Et tu, Brute? Side-Channel Assisted Chosen Ciphertext Attacks using Valid Ciphertexts on HQC KEM

Thales Paiva*[1], Prasanna Ravi[2], Dirmanto Jap[2] and Shivam Bhasin[2]

[1] Fundep and CASNAV, Brazil
[2] Temasek Laboratories, Nanyang Technological University, Singapore
thalespaiva@gmail.com prasanna.ravi@ntu.edu.sg djap@ntu.edu.sg
sbhasin@ntu.edu.sg

Abstract. HQC is a code-based key encapsulation mechanism (KEM) that was selected to move to the fourth round of the NIST post-quantum standardization process. While this scheme was previously targeted by side-channel assisted chosen-ciphertext attacks for key recovery, all these attacks have relied on malformed ciphertexts for key recovery. Thus, all these attacks can be easily prevented by deploying a detection based countermeasures for invalid ciphertexts, and refreshing the secret key upon detection of an invalid ciphertext. This prevents further exposure of the secret key to the attacker and thus serves as an attractive option for protection against prior attacks. Thus, in this work, we present a critical analysis of the detection based countermeasure, and present the first side-channel based chosen-ciphertext attack that attempts to utilize only valid ciphertexts for key recovery, thereby defeating the detection based countermeasure. We propose novel attacks exploiting leakage from the ExpandAndSum and FindPeaks operations within the Reed-Muller decoder for full key recovery with 100% success rate. We show that our attacks are quite robust to noise in the side-channel measurements, and we also present novel extensions of our attack to the shuffling countermeasure on both the ExpandAndSum and FindPeaks operation, which renders the shuffling countermeasure ineffective. Our work therefore shows that low-cost detection based countermeasures can be rendered ineffective, and cannot offer standalone protection against CC-based side-channel attacks. Thus, our work encourages more study towards development of new low-cost countermeasures against CC-based side-channel attacks.

Keywords: Code-based cryptography · Electromagnetic Side-Channel Attack · HQC · Key Encpasulation Mechanism · Chosen Ciphertext Attack

1 Introduction

The NIST standardization process for *post-quantum cryptography* concluded the third round in 2022 with the announcement of 4 winners (1 KEM and 3 digital signature schemes) for standardization. Three code based KEMs (HQC, BIKE and Classic McEliece) advanced to the fourth round of the standardization process, and at least one among these KEMs will be standardized at the end of the fourth round. While implementation performance and theoretical security served as the main criteria in the initial rounds, resistance against *side-channel attacks* (SCA) and *fault injection attacks* (FIA) emerged as an important criterion in the final round, as clearly stated by NIST at several instances [AH21, RR21].

*Part of this work was done while the author was a PhD student at the University of São Paulo.

This work focuses on HQC [MAB+21], a code-based KEM whose security is based on the syndrome decoding problem for quasi-cyclic codes. HQC can be seen as an intermediate scheme between the other two code-based KEMs currently being considered by NIST, namely, Classic McEliece [BCL+19] and BIKE [ABB+21]. Its quasi-cyclic structure improves efficiency compared to Classic McEliece, while avoiding the reliance on a secret sparse structure found in BIKE, making HQC a more conservative choice. Similar to LWE-based schemes, HQC employs error-correction mechanisms to recover the message after errors accumulate during decryption. To achieve this, HQC leverages a combination of well-known Reed-Muller and Reed-Solomon error-correction codes.

There have been several side-channel attacks, particularly attacks exploiting the power/EM side-channel reported on HQC KEM [SRSWZ21, SHR+22, GLG22]. Most of these attacks target the decapsulation procedure with chosen-ciphertexts for key recovery. All these attacks fall under the category of side-channel assisted chosen-ciphertext attacks. These attacks use malformed ciphertexts to trigger decapsulation failure and exploit corresponding side-channel leakage for key recovery. Moreover, there have been several timing and cache-timing side-channel attacks [HSC+23, GHJ+22], all of which also only utilize invalid/malicious ciphertexts for key recovery.

One of the standard ways to protect against these attacks is to employ masking countermeasures for the decapsulation procedure, but we expect them to come at a significant performance penalty, similar to other PQC schemes such as Kyber. For schemes such as Kyber, there have been alternative approaches that propose low-cost countermeasures against CC based attacks [XPRO20, RCDB22]. These countermeasures rely on detecting malicious ciphertexts and refreshing the secret key upon detection of an invalid ciphertext to prevent further exposure. Such countermeasures can also be effective for HQC KEM, as all existing side-channel attacks on key recovery only utilize malformed ciphertexts. Especially since these countermeasures come at a low-cost, they serve as an attractive option for designers for protecting against CC-based side-channel attacks on HQC KEM.

In this work, we perform a critical analysis of the decapsulation failure check countermeasure, and present novel CC-based side-channel attacks that only work with valid ciphertexts, thereby bypassing the decapsulation failure check countermeasure for HQC KEM. Moreover, we also identify new types of side-channel vulnerabilities within the RM decoding operation of HQC KEM within the decapsulation procedure, leading to novel key recovery attacks using valid ciphertexts.

Our Contribution

In this work, we present the following new contributions:

1. We demonstrate novel CC-based side-channel attacks on HQC KEM, that only rely on valid ciphertexts to carry out the attack for key recovery. In order to detect leakage from valid ciphertexts, we exploit two operations within the Reed-Muller decoder: ExpandAndSum and FindPeaks, and these operations have not been targeted by previous side-channel attacks on HQC KEM.

2. We propose novel key recovery algorithms exploiting leakage from the ExpandAndSum and FindPeaks operations for full key recovery with 100% success rate. While our attack on the ExpandAndSum operation leads to key recovery in only 2 traces, our attack on the FindPeaks operation requires about 200 thousand traces for full key recovery.

3. We also show that our key recovery attacks are quite robust to noise in the side-channel measurements, and full key recovery is possible even in the presence of significant errors of up to 40% bit error rate in the recovered codeword.

4. We also demonstrate novel key recovery attacks on the shuffled implementations of both the ExpandAndSum and FindPeaks, and demonstrate impact of the shuffling countermeasure on key recovery. While shuffling increases the attacker's complexity for key recovery, it does not concretely prevent the attack. Our novel key recovery algorithms are designed in such a way that they are invariant with respect to the shuffling countermeasure. This therefore calls for the design of strong side-channel countermeasures to protect against CC-based side-channel attacks.

We perform experimental validation of our CC attack using valid ciphertexts, on the unprotected and proprietary shuffled implementation variants of HQC KEM taken from the pqm4 library [KRSS]. Table 1 summarizes our results and shows a comparison with previous works. Our attack is the only one that is not easily thwarted by refresh-on-failure low-cost countermeasures. We remark that we do not target masking countermeasures since we are not aware of a masking scheme for HQC KEM, as well as an available masked implementation of HQC KEM for analysis.

Our work therefore shows that low-cost detection based countermeasures can be rendered ineffective, and cannot offer standalone protection against CC-based side-channel attacks. Thus, our work encourages more study towards development of new low-cost countermeasures against CC-based side-channel attacks.

Table 1. Comparison with previous power/EM based side-channel attacks when attacking HQC with 128-bit parameters.

Reference	Target operation	Number of traces for key recovery			Avoided with refresh-on-failure countermeasures?
		Unprotected	Shuffled (Low-level)	Shuffled (Top-level)	
[SRSWZ21]	BCH decoder (replaced)	10,000	–	–	Yes
[SHR+22]	Reed-Solomon decoder	53,000	–	–	Yes
[GLG22]	Hadamard transform	20,000	–	–	Yes
This work	ExpandAndSum	2	3682	64,200	**No**
This work	FindPeaks	200,000	20,000,000	–	**No**

2 Preliminaries

Vectors and matrices are denoted by bold lowercase and uppercase letters. We use zero-based indexing for vectors, and $\mathbf{a}[i]$ denotes the i-th entry of $\mathbf{a}$. We sometimes abuse the notation $\mathbf{a}[i]$ to represent the entry of $\mathbf{a}$ whose index is $(i \bmod n)$. The cyclic rotation of a vector $\mathbf{a}$ by i positions to the right is denoted by $\mathbf{a} \gg i$. In our analysis, it will be useful to talk about contiguous parts of a cyclic vector, which are called slices. A k-bit slice of $\mathbf{a}$ starting at position i is denoted as $\text{slice}_i^k(\mathbf{a}) = [\mathbf{a}[i], \ldots, \mathbf{a}[i+k-1]]$. We let slices wrap around the end of a vector, therefore $\text{slice}_{n-1}^3(\mathbf{a}) = [\mathbf{a}[n-1], \mathbf{a}[0], \mathbf{a}[1]]$. This, allows for a more concise notation and somewhat more intuitive descriptions in specific parts of our analysis. We denote the space of all byte arrays of length n bytes as $\mathcal{B}^n$. If $\mathbf{b} \in \mathcal{B}^n$, then $\mathbf{b}[i]$ denotes its i-th byte, while $\mathbf{b}[i]_k$ denotes the k^{th} bit of $\mathbf{b}[i]$.

A binary $[n, k]$-linear code is a k-dimensional linear subspace of $\mathbb{F}_2^n$, where $\mathbb{F}_2$ denotes the binary field. The Hamming weight of a vector $\mathbf{v}$, denoted by $\mathrm{w}(\mathbf{v})$, is the number of its non-zero entries. The Hamming distance between two vectors is the number of coordinates in which they differ. The support of a binary vector $\mathbf{v} \in \mathbb{F}_2^n$, denoted as $\text{supp}(\mathbf{v})$, is the set of indexes of its non-null entries. We denote by $\mathcal{L}(w)$ the subset of $\mathbb{F}_2^n$ consisting only

of elements of weight w. The product[1] of two binary vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}_2^n$ is defined as

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i \in \mathrm{supp}(\mathbf{u})} (\mathbf{v} \gg i) = \sum_{i \in \mathrm{supp}(\mathbf{v})} (\mathbf{u} \gg i).$$

3 HQC

This section presents the parameters and algorithms used by HQC, followed by a brief discussion of previous SCA attacks against the scheme.

3.1 Parameters and Algorithms

HQC provides its parameter sets defining $(n, k, \delta, w, w_{\mathbf{r}}, w_{\mathbf{e}})$ for each security level, as shown in Table 2. The integers n and k are the length and dimension, respectively, of the public error correction code $\mathcal{C}$ used by HQC. Parameters $w, w_{\mathbf{r}}$ and $w_{\mathbf{e}}$ correspond to the weights of the sparse vectors used for key generation and encryption. Parameter $M = n_2/128$ is the multiplicity of the repeated Reed-Muller code, which is a building block of code $\mathcal{C}$ and it is an important parameter used in our attacks.

Table 2. Parameter sets for HQC [MAB+21].

Security level	n_1	n_2	M	n	k	w	$w_{\mathbf{r}} = w_{\mathbf{e}}$	Failure probability (upper bound)
128	46	384	3	17669	128	66	77	2^{-128}
192	56	640	5	35851	192	100	114	2^{-192}
256	90	640	5	57637	256	133	149	2^{-256}

For each parameter set, HQC uses a public binary $[n, k]$-linear code $\mathcal{C}$, which is defined by the concatenation between a repeated Reed-Muller (RM) code and a Reed-Solomon (RS) code. The repeated Reed-Muller code is a binary $[n_2, 8]$-linear code whose minimum distance is $n_2/2$, while the Reed-Solomon is a non-binary linear code. Since the internal details of the RS codes are not important in this work, we simply consider that they take a string of k bits, then add redundancy to it, and return $8n_1$ bits. Figure 1 illustrates the encoding process of code $\mathcal{C}$.

Code $\mathcal{C}$ comes with an efficient pair of algorithms for encoding and decoding. Intuitively, if $\mathbf{m} \in \mathbb{F}_2^k$, then $\mathrm{Encode}(m)$ is responsible to add redundancy to it. If $\mathbf{c} = \mathrm{Encode}(\mathbf{m}) + \mathbf{e}$ for some error vector $\mathbf{e} \in \mathbb{F}_2^n$, then $\mathrm{Decode}(\mathbf{c})$ returns $\mathbf{m}$ as long as the weight of $\mathbf{e}$ is not too large. Notice that the decoding of a corrupted codeword goes in the inverse direction of Figure 1, that is, first remove the padding bits, then use the decoding algorithms for the Reed-Muller code in each block, followed by the Reed-Solomon decoder to the $8n_1$ bits.

Decoding Repeated Reed-Muller codes. We now review the RM decoding in detail, as it is the main focus of our work. Let M be the multiplicity[2] of the RM code and let $n_2 = 128M$. Suppose $\mathbf{w} \in \mathbb{F}_2^8$ was encoded using the repeated RM code as $\mathbf{z} = \mathrm{Encode}_{\mathrm{RM}}(\mathbf{w}) \in \mathbb{F}_2^{n_2}$. Let $\mathbf{e} \in \mathbb{F}_2^{n_2}$ be an error vector of weight $\mathrm{w}(\mathbf{e}) < \lfloor n_2/4 \rfloor$. Given a corrupted codeword $\mathbf{z}' = \mathbf{z} + \mathbf{e}$, the RM decoding procedure recovers $\mathbf{w}$ with the following 3 steps.

1. ExpandAndSum: This operation over $\mathbf{z}'$ returns vector $\mathbf{a} \in \mathbb{Z}^{128}$, such that $\mathbf{a}[i] = \sum_{m=0}^{M-1} \mathbf{z}'[i + 128m]$. That is, ExpandAndSum sums the M bits corresponding to the (possibly corrupted) repetitions of each of the 128 bits of the original RM codeword.

[1] It is well-known that $\mathbb{F}_2^n$ equipped with this product is isomorphic to the polynomial ring $\mathbb{F}_2[x]/(x^n - 1)$.
[2] The multiplicity is the number of times the Reed-Muller code is repeated.

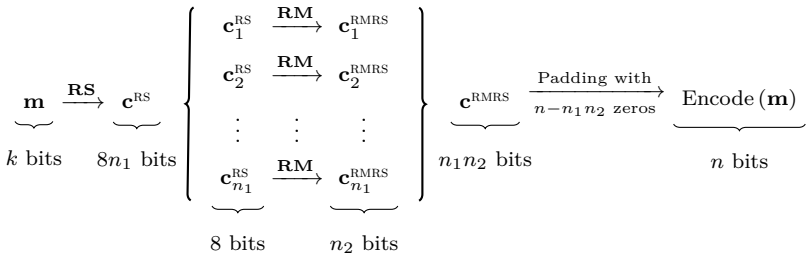


Figure 1. The encoding process using concatenated Reed-Muller and Reed-Solomon codes with padding.

2. HADAMARD transform: This operation outputs $\texttt{transform} \leftarrow \text{HADAMARD}(\mathbf{a}) \in \mathbb{Z}^{128}$. It can be shown that $\texttt{transform}[i]$ represents how likely the corrupted codeword $\mathbf{z}'$ resulted from the RM encoding of the binary representations of i or $(128 + i)$, depending on the sign of $\texttt{transform}[i]$ [MS77].

3. FINDPEAKS: The final step is then to compute $\mathbf{w} \leftarrow \text{FINDPEAKS}(\texttt{transform})$. This operation first finds the index $\texttt{pos}$ with the largest absolute value in $\texttt{transform}$. Then, if $\texttt{transform}[\texttt{pos}] \geq 0$, it sets $w = \texttt{pos} + 128$, otherwise it sets $w = \texttt{pos}$. Finally, it returns vector $\mathbf{w} \in \mathbb{F}_2^8$ as the binary representation of w.

Algorithms used by HQC. The IND-CPA encryption scheme underlying the IND-CCA secure HQC KEM is presented in Algorithm 1. While the key generation and encryption are more directly understood, let us see why decryption works. If we expand vector $\mathbf{c}' = \mathbf{v} - \mathbf{u} \cdot \mathbf{y}$, as computed in the decryption algorithm, we obtain $\mathbf{c}' = \text{Encode}(\mathbf{m}) + \mathbf{x} \cdot \mathbf{r}_2 - \mathbf{r}_1 \cdot \mathbf{y} + \mathbf{e}$. Intuitively, since $\mathbf{x}, \mathbf{y}, \mathbf{r}_1, \mathbf{r}_2$, and $\mathbf{e}$ all have low weight, we expect $\mathbf{e}' = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{r}_1 \cdot \mathbf{y} + \mathbf{e}$ to have a relatively low weight. HQC parameters are carefully chosen to ensure that $\text{w}(\mathbf{e}')$ is sufficiently low for it to be corrected out of $\mathbf{c}'$ with overwhelming probability, using the concatenated decoding algorithm $\mathcal{C}$.Decode. The concatenated decoding of $\mathcal{C}$, first applies the internal Reed-Muller decoder followed by the external Reed-Solomon decoder to correct the errors in $\mathbf{c}'$, and hopefully obtain the original message $\mathbf{m}$.

The scheme uses the implicit-rejection Fujisaki-Okamoto transformation [HHK17] in order to build upon the IND-CPA secure HQC encryption scheme into an IND-CCA secure HQC KEM. This enables the detection of invalid/malicious ciphertexts with overwhelming probability, thereby providing theoretical security against chosen-ciphertext attacks.

3.2 Prior Works and Motivation

The decapsulation procedure of HQC KEM has been subjected to several timing and power side-channel attacks to recover the long-term secret key [BDH+21, RRCB20, HHP+21, RRD+23]. In the following, we briefly describe prior side-channel attacks on the decapsulation procedure of HQC KEM.

The first power side-channel attack in this context was proposed by Schamberger et al. [SRSWZ21], targeting the original version of HQC based on BCH codes. It used malformed ciphertexts to query the decapsulation oracle, uses side-channel leakage from the BCH decoder to realize a Plaintext Checking (PC) oracle that returns if the error has been corrected for a given chosen-ciphertext. They were able to recover full key in $\approx$ 10000 traces. The same authors adapted their attack to the updated version of HQC based on concatenated Reed-Muller and Reed-Solomon codes, by exploiting leakage from the Reed-Solomon decoder, realizing a similar PC oracle for key recovery [SHR+22]. Subsequently, Goy et al. [GLG22] presented a simplified version of this attack, showing that leakage from the Hadamard transform operation in the RM decoder, for malformed/invalid

Algorithm 1. IND-CPA secure HQC PKE Scheme [MAB+21].

procedure CPAPKE.KeyGen
  $\mathbf{h} \leftarrow \mathbb{F}_2^n$
  $(\mathbf{x}, \mathbf{y}) \leftarrow (\mathcal{L}(w))^2$
  $\mathbf{s} \leftarrow \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$
  Return $\mathsf{pk} \leftarrow (\mathbf{h}, \mathbf{s}), \mathsf{sk} \leftarrow (\mathbf{x}, \mathbf{y})$

procedure CPAPKE.Encrypt($\mathsf{pk}, \mathbf{m} \in \mathcal{B}^*, \mathsf{seed} \in \mathcal{B}^*$)
  $(\mathbf{e}, \mathbf{r}_1, \mathbf{r}_2) \leftarrow$ Pseudorandom sample from $(\mathcal{L}(w_{\mathbf{e}}) \times \mathcal{L}(w_{\mathbf{r}}) \times \mathcal{L}(w_{\mathbf{r}}))$ using seed
  $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
  $\mathbf{v} \leftarrow \mathcal{C}.\text{Encode}(\mathbf{m}) + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$
  Return $\mathsf{ct} = (\mathbf{u}, \mathbf{v})$

procedure CPAPKE.Decrypt(ct, sk)
  $\mathbf{c}' = \mathbf{v} - \mathbf{u} \cdot \mathbf{y}$
  $\mathbf{m}' = \mathcal{C}.\text{Decode}(\mathbf{c}')$
  Return $\mathbf{m}'$

ciphertexts for full key recovery in $\approx$ 20000 traces. Similarly, there have been timing side-channel [GHJ+22, PT20] as well as cache-timing attacks [HSC+23], targeting leakage from the re-encryption procedure for key recovery, to realize a PC oracle that also relies on malformed/invalid ciphertexts for key recovery.

All the aforementioned Chosen-Ciphertext (CC) based side-channel attacks can be prevented using the masking countermeasure, but masking the entire decapsulation procedure of HQC KEM is expected to incur a significant performance penalty, as has been observed for other PQC schemes [OSPG18, BGR+21]. Moreover, we are not aware of a masking scheme for HQC KEM. This therefore calls for potential low-cost countermeasures to counter such attacks against HQC KEM.

In this context, we make a key observation that all the aforementioned attacks rely on invalid/malformed ciphertexts that always induce a decapsulation failure. Thus, an alternative approach towards countering such attacks has been towards development of low-cost countermeasures that attempt to detect such malicious ciphertexts [XPRO20,RCDB22]. If detected as malicious, the DUT can choose to refresh the secret key, ensuring further exposure for attacks is prevented. Thus, the attacker is restricted to recovering the secret key with only a single trace, which therefore offers concrete protection against all the previously proposed CC based side-channel attacks on HQC KEM. We refer to this countermeasure as the *decapsulation failure check* countermeasure.

3.3 Attacking Decapsulation Failure Check Countermeasure against CC based SCA

In this work, we perform a critical analysis of the decapsulation failure check countermeasure and construct novel side-channel attacks using valid chosen-ciphertexts that does not trigger refresh of the secret key. This calls for a completely new approach that uses valid chosen-ciphertexts for key recovery. This leads to identification of new side-channel vulnerabilities in the decapsulation procedure, that provides new type of information about the secret key. This also leads to identification of novel key recovery algorithms for practical key recovery attacks. This therefore makes our attack unique compared to prior side-channel attacks on HQC KEM, which can be easily defeated by the decapsulation failure check countermeasure. Moreover, we also demonstrate novel attacks on the shuffling countermeasure that protects the targeted operations by our attack.

Since our attack only relies on valid ciphertexts to defeat the decapsulation failure check countermeasure, we are limited to targeting operations in the decryption procedure for key recovery. In this respect, we identify two types of leakages from the RM decoder in the

decryption procedure that can be exploited for key recovery with valid chosen ciphertexts. They are: 1) ExpandAndSum Operation 2) FindPeaks Operation. We remark that our work is the first to exploit leakage from these two operations for key recovery. In the following, we demonstrate novel side-channel attacks exploiting leakage individually from both these operations for key recovery. We start with briefly describing the adversary model for our attack.

Adversary Model. The attacker's target is to recover the long-term secret key sk used by the target's decapsulation procedure of Kyber KEM. We assume physical access to DUT performing decapsulation for power/EM measurements. Since our attack is a CC-based attack, we assume the attacker's ability to communicate with the target decapsulation procedure with chosen ciphertexts of their choice. We note that this is a standard adversarial model used in several CC-based side-channel attacks [RRCB20, XPRO20, BDH+21]. All the chosen ciphertexts used in our attack for key recovery are valid ciphertexts. Since it is a profiled attack, the attacker has access to a clone device, on which they can control the secret key, so as to build side-channel templates for any intermediate variable of their choice.

Experimental Setup. We target the HQC implementation from the *pqm4* library [KRSS], a benchmarking and testing framework for PQC schemes on the 32-bit ARM Cortex-M4 microcontroller. This is the main optimization target recommended by NIST for embedded software implementations. Since we are not aware of any assembly optimized implementations of HQC for the embedded microcontrollers, all of our analysis was conducted on the reference implementation. We demonstrate our attacks using the EM side-channel measurements captured from an STM32F4 microcontroller (running at 24 MHz) using a Langer RF-U 5-2 near-field EM probe placed on top of the chip and are then collected using a Lecroy 610Zi oscilloscope at a sampling rate of 1.25 GSam/sec, amplified 30dB with a pre-amplifier. We also used a 48 MHz analog low-pass filter to remove high frequency noise in our traces.

4 SCA of ExpandAndSum Operation

The Reed-Muller decoding procedure is computed separately over n_1 RM corrupted codewords of $\mathbf{c}'$ in the decryption procedure, and each of the RM codewords is passed as variable `src` to ExpandAndSum. The ExpandAndSum is the first operation of the Reed-Muller decoder and refer to Figure 2 for its implementation as a C code snippet. The input codeword of $16 \cdot M$ bytes, or n_2 bits, is stored in the `src` array. The operation runs in two steps. In the first step (Lines 6 to 10), each bit of the first 16 bytes are extracted and stored separately as individual elements in the output array `dest` of 128 integers. In the second step (Lines 11 to 18), each bit of the successive $(M-1)$ segments of 16 bytes are extracted and accumulated into the corresponding indices in the output array `dest`.

Thus, we can clearly observe a *bitwise manipulation* behaviour of the codeword in `src` in both the steps of the ExpandAndSum operation. Similar behavior has also been observed in other PQC schemes such as Kyber and Frodo [ABD+20] leading to message and key recovery attacks [RR21, RBRC20, NDGJ21]. However, this is the first time that such behavior is reported for HQC, and in the following, we demonstrate exploitation of side-channel leakage to extract the complete codeword $\mathbf{c}'$ in `src` one bit at a time.

4.1 Leakage Detection and Template Building

We use the well-known TVLA metric for leakage detection [GJJR11], and particularly focus on detecting leakage from single bits of the codeword stored in the `src` array. To test

leakage of the first bit of the codeword (i.e.) $\texttt{src}_0$, we capture side-channel measurements for ciphertexts from two separate sets, CT_0 and CT_1, defined next. For CT_0 and CT_1, we collect 10,000 ciphertexts such that the first bit $\texttt{src}_0 = 0$ and $\texttt{src}_0 = 1$ respectively, while all the other bits of the codeword are random. This can be chosen by the attacker on a clone device, where they control the secret key.

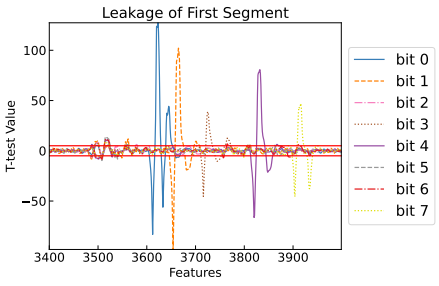
We now obtain two sets of side-channel measurements, $\mathcal{T}_0$ and $\mathcal{T}_1$, corresponding to decapsulation of ciphertexts CT_0 and CT_1, respectively. We normalize each trace and compute the Welch's t-test to identify the differentiating features between the trace sets. Figures 3a and 3b show 8 t-test plots validating leakage from the bits processed in Steps 1 and 2 of ExpandAndSum, respectively. We can see noticeable peaks in the t-test corresponding to single bits of the codeword, clearly demonstrating presence of leakage due to the bitwise manipulation from both steps of ExpandAndSum. In the following, we demonstrate how the detected leakage can be exploited to recover single bits of the codeword one at a time.

4.2 Recovery of the Corrupted Codeword c′ in Two Phases

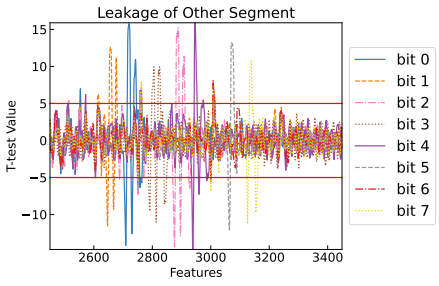
We now show an attack to recover bits of `src` that are processed either in Step 1 or 2 of ExpandAndSum. The attack can then be repeated to recover all bits of `src`, separately,

```
1  void expand_and_sum(uint16_t dest[128], uint8_t src[16*M]) {
2    // Remember that src represents one RM block of the corrupted codeword c'
3    size_t seg, bytepos, bitpos;
4
5    /* Bitwise manipulation of codeword in first 16-byte segment */
6    for (bytepos = 0; bytepos < 16; bytepos++) {
7      for (bitpos = 0; bitpos < 8; bitpos++) {
8        dest[bytepos*8+bitpos] = (src[bytepos] >> bitpos) & 1;
9      }
10   }
11   /* Bitwise manipulation of codeword in remaining M-1 16-byte segments */
12   for (block = 1; block < M; block++) {
13     for (bytepos = 0; bytepos < 16; bytepos++) {
14       for (bitpos = 0; bitpos < 8; bitpos++) {
15         dest[bytepos*8+bitpos] += (src[16*block+bytepos] >> bitpos) & 1;
16       }
17     }
18   }
19 }
```

Figure 2. C code snippet of ExpandAndSum operation in RM decoder of HQC KEM. The target operations/variables for SCA are highlighted in red.



(a) Leakage of the 8 bits in $\texttt{src}[0] = (\texttt{src}_0, \ldots, \texttt{src}_7)$ from Step 1 of ExpandAndSum operation.



(b) Leakage of the 8 bits in $\texttt{src}[16] = (\texttt{src}_{128}, \ldots, \texttt{src}_{135})$ from Step 2 of ExpandAndSum operation.

Figure 3. Observed t-test leakage of single bits of the codeword in `src`, considering Steps 1 and 2 of the ExpandAndSum algorithm.

and would ultimately provide a full recovery of $\mathbf{c}'$. The attack is done in two phases. First we build templates for a target device, and then we exploit exploit the ExpandAndSum leakage to recover $\mathbf{c}'$.

4.2.1 One-time Template Building For a Given Target

This phase involves building side-channel templates for the targeted bit of the codeword. It is a one-time process for a given target device as the templates are independent of the secret key, thus the same templates can be used to perform multiple attacks. We use the t-test plots from Figure 3 obtained from leakage detection of $\texttt{src}_0$, and select those features whose absolute t-test value is above a certain threshold. This gives us the set $\mathcal{P}_0$ of our points of interest (PoI). We stress that the threshold is a parameter of the experimental setup, and can be empirically determined. Using selected features $\mathcal{P}_0$, we build a reduced trace set $\mathcal{RT}_{(0,i)}$ from each $\mathcal{T}_i$ for $i = \{0, 1\}$. We can then compute the mean and co-variance matrix of each reduced trace set $\mathcal{RT}_{(0,i)}$, which we denote as $\mu_{(0,i)} \in \mathbb{R}^{\|\mathcal{P}_0\|}$ and $\Sigma_{(0,i)} \in \mathbb{R}^{(\|\mathcal{P}_0\|)\times(\|\mathcal{P}_0\|)}$, respectively. Thus, the reduced template for $\texttt{src}[0]_0 = i$ is denoted as $\texttt{Template}_{(0,i)} = (\mu_{(0,i)}, \Sigma_{(0,i)})$, for $i = 0$ or $i = 1$. An analogous process can be used to build templates $\texttt{Template}_{(j,i)}$ for all bits $\texttt{src}_j$, for $j = 0$ to $n_2 - 1$.

4.2.2 Exploitation Phase with Valid Ciphertexts

In the exploitation phase, the attacker tries to recover the first bit $\texttt{src}_0$ from a given trace tr as follows. The PoI set $\mathcal{P}_0$ is used to build a reduced trace tr'_0, corresponding to $\mathcal{P}_0$. Then the the maximum likelihood $f(\mathsf{tr}'_0)_i$ of the reduced trace tr'_0 for $i \in \{0, 1\}$ is computed, to match with the templates $\texttt{Template}_{(0,i)}$ using the following expression:

$$f(\mathsf{tr}'_0)_i = \frac{1}{\sqrt{(2\pi)^k \|\Sigma\|}} \exp\left(-\left(\mathsf{tr}'_0 - \mu_{(0,i)}\right)^T \cdot \Sigma^{-1} \cdot \left(\mathsf{tr}'_0 - \mu_{(0,i)}\right)\right).$$

This is used to classify whether $\texttt{src}_0 = 0$ or $\texttt{src}_0 = 1$ based on the highest value of $f(\mathsf{tr}'_0)_i$. The same approach can be used to recover all the bits of the codeword one bit at a time.

4.2.3 Experimental Results

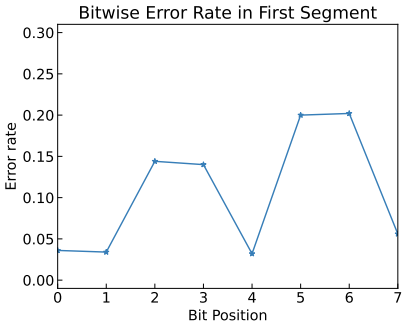
Figures 4a and 4b show the error rate when recovering the first 8 bits from the first segment (i.e.) $\texttt{src}[0]$ and first 8 bits from the second segment (i.e.) $\texttt{src}[16]$, which are processed in Steps 1 and 2 of ExpandAndSum, respectively. In both figures, we see a non-negligible error rate between 4% and 20% when recovering single bits of the codeword. The leakage is stronger in the first step, because it involves storing the single bits of the codeword in memory, while the second step only involves extracting the bit of the codeword in the registers, which manifests as weaker leakage.

We also observe the exact very similar success rates for other bits of the codeword, since they are manipulated sequentially, by the same instructions. This gives rise to an error rate of $\approx 14\%$ for recovering each bit of the codeword $\mathbf{c}'$ with $\mathbf{c}' = \mathbf{mG} + \hat{\mathbf{e}}$, where $\hat{\mathbf{e}} = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$, except for the last $n - n_1 n_2$ padding bits which are not dropped before being processed by ExpandAndSum.

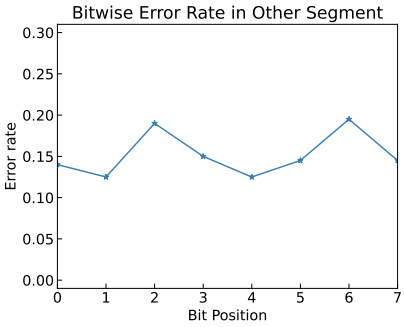
While we are not able to achieve perfect recovery of $\mathbf{c}'$ through SCA, we show that an attacker can still recover the secret key in the presence of noise through our novel key recovery algorithm.

4.3 Key Recovery

We now analyze how to use information on the corrupted codeword $\mathbf{c}' = \text{Encode}(\mathbf{m}) + \hat{\mathbf{e}}$ to recover the secret key. Suppose we generate a number η of valid ciphertexts $\left(\mathbf{u}^i, \mathbf{v}^i\right)$, that



(a) Each of the 8 bits in $\texttt{src}[0] = (\texttt{src}_0, \ldots, \texttt{src}_7)$ from Step 1 of ExpandAndSum operation.



(b) Each of the 8 bits in $\texttt{src}[16] = (\texttt{src}_{128}, \ldots, \texttt{src}_{135})$ from Step 2 of ExpandAndSum operation.

Figure 4. Error rate in recovering single bits of the codeword from leakage of the ExpandAndSum operation.

are created using values $\left(\mathbf{m}^i, \mathbf{r}_1^i, \mathbf{r}_2^i, \mathbf{e}^i\right)$, for each $i = 1$ to η. Then we use SCA on η traces to get an approximation of each corrupted codeword as $\left(\mathbf{c}'\right)^i \approx \text{Encode}\left(\mathbf{m}^i\right) + \mathbf{r}_2^i\mathbf{x} + \mathbf{r}_1^i\mathbf{y} + \mathbf{e}^i$.

Notice that the quality of the approximation depends on the strength of the SCA leakage. We can then write η approximate linear equations as

$$\left(\mathbf{c}'\right)^i - \text{Encode}\left(\mathbf{m}^i\right) \approx \sum_{j \in \text{supp}(\mathbf{x})} \left(\mathbf{r}_2^k \gg j\right) + \sum_{j \in \text{supp}(\mathbf{y})} \left(\mathbf{r}_1^k \gg j\right) + \mathbf{e}^k.$$

Remember that the main argument for the effectiveness of HQC decoding is that the products $\mathbf{x} \cdot \mathbf{r}_2^k$ and $\mathbf{y} \cdot \mathbf{r}_1^k$ are somewhat sparse, and so is their sum. Because of this sparsity, we expect $\left(\mathbf{c}'\right)^k - \text{Encode}\left(\mathbf{m}^k\right)$ to have a large number of ones in the same locations as the circular shifts of $\mathbf{r}_2^k$ and $\mathbf{r}_1^k$ that are selected by the non-null entries of $\mathbf{x}$ and $\mathbf{y}$, respectively, in the corresponding products of the equation above.

This motivates us to define vectors $\texttt{likelihoods}_\mathbf{x}$ and $\texttt{likelihoods}_\mathbf{y}$, whose entries represent the likelihoods of the corresponding index being 1 in $\mathbf{x}$ or $\mathbf{y}$, by computing the similarity between $\left(\mathbf{c}'\right)^k - \text{Encode}\left(\mathbf{m}^k\right)$ and the vectors $\mathbf{r}_2^k$ and $\mathbf{r}_1^k$, respectively. Formally, we let each entry j of the likelihood vector associated it $\mathbf{x}$ to be defined as[3]

$$\texttt{likelihoods}_\mathbf{x}[j] = \sum_{k=1}^{\eta} \left| \text{supp}\left(\left(\mathbf{c}'\right)^k - \text{Encode}\left(\mathbf{m}^k\right) \right) \cap \text{supp}\left(\mathbf{r}_2^k \gg j\right) \right|.$$

The definition of $\texttt{likelihoods}_\mathbf{y}$ is analogous, but uses $\mathbf{r}_1$ instead of $\mathbf{r}_2$. We emphasize that, although these likelihood vectors do not represent a probability, we expect that entries $i \in \text{supp}\left(\mathbf{x}\right)$ would show a high value of $\texttt{likelihoods}_\mathbf{y}[i]$, and analogously for $\mathbf{y}$.

Figure 5 shows the histogram of entries $\texttt{likelihoods}_\mathbf{x}[j]$ when j is, or is not, a non-null of $\mathbf{x}$. To generate these values, we simulated the case when $\eta = 10$ SCA decapsulation traces were used, and assumed that the obtained information on the bits of $\mathbf{c}'$ have an error rate of $\varepsilon = 25\%$. Even under these noisy SCA assumptions, we can see that $\texttt{likelihoods}_\mathbf{x}[i]$ indeed tends to be higher for values i for which $\mathbf{x}[i] = 1$. In particular, while the separation between the distributions is not perfect, if we take the values of j with the lowest values of $\texttt{likelihoods}_\mathbf{x}[j]$, our results suggest that we can be rather confident that they correspond to zero entries in $\mathbf{x}$.

[3] We remark that the definition of these likelihood vectors is somewhat similar to the computation of the counters of unsatisfied parity-check equations used by Gallagher's [Gal62] well-known decoding algorithm for low-density parity-check codes.

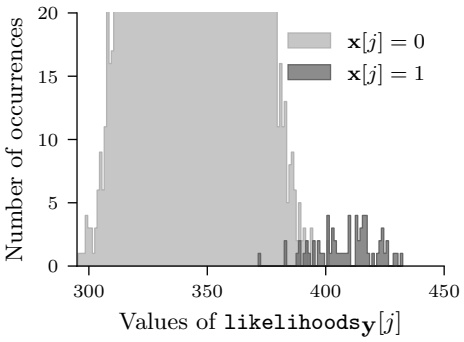


Figure 5. The distribution of the likelihood values, using $\eta = 10$ simulated SCA traces, assuming bit error rate of $\varepsilon = 25\%$ for the SCA recovery of $\mathbf{c}'$.

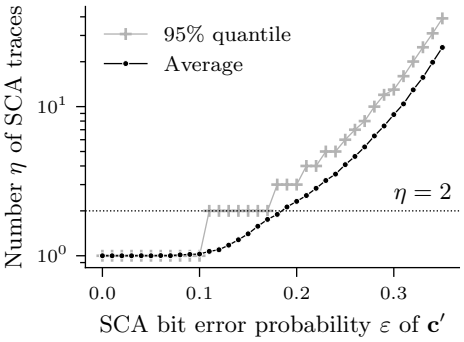


Figure 6. The number of SCA traces of ExpandAndSum to recover the key for increasing bit error rate ε, considering parameters for 128 bits of security.

Algorithm 2. Solving the key equation using likelihood vectors.

procedure TryToSolveKeyEquation(likelihoods$_{\mathbf{x}}$, likelihoods$_{\mathbf{y}}$)
  for n_zeros_in_x $= 1$ to n **do**
    n_zeros_in_y $\leftarrow n -$ n_zeros_in_x
    $Z_{\mathbf{x}} \leftarrow$ Indexes of the n_zeros_in_x lowest values of likelihoods$_{\mathbf{x}}$
    $Z_{\mathbf{y}} \leftarrow$ Indexes of the n_zeros_in_y lowest values of likelihoods$_{\mathbf{y}}$
    Fix $\mathbf{x}[i] = 0$ for each $i \in Z_{\mathbf{x}}$, and $\mathbf{y}[j] = 0$ for each $j \in Z_{\mathbf{y}}$
    Try to solve linear system $\mathbf{s} = \mathbf{x} + \mathbf{hy}$ for the n non-fixed values
    if low weight solutions $\mathbf{x}$ and $\mathbf{y}$ are found **then**
      return $(\mathbf{x}, \mathbf{y})$
  return $\perp$ ▷ More SCA traces needed to build better likelihood vectors

Now we have to actually use the likelihood vectors to find the key secret $(\mathbf{x}, \mathbf{y})$. While likelihoods vectors can be combined with advanced information-set decoding algorithms [HPR+22], we propose a simple linear algebra algorithm. This makes the evaluation of the key recovery more straightforward, since we do not need to take into account complicated attacker trade-offs between key recovery complexity and number of traces.

The key recovery algorithm from the likelihoods vectors is shown as Algorithm 2. At each iteration, the algorithm fixes a number of zeros in $\mathbf{x}$ and in $\mathbf{y}$ based on their corresponding likelihood vectors, and tries to solve for the missing part, which always consists of n bits. Since the key equation $\mathbf{s} = \mathbf{x} + \mathbf{hy}$ can be written as n binary linear equations, the algorithm should be able to find the key if the entries fixed as 0 were correctly guessed.

4.3.1 Experimental Results for Key Recovery targeting ExpandAndSum

Figure 6 shows the number of traces for full key recovery, considering different bit error rates ε. With very high probability, the attack works with only one query for values of ε up to 10%. Furthermore, the attack works for relatively high error rates of 35% with about 25 traces on average. Since the observed error rate in our real SCA experiments was about $\varepsilon \approx 14\%$, we expect that only $\eta = 2$ SCA traces should be enough for key recovery. Comparing the number of traces for key recovery with prior works with invalid ciphertexts [SMS19, SRSWZ21, GLG22], we observe that bitwise leakage of the codeword from the ExpandAndSum operation enables full key recovery in a few tens of traces, while prior works requires ≈ 10 thousand to ≈ 20 thousand traces for key recovery.

5 SCA of FindPeaks Operation

In this section, we demonstrate how leakage from the FindPeaks operation, which is the final operation in the RM decoding procedure can be exploited using valid chosen-ciphertexts for full key recovery.

5.1 Using SCA to Obtain the Peak Position

Refer to Figure 7 for the C-code snippet of the FindPeaks operation. Given an array `transform` as input, the main loop is responsible for finding the index with the highest absolute value, which is the peak. The loop iterates over every element of `transform`, and computes its corresponding absolute value `abs` in Line 8. Then, it computes variable `mask` (Line 10) based on the absolute value of the previous peak (`peak_abs`) and `abs`. The variable `mask` is defined as `mask` = `0x0000` when `abs` $\leq$ `peak_abs`, and `mask` = `0xFFFF` when `abs` $>$ `peak_abs`. The `mask` variable is then used to update the values of variables `pos`, `peak`, and `peak_abs` whenever a change is required, which is characterized by `mask` = `0xFFFF`.

```
1  uint8_t find_peaks(uint16_t transform[128]) {
2      uint16_t peak_abs = 0;
3      uint16_t peak = 0;
4      uint16_t pos = 0;
5      uint16_t t, abs, mask;
6      for (uint16_t i = 0; i < 128; i++) {
7          t = transform[i];
8          abs = t ^ ((-(t >> 15)) & (t ^ -t));
9          /* Computation of Mask Variable */
10         mask = -(((uint16_t) (peak_abs - abs)) >> 15);
11         /* Utilization of Mask Variable */
12         peak ^= mask & (peak ^ t);
13         pos ^= mask & (pos ^ i);
14         peak_abs ^= mask & (peak_abs ^ abs);
15     }
16     pos |= 128 & ((peak >> 15) - 1);
17     return (uint8_t) pos;
18 }
```

Figure 7. C code snippet of the FindPeaks operation used by the Reed-Muller decoder in HQC. The target operations and variables for SCA are highlighted in red.

Therefore, we identify that tracking the value of the `mask` variable enables to recover the index of the peak. In particular, the last occurrence of `mask` = `0xFFFF` in the 128 iterations of FindPeaks provides the final position `pos` of the peak variable, as updated in Line 13. Furthermore, notice that this value corresponds to the 7 least significant bits of the output of FindPeaks. Now, remember that variable `mask` only takes two possible values: `0x0000` or `0xFFFF`. Since the values have a high Hamming distance of 16 bits, these cases should be easily distinguishable through power and EM side-channel analysis. In order to distinguish between `mask` = `0x0000` or `0xFFFF`, we use the same t-test based template technique used to recover single bits of the RM codeword using leakage from ExpandAndSum in Section 4.

Compared with leakage from the ExpandAndSum operation, leakage of the `mask` in FindPeaks is much more pronounced because of the large Hamming distance between its two possible values. We propose to build a template for the `mask` = `0x0000` or `0xFFFF`, in each of the 128 iterations of the FindPeaks operation. This template can then be used to recover the value of the `mask` variable in every iteration of the FindPeaks operation.

Exploiting Leakage of `mask` Variable. Figure 8 shows the t-test leakage due to the `mask` variable in the first four iterations FindPeaks operation. We can observe corresponding

peaks in the t-test plot, and this leakage can be used to build template for `mask` = `0x0000/0xFFFF` for each of the iterations, and this can be done for all iterations of the FindPeaks operation. Given the clear leakage of `mask`, we obtained 100% success rate in recovering the `mask` variable even with single traces. Thus, we observe that this leakage can be used to easily recover the position of the peak, which in turn corresponds to the least 7 significant bits of the output of FindPeaks.

5.2 Key Recovery

In the following, we demonstrate a novel key recovery algorithm that can exploit information about the output of the FindPeaks operation, extracted through side-channels for full key recovery, when a sufficiently large number of challenges is analyzed.

5.2.1 The Relation between the Side-Channel Information and the Secret Key

Let us first show how we can encode the 7-bit value obtained through SCA in a meaningful way. Remember that FindPeaks is called n_1 times, that is, one call for each RM block. Let $\mathbf{c}_j^{\text{FP}} \in \mathbb{F}_2^8$ denote the output of FindPeaks applied to the j-th RM block. FindPeaks is the last step of the RM decoding, therefore there are two possibilities: either the RM decoder corrected all errors in the j-th block, or it did not. If the RM decoder successfully corrected all errors in the block, then $\mathbf{c}_j^{\text{FP}} = \mathbf{c}_j^{\text{RS}}$, where $\mathbf{c}_j^{\text{RS}}$ denotes the j-th 8-bit block of the Reed-Solomon encoding of the message $\mathbf{m}$, as shown in Figure 1.

But notice that, since the attacker knows $\mathbf{m}$, they can easily compute the exact value of each $\mathbf{c}_j^{\text{RS}}$ by simply encoding $\mathbf{m}$ with the Reed-Solomon code. Therefore, the attacker can use the side-channel information on the FindPeaks output to learn if the RM decoder did not corrected all errors in the j-th block. In particular, when the 7 least significant bits of $\mathbf{c}_j^{\text{FP}}$ and $\mathbf{c}_j^{\text{RS}}$ are different, then the RM decoder failed to decode the block. Formally, we can represent the side-channel information by a binary vector $\mathbf{d} \in \mathbb{F}_2^{n_1}$, whose entries j are defined as

$$\mathbf{d}[j] = \begin{cases} 0, & \text{if the 7 least significant bits of } \mathbf{c}_j^{\text{FP}} \text{ and } \mathbf{c}_j^{\text{RS}} \text{ are equal,} \\ 1, & \text{otherwise.} \end{cases}$$

Let us then investigate what kind of information on the secret key we can learn from the SCA result encoded in vector $\mathbf{d}$. Remember that the input to the decoding process is $\mathbf{c}' = \text{Encode}(\mathbf{m}) + \hat{\mathbf{e}}$, where $\hat{\mathbf{e}} = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$. The decoding procedure begins by dropping the last $n - n_1 n_2$ bits of $\mathbf{c}'$, and the resulting $n_1 n_2$-bit vector is broken into n_1 blocks of n_2 bits, and each of them is a RM codeword with some added noise.

Let $\mathbf{c}'_j$ and $\hat{\mathbf{e}}_j$ denote the corresponding blocks of n_2 bits of $\mathbf{c}'$ and $\hat{\mathbf{e}}$, respectively. That is, $\mathbf{c}'_j = \text{slice}_{jn_2}^{n_2}(\mathbf{c}')$ and $\hat{\mathbf{e}}_j = \text{slice}_{jn_2}^{n_2}(\hat{\mathbf{e}})$. Since the repeated Reed-Muller code used to

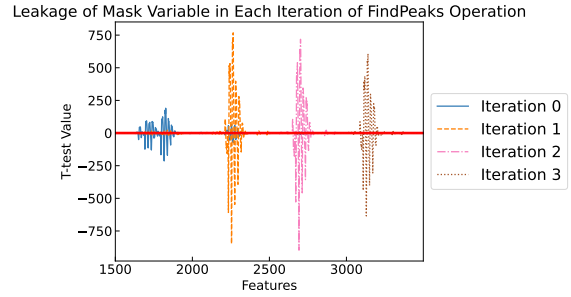


Figure 8. Observed t-test leakage corresponding to variable `mask` in the first four iterations of the FindPeaks operation

encode each block is a linear code with minimum distance $n_2/2$, then the RM decoder will be able to correct the errors in $\mathbf{c}'_j$ if the weight of $\hat{\mathbf{e}}_j$ is $\mathrm{w}(\hat{\mathbf{e}}_j) < \lfloor n_2/4 \rfloor$. Therefore, if the j-th bit of the SCA output $\mathbf{d}$ is equal to 1, then we know that the weight of block $\hat{\mathbf{e}}_j$ must be larger than $\lfloor n_2/4 \rfloor$.

But by definition $\hat{\mathbf{e}} = \left(\sum_{s\in\mathrm{supp}(\mathbf{r}_2)} (\mathbf{x} \gg s) + \sum_{s\in\mathrm{supp}(\mathbf{r}_1)} (\mathbf{y} \gg s) + \mathbf{e}\right)$, and since $\hat{\mathbf{e}}$ is sparse, the position of non-null entries of $\hat{\mathbf{e}}$ correlate with the non-null entries of the shifts of $\mathbf{x}$ and $\mathbf{y}$ that are selected by the binary vectors $\mathbf{r}_2$ and $\mathbf{r}_1$, respectively, in the summations above. Our main observation is that the attacker can then use the knowledge on the failures on each RM block, together with $\mathbf{r}_1$ and $\mathbf{r}_2$, to infer the number of non-null entries in the shifts of $\mathbf{y}$ and $\mathbf{x}$, respectively.

Let us make this observation more formal. For concreteness, consider the pair of vectors $\mathbf{y}$ and $\mathbf{r}_1$, but notice that we could have chosen $\mathbf{x}$ and $\mathbf{r}_2$ without loss of generality. Take the j-th RM block $\hat{\mathbf{e}}_j$, and notice that it can be written as

$$\hat{\mathbf{e}}_j \approx \sum_{s\in\mathrm{supp}(\mathbf{r}_1)} \mathrm{slice}^{n_2}_{jn_2}(\mathbf{y} \gg s) \approx \sum_{s\in\mathrm{supp}(\mathbf{r}_1)} \mathrm{slice}^{n_2}_{(jn_2-s)}(\mathbf{y}).$$

Therefore, from $\mathbf{d}[j]$ and $\mathbf{r}_1$, we can learn a small amount of information on the weight of $\mathrm{slice}^{n_2}_{(jn_2-s)}(\mathbf{y})$ for each s in $\mathrm{supp}(\mathbf{r}_1)$. For example, if $\mathbf{d}[j] = 1$, then, for each s in $\mathrm{supp}(\mathbf{r}_1)$, we increase our belief that the weight of $\mathrm{slice}^{n_2}_{(jn_2-s)}(\mathbf{y})$ is higher than previously thought, otherwise we decrease it. Furthermore, if we have a sufficiently large number of pairs of $(\mathbf{r}_1, \mathbf{d})$, the values of $(jn_2 - s) \bmod n$ should cover the full set of integers modulo n, which means we must be able to learn the weight of $\mathrm{slice}^{n_2}_i(\mathbf{y})$ for each possible $i = 0$ to $n - 1$.

This motivates us to define two vectors, $\mathbf{w}_1$ and $\mathbf{w}_2$ in $\mathbb{R}^n$, as follows. Each entry $\mathbf{w}_1[i]$ represents the average number of failures when $\mathrm{slice}^{n_2}_i(\mathbf{y})$ was selected by a non-null element of $\mathbf{r}_1$. Similarly, each entry $\mathbf{w}_2[i]$ represents the same quantity but for $\mathbf{x}$ and $\mathbf{r}_2$. Figure 9 illustrates how, for a sufficiently large number η of SCA observations, $\mathbf{w}_1$ and $\mathbf{w}_2$ can be seen as approximations of the weights of the possible slices of n_2 bits of $\mathbf{y}$ and $\mathbf{x}$, respectively. In particular, the figure shows this comparison for $\mathbf{w}_1$ and $\mathbf{y}$, considering $\eta = 5$ million traces.

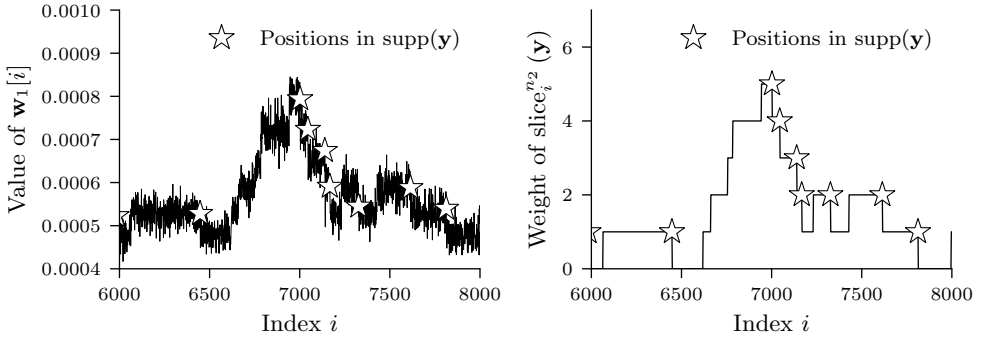


Figure 9. A comparison between $\mathbf{w}_1$ and the weight of each slice of n_2 bits of $\mathbf{y}$, considering η SCA traces and parameters for 128 bits of security.

5.2.2 Recovering the key

We now discuss how to use the information to effectively recover the key from vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, that are computed from observing SCA on FindPeaks for a number η of decryption challenges. Once again, we focus on secret vector $\mathbf{y}$, but the procedure works analogously for $\mathbf{x}$. Let us denote the weight of the i-th slice of n_2 bits of $\mathbf{y}$ by

$$\Delta_{\mathbf{y}}(i) = \mathrm{w}\left(\mathrm{slice}_i^{n_2}(\mathbf{y})\right).$$

Notice that, as shown in Figure 9, this definition implies that vector $\mathbf{w}_1$ can be seen as a noisy representation of $\Delta_{\mathbf{y}}$. Analyzing Figure 9, we can point two distinguishing patterns that allow us to distinguish some of the non-null entries of $\mathbf{y}$ from $\Delta_{\mathbf{y}}$.

- If $\Delta_{\mathbf{y}}(i) > \Delta_{\mathbf{y}}(i+1)$, then $\mathbf{y}[i] = 1$. This happens because there is a 1 in $\mathrm{slice}_i^{n_2}(\mathbf{y})$ but this 1 is not in $\mathrm{slice}_{i+1}^{n_2}(\mathbf{y})$. Since the slices differ only in the first and last entries, the only way to observe this is when $\mathbf{y}[i] = 1$ and $\mathbf{y}[i+n_2] = 0$.
- If $\Delta_{\mathbf{y}}(i-n_2) < \Delta_{\mathbf{y}}(i-n_2+1)$, then $\mathbf{y}[i] = 1$. In this case, a new one was found when going from $\mathrm{slice}_{i-n_2}^{n_2}(\mathbf{y})$ to $\mathrm{slice}_{i-n_2+1}^{n_2}(\mathbf{y})$. Since $\mathrm{slice}_{i-n_2+1}^{n_2}(\mathbf{y})$ contains $\mathbf{y}[i]$ and $\mathrm{slice}_{i-n_2}^{n_2}(\mathbf{y})$ does not, then this happens when $\mathbf{y}[i] = 1$ and $\mathbf{y}[i-n_2] = 0$.

Intuitively, these observations mean that indexes of non-null entries of $\mathbf{y}$ are typically located in the right edges of the graphs, but not on the left ones, as shown in Figure 9. We propose a simple approach to solve the problem of, given vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, computing the likelihoods of indexes being in the supports of $\mathbf{y}$ and $\mathbf{x}$, respectively. The approach is based on comparisons of moving averages to determine, for each index i, the likelihood of $\mathbf{w}_1[i]$ and $\mathbf{w}_2[i]$ being right edges. While points closer to a right edge indeed tend to correlate with non-null entries in the secret vectors, there are some important corner cases that have to be considered. However, because of space limitations, we leave a detailed description to the full version of this paper. After computing the likelihood vectors, we can feed them to Algorithm 2 to recover the key.

5.2.3 Experimental Results for Key Recovery targeting FindPeaks

Figure 10 shows the number of SCA traces needed for a successful attack against 128 bits of security. To simulate possible errors due to SCA, we consider that the SCA wrongly classifies a change in the peak value with probability ε. Notice that even a small increase in ε from 0 to 0.0025 has a significant impact on the number of traces needed, which goes from about 200,000 to more than a million. This is unlike ExpandAndSum operation where the noise in side-channel information on the attacker's effort (number of traces) had much lesser impact. However, the observed SCA output was 100% correct in all cases, that is, the SCA experimental error was $\varepsilon = 0$. Therefore, key recovery requires $\eta = 188{,}333$ SCA traces on average for full key recovery. As can be seen, leakage from the FindPeaks operation provides much lesser information compared to ExpandAndSum, which enables key recovery with less than a hundred traces (refer to Section 4.3.1).

6 Attacking Shuffled Implementations

We have shown that both the ExpandAndSum and FindPeaks operation leak information about the erroneous codeword $\mathbf{c}'$, which can be used for key recovery with valid ciphertexts. Both the targeted operations are sequential in nature, and we exploit information from variables manipulated in every iteration of both the operations. Thus, shuffling both the operations serve as a low-cost countermeasure to protect against the aforementioned attacks.

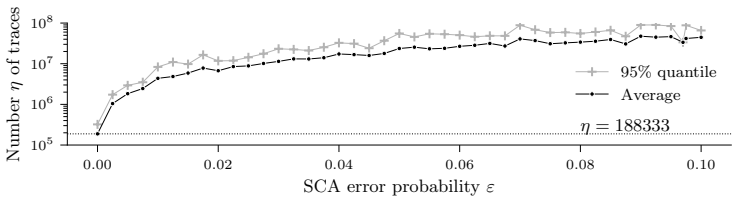


Figure 10. Number of SCA traces of the FindPeaks operation needed to recover the key for different levels ε of SCA error, considering parameters for 128 bits of security.

When the operations are shuffled, then the attacker does not know the value of the individual bits that are processed, and hence the attacker cannot perform leakage detection that was possible on unprotected implementations. But, notice that, the leakage of the bits of the codeword as well as the `mask` variable still exists. The attacker can therefore perform leakage detection and build templates on a clone device , on which the attacker has knowledge of the random permutation used for every execution. Thus, we assume that the attacker builds side-channel templates on the clone device, and uses the templates to perform attack on the target device. In the following, we demonstrate novel attacks on the shuffled implementations of the ExpandAndSum and FindPeaks operations.

6.1 Attacking Shuffled ExpandAndSum

We consider two variants of the shuffling countermeasure on the ExpandAndSum operation: (1) Intra-codeword Shuffling and (1) Inter-codeword Shuffling. We now refer to Fig. 2 for the C code snippet of the ExpandAndSum operation. For the intra-codeword shuffling, we shuffle the order in which the individual bits of the codeword are accessed within the ExpandAndSum operation of a given RM codeword. We generate a random permutation using the Fisher-Yates shuffle algorithm for the 128 bits accessed in each segment of the codeword. In this case, the attacker can extract the Hamming Weight of each of the n_1 n_2-bit RM codewords.

For inter-codeword shuffling, one can generate a random shuffling order in which the the n_1 codewords are decoded using the RM decoder algorithm. This performs a higher level shuffling on the processing of the codewords themselves, without altering the underlying bitwise operations within ExpandAndSum. From this shuffling variant, the attacker can only obtain HW of all the n_1 n_2-bit codewords, which is much less information compared to Intra-codeword Shuffling, where the attacker can extract HW of each of the n_1 codewords.

Extracting Side-Channel information from Shuffled ExpandAndSum. In both the shuffling variants, the leakage of the individual bits of the codeword still exists and is similar to the unprotected implementation. But the attacker does not know the shuffling order and hence the attacker can only build side-channel templates on a clone device where they have knowledge of the random permutation. Thus, we are still in a profiled setting, and we can perform similar experiments as that on the unprotected implementation. We performed experiments to recover single bits of the codeword from the shuffled ExpandAndSum operation, and observed an average bit error rate of 29% in recovering every bit of the codeword.

6.1.1 Key Recovery using the SCA Information

We know that the attacker can still build templates for the single bits of the codeword from ExpandAndSum, provided they are given access to a clone device with knowledge of the random permutation on the clone device. With this, the attacker can recover the

HW of the individual RM codewords or all the RM codewords together, depending on the shuffling variant.

We now discuss novel key recovery algorithms to perform full key recovery that applies to both the shuffling variants, that only uses information from the approximation of the Hamming weight RM codewords. We first demonstrate key recovery for the intra-codeword shuffling variant, and show extensions to the inter-codeword shuffling variant.

The Likelihoods Vectors. Suppose the attacker generates a valid ciphertext $(\mathbf{u}, \mathbf{v})$ using $\mathbf{m}, \mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{e}$, following an honest execution of the encryption algorithm. After asking the target to decrypt $(\mathbf{u}, \mathbf{v})$, the attacker uses SCA to obtain an array $\mathbf{w} \in \mathbb{Z}^{n_1}$, which is a list of the n_1 approximate Hamming weights of the n_2-bit corrupted RM codewords.

Let us then see how $\mathbf{w}$ relates to the secret key $(\mathbf{x}, \mathbf{y})$. Let $\mathbf{c}_j^{\text{RMRS}}$ denote the j-th RM codeword of $\mathbf{c}^{\text{RMRS}} = \text{Encode}(\mathbf{m})$, which is known by the attacker who generated the ciphertext. Similarly, let $\hat{\mathbf{e}}_j$ be the j-th block of n_2 bits of the accumulated noise $\hat{\mathbf{e}} = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$. Then, by definition, $\mathbf{w}[j] \approx \text{w}\left(\mathbf{c}_j^{\text{RMRS}} + \hat{\mathbf{e}}_j\right)$, for each entry $j = 0$ to $n_1 - 1$. The attacker, who holds vectors $\mathbf{w}$ and $\mathbf{c}^{\text{RMRS}}$, can compare them and reason as follows.

1. When $\mathbf{w}[j] > \text{w}\left(\mathbf{c}_j^{\text{RMRS}}\right)$, then at least one of the non-null entries of $\hat{\mathbf{e}}_j$ was responsible for flipping a bit of $\mathbf{c}_j^{\text{RMRS}}$ from a 0 to a 1.

2. Similarly, when $\mathbf{w}[j] < \text{w}\left(\mathbf{c}_j^{\text{RMRS}}\right)$, then there was an entry in $\hat{\mathbf{e}}_j$ responsible for flipping a 1 to a 0.

This motivates us to define the vector $\texttt{flip\_candidates} \in \{0,1\}^{n_1 n_2}$ such that

$$\texttt{flip\_candidates}[i] = \begin{cases} 1, \text{ if } \mathbf{c}^{\text{RMRS}}[i] = 0 \text{ and } \mathbf{w}\left[\lfloor i/n_2 \rfloor\right] > \text{w}\left(\mathbf{c}^{\text{RMRS}}_{\lfloor i/n_2 \rfloor}\right), \\ 1, \text{ if } \mathbf{c}^{\text{RMRS}}[i] = 1 \text{ and } \mathbf{w}\left[\lfloor i/n_2 \rfloor\right] < \text{w}\left(\mathbf{c}^{\text{RMRS}}_{\lfloor i/n_2 \rfloor}\right), \\ 0, \text{otherwise.} \end{cases} \tag{1}$$

Intuitively, if $\texttt{flip\_candidates}[i] = 1$, then $\mathbf{c}^{\text{RMRS}}[i]$ may have been one of the bits flipped by the error $\hat{\mathbf{e}}$ when computing $\mathbf{c}' = \mathbf{c}^{\text{RMRS}} + \hat{\mathbf{e}}$. In other words, if the weight of the j-th block increased when going from $\mathbf{c}^{\text{RMRS}}$ to $\mathbf{c}'$, then all of the null entries in block j are marked as 1. Alternatively, if the weight of the j-th block decreased, then all of the non-null entries in block j are marked as 1. If no change is observed, that is $\mathbf{w}\left[\lfloor i/n_2 \rfloor\right] = \text{w}\left(\mathbf{c}^{\text{RMRS}}_{\lfloor i/n_2 \rfloor}\right)$, then the entries of the j-th block are marked as 0.

We are now ready to define the likelihood vectors for entries of $\mathbf{x}$ and $\mathbf{y}$ as follows. Suppose the attacker creates η valid ciphertexts $\left(\mathbf{u}^k, \mathbf{v}^k\right)$ with known values $\left(\mathbf{m}^k, \mathbf{r}_1^k, \mathbf{r}_2^k\right)$. Let $\mathbf{w}^k$ denote the information obtained from the SCA of the decryption of the k-th ciphertext $(\mathbf{u}, \mathbf{v})$. Let $\texttt{flip\_candidates}^k$ denote the vector computed with Equation 1, using $\mathbf{c}^{\text{RMRS}} \leftarrow \text{Encode}\left(\mathbf{m}^k\right)$ and $\mathbf{w}^k$. Then we compute the likelihood vectors $\texttt{likelihoods}_\mathbf{x}$ as

$$\texttt{likelihoods}_\mathbf{x}[j] = \sum_{k=1}^{\eta} \left|\text{supp}\left(\texttt{flip\_candidates}^k\right) \cap \text{supp}\left(\mathbf{r}_2^k \gg i\right)\right|.$$

Analogously, the likelihoods vector $\texttt{likelihoods}_\mathbf{y}$ is computed using $\mathbf{r}_1$ instead of $\mathbf{r}_2$. It is interesting to notice that the likelihood vectors used for the non-shuffled implementation can be seen as a particularization of the ones above when the shuffling block has length 1, and not n_2.

Key Recovery. Now, with the likelihood vectors in hand, we can use them together with Algorithm 2 to try to solve the key equation. Figure 11 shows the number of decryption SCA traces needed for a successful attack against 128-bit HQC parameters. To simulate SCA errors when classifying between 0 and 1, we use the bit-error probability ε. Since the observed SCA bit-error rate was around 29%, we consider that the attack is successful with just 681 traces on average for full key recovery.

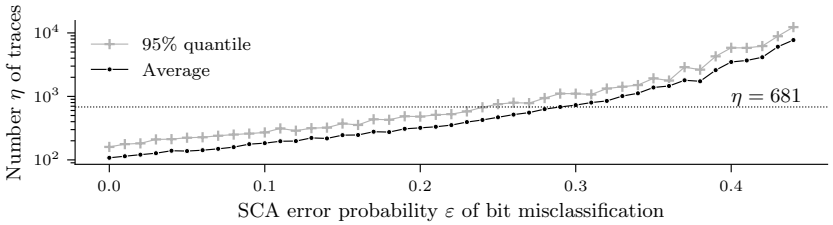


Figure 11. The number of decryption SCA traces needed to recover the key, when ExpandAndSum is implemented with the intra-codeword shuffling countermeasure, considering 128-bit security parameters.

Extending Key Recovery to Inter-codeword Shuffling. Our key recovery algorithm can easily be extended to work with the Hamming weight of segments of $\mathbf{c}'$ of arbitrary length. The main step for this generalization is to replace n_2 in Equation 1 with the lengths of the blocks whose approximate Hamming weights can be obtained through SCA, which depends on how granular is the implemented shuffling. Remarkably, this allows us to recover the key even when shuffling is done by mixing RM blocks among themselves. In this case, SCA would give us the Hamming weight of the full $\mathbf{c}'$, except for the last $n - n_1 n_2$ bits. Assuming perfect SCA information (i.e. $\varepsilon = 0$), our key recovery algorithm requires 17,500 ciphertexts on average. However, considering our empirical bit error rate of $\varepsilon = 29\%$, we were able to fully recover the key with 64,200 traces.

6.2 Attacking Shuffled FindPeaks

For the shuffled FindPeaks, we generate a random permutation for the main loop of FindPeaks using the Fisher-Yates algorithm, which is equivalent to say that the input `transform` is given in its shuffled form `shuff(transform)` (refer to C code of unprotected FindPeaks in Fig. 7). Similar to the shuffling assumption used in the previous section for ExpandAndSum, all other operations remain the same, and we stress that this corresponds to the most randomized assumption possible within FindPeaks. Since the leakage in the shuffled FindPeaks operation is the same as that of the unprotected implementation, the attacker can use leakage from a clone device to build templates for the leaky `mask` variable in the same manner as an unprotected implementation. From our practical experiments on the shuffled FindPeaks, we were able to recover the `mask` variable with 99.99% success rate.

The attacker's main technique for key recovery is to recover the number p of times that the temporary peak changes when FindPeaks is processing the input `shuff(transform)`. While the number p clearly depends on the actual permutation of `transform` obtained through shuffling, we show that, on average, p is higher when the number of unique values in `transform` is larger. Now, the number of unique values is invariant with respect to shuffling, and we demonstrate a novel technique to exploit this information to recover the secret key. We remark that this is the first time this type of correlation is used for attacking shuffled implementations.

6.2.1 Key Recovery Attack

Using SCA on the shuffled FindPeaks, the attacker can recover the number of times the temporary peak has changed when processing `shuff(transform)`. Surprisingly, we now show that this is enough to find the key, although a high number of traces is needed.

The Likelihoods Vectors. For each full decryption, the SCA information can be represented by a vector $\mathbf{p} \in \mathbb{Z}^{n_1}$. Each entry $\mathbf{p}[j]$ is the number of times there is a change of value in the temporary peak when iterating over `shuff(transform)` array of the j-th Reed-Muller corrupted codeword. Formally, we can write

$$\mathbf{p}[j] = |\{\ell : |\texttt{shuff(transform)}[\ell]| > |\texttt{shuff(transform)}[i]|\,,\ \text{for all } 0 \leq i < \ell < 128\}|\,.$$

Now let us consider what affects $\mathbf{p}[j]$ for a fixed n_2-bit block j. In particular, we can point two conditions for the number of peak changes $\mathbf{p}[j]$ to be high.

1. When the peak is located in the end of array `shuff(transform)`, so there is more opportunity for the current peak variable to be changed during the computation.

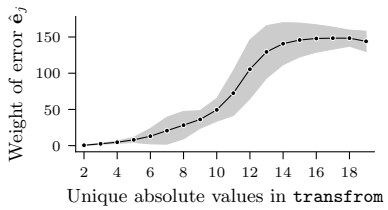
2. When there is a large number of unique entries `shuff(transform)`. This makes it more likely that the variable with the current peak will take different values before finding the peak.

The first condition is rather useless, because the shuffling removes any useful information related the position of the peak. Interestingly, however, the second condition can be exploited, as we explain next.

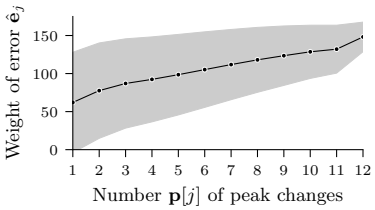
Remember that the `transform` vector, which is the output of the Hadamard transform, contains information on the distance from the corrupted RM codeword and each possible valid codeword. Intuitively, then, when the noise is high, the number of different values in the entries of `transform` should also be higher. In more detail, when the error block $\hat{\mathbf{e}}_j$ added to the RM codeword $\mathbf{c}_j^{\text{RMRS}}$ has high Hamming weight, there are more possibilities of binary combinations for the Hamming differences from the 2^8 Reed-Muller codewords, resulting in a richer set of possible values in `transform`. The importance of this observation comes from the fact that the number of different values in `transform` is invariant when the vector is shuffled.

Figure 12a illustrates this property showing that the number of unique values in `shuff(transform)` is indeed positively correlated with the weight of the error block $\hat{\mathbf{e}}_j$. Now, if we want to use the values of $\mathbf{p}$ to attack the shuffled implementation of FindPeaks we use the following information flow. The value of $\mathbf{p}[j]$ gives some information on the number of unique values in `shuff(transform)` associated with block j, which in turn is related to the weight of the j-th block of n_2 bits in the error $\hat{\mathbf{e}}$. Figure 12b demonstrates that this correlation holds. In fact, we can see that larger values of $\mathbf{p}[j]$, on average, correspond to higher error weights in $\hat{\mathbf{e}}_j$. However, the standard deviation, which is shown as the grey error bands, is relatively large. This means that, if we want to use this information for key recovery, we may need a large number of traces to get rid of the noise.

Now, we just concluded that the values in $\mathbf{p}$ are higher when the weight of the error $\hat{\mathbf{e}}$ in each corresponding block is also higher. But this is exactly the same property observed for vector $\mathbf{d}$, which represents the SCA information obtained for the non-shuffled FindPeaks. This means that we can build vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, using $\mathbf{p}$ instead of $\mathbf{d}$, just as described in Section 5.2.2. For these newly computed vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, we observe the same properties illustrated in Figure 9, although with much more noise when the same number η of traces is used. Finally, we can also compute likelihood vectors $\texttt{likelihoods}_{\mathbf{x}}$ and $\texttt{likelihoods}_{\mathbf{y}}$ with our algorithm based on edge-detection, as used in the non-shuffled attack, applied to $\mathbf{w}_2$ and $\mathbf{w}_1$, respectively.



(a) Correlation between error weight and the number of unique absolute values in `shuff(transform)`.



(b) The correlation between $\mathbf{p}[j]$ and the weight of the j-th error block $\hat{\mathbf{e}}_j$ of n_2 bits.

Figure 12. The correlations observed that allow key recovery, considering parameters for 128 bits of security. The grey error bands illustrate the standard deviation.

Key Recovery. Figure 13 shows the attack performance against the 128-bit security parameters of HQC, when FindPeaks is implemented with shuffling. We can see that the number of traces needed for the attack is about 100× larger than what is needed to attack the non-shuffled implementation of FindPeaks. Furthermore, we can see that even a very small probability ε of wrongly detecting a peak change with SCA greatly impacts the recovery performance. Luckily, however, our side-channel analysis of the shuffled implementation of FindPeaks achieves 100% accuracy, and therefore the attack should be possible with about 21 million challenges, on average, as shown in the figure.

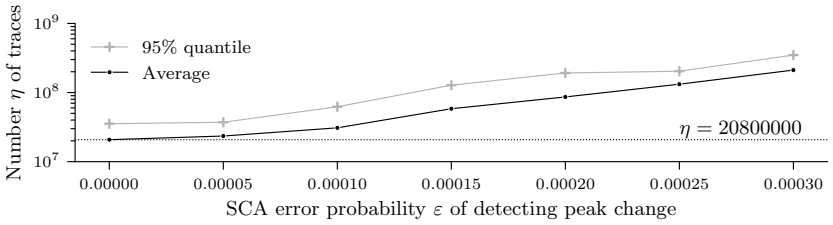


Figure 13. The number of SCA traces needed to recover the key on Shuffled FindPeaks operation for 128-bit security parameters.

7 Countermeasures and Concluding Remarks

We have clearly demonstrated that decapsulation failure check countermeasure does not provide protection, as leakages from the ExpandAndSum and FindPeaks operation can be efficiently exploited for key recovery using only valid ciphertexts. Our attacks exploit leakages from the ExpandAndSum and FindPeaks operations for efficient key recovery, and we have also proposed novel key recovery algorithms that enable 100% key recovery, even in the presence of significant noise in the side-channel measurements. Moreover, our results show that shuffling has no effect in hindering the attack, when targeting ExpandAndSum, allowing key recovery with less than 4000 traces. On the other hand shuffling does help FindPeaks operation, however, key recovery is still possible with 100× more traces. This makes it important to consider masking strategies for ExpandAndSum and FindPeaks, which may require a redesign of the decoding procedure. It is possible that masking and shuffling together might serve as a strong countermeasure against our proposed attacks, but an extensive study of the same is left for future work. Thus, our work highlights urgent need for research on efficient masking schemes for HQC, which currently is one of the few NIST candidates without a masked implementation.

References

[ABB+21] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Carlos Aguilar-Melchor, Rafael Misoczki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit flipping key encapsulation, 2021. https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf.

[ABD+20] Erdem Alkim, Joppe W. Bos, Leo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM learning with errors key encapsulation: Algorithm specifications and supporting documentation (September 30, 2020). *Submission to the NIST post-quantum project*, 2020.

[AH21] Daniel Apon and James Howe. Attacks on NIST PQC 3rd Round Candidates, 2021. Invited talk at Real World Crypto 2021, https://iacr.org/submit/files/slides/2021/rwc/rwc2021/22/slides.pdf.

[BCL+19] Daniel J Bernstein, Tung Chou, Tanja Lange, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Peter Schwabe, Jakub Szefer, and Wen Wang. Classic McEliece: conservative code-based cryptography. 2019.

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):334–359, Jul. 2021.

[BGR+21] Joppe W Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First-and higher-order implementations. *IACR Cryptol. ePrint Arch.*, 2021:483, 2021.

[Gal62] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 223–263, 2022.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, volume 7, pages 115–136, 2011.

[GLG22] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on HQC with chosen ciphertext. In *Post-Quantum Cryptography: 13th International Workshop, PQCrypto 2022, Virtual Event, September 28–30, 2022, Proceedings*, pages 353–371. Springer, 2022.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 88–113, 2021.

[HPR+22] Anna-Lena Horlemann, Sven Puchinger, Julian Renner, Thomas Schamberger, and Antonia Wachter-Zeh. Information-set decoding with hints. In *Code-Based Cryptography: 9th International Workshop, CBCrypto 2021 Munich, Germany, June 21–22, 2021 Revised Selected Papers*, pages 60–83. Springer, 2022.

[HSC+23] Senyang Huang, Rui Qi Sim, Chitchanok Chuengsatiansup, Qian Guo, and Thomas Johansson. Cache-timing attack against hqc. *Cryptology ePrint Archive*, 2023.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. mupq/pqm4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[MAB+21] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérôme Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. Hamming Quasi-Cyclic: HQC, 2021. https://pqc-hqc.org/doc/hqc-specification_2021-06-06.pdf.

[MS77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error correcting codes*, volume 16. Elsevier, 1977.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure Saber KEM. *IACR Cryptol. ePrint Arch.*, 2021:079, 2021.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 142–174, 2018.

[PT20] Thales Bandiera Paiva and Routo Terada. A timing attack on the HQC encryption scheme. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, pages 551–573, Cham, 2020. Springer International Publishing.

[RBRC20] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) NIST PQC candidates for practical message recovery and key recovery attacks. *IACR Cryptol. ePrint Arch.*, 2020:1559, 2020.

[RCDB22] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. *Cryptology ePrint Archive*, 2022.

[RR21] Prasanna Ravi and Sujoy Sinha Roy. Side-channel analysis of lattice-based PQC candidates. *Round 3 Seminars, NIST Post Quantum Cryptography*, 2021.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):307–335, 2020.

[RRD+23] Gokulnath Rajendran, Prasanna Ravi, Jan-Pieter D'anvers, Shivam Bhasin, and Anupam Chattopadhyay. Pushing the limits of generic side-channel attacks on lwe-based kems-parallel pc oracle attacks on kyber kem and beyond. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[SHR+22] Thomas Schamberger, Lukas Holzbaur, Julian Renner, Antonia Wachter-Zeh, and Georg Sigl. A power side-channel attack on the Reed-Muller Reed-Solomon version of the HQC cryptosystem. In *Post-Quantum Cryptography: 13th International Workshop, PQCrypto 2022, Virtual Event, September 28–30, 2022, Proceedings*, pages 327–352. Springer, 2022.

[SMS19] Thomas Schamberger, Oliver Mischke, and Johanna Sepulveda. Practical evaluation of masking for NTRUEncrypt on ARM Cortex-M4. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2019.

[SRSWZ21] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the CCA2-secure HQC KEM. In *Smart Card Research and Advanced Applications: 19th International Conference, CARDIS 2020, Virtual Event, November 18–19, 2020, Revised Selected Papers 19*, pages 119–134. Springer, 2021.

[XPRO20] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David Oswald. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of Kyber. *IACR Cryptol. ePrint Arch.*, 2020:912, 2020.