend that future HQC specifications and software libraries enforce constant-time execution through inline assembly or hardened compiler direc-

tives. Relying solely on C-level logic is risky, as it can be easily undermined by modern optimization heuristics.

Looking ahead, there are several promising avenues for extending this research. It remains an open challenge to further optimize our open-source Soft-ISD artifact and apply it to a broader range of code-based cryptographic applications. Furthermore, as post-quantum algorithms migrate to diverse hardware platforms, it is crucial to investigate how similar compiler-induced leakages might manifest on non-`x86` architectures. Developing automated tools to detect these compiler-induced vulnerabilities will be essential for securing our future public-key infrastructure.

Acknowledgment

We used OpenAI Codex as a debugging aid during the implementation of Soft-ISD software on Metal GPU with Unified Memory. This work was supported by the Swedish Research Council (grant numbers 2021-04602 and 2023-03654), the ELLIIT program, and the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation. The computations were partly enabled by resources provided by LUNARC.

References

1. Albrecht, M., Bard, G., Hart, W.: Algorithm 898: Efficient multiplication of dense matrices over gf(2). ACM Transactions on Mathematical Software (TOMS) **37**(1), 1–14 (2010)
2. Aldaya, A.C., Brumley, B.B.: HyperDegrade: From GHz to MHz effective CPU frequencies. In: Butler, K.R.B., Thomas, K. (eds.) USENIX Security 2022. pp. 2801–2818. USENIX Association (Aug 2022), `https://www.usenix.org/conference/usenixsecurity22/presentation/aldaya`
3. Allan, T., Brumley, B.B., Falkner, K., Van de Pol, J., Yarom, Y.: Amplifying side channels through performance degradation. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. pp. 422–435 (2016)
4. Băetu, C., Durak, F.B., Huguenin-Dumittan, L., Talayhan, A., Vaudenay, S.: Misuse attacks on post-quantum cryptosystems. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 747–776. Springer, Cham (May 2019). https://doi.org/10.1007/978-3-030-17656-3_26
5. Becker, A., Joux, A., May, A., Meurer, A.: Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 520–536. Springer, Berlin, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4_31
6. Bernstein, D.J., Bhargavan, K., Bhasin, S., Chattopadhyay, A., Chia, T.K., Kannwischer, M.J., Kiefer, F., Paiva, T.B., Ravi, P., Tamvada, G.: KyberSlash: Exploiting secret-dependent division timings in kyber implementations. IACR TCHES **2025**(2), 209–234 (2025). https://doi.org/10.46586/tches.v2025.i2.209-234
7. Bulck, J.V., Piessens, F., Strackx, R.: Sgx-step: A practical attack framework for precise enclave execution control. In: Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX '17). pp. 4:1–4:6. ACM, Shanghai, China

(Oct 2017). https://doi.org/10.1145/3152701.3152706, https://vanbulck.net/files/systex17-sgxstep.pdf, open-access author copy
8. Chen, B., Wang, Y., Shome, P., Fletcher, C.W., Kohlbrenner, D., Paccagnella, R., Genkin, D.: GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru
9. Dong, H., Guo, Q.: Multi-value plaintext-checking and full-decryption oracle-based attacks on HQC from offline templates. IACR TCHES **2025**(4), 254–289 (2025). https://doi.org/10.46586/tches.v2025.i4.254-289
10. Dong, H., Guo, Q.: OT-PCA: New key-recovery plaintext-checking oracle based side-channel attacks on HQC with offline templates. IACR TCHES **2025**(1), 251–274 (2025). https://doi.org/10.46586/tches.v2025.i1.251-274
11. Dong, H., Guo, Q., Nabokov, D.: Single-trace key recovery attacks on HQC using valid and invalid ciphertexts. In: Advances in Cryptology – EUROCRYPT 2026 (2026), https://eprint.iacr.org/2025/1987, available as IACR Cryptology ePrint Archive, Report 2025/1987
12. Ducas, L., Esser, A., Etinski, S., Kirshanova, E.: Asymptotics and improvements of sieving for codes. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part VII. LNCS, vol. 14657, pp. 151–180. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58754-2_6
13. Finiasz, M., Sendrier, N.: Security bounds for the design of code-based cryptosystems. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 88–105. Springer, Berlin, Heidelberg (Dec 2009). https://doi.org/10.1007/978-3-642-10366-7_6
14. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 537–554. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_34
15. Gaborit, P., Aguilar-Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Persichetti, E., Zémor, G., Bos, J., Dion, A., Lacan, J., Robert, J.M., Véron, P., Barreto, P.L., Ghosh, S., Gueron, S., Güneysu, T., Misoczki, R., Richter-Brokmann, J., Sendrier, N., Tillich, J.P., Vasseur, V.: Hamming quasi-cyclic (hqc). https://pqc-hqc.org (aug 2025), https://pqc-hqc.org, specification document, version 2025/08/22
16. Gast, S., Weissteiner, H., Schröder, R.L., Gruss, D.: CounterSEVeillance: Performance-counter attacks on AMD SEV-SNP. In: NDSS 2025. The Internet Society (Feb 2025)
17. Goy, G., Loiseau, A., Gaborit, P.: A new key recovery side-channel attack on HQC with chosen ciphertext. In: Cheon, J.H., Johansson, T. (eds.) PQCrypto 2022. pp. 353–371. Springer, Cham (Sep 2022). https://doi.org/10.1007/978-3-031-17234-2_17
18. Goy, G., Maillard, J., Gaborit, P., Loiseau, A.: Single trace HQC shared key recovery with SASCA. IACR TCHES **2024**(2), 64–87 (2024). https://doi.org/10.46586/tches.v2024.i2.64-87
19. Guo, Q., Hlauschek, C., Johansson, T., Lahr, N., Nilsson, A., Schröder, R.L.: Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. IACR TCHES **2022**(3), 223–263 (2022). https://doi.org/10.46586/tches.v2022.i3.223-263

20. Guo, Q., Johansson, T., Mårtensson, E., Stankovski Wagner, P.: Some cryptanalytic and coding-theoretic applications of a soft stern algorithm. Advances in Mathematics of Communications **13**(4) (2019)
21. Guo, Q., Johansson, T., Nguyen, V.: A new sieving-style information-set decoding algorithm. IEEE Transactions on Information Theory **70**(11), 8303–8319 (2024)
22. Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 359–386. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56880-1_13
23. Guo, Q., Mårtensson, E., Åström, A.: The perils of limited key reuse: Adaptive and parallel mismatch attacks with post-processing against Kyber. CiC **1**(3), 21 (2024). https://doi.org/10.62056/a3n5qj888
24. Guo, Q., Nabokov, D., Nilsson, A., Johansson, T.: SCA-LDPC: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes. In: Guo, J., Steinfeld, R. (eds.) ASIACRYPT 2023, Part IV. LNCS, vol. 14441, pp. 203–236. Springer, Singapore (Dec 2023). https://doi.org/10.1007/978-981-99-8730-6_7
25. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017, Part I. LNCS, vol. 10677, pp. 341–371. Springer, Cham (Nov 2017). https://doi.org/10.1007/978-3-319-70500-2_12
26. Huang, S., Sim, R.Q., Chuengsatiansup, C., Guo, Q., Johansson, T.: Cache-timing attack against HQC. IACR TCHES **2023**(3), 136–163 (2023). https://doi.org/10.46586/tches.v2023.i3.136-163
27. Huang, Z., Wang, W., Zhou, X., Yu, Y.: Dpa-style attacks on hqc. IACR Cryptology ePrint Archive, Report 2025/1966 (2025)
28. Lai, Z., Zhang, R., Zhang, Z., Hermelink, J., Schwarz, M., Pham, V.T., Parampalli, U.: You only decapsulate once: Ciphertext-independent single-trace passive side-channel attacks on hqc. IACR Cryptology ePrint Archive, Report 2025/2162 (2025)
29. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy. pp. 605–622. IEEE Computer Society Press (May 2015). https://doi.org/10.1109/SP.2015.43
30. Maillet, N., Nugier, C., Migliore, V., Deneuville, J.C.: Key recovery from side-channel power analysis attacks on non-SIMD HQC decryption. In: Kalai, Y.T., Kamara, S.F. (eds.) CRYPTO 2025, Part V. LNCS, vol. 16004, pp. 70–102. Springer, Cham (Aug 2025). https://doi.org/10.1007/978-3-032-01901-1_3
31. May, A., Meurer, A., Thomae, E.: Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 107–124. Springer, Berlin, Heidelberg (Dec 2011). https://doi.org/10.1007/978-3-642-25385-0_6
32. May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 203–228. Springer, Berlin, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_9
33. Mondal, P., Kundu, S., Bhattacharya, S., Karmakar, A., Verbauwhede, I.: A practical key-recovery attack on LWE-based key-encapsulation mechanism schemes using rowhammer. In: Pöpper, C., Batina, L. (eds.) ACNS 2024, Part III. LNCS, vol. 14585, pp. 271–300. Springer, Cham (Mar 2024). https://doi.org/10.1007/978-3-031-54776-8_11

34. Ngo, K., Dubrova, E., Guo, Q., Johansson, T.: A side-channel attack on a masked IND-CCA secure Saber KEM implementation. IACR TCHES **2021**(4), 676–707 (2021). https://doi.org/10.46586/tches.v2021.i4.676-707, https://tches.iacr.org/index.php/TCHES/article/view/9079
35. Ngo, K., Dubrova, E., Johansson, T.: A side-channel attack on a masked and shuffled software implementation of Saber. Journal of Cryptographic Engineering **13**(4), 443–460 (Nov 2023). https://doi.org/10.1007/s13389-023-00315-3
36. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Berlin, Heidelberg (Feb 2006). https://doi.org/10.1007/11605805_1
37. Paiva, T.B., Ravi, P., Jap, D., Bhasin, S., Das, S., Chattopadhyay, A.: Et tu, brute? Side-channel assisted chosen ciphertext attacks using valid ciphertexts on HQC KEM. In: Niederhagen, R., Saarinen, M.J.O. (eds.) Post-Quantum Cryptography - 16th International Workshop, PQCrypto 2025, Part II. pp. 294–321. Springer, Cham (Apr 2025). https://doi.org/10.1007/978-3-031-86602-9_11
38. Paiva, T.B., Terada, R.: A timing attack on the HQC encryption scheme. In: Paterson, K.G., Stebila, D. (eds.) SAC 2019. LNCS, vol. 11959, pp. 551–573. Springer, Cham (Aug 2019). https://doi.org/10.1007/978-3-030-38471-5_22
39. Pessl, P., Mangard, S.: Enhancing side-channel analysis of binary-field multiplication with bit reliability. In: Sako, K. (ed.) CT-RSA 2016. LNCS, vol. 9610, pp. 255–270. Springer, Cham (Feb / Mar 2016). https://doi.org/10.1007/978-3-319-29485-8_15
40. PQShield: PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity. https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber-ml-kem-to-improve-pqc-implementation-maturity/ (Jun 2024), accessed: 2026-02-09. Vulnerability tracked as CVE-2024-37880
41. Prange, E.: The use of information sets in decoding cyclic codes. IRE Transactions on Information Theory **IT-8**, 5–9 (1962)
42. Rajendran, G., Ravi, P., D'Anvers, J.P., Bhasin, S., Chattopadhyay, A.: Pushing the limits of generic side-channel attacks on LWE-based KEMs - parallel PC oracle attacks on Kyber KEM and beyond. IACR TCHES **2023**(2), 418–446 (2023). https://doi.org/10.46586/tches.v2023.i2.418-446
43. Ravi, P., Ezerman, M.F., Bhasin, S., Chattopadhyay, A., Roy, S.S.: Will you cross the threshold for me? Generic side-channel assisted chosen-ciphertext attacks on NTRU-based KEMs. IACR TCHES **2022**(1), 722–761 (2022). https://doi.org/10.46586/tches.v2022.i1.722-761
44. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR TCHES **2020**(3), 307–335 (2020). https://doi.org/10.13154/tches.v2020.i3.307-335, https://tches.iacr.org/index.php/TCHES/article/view/8592
45. Rizin: Rizin: Unix-like reverse engineering framework (2025), https://rizin.re
46. Schamberger, T., Holzbaur, L., Renner, J., Wachter-Zeh, A., Sigl, G.: A power side-channel attack on the reed-muller reed-solomon version of the HQC cryptosystem. In: Cheon, J.H., Johansson, T. (eds.) PQCrypto 2022. pp. 327–352. Springer, Cham (Sep 2022). https://doi.org/10.1007/978-3-031-17234-2_16
47. Schröder, R.L., Gast, S., Guo, Q.: Divide and surrender: Exploiting variable division instruction timing in HQC key recovery attacks. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/schr%C3%B6der

48. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D., Ding, J.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
49. Shao, M., Liu, Y., Zhou, Y.: Pairwise and parallel: Enhancing the key mismatch attacks on Kyber and beyond. In: Zhou, J., Quek, T.Q.S., Gao, D., Cárdenas, A.A. (eds.) ASIACCS 24. ACM Press (Jul 2024). https://doi.org/10.1145/3634737.3637661
50. Stern, J.: A method for finding codewords of small weight. In: International colloquium on coding theory and applications. pp. 106–113. Springer (1988)
51. Tanaka, Y., Ueno, R., Xagawa, K., Ito, A., Takahashi, J., Homma, N.: Multiple-valued plaintext-checking side-channel attacks on post-quantum KEMs. IACR TCHES **2023**(3), 473–503 (2023). https://doi.org/10.46586/tches.v2023.i3.473-503
52. Torres, R.C., Sendrier, N.: Analysis of information set decoding for a sub-linear error weight. In: Takagi, T. (ed.) PQCrypto 2016. pp. 144–161. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29360-8_10
53. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. IACR TCHES **2022**(1), 296–322 (2022). https://doi.org/10.46586/tches.v2022.i1.296-322
54. Wafo-Tapa, G., Bettaieb, S., Bidoux, L., Gaborit, P., Marcatel, E.: A practicable timing attack against HQC and its countermeasure. Cryptology ePrint Archive, Report 2019/909 (2019), https://eprint.iacr.org/2019/909
55. Wang, Y., Paccagnella, R., He, E.T., Shacham, H., Fletcher, C.W., Kohlbrenner, D.: Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In: Butler, K.R.B., Thomas, K. (eds.) USENIX Security 2022. pp. 679–697. USENIX Association (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen
56. Wilke, L., Wichelmann, J., Rabich, A., Eisenbarth, T.: SEV-step A single-stepping framework for AMD-SEV. IACR TCHES **2024**(1), 180–206 (2024). https://doi.org/10.46586/tches.v2024.i1.180-206
57. Xagawa, K., Ito, A., Ueno, R., Takahashi, J., Homma, N.: Fault-injection attacks against NIST's post-quantum cryptography round 3 KEM candidates. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part II. LNCS, vol. 13091, pp. 33–61. Springer, Cham (Dec 2021). https://doi.org/10.1007/978-3-030-92075-3_2
58. Xu, Z., Pemberton, O., Roy, S.S., Oswald, D.F.: Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. IACR Cryptol. ePrint Arch. p. 912 (2020), https://eprint.iacr.org/2020/912
59. Yarom, Y.: Mastik: A micro-architectural side-channel toolkit (2016), https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf
60. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: Fu, K., Jung, J. (eds.) USENIX Security 2014. pp. 719–732. USENIX Association (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

Auxiliary Supporting Material

A Online Attack in [9] and New Attacks' Modifications

Algorithm 2 Online Attack and Obtaining Reliability Scores in [9]

Input: $pk = (h, s)$, $m = 0$ by default, *salt*, scheme parameters $(n_1, n_2, \mathbf{G})$
$\mathcal{E} = \{\epsilon_1, \ldots, \epsilon_\tau\}$, offline template $\mathcal{T}_{\epsilon,m}$ for each $\epsilon \in \mathcal{E}$
Output: Reliability score matrix $R \in \mathbb{R}^{\{\text{"X"},\text{"Y"}\} \times [0, n_1 n_2 - 1]}$

$R_{i,\ell} \leftarrow 1.0 \quad \forall i \in \{\text{"X"}, \text{"Y"}\},\ \forall \ell \in [0, n_1 n_2 - 1]$
for $i \in \{\text{"X"}, \text{"Y"}\}$ **do**
  for each $\epsilon \in \mathcal{E}$ **do**
    if $i =$ "X" **then**
      $r_1 \leftarrow 0,\ r_2 \leftarrow 1$
    else if $i =$ "Y" **then**
      $r_1 \leftarrow 1,\ r_2 \leftarrow 0$
    end if
    $e_b \leftarrow \epsilon \quad \forall b \in [0, n_1 - 1]$
    $e \leftarrow (e_0 \,\|\, e_1 \,\|\, \cdots \,\|\, e_{n_1 - 1})$
    $c \leftarrow \big(r_1 + h \cdot r_2,\ m\mathbf{G} + s \cdot r_2 + e,\ \mathit{salt}\big)$
    $(o_0, \ldots, o_{n_1 - 1}) \leftarrow \mathcal{O}_{\text{FD}}(c)$
    for $b \in [0, n_1 - 1]$ **do**
      for $j \in [0, n_2 - 1]$ **do**
        $P \leftarrow \Pr(\xi_j = 1 \mid o_b)$ from $\mathcal{T}_{\epsilon,m}$
        $\ell \leftarrow j + b \cdot n_2$
        $R_{i,\ell} \leftarrow R_{i,\ell} \cdot P$
      end for
    end for
  end for
end for
return $R_{i,\ell},\ \forall i \in \{\text{"X"}, \text{"Y"}\},\ \forall \ell \in [0, n_1 n_2 - 1]$

For Method 1:

In Algorithm 2, replace Lines 9–18 by:

for $b \in [0, n_1 - 1]$ **do**
  $e_a \leftarrow 0^{n_2} \quad \forall a \in [0, n_1 - 1]$
  $e_b \leftarrow \epsilon$
  $e \leftarrow (e_0 \| e_1 \| \cdots \| e_{n_1 - 1})$
  $c \leftarrow (r_1 + h \cdot r_2,\ m\mathbf{G} + s \cdot r_2 + e,\ \mathit{salt})$
  $o_b \leftarrow \mathcal{O}_{FD}(c)$
  for $j \in [0, n_2 - 1]$ **do**
    $P \leftarrow \Pr(\xi_j = 1 \mid o_b)$ from $\mathcal{T}_{\epsilon,m}$
    $\ell \leftarrow j + b \cdot n_2$
    $R_{i,\ell} \leftarrow R_{i,\ell} \cdot P$

end for
end for

For Method 2:

In Algorithm 2, replace Lines 9–18 by:

for $\hat{r} \in [0, 12]$ **do**
  $e_0 \leftarrow \epsilon_H$
  $e_a \leftarrow 0^{n_2} \quad \forall a \in [1, 6]$
  for $a \in [7, n_1 - 1]$ **do**
    if $a \equiv \hat{r} \pmod{13}$ **then**
      $e_a \leftarrow \epsilon$
    else
      $e_a \leftarrow 0^{n_2}$
    end if
  end for
  $e \leftarrow (e_0 \| e_1 \| \cdots \| e_{n_1 - 1})$
  $c \leftarrow (r_1 + h \cdot r_2,\ m\mathbf{G} + s \cdot r_2 + e,\ salt)$
  $\mathcal{B}_{\hat{r}} \leftarrow \{b \in [7, n_1 - 1] : b \equiv \hat{r} \pmod{13}\}$
  $(o_b)_{b \in \mathcal{B}_{\hat{r}}} \leftarrow \mathcal{O}_{FD}(c)$
  for $b \in \mathcal{B}_{\hat{r}}$ **do**
    for $j \in [0, n_2 - 1]$ **do**
      $P \leftarrow \Pr(\xi_j = 1 \mid o_b)$ from $\mathcal{T}_{\epsilon,m}$
      $\ell \leftarrow j + b \cdot n_2$
      $R_{i,\ell} \leftarrow R_{i,\ell} \cdot P$
    end for
  end for
end for

After Algorithm 2 Line 19, insert:

$p \leftarrow \omega / n$
for $a \in [0, 6]$ **do**
  for $j \in [0, n_2 - 1]$ **do**
    $\ell \leftarrow j + a \cdot n_2$
    $R_{i,\ell} \leftarrow R_{i,\ell} \cdot p$
  end for
end for

B New Cache Timing Attack Flow

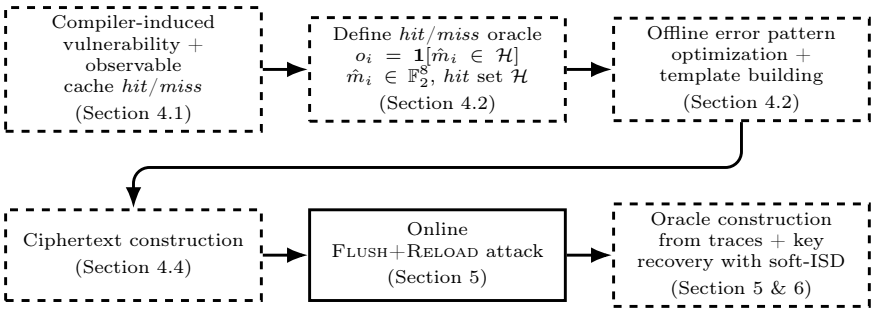


Fig. B.1: Cache-timing attack analysis and flow

C Offline Template for One Optimized Error pattern

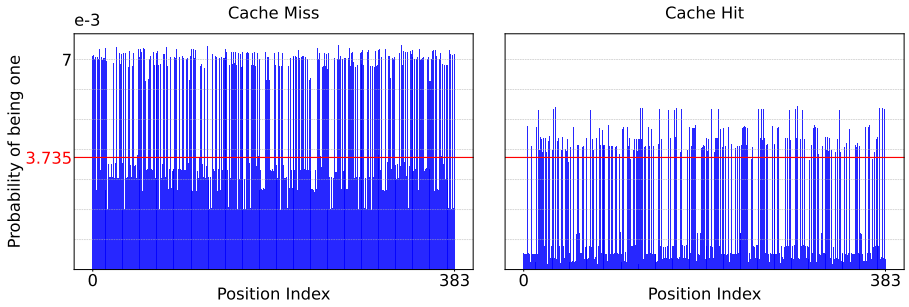


Fig. C.1: Empirical probability of being 1 for each position in the secret key block conditional on cache *miss*/*hit*. The red line indicates the a priori probability of the bit position being 1 for hqc-1, which is $p = \frac{\omega}{n}$.

D Method 1 Result Illustration

Figure D.1 plots the total *hit* count from $R_1 = 50$ runs for each ciphertext targeting each of the n_1 blocks for a given error pattern and key part. Taking the first five blocks as an example, the oracle outputs correspond to a *miss* for blocks 0, 1, 2, and 4, and a *hit* for block 3.

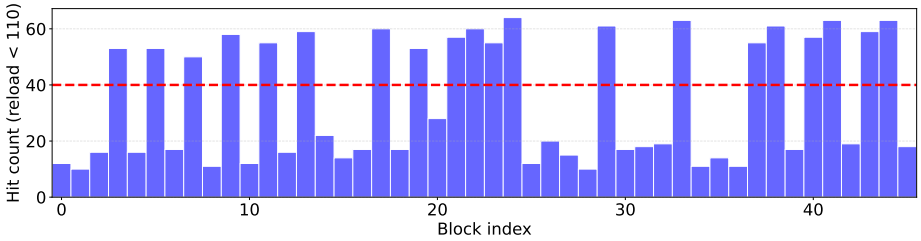


Fig. D.1: Total *hit* count (reload time < 110 cycles) from $R_1 = 50$ runs for each of the n_1 target blocks, given a fixed key part and error pattern. If the count exceeds 40 (80% of R_1), the oracle output for that block is a *hit*; otherwise, it is a *miss*.

E Architecture of our Soft ISD implementation

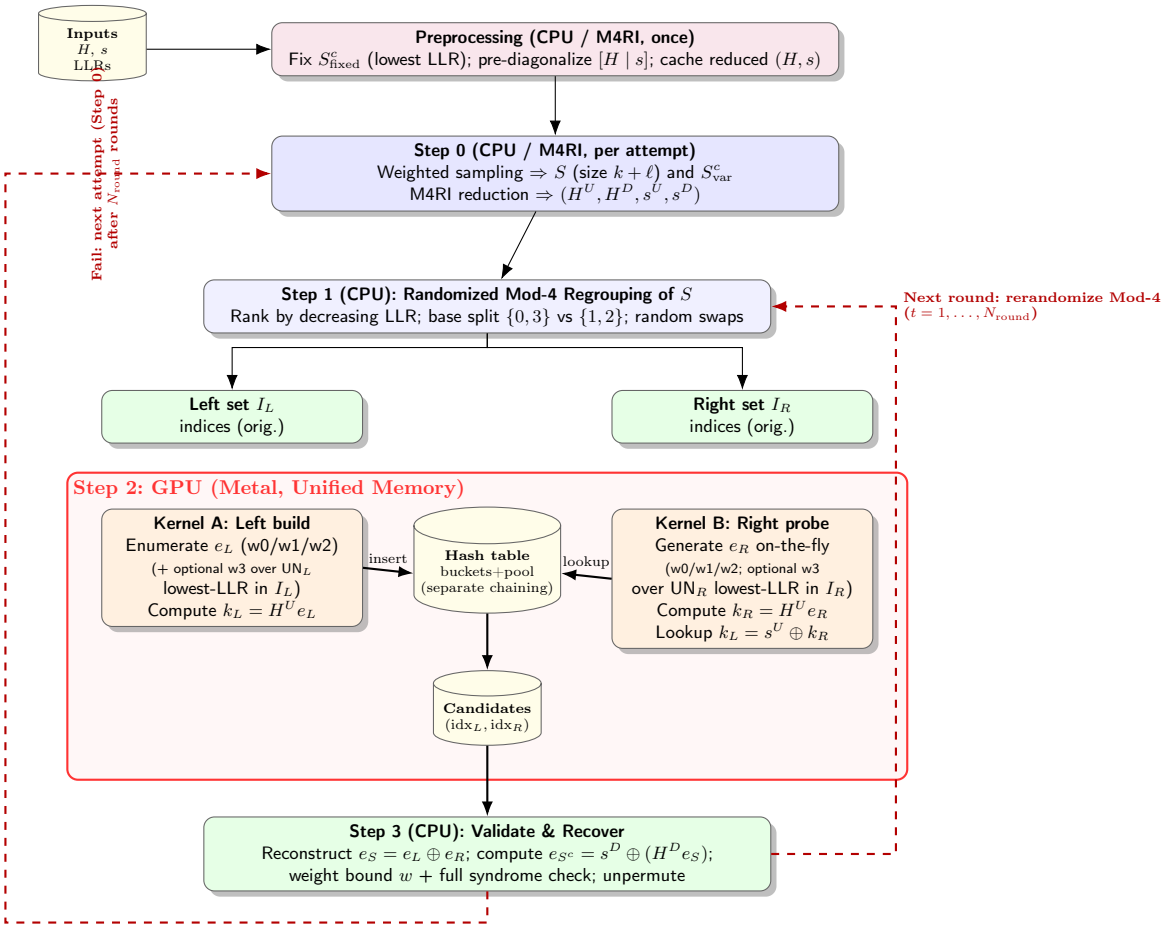


Fig. E.1: High-level implementation architecture of the two-list Soft ISD decoder on unified memory: the CPU performs preprocessing and partitioning, the GPU runs a hash-based join, and the CPU validates candidate collisions.

F Two-List MITM Soft ISD Decoder [11]

This variant, described in [11], introduces a multi-pass ISD framework tailored for soft information. Unlike SoftStern [20], which typically performs a single pass generating lists of the *most probable* error patterns, this approach employs a probabilistic strategy to refine the information set selection while retaining a traditional collision search structure.

The algorithm begins by restricting the search space to a candidate pool of the $\phi \cdot n_c$ most reliable bit positions (those most likely to be zero). In each iterative trial, the information set is constructed via weighted sampling, where positions are drawn from this pool with probabilities proportional to their reliability scores. This ensures that the subsets S_1 and S_2 are statistically biased to contain fewer errors than a random selection.

Once the information set is established, the method utilizes a standard Stern-like approach: it exhaustively enumerates all error patterns within the subsets up to a fixed Hamming weight of w_0. A collision search is then performed on the partial syndromes of these weight-w_0 patterns to identify candidate secret keys.

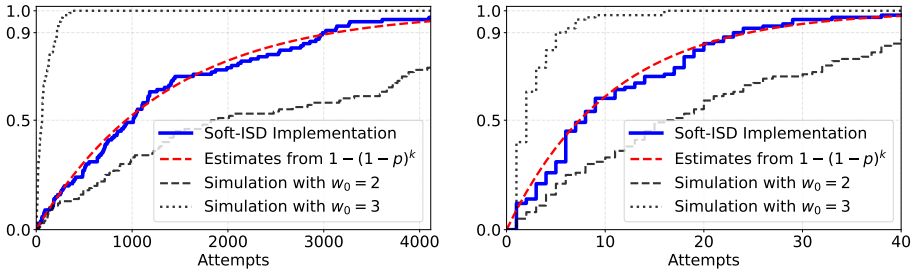
Algorithm 3 Two-List MITM Soft ISD Decoder with Candidate Pool [11]

Input: Parity check matrix $\mathbf{H}$, Syndrome $\mathbf{s}$, Reliability vector L, Parameters $n_c, k\ell, w_0$, and candidate pool factor ϕ

Output: Unique secret key (x, y)

Preprocessing:
Identify the candidate pool C of size $\phi \cdot n_c$ bit positions with the highest reliability in L.
Partition C into two disjoint sets P_1, P_2 by segmenting C into blocks of 4 and randomly assigning $\{0, 3\}$ to one pool and $\{1, 2\}$ to the other.
while failure **do**
    $S_1 \leftarrow \textsc{WeightedSampling}(P_1, \frac{k+\ell}{2}, L)$
    $S_2 \leftarrow \textsc{WeightedSampling}(P_2, \frac{k+\ell}{2}, L)$
    $S \leftarrow S_1 \cup S_2$
    $\tilde{\mathbf{H}}, \tilde{\mathbf{s}} \leftarrow \text{GaussianElimination}(\mathbf{H}, \mathbf{s}, S)$
    $HT \leftarrow \text{InitializeHashTable}()$
    for each $\mathbf{e}_L \in \text{Patterns}(S_1, \{0, \ldots, w_0\})$ **do** ▷ *Left List*
        $\mathbf{v}_L \leftarrow \mathbf{H}^U \cdot \mathbf{e}_L$
        $HT.\text{Insert}(\mathbf{v}_L, \mathbf{e}_L)$
    end for
    for each $\mathbf{e}_R \in \text{Patterns}(S_2, \{0, \ldots, w_0\})$ **do** ▷ *Right List*
        $\mathbf{v}_R \leftarrow \mathbf{H}^U \cdot \mathbf{e}_R$
        $\mathbf{T} \leftarrow \mathbf{v}_R \oplus \mathbf{s}^U$
        if $\mathbf{T} \in HT$ **then**
            $\mathbf{e}_L \leftarrow HT.\text{Get}(\mathbf{T})$
            $\mathbf{x} \leftarrow \text{ConstructCandidate}(\mathbf{e}_L, \mathbf{e}_R)$
            if $\text{Verify}(\mathbf{x}, \mathbf{H}, \mathbf{s})$ **then** ▷ *Full Syndrome Check*
                return $\mathbf{x}$
            end if
        end if
    end for
end while

G Comparison of Our Soft-ISD-Solver with the Soft-ISD Algorithm in [11]



(a) Key 1 (b) Key 2

Fig. G.1: Cumulative success rates for full key recovery over 100 trials. We target two distinct secret keys exhibiting different noise levels: (a) Key 1 (high noise, capped at 4116 attempts) and (b) Key 2 (low noise, capped at 40 attempts). The plots compare the empirical performance of our GPU-accelerated implementation (solid blue) against our theoretical estimation (dashed red) and simulations of the state-of-the-art Soft-ISD algorithm [11] with parameters $w_0 = 2, 3$.

H Results from Desktop with Intel i7-10700K CPU

The attacks were also reproduced on a desktop workstation with an Intel Core i7-10700K processor running Ubuntu 20.04 LTS, accessed and monitored remotely via Chrome Remote Desktop. This processor features 8 physical cores (16 logical threads via SMT), each with private L1 and L2 caches. The LLC is inclusive and shared among all physical cores. The optimized HQC implementation was compiled using `clang-14` v14.0.6. As this build differs from the one in the main text, we monitored the comparable offset `0xbcc4`.

The RM decoding duration per block on this target differs from the main text. Figure H.1 shows the distribution. The average duration is used to align the traces for Method 2. Apart from this specific calibration, the attack methods remain identical to those described in the main text. Figure H.2 illustrates oracle predictions for 46 blocks with Method 1. Figure H.3 illustrates trace alignment for one ciphertext in Method 2. Finally, Figure H.4 plots the Hamming weight distribution within the first n positions of the reordered secret key vector across 30 distinct keys against the number of traces per key.

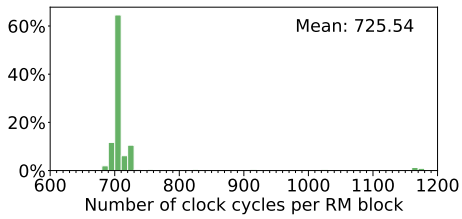


Fig. H.1: Distribution of the estimated number of cycles per RM block decoding from 10 000 runs.

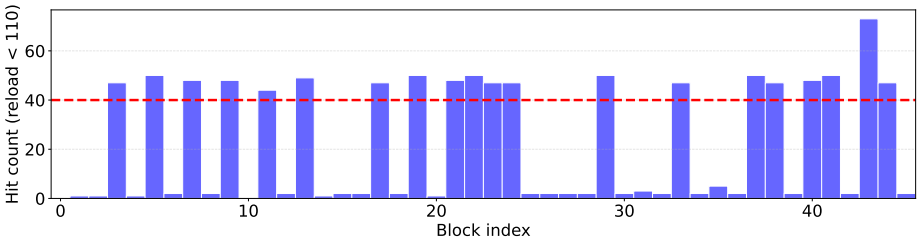


Fig. H.2: Total *hit* count (reload time < 110 cycles) from $R_1 = 50$ runs for each of the n_1 target blocks, given a fixed key part and error pattern. If the count exceeds 40 (80% of R_1), the oracle output for that block is a *hit*; otherwise, it is a *miss*.

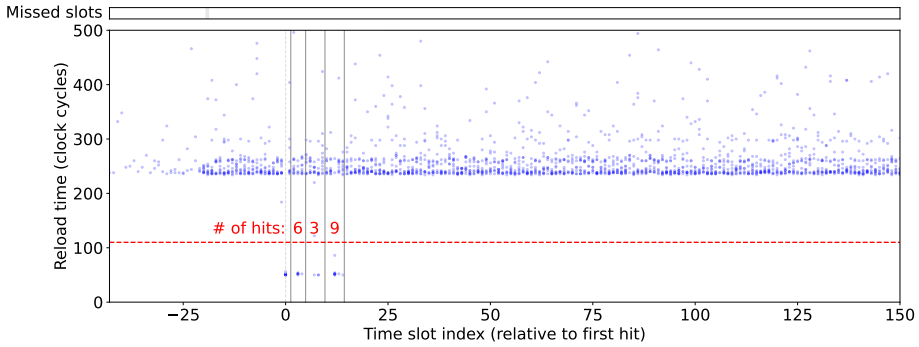


Fig. H.3: Reload times across $R_2 = 11$ runs of one ciphertext, aligned to the first-*hit* time slot. Shaded areas in the missed slots panel indicate at least one run does not have samples in the corresponding time slot. The ranges for the three target blocks are delineated by solid gray vertical lines. With 6, 3, and 9 *hits* (out of $R_2 = 11$ signals each) observed in the three blocks' ranges respectively, the oracle outputs are *hit*, *miss*, and *hit*.

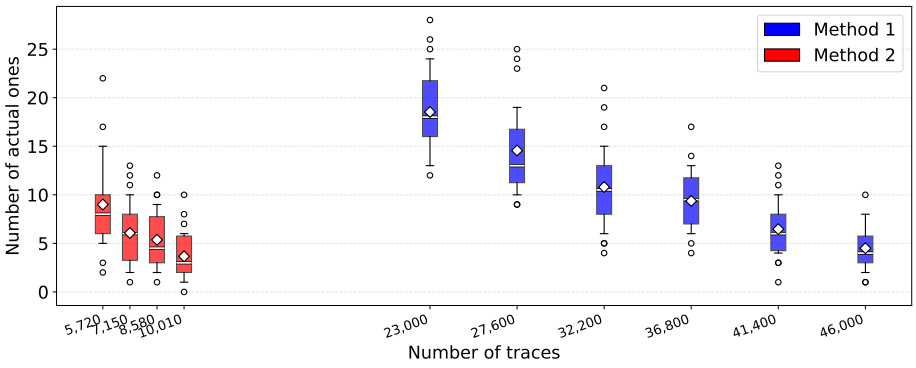


Fig. H.4: Distribution of the Hamming weight within the first n positions of the reordered secret key vector across 30 distinct keys. The data is plotted against the number of traces per key on the x-axis. Whiskers denote the 10th–90th percentiles. The white bar and diamond denote the median and mean, respectively.

I LLC Hit vs. L1 Hit

While not our threat model, we also tried pinning the victim and adversary programs to SMT siblings sharing the same physical core and thus the L1 cache. We ran the attack using Method 1. Figure I.1 illustrates the reload time distribution for *hit* samples (< 110 clock cycles) in this shared-L1-cache experiment versus in the main text's cross-core shared-LLC experiment. The two distributions are clearly distinct.

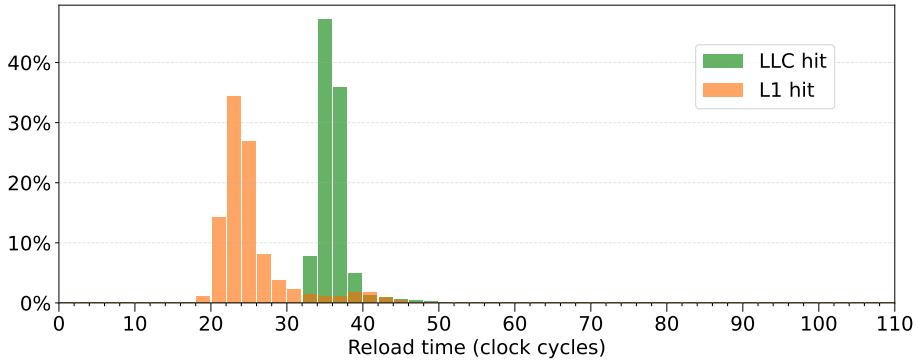


Fig. I.1: Reload time distribution for LLC hit vs. L1 hit.

J Full Conditional Branching Part in Disassembly

Listing 3: Full conditional branching in disassembly regarding Line 232–233 in `reed_muller.c` for `avx256`

```
0xb5b9    mov    edi, ecx                ; reed_muller.c
0xb5bb    and    edi, 0xf
0xb5be    test   rdi, rdi                ; reed_muller.c:232
0xb5c1    je     0xb760                  ; reed_muller.c:233
0xb5c7    mov    dword [var_270h], 0     ; reed_muller.c
0xb5d2    cmp    rdi, 1                  ; reed_muller.c:232
0xb5d6    jne    0xb778                  ; reed_muller.c:233
0xb5dc    vpextrw edx, xmm0, 1           ; reed_muller.c
0xb5e1    mov    dword [var_290h], edx
0xb5e5    cmp    rdi, 2                  ; reed_muller.c:232
0xb5e9    jne    0xb78a                  ; reed_muller.c:233
0xb5ef    vpextrw edx, xmm0, 2           ; reed_muller.c
0xb5f4    cmp    rdi, 3                  ; reed_muller.c:232
0xb5f8    jne    0xb796                  ; reed_muller.c:233
0xb5fe    vpextrw esi, xmm0, 3           ; reed_muller.c
0xb603    mov    dword [var_2b0h], esi
0xb607    cmp    rdi, 4                  ; reed_muller.c:232
0xb60b    jne    0xb7a8                  ; reed_muller.c:233
0xb611    vpextrw esi, xmm0, 4           ; reed_muller.c
0xb616    mov    dword [var_2d0h], esi
0xb61a    cmp    rdi, 5                  ; reed_muller.c:232
0xb61e    mov    qword [var_2d8h], rbx   ; reed_muller.c
0xb623    jne    0xb7bf                  ; reed_muller.c:233
0xb629    vpextrw ebx, xmm0, 5           ; reed_muller.c
0xb62e    cmp    rdi, 6                  ; reed_muller.c:232
0xb632    jne    0xb7cb                  ; reed_muller.c:233
0xb638    vpextrw r15d, xmm0, 6          ; reed_muller.c
0xb63d    cmp    rdi, 7                  ; reed_muller.c:232
0xb641    jne    0xb7d8                  ; reed_muller.c:233
0xb647    vpextrw esi, xmm0, 7           ; reed_muller.c
0xb64c    vextracti128 xmm0, ymm0, 1     ; reed_muller.c:233
0xb652    cmp    rdi, 8                  ; reed_muller.c:232
0xb656    jne    0xb7ea                  ; reed_muller.c:233
0xb65c    vpextrw r14d, xmm0, 0          ; reed_muller.c
0xb661    cmp    rdi, 9                  ; reed_muller.c:232
0xb665    jne    0xb7f7                  ; reed_muller.c:233
0xb66b    vpextrw r12d, xmm0, 1          ; reed_muller.c
0xb670    cmp    rdi, 0xa                ; reed_muller.c:232
0xb674    jne    0xb804                  ; reed_muller.c:233
0xb67a    vpextrw r13d, xmm0, 2          ; reed_muller.c
0xb67f    cmp    rdi, 0xb                ; reed_muller.c:232
0xb683    jne    0xb811                  ; reed_muller.c:233
0xb689    vpextrw r8d, xmm0, 3           ; reed_muller.c
0xb68e    cmp    rdi, 0xc                ; reed_muller.c:232
0xb692    jne    0xb81e                  ; reed_muller.c:233
0xb698    vpextrw r10d, xmm0, 4          ; reed_muller.c
0xb69d    cmp    rdi, 0xd                ; reed_muller.c:232
0xb6a1    jne    0xb82b                  ; reed_muller.c:233
0xb6a7    vpextrw r9d, xmm0, 5           ; reed_muller.c
0xb6ac    cmp    rdi, 0xe                ; reed_muller.c:232
0xb6b0    jne    0xb838                  ; reed_muller.c:233
0xb6b6    vpextrw r11d, xmm0, 6          ; reed_muller.c
0xb6bb    cmp    rdi, 0xf                ; reed_muller.c:232
0xb6bf    jne    0xadc0                  ; reed_muller.c:233
0xb6c5    jmp    0xb845
..
0xb6d0    mov    al, dl                  ; reed_muller.c
0xb6d2    shl    rax, 4
0xb6d6    or     rax, 0x60
0xb6da    mov    rcx, rax
0xb6dd    vpcmpgtw ymm7, ymm15, ymm10    ; reed_muller.c:187
```

```
0xb6e2   vptest ymm7, ymm7            ; reed_muller.c:188
0xb6e7   mov    eax, 0x40             ; '@'
0xb6ec   jne    0xb2c0                ; reed_muller.c:189
0xb6f2   mov    rax, rcx              ; reed_muller.c
0xb6f5   vpcmpgtw ymm7, ymm1, ymm10   ; reed_muller.c:187
0xb6fa   vptest ymm7, ymm7            ; reed_muller.c:188
0xb6ff   mov    ecx, 0x30             ; '0'
0xb704   jne    0xb2d5                ; reed_muller.c:189
0xb70a   mov    rcx, rax              ; reed_muller.c
0xb70d   vpcmpgtw ymm7, ymm5, ymm10   ; reed_muller.c:187
0xb712   vptest ymm7, ymm7            ; reed_muller.c:188
0xb717   mov    eax, 0x20             ; "@"
0xb71c   jne    0xb2ea                ; reed_muller.c:189
0xb722   mov    rax, rcx              ; reed_muller.c
0xb725   vpcmpgtw ymm7, ymm6, ymm10   ; reed_muller.c:187
0xb72a   vptest ymm7, ymm7            ; reed_muller.c:188
0xb72f   mov    edx, 0x10
0xb734   jne    0xb2ff                ; reed_muller.c:189
0xb73a   mov    rdx, rax              ; reed_muller.c
0xb73d   vpcmpgtw ymm7, ymm9, ymm10   ; reed_muller.c:187
0xb742   xor    eax, eax
0xb744   vptest ymm7, ymm7            ; reed_muller.c:188
0xb749   jne    0xb30d                ; reed_muller.c:189
0xb74f   jmp    0xb30f
..
0xb760   vmovd edx, xmm0              ; reed_muller.c
0xb764   movzx edx, dx
0xb767   mov    dword [var_270h], edx
0xb76e   cmp    rdi, 1                ; reed_muller.c:232
0xb772   je     0xb5dc                ; reed_muller.c:233
0xb778   mov    dword [var_290h], 0   ; reed_muller.c
0xb780   cmp    rdi, 2                ; reed_muller.c:232
0xb784   je     0xb5ef                ; reed_muller.c:233
0xb78a   xor    edx, edx              ; reed_muller.c
0xb78c   cmp    rdi, 3                ; reed_muller.c:232
0xb790   je     0xb5fe                ; reed_muller.c:233
0xb796   mov    dword [var_2b0h], 0   ; reed_muller.c
0xb79e   cmp    rdi, 4                ; reed_muller.c:232
0xb7a2   je     0xb611                ; reed_muller.c:233
0xb7a8   mov    dword [var_2d0h], 0   ; reed_muller.c
0xb7b0   cmp    rdi, 5                ; reed_muller.c:232
0xb7b4   mov    qword [var_2d8h], rbx ; reed_muller.c
0xb7b9   je     0xb629                ; reed_muller.c:233
0xb7bf   xor    ebx, ebx              ; reed_muller.c
0xb7c1   cmp    rdi, 6                ; reed_muller.c:232
0xb7c5   je     0xb638                ; reed_muller.c:233
0xb7cb   xor    r15d, r15d            ; reed_muller.c
0xb7ce   cmp    rdi, 7                ; reed_muller.c:232
0xb7d2   je     0xb647                ; reed_muller.c:233
0xb7d8   xor    esi, esi              ; reed_muller.c
0xb7da   vextracti128 xmm0, ymm0, 1   ; reed_muller.c:233
0xb7e0   cmp    rdi, 8                ; reed_muller.c:232
0xb7e4   je     0xb65c                ; reed_muller.c:233
0xb7ea   xor    r14d, r14d            ; reed_muller.c
0xb7ed   cmp    rdi, 9                ; reed_muller.c:232
0xb7f1   je     0xb66b                ; reed_muller.c:233
0xb7f7   xor    r12d, r12d            ; reed_muller.c
0xb7fa   cmp    rdi, 0xa              ; reed_muller.c:232
0xb7fe   je     0xb67a                ; reed_muller.c:233
0xb804   xor    r13d, r13d            ; reed_muller.c
0xb807   cmp    rdi, 0xb              ; reed_muller.c:232
0xb80b   je     0xb689                ; reed_muller.c:233
0xb811   xor    r8d, r8d              ; reed_muller.c
0xb814   cmp    rdi, 0xc              ; reed_muller.c:232
0xb818   je     0xb698                ; reed_muller.c:233
0xb81e   xor    r10d, r10d            ; reed_muller.c
0xb821   cmp    rdi, 0xd              ; reed_muller.c:232
0xb825   je     0xb6a7                ; reed_muller.c:233
```

```
0xb82b    xor    r9d, r9d              ; reed_muller.c
0xb82e    cmp    rdi, 0xe              ; reed_muller.c:232
0xb832    je     0xb6b6                ; reed_muller.c:233
0xb838    xor    r11d, r11d            ; reed_muller.c
0xb83b    cmp    rdi, 0xf              ; reed_muller.c:232
0xb83f    jne    0xadc0                ; reed_muller.c:233
```