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

A Case Study 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, we notice that all of these attacks use malformed ciphertexts, which can be easily detected since they cause a decapsulation failure. In this case, designers may chose as a countermeasure to refresh the key whenever a failure occurs, making these previous attacks ineffective. In this work, we present the first side-channel assisted chosen-ciphertext attacks using valid ciphertexts which can be carried out in a stealthy manner for key recovery. Our attacks target side-channel leakage from two different operations within the Reed-Muller decoder used for decryption, and can recover the secret key with 100% success rate, even in the presence of errors in side-channel information. All our experiments are performed on the open-source implementation of HQC KEM taken from the *pqm4* library, with our attacks validated using both the power and EM side-channel. We also demonstrate novel key recovery attacks which also work on shuffled implementations, and discuss applicability of our attack to masking countermeasures. To the best of our knowledge, we are not aware of a side-channel protected design for HQC KEM, and thus we believe our work stresses the need towards more research on secure and efficient masking and hiding countermeasures for HQC KEM.

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

1 Introduction

In 2022, the NIST standardization process for *post-quantum cryptography* has finished its third round [AAC+22]. In this round, one lattice-based key encapsulation mechanism (KEM) and 3 digital signature schemes were selected for standardization, while three code-based KEMs were selected to advance to 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 also clearly stated by NIST at several instances [AH21,RR21]. Of the three code-based schemes at least one is likely to be chosen by NIST for standardization. This makes it important to study their resistance against SCA.

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

In this work, we focus on HQC [MAB+21], which is 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 makes it more efficient than Classic McEliece, but it does not rely on a secret sparse structure as BIKE, making it a slightly more conservative choice.

In its first revision, HQC was subject to timing attacks [PT20, WTBBG19], which mainly exploited leakage from non-constant implementation of the decoding process for the error correction codes used for decryption. After these timing attacks were proposed, the HQC team updated its implementation to use constant-time decoders, but these were targeted side-channel attacks using electromagnetic emanations (EM) and power consumption [SRSWZ21, SHR+22, GLG22]. These attacks can be commonly referred to as side-channel assisted chosen-ciphertext attacks. One of the main characteristics of these attacks is that they rely on querying the decapsulation device with malformed ciphertexts, and utilizing leakage from targeted operations in the decapsulation procedure for key recovery. One of the main downsides of these attacks is that the malformed ciphertexts used for the attack, can be detected with a very high probability. Such chosen-ciphertext attacks using malformed ciphertexts have also been used to target other PQC schemes as well [RR21]. This makes it natural for a designer to implement a simple protection: refresh the secret key every time he/she observes a decapsulation failure, since decapsulation failures for valid ciphertexts occur negligible probability.

This raises a natural question on whether "it is possible to perform chosen-ciphertext attacks on HQC with valid ciphertexts?". This represents a more stealthy approach towards chosen-ciphertext attacks, as valid ciphertexts cannot be detected as malicious by the decapsulation procedure, as they do not trigger decapsulation failure. In this work, we answer this question positively by demonstrating the first side-channel assisted chosen-ciphertext attack on HQC KEM with valid ciphertexts.

Our Contribution

The contribution of this work is manifold.

1. Our chosen-ciphertext attacks rely on leakage from two types of operations in the Reed-Muller decoder - ExpandAndSum and FindPeaks, which have not been targeted any previous side-channel attacks on HQC KEM. We show that it is possible to recover significant information about the decrypted codeword using leakage from these operations for key recovery.

2. We propose novel key recovery attacks, capable of exploiting leakage from the aforementioned operations to recover the key with 100% success rate. Remarkably, we show that our key recovery attacks are robust to noise in the side-channel measurements, and that full key recovery is possible even with errors in the recovered side-channel information.

3. We perform experimental validation of our attacks on the open-source implementation of HQC KEM taken from the *pqm4* library, with our attacks validated using both the power and EM side-channel.

4. We also demonstrate novel key recovery attacks which work on shuffled implementations of the ExpandAndSum and FindPeaks operation, thereby demonstrating that shuffling increases the attacker's complexity for key recovery, but does not concretely prevent the attack. Similarly, we also discuss the applicability of our attack to masking countermeasures.

5. To the best of our knowledge, we are not aware of a side-channel protected design for HQC KEM, and thus we believe our work stresses the need towards more research on secure and efficient masking and hiding countermeasures for HQC KEM.

2 Background

2.1 Notation

Vectors and matrices are denoted by bold lowercase and uppercase letters, respectively, such as $\mathbf{a}$ and $\mathbf{A}$. We use zero-based indexing, and if $\mathbf{a}$ is a vector of length n, then $\mathbf{a}[i]$ denotes its i-th entry, for $i = 0, \ldots, n-1$. We sometimes abuse the notation $\mathbf{a}[i]$ to represent the $(i \bmod n)$-th entry of $\mathbf{a}$, for $i \in \mathbb{Z}$. 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 cyclic a vector, which are called slices. A k-bit slice of $\mathbf{a}$ starting at position i is denoted as $\operatorname{slice}_i^k(\mathbf{a}) = [\mathbf{a}[i], \ldots, \mathbf{a}[i+k-1]]$. We let slices wrap around the end of a vector, therefore $\operatorname{slice}_{n-1}^3(\mathbf{a}) = [\mathbf{a}[n-1], \mathbf{a}[0], \mathbf{a}[1]]$. This, we believe, allows for a more concise notation and somewhat more intuitive descriptions in specific parts of our analysis. We denote the 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. If $\mathcal{C}$ is a binary $[n,k]$-linear code spanned by the rows of a matrix $\mathbf{G}$ of $\mathbb{F}_2^{k\times n}$, we say that $\mathbf{G}$ is a *generator matrix* of $\mathcal{C}$. Similarly, if $\mathcal{C}$ is the kernel of a matrix $\mathbf{H}$ of $\mathbb{F}_2^{r\times n}$, we say that $\mathbf{H}$ is a *parity-check matrix* of $\mathcal{C}$. 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. If $\mathcal{C}$ is a binary $[n,k]$-linear code, and the minimum Hamming distance between two codewords in $\mathcal{C}$ is d, we say that $\mathcal{C}$ is an $[n,k,d]$-linear code. The *support* of a binary vector $\mathbf{v} \in \mathbb{F}_2^n$, denoted as $\operatorname{supp}(\mathbf{v})$, is the set $\operatorname{supp}(\mathbf{v}) = \{i \in \{0, \ldots, n-1\} : \mathbf{v}[i] = 1\}$. We denote by $\mathcal{L}(w)$ the subset of $\mathbb{F}_2^n$ consisting only of elements of weight w.

The *syndrome* $\mathbf{z}$ of a vector $\mathbf{e}$ with respect to a parity check matrix $\mathbf{H}$ is the vector $\mathbf{z} = \mathbf{e}\mathbf{H}^\top$. If the vector $\mathbf{e}$ is sufficiently sparse and the linear code defined by $\mathbf{H}$ is sufficiently good, it may be possible to recover $\mathbf{e}$ from the syndrome $\mathbf{z}$ by using efficient decoding algorithms.

The *circulant matrix* defined by a vector $\mathbf{v} = [v_0, \ldots, v_{n-1}]$, is the matrix

$$\operatorname{rot}(\mathbf{v}) = \begin{bmatrix} v_0 & v_{n-1} & \cdots & v_1 \\ v_1 & v_0 & \cdots & v_2 \\ \vdots & \vdots & \ddots & \vdots \\ v_{n-1} & v_{n-2} & \cdots & v_0 \end{bmatrix}.$$

It is well known that binary circulant matrices of dimension n form a ring that is isomorphic to the polynomial ring $\mathbb{F}_2[X]/(X^n-1)$. Since each circulant matrix is defined by one vector, we can identify polynomials in $\mathbb{F}_2[X]/(X^n-1)$ and vectors in $\mathbb{F}_2^n$. This motivates the definition of the *product of two vectors* $\mathbf{u}$ and $\mathbf{v}$ of $\mathbb{F}_2^n$ as

$$\mathbf{u}\cdot\mathbf{v} = \mathbf{u}\operatorname{rot}(\mathbf{v})^T = \left(\operatorname{rot}(\mathbf{v})\,\mathbf{u}^T\right)^T = \mathbf{v}\operatorname{rot}(\mathbf{u})^T = \mathbf{v}\cdot\mathbf{u}.$$

Notice that the transpose of the circulant matrix generated by $\mathbf{v}$ consists of a sequence of cyclic shifts of $\mathbf{v}$, and therefore the product of two vectors can also be written as

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

In this paper, we prefer the binary linear algebra interpretation of vectors instead of polynomials, as most of the attacks are described in this way.

2.2 HQC

In this section, we present the parameters and algorithms used by HQC, together with its security.

2.2.1 Parameters and Algorithms

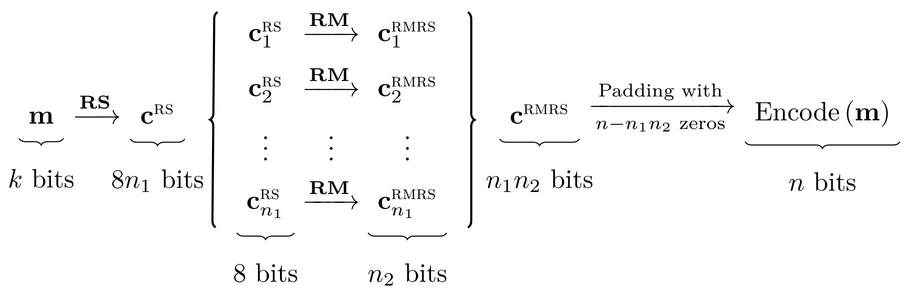
HQC provides its parameter sets defining $(n, k, \delta, w, w_{\mathbf{r}}, w_{\mathbf{e}})$ for each security level, as shown in Table 1. Parameters n and k define the public error correction code $\mathcal{C}$ used by HQC, while parameters $w, w_{\mathbf{r}}$ and $w_{\mathbf{e}}$ correspond to the weights of the sparse vectors defined and used in the next sections. 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 1: 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, the scheme uses a public binary $[n, k]$-linear code $\mathcal{C}$ which consists of a concatenated code between two codes with good error correction properties: 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, while the Reed-Solomon is a non-binary linear code. Since, in this work, the details of the Reed-Solomon code are not particularly important we can think of the Reed-Solomon as an encoding algorithm that, given k bits, returns $8n_1$ bits corresponding to representation of the original k bits with redundancy.

Therefore, code $\mathcal{C}$ comes with an efficient pair of algorithms for encoding and decoding. Intuitively, if $\mathbf{m} \in \mathbb{F}_2^k$, then $\text{Encode}(m)$ is responsible to add redundancy to it. If $\mathbf{c} = \text{Encode}(\mathbf{m}) + \mathbf{e}$ for some error vector $\mathbf{e} \in \mathbb{F}_2^n$, then $\text{Decode}(\mathbf{c})$ 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.



$\underbrace{\mathbf{m}}_{k \text{ bits}} \xrightarrow{\mathbf{RS}} \underbrace{\mathbf{c}^{\text{RS}}}_{8n_1 \text{ bits}}$
$\mathbf{c}_1^{\text{RS}} \xrightarrow{\mathbf{RM}} \mathbf{c}_1^{\text{RMRS}}$
$\mathbf{c}_2^{\text{RS}} \xrightarrow{\mathbf{RM}} \mathbf{c}_2^{\text{RMRS}}$
$\underbrace{\mathbf{c}_{n_1}^{\text{RS}}}_{8 \text{ bits}} \xrightarrow{\mathbf{RM}} \underbrace{\mathbf{c}_{n_1}^{\text{RMRS}}}_{n_2 \text{ bits}}$
$\underbrace{\mathbf{c}^{\text{RMRS}}}_{n_1 n_2 \text{ bits}} \xrightarrow[n-n_1n_2 \text{ zeros}]{\text{Padding with}} \underbrace{\text{Encode}(\mathbf{m})}_{n \text{ bits}}$

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

At its core, HQC uses a chosen-plaintext secure PKE scheme (IND-CPA), consisting of three procedures: Key Generation (CPAPKE.KeyGen), Encryption (CPAPKE.Encrypt), Decryption (CPAPKE.Decrypt). These are reviewed next.

IND-CPA algorithms used by HQC. The three procedures of the IND-CPA secure HQC PKE scheme are described as Algorithm 1. While the key generation and encryption are more directly understood, let us see why decryption works. Notice that vector $\mathbf{c}' = \mathbf{v} - \mathbf{u} \cdot \mathbf{y}$, as computed in the decryption algorithm is equal to

$$\begin{aligned}\mathbf{c}' &= \text{Encode}(\mathbf{m}) + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e} - (\mathbf{r}_1 + \mathbf{r}_2 \cdot \mathbf{h}) \cdot \mathbf{y} \\ &= \text{Encode}(\mathbf{m}) + (\mathbf{x} + \mathbf{y} \cdot \mathbf{h}) \cdot \mathbf{r}_2 + \mathbf{e} - (\mathbf{r}_1 + \mathbf{r}_2 \cdot \mathbf{h}) \cdot \mathbf{y} \\ &= \text{Encode}(\mathbf{m}) + \mathbf{x} \cdot \mathbf{r}_2 - \mathbf{r}_1 \cdot \mathbf{y} + \mathbf{e}.\end{aligned}$$

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 $\mathrm{w}(\mathbf{e}')$ is sufficiently low for it to be corrected out of $\mathbf{c}'$ with overwhelming probability, using the decoder $\mathcal{C}$.Decode.

This decoder is for the concatenated code $\mathcal{C}$, that is, first the internal Reed-Muller decoder followed by the external Reed-Solomon decoder to correct the errors in $\mathbf{c}'$ and yield the same message $\mathbf{m}' = \mathbf{m}$. In general, schemes based on syndrome decoding have to take care to avoid generic attacks based on Information Set Decoding [Pra62, Ste88, TS16]. Furthermore, the quasi-cyclic structure of the code used to secure the secret key can make the scheme vulnerable to DOOM [Sen11], or other structural attacks [GJL15, LJS+16]. To instantiate the scheme, the authors propose parameters for which they prove very low decryption error probability and resistance to the attacks mentioned.

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

```
1: procedure CPAPKE.KeyGen
2:     $\mathbf{h} \leftarrow \mathbb{F}_2^n$
3:     $(\mathbf{x}, \mathbf{y}) \leftarrow (\mathcal{L}(w))^2$
4:     $\mathbf{s} \leftarrow \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$
5:     Return $\mathsf{pk} \leftarrow (\mathbf{h}, \mathbf{s}), \mathsf{sk} \leftarrow (\mathbf{x}, \mathbf{y})$
6: end procedure
------------------------------------------------------------
7: procedure CPAPKE.Encrypt($\mathsf{pk}, \mathbf{m} \in \mathcal{B}^*, \mathsf{seed} \in \mathcal{B}^*$)
8:     $(\mathbf{e}, \mathbf{r}_1, \mathbf{r}_2) \leftarrow$ Pseudorandom sample from $(\mathcal{L}(w_\mathbf{e}) \times \mathcal{L}(w_\mathbf{r}) \times \mathcal{L}(w_\mathbf{r}))$ using seed
9:     $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
10:    $\mathbf{v} \leftarrow \mathcal{C}.\text{Encode}(\mathbf{m}) + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$
11:    Return $\mathsf{ct} = (\mathbf{u}, \mathbf{v})$
12: end procedure
------------------------------------------------------------
13: procedure CPAPKE.Decrypt(ct, sk)
14:    $\mathbf{c}' = \mathbf{v} - \mathbf{u} \cdot \mathbf{y}$
15:    $\mathbf{m}' = \mathcal{C}.\text{Decode}(\mathbf{c}')$
16:    Return $\mathbf{m}'$
17: end procedure
```

2.2.2 CCA Transformation

The aforementioned PKE scheme is only secure in the IND-CPA security model, but is insecure in the IND-CCA security model, and therefore susceptible to chosen-ciphertext attacks. Thus, the well-known Hofheinz-Hövelmanns-Kiltz (HHK) transformation [HHK17] is used to turn the IND-CPA secure PKE into an IND-CCA2 secure KEM.

Refer to Alg. 2 for the encapsulation and decapuslation procedures of IND-CCA secure HQC KEM. The core idea of this transformation is to make the encryption procedure be deterministic, such that the generated ciphertext ct (Lines 3 to 4 in CCA.Encaps) depends entirely upon the message $\mathbf{m}$. This enables the decapsulation procedure to encrypt the message $\mathbf{m}$ (Line 12) and recompute the ciphertext ct'. Then, the received ciphertext ct is compared with ct' (Line 14), and the comparison succeeds for a valid ciphertext with overwhelming probability, to generate a valid session key (Line 16). However, the comparison fails with an overwhelming probability for an invalid ciphertext, which typically means a malicious party manipulated the ciphertext, and this results in aborting the decapsulation procedure (Line 18).

Algorithm 2 IND-CCA2 secure HQC KEM (Using HHK Transform [HHK17])

```
1:  procedure CCAKEM.Encaps($\mathsf{pk} = (\mathbf{h}, \mathbf{s})$)
2:      $\mathbf{m} \leftarrow \mathbb{F}_2^k$
3:      $\mathsf{seed} \leftarrow \mathcal{G}(\mathbf{m})$
4:      $\mathsf{ct} = \mathsf{CPAPKE.Encrypt}(\mathbf{m}, \mathsf{pk}, \mathsf{seed})$        ▷ Deterministic Encryption
5:      $K = \mathcal{K}(\mathbf{m}, \mathsf{ct})$
6:      $\mathbf{d} = \mathcal{H}(\mathbf{m})$
7:      Return $(\mathsf{ct}, \mathbf{d})$
8:  end procedure
9:  procedure CCA.Decaps($\mathsf{sk}, \mathsf{pk}, \mathsf{ct}$)
10:     $\mathbf{m}' = \mathsf{CPAPKE.Decrypt}(\mathsf{ct}, \mathsf{sk})$
11:     $\mathsf{seed}' = \mathcal{G}(\mathbf{m}')$
12:     $\mathsf{ct}' = \mathsf{CPAPKE.Encrypt}(\mathbf{m}', \mathsf{pk}, \mathsf{seed}')$
13:     $\mathbf{d}' = \mathcal{H}(\mathbf{m}')$
14:     if $\mathsf{ct}' = \mathsf{ct}$ and $\mathbf{d}' = \mathbf{d}$ then
15:         $K = \mathcal{K}(\mathbf{m}, \mathsf{ct})$
16:         Return $K$
17:     else
18:         Return $\perp$
19:     end if
20: end procedure
```

2.3 Prior SCA on HQC and Motivation

HQC KEM has been subjected to a number of side-channel attacks mainly targeting the decapsulation procedure to recover the long-term secret key. They can be broadly split into two categories: (1) Timing Attacks and (2) Power/EM Side-Channel Attacks. Timing attacks on HQC KEM have exploited the inherent non-constant time behavior of error correcting codes or the rejection sampling procedure during re-encryption for key recovery. The first timing attack on the CCA secure decapsulation procedure was proposed in [PT20], which worked by querying the decapsulation device with valid ciphertexts, and subsequently utilizing the correlation between the weight of the decoded error and the computation time of the decoder for full key recovery. Subsequently, [GHJ+22] proposed novel timing attacks exploiting non-constant time rejection sampling procedures used in the re-encryption operation after decryption. They query the decapsulation device with malformed/invalid ciphertexts and utilize timing information from the rejection sampling procedure to obtain information about the decrypted message, resulting in full key recovery. However, the attacks can be trivially prevented by using constant time implementation of the decoding procedure.

With respect to the power/EM side-channel attacks, Schamberger *et al.* [SRSWZ21] presented the first power side-channel attack on the original version of HQC KEM based on the BCH code. It also utilizes malformed/invalid ciphertexts to query the decapsulation oracle, and subsequently utilizes leakage from the BCH decoder to recover the corresponding codewords, resulting in full key recovery. Subsequently, the same authors adapted their attack to the updated version of HQC KEM, based on the concatenated Reed-Muller and Reed-Solomon code, by exploiting leakage from the Reed-Solomon decoder for key recovery [SHR+22]. Goy *et al.* [GLG22] presented a simplified version of this attack, where they demonstrated that leakage exploited from the Hadamard transform operation for malformed/invalid ciphertexts can also result in key recovery attacks, potentially with a simpler technique to build chosen-ciphertexts.

We observed that all power/EM-based key recovery attacks on HQC KEM, rely on utilization of malformed ciphertexts which can be easily detected by the decapsulation procedure as invalid ones. Moreover, all these attacks require a few thousand chosen-ciphertext queries for full key recovery. Given the almost negligible decapsulation failure rate for valid ciphertexts, observing a few thousand decapsulation failures will definitely raise suspicion. Moreover, it is easy for a designer to implement a countermeasure that simply refreshes the secret key upon a decapsulation failure, as a precaution. In such a scenario, all the existing side-channel attacks cannot be used for key recovery. In this work, we demonstrate the first power/EM side-channel assisted chosen-ciphertext attack that can be done using valid ciphertexts. We demonstrate that our novel key recovery attacks can work even in the presence of errors in the side-channel information, and are also applicable to implementations protected with shuffling and masking countermeasures.

2.4 Test Vector Leakage Assessment (TVLA) [GJJR11]

TVLA is a popular conformance-based evaluation methodology widely used to perform side-channel evaluation of cryptographic implementations. It involves computation of the univariate Welch's t-test over two sets of side-channel measurements to identify differentiating features. The TVLA formulation over two datasets $\mathcal{T}_r$ and $\mathcal{T}_f$ is given by:

$$\mathsf{TVLA} = \frac{\mu_r - \mu_f}{\sqrt{\frac{\sigma_r^2}{m_r} + \frac{\sigma_f^2}{m_f}}} \ , \tag{1}$$

where μ_r, σ_r and m_r (resp. μ_f, σ_f and m_f) are univariate mean, standard deviation and cardinality of the trace set $\mathcal{T}_r$ (resp. $\mathcal{T}_f$). The null hypothesis (two means are equal) is rejected with a confidence of 99.9999% when the absolute value of the t-test score is greater than 4.5 [GJJR11]. A rejected hypothesis implies that the two sets are different and hence could leak some side-channel information.

3 Our Approach to Key Recovery

We observe that the Reed-Muller decoder operates upon the erroneous codeword $\mathbf{c}'$, as it serves as the external code in the concatenated code of HQC (Line 15 in the CPAPKE.Decrypt) procedure). As we show later, the codeword $\mathbf{c}'$ is a sensitive variable, as the error component $\mathbf{e}'$ that masks the correct codeword $\mathbf{c}$ within $\mathbf{c}'$, is dependent upon the secret key components $\mathbf{x}$ and $\mathbf{y}$ (Refer to description of IND-CPA secure CPAPKE.Decrypt procedure of HQC KEM). We demonstrate that leakage from the Reed-Muller decoder during decapsulation of valid ciphertexts can be exploited to recover incremental information about the secret key, resulting in full key recovery attacks. Thus, operations within the Reed-Muller decoder serve as the primary target for the side-channel attacks presented in this work.

In the following, we first discuss our approach to key recovery from side-channel information, obtained from the Reed-Muller decoder. We start by discussing the types of oracles that are used to model the side-channel information. We then we define our basic approach for key recovery using partial information, which consists of a simple linear algebra method. We finish by providing an abstract overview on what we call likelihood functions, which are used to extract secret information from the oracles' outputs and are a central tool to allow the key recovery in all of our attacks.

3.1 The Oracles

To model the information we can get from side-channel analysis of the steps of the Reed-Muller decoding operation, we define a number of oracles throughout the paper. Since we are interested in attacks using valid ciphertexts, we let each oracle $\mathcal{O}$ output, together with the side-channel information `sca_information` on a targeted step of the Reed-Muller decoding, the message $\mathbf{m}$ and the random vectors $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{e}$ used to encrypt the message under a public key pk. A little more formally, each oracle $\mathcal{O}$ takes as input a public key pk, and returns a freshly generated tuple

$$(\texttt{sca\_information}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e}) \leftarrow \mathcal{O}(\mathsf{pk}).$$

Notice that this notation makes it explicit that, in this paper, we do not require the attacker to choose $\mathbf{m}$, $\mathbf{r}_1$, $\mathbf{r}_2$, nor $\mathbf{e}$: they only need to know these values from an honest encryption, together with the corresponding `sca_information`, to perform the attack.

The quality of the information provided by side-channel analysis is dependent on both the hardware and the setup for the attack. Furthermore it may not be possible to perfectly recover the side-channel information in some setups. To account for possible noise in the setup, we also define, for each oracle $\mathcal{O}$, a noisy oracle $\mathcal{O}^{\varepsilon}$. A noisy oracle $\mathcal{O}^{\varepsilon}$, calls its perfect parent oracle $\mathcal{O}$ and adds some noise to `sca_information` before returning the tuple, while parameter ε controls the amount of noise. The way in which noise is added to `sca_information` depends on a reasonable noise modeling for each side-channel implementation, and therefore they are explained in each section.

3.2 Solving The Key Equation with Partial Information

In HQC, the secret key $(\mathbf{x}, \mathbf{y})$ is related to the public key $(\mathbf{h}, \mathbf{s})$ by the following binary linear equation

$$\mathbf{y} \cdot \mathbf{h} = \mathbf{y} \operatorname{rot}(\mathbf{h})^{\top} = \mathbf{s} + \mathbf{x}.$$

Notice that, if we know either $\mathbf{y}$ or $\mathbf{x}$, we can easily recover the other. The problem we face, when attacking HQC, is that $\mathbf{x}$ and $\mathbf{y}$ are not directly recoverable purely from the side-channel observations. Therefore we must build algorithms that allow us to solve the key equation from information on both $\mathbf{x}$ and $\mathbf{y}$.

Suppose that, using side-channel analysis together with some additional algorithms, we are able to build two sets of indexes $U_{\mathbf{y}}$ and $E_{\mathbf{x}}$ as described next. Let $U_{\mathbf{y}}$ be some set of indexes of $\mathbf{y}$ that is known to contain each index of non-null entries in $\mathbf{y}$, that is, $\operatorname{supp}(\mathbf{y}) \subset U_{\mathbf{y}}$. Let $E_{\mathbf{x}}$ be a set of indexes of $\mathbf{x}$ that contains only indexes of null entries of $\mathbf{x}$, that is, $\operatorname{supp}(\mathbf{x}) \cap E_{\mathbf{x}} = \emptyset$. In the next paragraph, we explain why $U_{\mathbf{y}}$ and $E_{\mathbf{x}}$ can be seen as the sets of *unknowns* and *equations*, respectively.

Notice how, since we know $U_{\mathbf{y}}$, we can reduce the number of variables in $\mathbf{y}$ from n to $|U_{\mathbf{y}}|$. Although we have n equations, this is still not enough to solve the system because, since we do not know $\mathbf{x}$, there is still some uncertainty on $\mathbf{s} + \mathbf{x}$. But $E_{\mathbf{x}}$ tells us a number of indexes that are 0 in $\mathbf{x}$, so if we use only these equations, together with the public vector $\mathbf{s}$, we may be able to solve the unknowns of $\mathbf{y}$.

From the discussion above, we need two conditions for being able to recover the secret key $\mathbf{y}$ using simple linear algebra. First, the number of unknowns should be lower than or equal to the number of equations, that is $|U_{\mathbf{y}}| \leq |E_{\mathbf{x}}|$. Second, the linear subsystem obtained from $\mathbf{y} \operatorname{rot}(\mathbf{h}) = \mathbf{s} + \mathbf{x}$ together with $U_{\mathbf{y}}$ and $E_{\mathbf{x}}$ should have full rank. This second condition is equivalent to say that the matrix $\hat{\mathbf{H}} \in \mathbb{F}_2^{|U_{\mathbf{y}}| \times |E_{\mathbf{x}}|}$ is full rank, where $\hat{\mathbf{H}}$

is constructed by taking, from $\text{rot}(\mathbf{h})^\top$, the rows and columns whose indexes are in $U_\mathbf{y}$ and $E_\mathbf{x}$, respectively.

An alternative to the simple linear algebra approach discussed above would be to combine sets $U_\mathbf{y}$ and $E_\mathbf{x}$ with information-set decoding (ISD) algorithms [Pra62, Ste88, Dum91]. While ISD algorithms are the best algorithms to find the secret key from the key equation when no information is known about the secret key, recently, Horlemann et al. [HPR+22] studied how partial information on the key improves the performance of ISD algorithms.

In some of the attacks we show, the attacker needs to try multiple attack parameters, and, for each of them, try to recover the key. In the simple linear algebra approach, the attacker either can recover the key, with the information obtained, or not, while for the ISD with hints algorithm, there is a variable work factor that depends on how much information the attacker was able to obtain. The problem of using the more powerful ISD with hints approach in our case is that the composition of work factors of the ISD, together with the multiple parameters the attacker needs to test, may result in an inefficient key recovery procedure.

Therefore, when simulating the number of oracle queries required for a key recovery attack, we take the pure linear algebra approach instead of the ISD with hints. The main reason is that this simplifies the analysis, at the cost of increasing the number of queries required for recovering the key. The result is that the estimates on the performance of the attacks presented in this paper are conservative.

In the next section, we describe the role of the likelihood functions, which serve as a bridge from the oracles information and to the construction of sets $U_\mathbf{y}$ and $E_\mathbf{x}$, that are required for the key recovery.

3.3 The Likelihood Vectors

To model the information we get from side-channel analysis, we use the oracles, as was previously mentioned. But this information is usually rather raw, and must be processed for its relation to the secret key to be clearly visible. For example, the side-channel information may consist of whether the Reed-Muller decoder was able to correct all the errors in each block, or be the weight of some part of the noisy codeword $\mathbf{c}'$.

The core of our key recovery procedure is building functions that take a number of independent oracle answers and output two vectors $\texttt{likelihoods}_\mathbf{x}$ and $\texttt{likelihoods}_\mathbf{y}$, that belong to $\mathbb{R}^n$. Ideally, for a sufficiently high number of oracle queries, it should hold that

- $\texttt{likelihoods}_\mathbf{x}[i] > \texttt{likelihoods}_\mathbf{x}[j]$, whenever $i \in \text{supp}(\mathbf{x})$ and $j \notin \text{supp}(\mathbf{x})$, and
- $\texttt{likelihoods}_\mathbf{y}[i] > \texttt{likelihoods}_\mathbf{y}[j]$, whenever $i \in \text{supp}(\mathbf{y})$ and $j \notin \text{supp}(\mathbf{y})$.

That is, the indexes of the highest likelihoods indicate the non-null entries of the secret key vectors $\mathbf{x}$ and $\mathbf{y}$.

These properties, however, are not always attainable for some of the oracles we use, even for a large number of queries. But remember that we do not need full separation between entries inside and outside the supports of $\mathbf{x}$ and $\mathbf{y}$ to fully recover the key. As we discussed in the previous section, we only need to be able to build a sufficiently large set of equations $E_\mathbf{x}$ and together with a small enough set of unknowns $U_\mathbf{y}$.

If we want to use only the likelihoods vector to define sets $E_\mathbf{x}$ and $U_\mathbf{y}$, then the maximum size of a valid $E_\mathbf{x}$, such that $E_\mathbf{x} \cap \text{supp}(\mathbf{x}) = \emptyset$, is given by

$$\max |E_\mathbf{x}| = |\{j : \texttt{likelihoods}[j] < \tau_\mathbf{x}\}|,$$

where $\tau_\mathbf{x} = \min_{i \in \text{supp}(\mathbf{x})} \texttt{likelihoods}[i]$.

Algorithm 3 Solving the key equation using likelihood vectors.

```
1: procedure TryToSolveKeyEquation($\texttt{likelihoods}_{\mathbf{x}}, \texttt{likelihoods}_{\mathbf{y}}$)
2:     for n_unknowns $= \omega$ to $n - \omega$ do
3:         $U_{\mathbf{y}} \leftarrow$ Indexes of the n_unknowns highest entries of $\texttt{likelihoods}_{\mathbf{y}}$
4:         $\hat{\mathbf{H}} \leftarrow$ Take rows from $\mathrm{rot}(\mathbf{h})^{\top}$ whose indexes are in $U_{\mathbf{y}}$
5:         n_equations $\leftarrow$ n_unknowns
6:         while $\hat{\mathbf{H}}$ is not full rank do
7:             $E_{\mathbf{x}} \leftarrow$ Indexes of the n_equations smallest entries of $\texttt{likelihoods}_{\mathbf{x}}$
8:             $\hat{\mathbf{H}} \leftarrow$ Columns from $\hat{\mathbf{H}}$ whose indexes are in $E_{\mathbf{x}}$
9:             n_equations $\leftarrow$ n_equations $+ 1$
10:        end while
11:        $\hat{\mathbf{s}} \leftarrow$ Entries of $\mathbf{s}$ whose indexes are in $E_{\mathbf{x}}$
12:        Solve full rank system $\hat{\mathbf{y}}\hat{\mathbf{H}} = \hat{\mathbf{s}}$
13:        if a solution $\hat{\mathbf{y}}$ exists and has weight $\omega$ then
14:            Reconstruct $\mathbf{y}$ from $\hat{\mathbf{y}}$ and $U_{\mathbf{y}}$
15:            return $\mathbf{y}$
16:        end if
17:    end for
18:    return $\perp$
19: end procedure
```

Similarly, the minimum size of a valid $U_{\mathbf{y}}$, such that $\mathrm{supp}(\mathbf{y}) \subset U_{\mathbf{y}}$, is

$$\min |U_{\mathbf{y}}| = |\{j : \texttt{likelihoods}[j] \geq \tau_{\mathbf{y}}\}|,$$

where $\tau_{\mathbf{y}} = \min_{i \in \mathrm{supp}(\mathbf{y})} \texttt{likelihoods}[i]$.

Now, when $\min |U_{\mathbf{y}}| \leq \max |E_{\mathbf{x}}|$, then key recovery may be possible, if the resulting linear system is full rank. However, remember that the attacker has no way of knowing $\min |U_{\mathbf{y}}|$ and $\max |E_{\mathbf{x}}|$ since they do not know $\mathbf{y}$ and $\mathbf{y}$, and therefore cannot compute $\tau_{\mathbf{y}}$ and $\tau_{\mathbf{x}}$. One possible solution is to iterate over possible values for the number `n_unknowns` of unknowns, and, by setting the number of equations `n_equations` $\geq$ `n_unknowns` just high enough to get a full rank system, we then try to solve the resulting linear system.

Algorithm 3 formalizes this approach. The algorithm iterates over possible values for `n_unknowns` and, for each value, build candidate sets $U_{\mathbf{y}}$ and $E_{\mathbf{x}}$. If the resulting linear system is full rank, then it solves it and returns the key $\mathbf{y}$, if the solution has low weight. If no key $\mathbf{y}$ is found, then it returns an error $\perp$, which indicates that the attacker should ask more queries to the oracle and update the likelihood vectors $\texttt{likelihoods}_{\mathbf{x}}$ and $\texttt{likelihoods}_{\mathbf{y}}$ accordingly, to hopefully get a sufficient separation between entries inside and outside the secret supports.

3.4 Reed-Muller Error Correcting Code

We briefly explain the operation of the repeated Reed-Muller error correcting code, whose side-channel leakage is used for our key recovery attacks. The repeated Reed-Muller code consists of repeating M times the codeword of an original Reed-Muller code with parameters $(r = 1, m = 7)$, which is a binary $[128, 8, 64]$-linear code. The resulting error correction codes are binary $[n_2 = 128M, 8, 64M]$-linear codes, for $M = 3$ or 5, depending on the security parameters from Table 1. In particular, for 128 bits of security, a binary $[384, 8, 192]$-linear code is used, while a binary $[640, 8, 320]$-linear code is used for security levels 192 and 256.

For simplicity, we visualize the duplicated codeword $\mathbf{a}$ with M blocks (i.e.) $\mathbf{a} = (\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{M-1})$ where $\mathbf{a}_i$ denotes the duplicated copies of $\mathbf{a}_0$, for $i \in [1, M-1]$. When encoding under the concatenated code $\mathcal{C}$, there is a total of n_1 Reed-Muller codewords of n_2 bits. In the decryption procedure, the Reed-Muller decoding procedure is applied over each of the n_1 codewords $\mathbf{a}_i \in \mathbb{F}_2^{n_2}$ for $i \in [0, n_1 - 1]$. For each of these n_2-bit blocks, the Reed-Muller decoding procedure consists of the following three distinct steps. We emphasize that, when we say Reed-Muller decoding in HQC, we actually mean the decoding of the corresponding repeated Reed-Muller code.

1. ExpandAndSum: The multiplicity M of the duplicated codeword $\mathbf{a}$ is removed by this operation, wherein the single bits of the copies $\mathbf{a}_i$ for $i \in [0, M-1]$ are simply added to each other, yielding a vector $\mathbf{a}' \in \mathbb{N}^{128}$, where each element $\mathbf{a}'[j]$ for $j \in [0, 127]$ is in the range of $[0, M]$. Refer to the pseudo-code of the ExpandAndSum procedure in Alg.4.

2. Hadamard transform: This operation essentially involves multiplication of $\mathbf{a}' \in \mathbb{N}^{128}$ with the Hadamard matrix. The Hadamard transform is a more efficient way of performing the same computation, which essentially computes the generalized discrete Fourier transform of $\mathbf{a}'$. This operation determines the weight distribution of the cosets, thereby allowing to decode using a maximum likelihood operation, and find the distance between the received message and every codeword. This operation returns a vector denoted as $\mathbf{a}'' \in \mathbb{Z}^{128}$.

3. FindPeaks: This is the final operation in the Reed-Muller decoding procedure, which returns a byte $\mathbf{b} \in \mathbb{F}_2^8$, whose seven (7) least significant bits correspond to the location of the highest absolute value in $\mathbf{a}'' \in \mathbb{Z}^{128}$. The most significant bit is given by the sign of the absolute maximum value. Refer to Alg. 4 for the pseudo-code of the FindPeaks procedure.

Algorithm 4 ExpandAndSum and FindPeaks Operation used within the Reed-Muller Decoder.

```
procedure ExpandAndSum Operation($\mathbf{a} \in \mathbb{F}_2^{(128 \cdot M)}$)
    $\mathbf{a} \leftarrow \mathbf{0} \in \mathbb{N}^{128}$
    for $i = 0$ to $M$ do
        for $j = 0$ to 128 do
            $\mathbf{a}'[j] += \mathbf{a}[128i + j]$
        end for
    end for
    return $\mathbf{a}'$
end procedure
```

```
procedure FindPeaks Operation($\mathbf{a}'' \in \mathbb{Z}^{128}$)
    peak_value, peak_abs_value, peak_pos $= (0, 0, 0)$
    for $i = 0$ to $M$ do
        $(t, t_{abs}) \leftarrow (\mathbf{a}''[i], \mathsf{abs}(\mathbf{a}''[i]))$
        if ($t_{abs} >$ peak_abs_value) then
            peak_value, peak_abs_value, peak_pos $\leftarrow (t, t_{abs}, i)$
        end if
    end for
    if peak_value $> 0$ then
        peak_neg $\leftarrow 1$
    else
        peak_neg $\leftarrow 0$
    end if
    peak_pos $\leftarrow$ peak_pos $+ 128$(peak_neg)
    return peak_pos
end procedure
```

In this work, we exploit leakage from the ExpandAndSum and FindPeaks operations within the Reed-Muller decoding procedure to recover incremental information about the codeword $\mathbf{c}' \in \mathbb{F}_2^n$, for full key recovery. We start by defining the adversary model that we utilize for all attacks mentioned in the paper.

3.5 Adversary Model

The attacker's main motive is to recover the long term secret key used by the target device for the decapsulation procedure of HQC KEM. We assume the following attacker capabilities:

- Physical access to DUT performing decapsulation for power/EM measurements.
- Ability to request the DUT to decapsulate arbitrary number of chosen ciphertexts.
- No knowledge of secret key of the DUT or any innate knowledge of the underlying implementation such as the source or compiled executable.

4 SCA of ExpandAndSum Operation

```c
void expand_and_sum(uint16_t dest[128], uint8_t src[16*M])
{
  size_t seg, bytepos, bitpos;
  /* Extract bits of first segment */
  /* Iterating over 16 bytes of segment */
  for (bytepos = 0; bytepos < 16; bytepos++){
    /* Iterating over 8 bits of the byte */
    for (bitpos = 0; bitpos < 8; bitpos++){
      /* Extracting the target bit */
      /* Storing the bit in dest */
      dest[bytepos*8+bitpos] = ((src[bytepos]>>bitpos)&1);
    }
  }
  /* Accumulating bits of other segments */
  /* Iterating over remaining M-1 segments */
  for (block = 1; block < M; block++){
    /* Iterating over 16 bytes of segment */
    for (bytepos = 0; bytepos < 16; bytepos++){
      /* Iterating over 8 bits of the byte */
      for (bitpos = 0; bitpos < 8; bitpos++){
        /* Extracting the target bit */
        /* Adding bit to entry in dest */
        dest[bytepos*8+bitpos] += ((src[16*block+bytepos]>>bitpos)&1);
      }
    }
  }
}
```

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

Our experiments were carried out on the reference implementation of the HQC scheme from the authors [MAB+21], taken from the open-source *pqm4* library, a well-known benchmarking and testing framework for the ARM Cortex-M4 microcontroller. This is the also same implementation that was submitted to the NIST PQC standardization process by the authors. Moreover, we are not aware of any assembly optimized implementations of HQC for the embedded microcontrollers, and thus, we report all our analysis on the reference implementation of HQC.

Recall that the Reed-Muller decoding procedure is computed over n_1 RM codewords in the total codeword $\mathbf{c}'$ in the decryption procedure (Line 15 in CPAPKE.Decrypt procedure in Alg.1). However, our analysis deals with decoding of one of the n_1 codewords (also denoted as $\mathbf{c}'$), while the same can be extended to all n_1 RM codewords. Refer to Fig.2 for the C code snippet of the ExpandAndSum operation which operates over the input codeword $\mathbf{c}'$ stored in the src array, organized as $16 \cdot M$ bytes. The codeword src can be visualized as containing M blocks, each of length 16 bytes. Thus, the i^{th} block (starting from 0) corresponds to the bytes src$[i \cdot 16 : (i+1) \cdot 16)$ for $i \in [0, M-1]$. The ExpandAndSum operation on src is implemented in two steps in the following manner:

1. Step-1: Firstly, every bit of the first block of src (i.e.) a total of 128 bits from bytes src[0] to src[15] are extracted, and subsequently stored in 128 successive locations of the dest array one bit at a time. This operation is shown between Lines 6-13 in Fig.2. We refer to this operation as First_Block_Extraction operation.

2. Step-2: After extracting the bits of the first segment, bits of the subsequent segments (1 to $M-1$) are extracted one bit at a time, and then added to the previous value in the corresponding location in the dest array. This operation is shown in Lines 16-26 in Fig.2. We refer to this operation as Other_Blocks_Accumulation.

We therefore observe that single bits of the codeword stored in the src array, are manipulated one bit at a time in both the First_Block_Extraction and Other_Blocks_Accumulation operations. For brevity, we refer to it as the Bitwise_Manipulation behaviour throughout this paper. In the following, we demonstrate that this Bitwise_Manipulation vulnerability can be exploited from both the operations to recover the entire src array containing the codeword $\mathbf{c}'$, one bit at a time. We will first present our side-channel analysis of the First_Block_Extraction followed by a similar analysis of the Other_Blocks_Accumulation operation.

4.1 SCA of First_Block_Extraction

We observe that the First_Block_Extraction operation (Lines 6-13 in Fig.2) iterates over the first block of the src array, one byte at a time, starting from src[0] until src[15], using the bytepos variable (Line 10). Then, for each byte, single bits are extracted in order starting from bit 0 until bit 7, using the bitpos variable (Line 13). The bits are sequentially extracted and stored in the index (bytepos $\cdot\, 8 +$ bitpos) of the dest array. This Bitwise_Manipulation behaviour leads to two possible leakage sources (i.e.) the bit extraction as well as the storage of the extracted bit in memory.

4.1.1 Experimental Setup

The implementations of the targeted schemes are taken from the public *pqm4* library [KRSS], a benchmarking and testing framework for PQC schemes on the 32-bit ARM Cortex-M4 microcontroller, which is a NIST recommended optimization target for embedded software implementations. To demonstrate the generic applicability of our attack, we perform experiments using both the EM and power side-channel.

EM Side Channel Setup

The EM side-channel measurements were 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.

Power Side Channel Setup

The power side-channel measurements were captured from an STM32F3 microcontroller (running at 7.372 MHz) using the ChipWhisperer CW308 setup. The measurements are then collected using a Lecroy 610Zi oscilloscope at a sampling rate of 1.25 GSam/sec.

4.1.2 Leakage Detection

We start by validating the presence of side-channel leakage due to the Bitwise_Manipulation behaviour. We utilize the well-known TVLA metric for the same, and detect leakage from the first bit of the src byte denoted as $\mathsf{src}[0]_0$. We capture two sets of side-channel measurements corresponding to $\mathsf{src}[0]_0 = 0$ and $\mathsf{src}[0]_0 = 1$ respectively, while the other bits of src are random. This however requires the attacker to control the value of the codeword $\mathbf{c}'$ on the target device, without the knowledge of the secret key.

Controlling the value of codeword $\mathbf{c}'$: The value of the codeword can be controlled by choosing maliciously crafted chosen-ciphertexts $\mathsf{ct} = \mathbf{u}, \mathbf{v}$, in the following manner. Choosing $\mathbf{u} = \mathbf{0} \in \mathcal{R}$, ensures that the codeword $\mathbf{c}' = \mathbf{v} - \mathbf{0} \cdot \mathbf{y}$ has the value of $\mathbf{c}' = \mathbf{v}$. Since the value of $\mathbf{v}$ can also be controlled by the attacker, he/she can observe side-channel leakage for codewords $\mathbf{c}'$ of his/her choice, without the knowledge of the secret key.

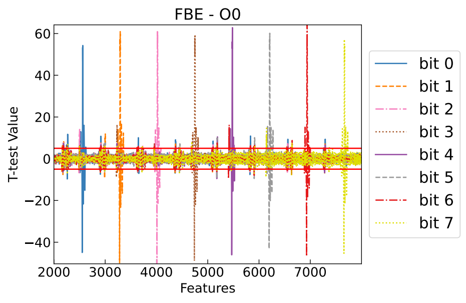
We acknowledge that leakage detection requires to utilize such invalid ciphertexts. However, this is not an issue as leakage detection only needs to be done once for a given target device. The same leakage profiles can then be used for multiple attacks, for different keys using valid ciphertexts. Even if the target device adopts a countermeasure to refresh the key pair after observing a decapsulation failure, profiling leakage of the codeword can still be carried out, since the value of the codeword is independent of the key during leakage evaluation.

To test for the leakage of $\mathsf{src}[0]_0$, we build two sets of ciphertexts. The first set is denoted as CT_0 whose $\mathbf{u} = 0$, but $\mathbf{v}$ is chosen such that its first bit $\mathbf{v}[0] = 0$, while all other bits are random. Similarly, the second set is denoted as CT_1 whose $\mathbf{u} = 0$, but $\mathbf{v}$ is chosen such that its first bit $\mathbf{v}[0] = 1$, while all of its other bits are random. We collect two sets of 10k side-channel measurements corresponding to decapsulation of the two ciphertexts in sets CT_0 and CT_1 denoted as $\mathcal{T}_0$ and $\mathcal{T}_1$ respectively. We normalize each trace and compute the Welch's t-test to identify the differentiating features between the trace sets.

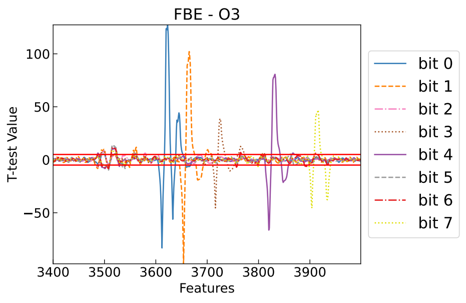
We performed experiments on two types of implementations: (1) `O0` compiler optimization: no optimization and (2) `O3` compiler optimization: highest optimization. Refer to Fig. 3 which shows the 8 t-test plots overlaid on top of each other, demonstrating leakage of 8 bits of the first byte $\mathsf{src}[0]$, for both the `O0` and `O3` optimized implementations. While Fig.3(a)-(b) denote results for the EM side-channel, Fig.3(a)-(b) denote results for the power side-channel.

In case of the `O0`-optimized implementation (Fig.3)(a) and Fig.3)(c)), we clearly observe eight distinct and uniformly distanced peaks corresponding to all the bits of $\mathsf{src}[0]$. However, in case of the `O3`-optimized implementation (Fig.3)(b) and Fig.3)(d)), we see that unlike for `O0`, the $t-$ test peaks are not uniformly spaced out and do not have similar heights.

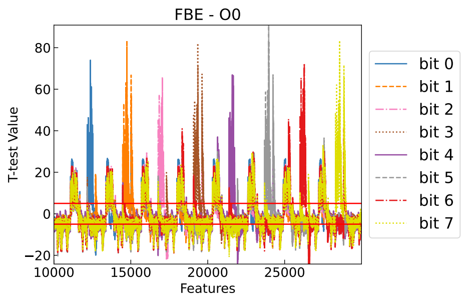
Effect of Compiler Optimization Level: In case of the `O0` optimized implementation, the compiler implements the individual iterations of the inner for loop (Line 8-12 in Fig.2) using the same set of assembly instructions. This leads to utilization of the same set of registers for all the bits, thereby giving rise to uniform leakage. However, in case of the `O3` optimized implementation, the compiler unrolls the inner for loop and utilizes different set of instructions and registers to manipulate the different bits. This results in non-uniform leakage for the individual bits within the src array.



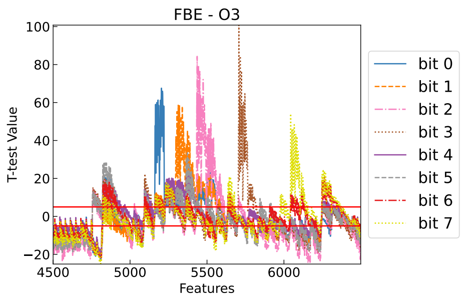
(a) `O0` optimized (EM)



(b) `O3` optimized (EM)



(c) `O0` optimized (Power)



(d) `O3` optimized (Power)

Figure 3: t-test leakage corresponding to 8 bits of the first byte of the src array in the First_Block_Extraction (FBE) within the ExpandAndSum operation. (a)-(b) show results for the EM side-channel, while (c)-(d) show results for the power side-channel.

We verified the same through analysis of the assembly level instructions for both the implementations. Though the leakage behaviour is different, we however detect significant leakage for all the bits in src[0], for both the `O0` and `O3` optimized implementations. Similarly, the attacker can detect leakage for the other 15 bytes of src, processed within the First_Block_Extraction operation.

In the following, we demonstrate how the detected leakage can be exploited to recover the first 16 bytes of the codeword stored in the src array.

4.2 Two Phase Codeword Recovery Attack

The attack is performed in two phases: (1) *Pre-Processing* phase and (2) *Exploitation* phase.

4.2.1 *Pre-Processing* Phase

This phase involves building side-channel templates for every bit in the first block of the src array (i.e.) bytes src$[0:15]$. It is a one-time process for a given target device, and the same templates can be used to perform multiple attacks. We present the methodology for building templates for the first bit src$[0]_0$, while the same technique can be used to build templates for all bits in src$[0:15]$.

We utilize the t-test plot obtained as a result of leakage detection for src$[0]_0$ (Refer to Fig.3) using the trace sets $\mathcal{T}_0$ and $\mathcal{T}_1$. We then select those features whose absolute t-test value is above a certain threshold Th_0, as our points of interest (PoI) denoted as $\mathcal{P}_0$. We stress that Th_0 is a parameter of the experimental setup, and can be experimentally determined. We use the selected features $\mathcal{P}_0$ to 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 $\Sigma_{(0,i)} \in \mathbb{R}^{(\|\mathcal{P}_0\|)\times(\|\mathcal{P}_0\|)}$. Thus, the reduced template for $\mathsf{src}[0]_0 = i$ is denoted as $tmp_{(0,i)} = (\mu_{(0,i)}, \Sigma_{(0,i)})$ for $i = \{0, 1\}$. The same procedure can be repeated to build side-channel templates for all 128 bits within the first block $\mathsf{src}[0 : 15]$.

Improving Efficiency for Template Creation: While it appears that separate/distinct trace sets are required to build templates for different bits, it is possible to build templates for all the bits using a single trace set. We can collect side-channel measurements corresponding to random ciphertexts $\mathsf{ct}_i = (\mathbf{u} = 0, \mathbf{v}_i)$ where $\mathbf{v}_i \in \mathcal{R}$ are chosen in random. In order to build templates for the bit $\mathsf{src}[0]_0$, we split the random ciphertexts into two sets $\mathcal{CT}_j$ for $j = \{0, 1\}$, such that the set $\mathcal{CT}_j$ corresponds to ciphertexts such that $\mathbf{u} = 0$ and $\mathbf{v}[0] = j$. Note that the exact number of traces required for profiling is an empirical parameter of the experimental setup. We merely establish that all bits of the codeword can be profiled simultaneously, thereby reducing the trace complexity of the pre-processing phase.

4.2.2 *Exploitation* Phase

In the exploitation phase, the attacker obtains a trace tr corresponding to decapsulation of a chosen valid ciphertext ct, whose codeword $\mathbf{c}'$ is intended to be recovered by the attacker. In order to recover the first bit $\mathsf{src}[0]_0$, we first utilize the PoI set $\mathcal{P}_0$ to build a reduced trace tr'_0, corresponding to $\mathcal{P}_0$. We then compute the maximum likelihood $f(\mathsf{tr}'_0)_i$ of the reduced trace tr'_0 for $i \in \{0, 1\}$, to match with the templates $tmp_{(0,i)}$ using the following formula:

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

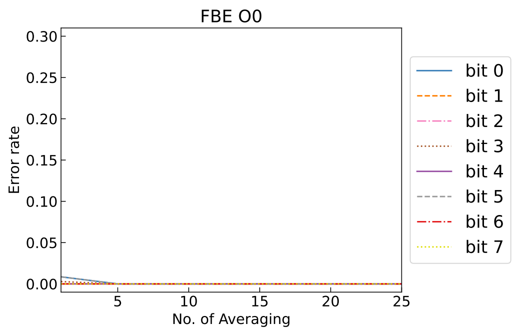
We then classify $\mathsf{src}[0]_0 = i$ based on the highest value of $f(\mathsf{tr}'_0)_i$. A similar approach can be used to recover all the bits of the first block of src array.

4.2.3 Experimental Results

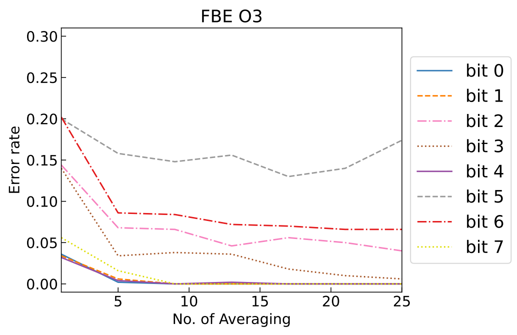
We performed experimental validation of the codeword recovery attack for both the `O0` and `O3` optimized implementations. Our attack targets leakage from single bit manipulation, and thus leakage only corresponds to a very few points. Thus, success rate of recovery is heavily dependent on the Signal to Noise Ratio (SNR). We therefore use the technique of averaging of repeated measurements to reduce the random noise, and thereby increase SNR to improve the success rate. Since the attacker has the ability to query with chosen-ciphertexts, he/she can obtain multiple side-channel measurements corresponding to the same inputs.

Fig.4 shows the error rate when recovering the 8 bits of the first byte $\mathsf{src}[0]$ for both the `O0` and `O3` optimized implementations. Fig.4(a)-(b) show results for the EM side-channel, while Fig.4(c)-(d) show results for the power side-channel. For single traces on `O0` optimized implementation, we obtain an error rate of ≈ 0.008 for the EM side-channel, and between $0.06 - 0.15$ for the power side-channel. However, for traces with an averaging of 25, we observe a much reduced error rate of 0 for the EM side-channel and $0.00 - 0.03$ for the power side-channel. This cleary demonstrates improved success rate with averaging in the traces.

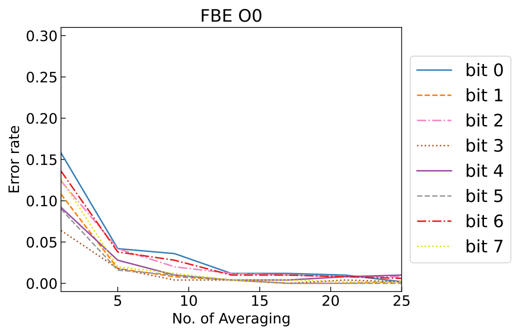
Similarly, for single traces on `O3` optimized implementation, we obtain an error rate of $\approx 0.04 - 0.20$ for the EM side-channel, and between $0.06 - 0.18$ for the power side-channel. However, for traces with an averaging of 25, we observe a slightly reduced error rate between $0.00 - 0.18$ for the EM side-channel and $0.02 - 0.22$ for the power



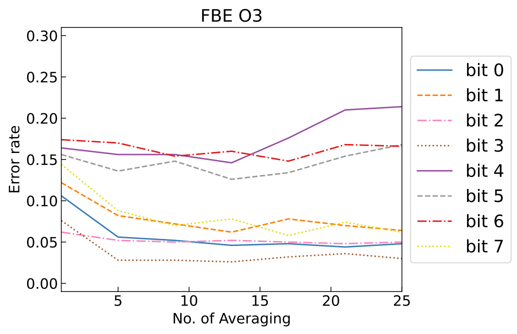
(a) `O0` optimized (EM)



(b) `O3` optimized (EM)



(c) `O0` optimized (Power)



(d) `O3` optimized (Power)

Figure 4: Error rate in recovering single bits of the codeword using leakage from the First_Block_Extraction (FBE) within the ExpandAndSum operation. (a)-(b) show results for the EM side-channel, while (c)-(d) show results for the power side-channel.

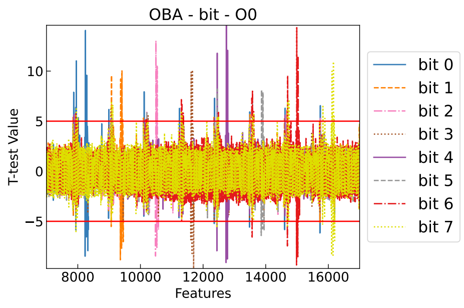
side-channel. The improvement due to averaging is slightly lesser compared to the `O0` optimized implementation.

An attacker can also other techniques such as employing high precision EM probes, hardware analog filters and advanced digital filtering to improve success rate, apart from averaging of replicated measurements. We hypothesize that a motivated attacker with a more sophisticated SCA setup and advanced post-processing techniques might be able to obtain achieve better error rates than the results obtained from our experiments.

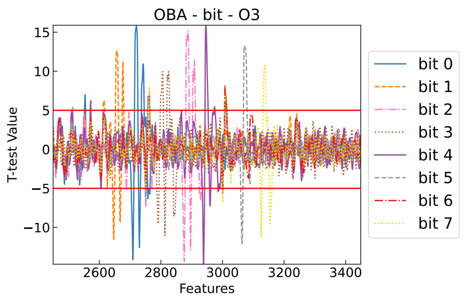
In this manner, it is possible to recover significant amount of information from the first block of 128 bits within every n_1 RM codewords in a single trace, thereby simultaneously recovering $(n_1 \cdot 128)$ bits out of the $(n_1 \cdot 128 \cdot M)$ bits of the total codeword.

4.3 Side-Channel Analysis of Other_Blocks_Accumulation

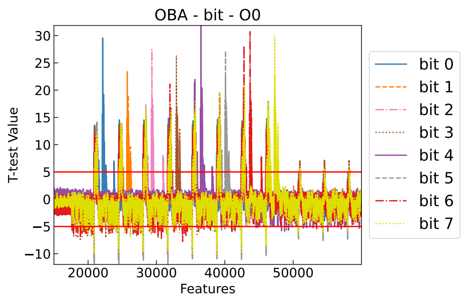
Similar to recovering the first block of the n_1 RM codewords, it is also possible to recover the other blocks of the RM codewords, exploiting leakage from the Other_Blocks_Accumulation operation within the ExpandAndSum operation. We observe that the Other_Blocks_Accumulation operation (Lines 16-26 in Fig.2) iterates over every bit (bitpos $=$ 0 to 8) within every byte (bytepos $=$ 0 to 15) in blocks 1 until $M-1$, and extracts every bit in index $(128 \cdot \mathit{block} + 8 \cdot \mathit{bytepos} + \mathit{bitpos})$ of the src array, and adds it to the index $(8 \cdot \mathit{bytepos} + \mathit{bitpos})$ in the dest array. Thus, leakage from extraction of the single bits can yet again be exploited to recover bits from the other blocks of the RM codeword. However, there is a subtle difference between leakage of the First_Block_Extraction and Other_Blocks_Accumulation operations. While First_Block_Extraction contained leakage from both the extraction as well as storage of the extracted bit in memory, leakage within



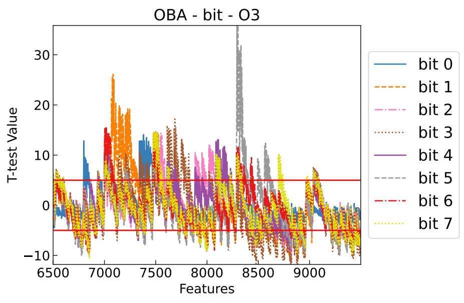
(a)



(b)



(c)



(d)

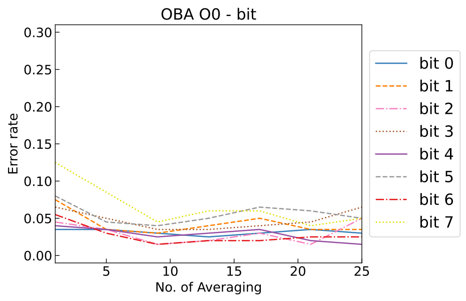
Figure 5: t-test leakage corresponding to 8 bits of the byte src[16] in the Other_Block_Accumulation (OBA) within the ExpandAndSum operation. (a)-(b) show results for the EM side-channel, while (c)-(d) show results for the power side-channel.

Other_Blocks_Accumulation is slightly weaker since leakage only arises from the extraction of the bit. Thus, we expect the success rate in recovering the bits of the other blocks to be slightly lower than that of the first block.

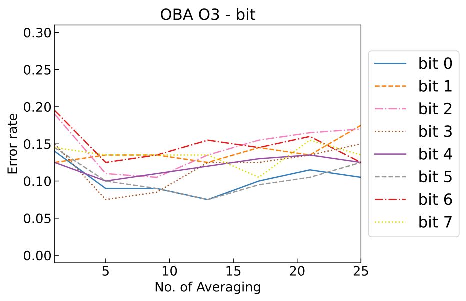
Refer to Fig. 5 which shows 8 t-test plots overlaid on top of each other, demonstrating leakage of 8 bits of the byte src[16] (first byte of the second block), for both the O0 and O3 optimized implementations. Similar to the case of the First_Block_Extraction operation, we observe that the t-test leakage is much more pronounced and uniform in case of O0 optimized implementation, compared to the O3 optimized implementation. Refer to Fig.6 that shows the error rate when recovering the 8 bits of the first byte of the second block (i.e.) src[16] for the O0 and O3 optimized implementations, and both the EM and power side-channel leakage.

For single traces on O0 optimized implementation, we obtain an error rate of $0.04 - 0.12$ for the EM side-channel, and between $0.06 - 0.09$ for the power side-channel. However, for traces with an averaging of 25, we observe a much reduced error rate of $0.02 - 0.08$ for the EM side-channel and $0.02 - 0.04$ for the power side-channel. This clearly demonstrates improved success rate with averaging in the traces. Similarly, for single traces on O3 optimized implementation, we obtain an error rate of $0.12 - 0.20$ for the EM side-channel, and between $0.20 - 0.30$ for the power side-channel. However, for traces with an averaging of 25, we observe a slightly reduced error rate between $0.10 - 0.17$ for the EM side-channel and $0.10 - 0.28$ for the power side-channel. The improvement due to averaging is slightly lesser compared to the O0 optimized implementation.

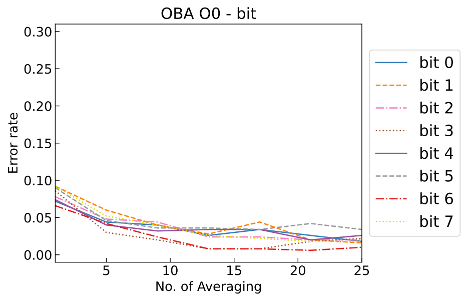
In this manner, side-channel analysis of the ExpandAndSum operation gives information on each bit of the corrupted codeword $\mathbf{c}' = \mathbf{m}\mathbf{G} + \hat{\mathbf{e}}$, where $\hat{\mathbf{e}} = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$. While we are not able to achieve perfect recovery of $\mathbf{c}'$ through SCA, we show that an attacker can still recover the secret key through our novel key recovery attacks.



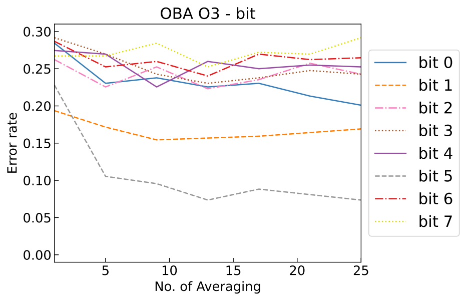
(a)



(b)



(c)



(d)

Figure 6: Error rate in recovering single bits of the byte src[16] in the Other_Block_Accumulation within the ExpandAndSum operation. (a)-(b) show results for the EM side-channel, while (c)-(d) show results for the power side-channel.

4.4 Key Recovery Attack

In this section, we analyze how to use information on $\mathbf{c}'$ to recover the secret key. We separate the analysis into two cases.

First, we assume we can recover $\hat{\mathbf{e}}$ without any errors, which is the case when the side-channel analysis of the expand and sum operation leaks each bit of $\mathbf{c}'$ with perfect accuracy. We then describe a new key recovery algorithm that works even when there is some error associated with the recovery of the bits of $\mathbf{c}'$.

4.4.1 Recovering the Key Without SCA Errors

In this case, the attacker generates valid ciphertexts and have access to an oracle that gives them the noisy codewords $\mathbf{c}'$ associated with the generated ciphertexts. Notice that, although the attacker does not manipulate the ciphertext, they know the values of $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{e}$. A bit more formally, this situation can be modeled as if the attacker has access to an oracle $\mathcal{O}_{\text{ES}}$ that, on each query with input pk, outputs a freshly generated tuple $(\mathbf{c}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$, where each $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{e}$ are are honestly chosen following the encryption algorithm described in Algorithm 1 using public key pk, and $\mathbf{c}'$ is the noisy codeword that is computed during decryption.

To recover the secret key $\mathbf{x}, \mathbf{y}$ associated with pk using queries to $\mathcal{O}_{\text{ES}}(\mathsf{pk})$, we observe that with only one query, together with the key equation $\mathbf{s} = \mathbf{x} + \mathbf{y} \cdot \mathbf{h}$, we can build a linear system with two equations in which the unknowns are precisely $\mathbf{x}$ and $\mathbf{y}$. Let $(\mathbf{c}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$ be an output of $\mathcal{O}_{\text{ES}}(\mathsf{pk})$, then we build the linear system

$$\begin{cases} \quad \mathbf{x} + \ \mathbf{y} \cdot \mathbf{h} = \mathbf{s}, \\ \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 = \hat{\mathbf{e}} - \mathbf{e}, \end{cases}$$

where $\hat{\mathbf{e}} = \mathbf{m}\mathbf{G} - \mathbf{c}'$.

By expanding the product between vectors into the multiplication of a vector and a circulant matrix, the linear system above can be written as the single linear equation

$$[\,\mathbf{x}\,|\,\mathbf{y}\,]\,\mathbf{T} = [\,\mathbf{s}\,|\,\hat{\mathbf{e}} - \mathbf{e}\,]\,,\ \text{where } \mathbf{T} = \begin{bmatrix} \mathbf{I} & \operatorname{rot}(\mathbf{r}_2)^\top \\ \operatorname{rot}(\mathbf{h})^\top & \operatorname{rot}(\mathbf{r}_1)^\top \end{bmatrix}.$$

The solution $[\,\mathbf{x}\,|\,\mathbf{y}\,]$ then can be easily recovered as long as the inverse of matrix $\mathbf{T}$ exists. Let us analyze when this is the case.

First notice that, if the inverse of the block matrix $\mathbf{T}$ exists, then it must be equal to

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{K}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}^{-1} \end{bmatrix} \begin{bmatrix} \operatorname{rot}(\mathbf{r}_1)^\top & \operatorname{rot}(\mathbf{r}_2)^\top \\ \operatorname{rot}(\mathbf{h})^\top & \mathbf{I} \end{bmatrix},\ \text{where } \mathbf{K} = \operatorname{rot}(\mathbf{r}_1)^\top + \operatorname{rot}(\mathbf{r}_2)^\top \operatorname{rot}(\mathbf{h})^\top,$$

which can be checked by evaluating the product $\mathbf{T}^{-1}\mathbf{T}$. This implies that $\mathbf{T}$ is invertible if and only if $\mathbf{K}$ is also invertible.

Notice that $\mathbf{K}$ is a circulant $n \times n$ matrix, since the product and sum of circulant matrices is also circulant. Furthermore, notice that 2 is a primitive root modulo n for all security levels supported by HQC. It is known that, for such values of n, a circulant $n \times n$ binary matrix is invertible if and only if the weight of its first row is odd [ABB+22, Section A.1.2]. We can see that the parity of the weight of the first row of $\mathbf{K}$ is equal to

$$\operatorname{w}(\mathbf{r}_1) + \operatorname{w}(\mathbf{r}_2)\operatorname{w}(\mathbf{h}) \bmod 2 = w_{\mathbf{r}} + w_{\mathbf{r}}\operatorname{w}(\mathbf{h}) \bmod 2.$$

Therefore, we have two cases. When $w_{\mathbf{r}}$ is odd, then $\mathbf{T}$ is invertible whenever $\operatorname{w}(\mathbf{h})$ is also odd, which happens with probability 1/2. However, when $w_{\mathbf{r}}$ is even, then $\mathbf{K}$ is never invertible, since the parity is always 0 in this case. Notice, from the security parameters shown in Table 1, that $w_{\mathbf{r}}$ is odd for levels 128 and 256, but it is even for 192 bits of security.

Although in the cases when $\mathbf{T}$ has no inverse we cannot directly solve the system with only one query, we can ask the oracle $\mathcal{O}_{\text{ES}}$ for more samples and build a larger system that should be full rank with overwhelming probability. The main limitation of this linear algebra approach to recover the key is that it does not work even if only a single bit of $\mathbf{c}'$ is wrong. Since in side-channel analysis there is usually some amount of error, this may significantly reduce the effectiveness of this algorithm. In the following section, we describe an algorithm that works well even when the bit error rate of $\mathbf{c}'$ is relatively high.

4.4.2 Recovering the Key Without SCA Errors

In this case, the attacker generates valid ciphertexts and have access to an oracle that gives them the noisy codewords $\mathbf{c}'$ associated with the generated ciphertexts. Notice that, although the attacker does not manipulate the ciphertext, they know the values of $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{e}$. A bit more formally, this situation can be modeled as if the attacker has access to an oracle $\mathcal{O}_{\text{ES}}$ that, on each query with input pk, outputs a freshly generated tuple $(\mathbf{c}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$, where each $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{e}$ are are honestly chosen following the encryption algorithm described in Algorithm 1 using public key pk, and $\mathbf{c}'$ is the noisy codeword that is computed during decryption.

To recover the secret key $\mathbf{x}, \mathbf{y}$ associated with pk using queries to $\mathcal{O}_{\text{ES}}(\mathsf{pk})$, we observe that with only one query, together with the key equation $\mathbf{s} = \mathbf{x} + \mathbf{y} \cdot \mathbf{h}$, we can build a linear system with two equations in which the unknowns are precisely $\mathbf{x}$ and $\mathbf{y}$. Let $(\mathbf{c}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$ be an output of $\mathcal{O}_{\text{ES}}(\mathsf{pk})$, then we build the linear system

$$\begin{cases} \mathbf{x} + \ \mathbf{y} \cdot \mathbf{h} = \mathbf{s}, \\ \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 = \hat{\mathbf{e}} - \mathbf{e}, \end{cases}$$

where $\hat{\mathbf{e}} = \mathbf{mG} - \mathbf{c}'$.

By expanding the product between vectors into the multiplication of a vector and a circulant matrix, the linear system above can be written as the single linear equation

$$[\,\mathbf{x}\,|\,\mathbf{y}\,]\,\mathbf{T} = [\,\mathbf{s}\,|\,\hat{\mathbf{e}} - \mathbf{e}\,]\,, \text{ where } \mathbf{T} = \begin{bmatrix} \mathbf{I} & \operatorname{rot}(\mathbf{r}_2)^\top \\ \operatorname{rot}(\mathbf{h})^\top & \operatorname{rot}(\mathbf{r}_1)^\top \end{bmatrix}.$$

The solution $[\,\mathbf{x}\,|\,\mathbf{y}\,]$ then can be easily recovered as long as the inverse of matrix $\mathbf{T}$ exists. Let us analyze when this is the case.

First notice that, if the inverse of the block matrix $\mathbf{T}$ exists, then it must be equal to

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{K}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}^{-1} \end{bmatrix} \begin{bmatrix} \operatorname{rot}(\mathbf{r}_1)^\top & \operatorname{rot}(\mathbf{r}_2)^\top \\ \operatorname{rot}(\mathbf{h})^\top & \mathbf{I} \end{bmatrix}, \text{ where } \mathbf{K} = \operatorname{rot}(\mathbf{r}_1)^\top + \operatorname{rot}(\mathbf{r}_2)^\top \operatorname{rot}(\mathbf{h})^\top,$$

which can be checked by evaluating the product $\mathbf{T}^{-1}\mathbf{T}$. This implies that $\mathbf{T}$ is invertible if and only if $\mathbf{K}$ is also invertible.

Notice that $\mathbf{K}$ is a circulant $n \times n$ matrix, since the product and sum of circulant matrices is also circulant. Furthermore, notice that 2 is a primitive root modulo n for all security levels supported by HQC. It is known that, for such values of n, a circulant $n \times n$ binary matrix is invertible if and only if the weight of its first row is odd [ABB+22, Section A.1.2]. We can see that the parity of the weight of the first row of $\mathbf{K}$ is equal to

$$\mathrm{w}(\mathbf{r}_1) + \mathrm{w}(\mathbf{r}_2)\,\mathrm{w}(\mathbf{h}) \bmod 2 = w_{\mathbf{r}} + w_{\mathbf{r}}\mathrm{w}(\mathbf{h}) \bmod 2.$$

Therefore, we have two cases. When $w_{\mathbf{r}}$ is odd, then $\mathbf{T}$ is invertible whenever $\mathrm{w}(\mathbf{h})$ is also odd, which happens with probability 1/2. However, when $w_{\mathbf{r}}$ is even, then $\mathbf{K}$ is never invertible, since the parity is always 0 in this case. Notice, from the security parameters shown in Table 1, that $w_{\mathbf{r}}$ is odd for levels 128 and 256, but it is even for 192 bits of security.

Although in the cases when $\mathbf{T}$ has no inverse we cannot directly solve the system with only one query, we can ask the oracle $\mathcal{O}_{\text{ES}}$ for more samples and build a larger system that should be full rank with overwhelming probability. The main limitation of this linear algebra approach to recover the key is that it does not work even if only a single bit of $\mathbf{c}'$ is wrong. Since in side-channel analysis there is usually some amount of error, this may significantly reduce the effectiveness of this algorithm. In the following section, we describe an algorithm that works well even when the bit error rate of $\mathbf{c}'$ is relatively high.

4.4.3 Recovering the Key with SCA errors

To account for the case when there may be some errors in the bits of $\mathbf{c}$', we now introduce the noisy oracle $\mathcal{O}^{\varepsilon}_{\text{ES}}$. On each query to $\mathcal{O}^{\varepsilon}_{\text{ES}}(\mathsf{pk})$, the oracle, similarly to its parent $\mathcal{O}_{\text{ES}}$, also returns a sequence $(\hat{\mathbf{c}}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$, where $\hat{\mathbf{c}}$ is a noisy representation of the corrupted codeword $\mathbf{c}'$ computed during decryption. Formally, vector $\hat{\mathbf{c}}$ can be written as

$$\hat{\mathbf{c}} = \mathbf{c}' + \mathbf{e}^\star = \mathbf{mG} + \hat{\mathbf{e}} + \mathbf{e}^\star,$$

where $\mathbf{e}^\star$ is an n-bit vector in which each entry is non-null with probability ε.

It is important to notice that, in each query, a fresh error vector $\mathbf{e}^\star$ is generated, and $\mathbf{e}^\star$ is unknown to the attacker as it represents the measurement errors from the side-channel analysis. As we discussed in the end of the previous section, the linear algebra approach alone is not effective to recover the key in this case because of the addition of $\mathbf{e}^\star$. To deal with this problem, we propose a different approach that exploits the sparsity of vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{r}_1$ and $\mathbf{r}_2$.

Suppose we make a number η of queries to oracle $\mathcal{O}^{\varepsilon}_{\text{ES}}(\mathsf{pk})$, obtaining as a list of results

$$\left(\hat{\mathbf{c}}^k, \mathbf{m}^k, \mathbf{r}_1^k, \mathbf{r}_2^k, \mathbf{e}^k\right) \leftarrow \mathcal{O}^{\varepsilon}_{\text{ES}}(\mathsf{pk}), \text{ for } k = 1 \text{ to } \eta.$$

Because of the unknown errors $\mathbf{e}^{\star}$ used to generate each sample, this would give us η approximate linear equations

$$\begin{aligned}\hat{\mathbf{c}}^k - \mathbf{m}^k\mathbf{G} \approx \hat{\mathbf{e}}^k &\approx \mathbf{x} \cdot \mathbf{r}_2^k + \mathbf{y} \cdot \mathbf{r}_1^k + \mathbf{e}^k \\ &\approx \sum_{j \in \text{supp}(\mathbf{x})} \left(\mathbf{r}_2^k \gg j\right) + \sum_{j \in \text{supp}(\mathbf{y})} \left(\mathbf{r}_1^k \gg j\right) + \mathbf{e}^k.\end{aligned}$$

Remember that one of the main arguments 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, just as their sum. Because of this sparsity, we expect that the left-hand side of the equation $\hat{\mathbf{c}}^k - \mathbf{m}^k\mathbf{G}$ should share a larger number of ones in the same locations with 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 in the equation above.

This motivates us to define the likelihoods vectors $\texttt{likelihoods}_{\mathbf{x}}$ and $\texttt{likelihoods}_{\mathbf{y}}$ in a way that they capture the similarity between $\hat{\mathbf{c}}^k - \mathbf{m}^k\mathbf{G}$ and the vectors $\mathbf{r}_2^k$ and $\mathbf{r}_1^k$. Formally, we let each entry j of the likelihood vectors be defined as[1]

$$\begin{aligned}\texttt{likelihoods}_{\mathbf{x}}[j] &= \sum_{k=1}^{\eta} \left|\text{supp}\left(\hat{\mathbf{c}}^k - \mathbf{m}^k\mathbf{G}\right) \cap \text{supp}\left(\mathbf{r}_2^k \gg j\right)\right|, \\ \texttt{likelihoods}_{\mathbf{y}}[j] &= \sum_{k=1}^{\eta} \left|\text{supp}\left(\hat{\mathbf{c}}^k - \mathbf{m}^k\mathbf{G}\right) \cap \text{supp}\left(\mathbf{r}_1^k \gg j\right)\right|.\end{aligned}$$

Figure 7 illustrates a real instance of the distributions of $\texttt{likelihoods}_{\mathbf{x}}[j]$ when j is in or is not in supp$(\mathbf{x})$. We can see that even for a relatively high error rate of $\varepsilon = 0.25$, just $\eta = 10$ queries to $\mathcal{O}^{\varepsilon}_{\text{ES}}$ are enough to see that $\texttt{likelihoods}_{\mathbf{x}}$ tends to be higher for values in supp$(\mathbf{x})$. Notice that the separation between the distributions is not enough to confidently decide if a given entry belongs to supp$(\mathbf{x})$. However, if we take the values of j with the lowest values of $\texttt{likelihoods}_{\mathbf{x}}(j)$, the figure suggests that we can be rather confident that they correspond to zero entries in $\mathbf{x}$.

We can then plug these likelihoods vectors into Algorithm 3 from Section 3.3, which will use them to try to solve the key equation. Figure 8 shows the results of our attack simulations on the necessary number of queries to $\mathcal{O}^{\varepsilon}_{\text{ES}}$ for full key recovery, considering different values of ε. We can see that this key recovery procedure is much better than the simple linear algebra approach. With very high probability, the attack works with only one query for values of ε up to 0.1. Furthermore, the attack works even for relatively high error rates of 0.35 with about 25 queries on average.

While Figure 8 gives us a clear picture on the effect of noise on the performance of key recovery, we need to combine this result with the real-world implementation of the side-channel attack. In particular, to quantify the interaction with the target device, it is important to compute the real number of challenges needed. That is, we must consider if we need to repeat challenges to average them out to get rid of SCA noise, as seen previously in the SCA sections.

[1] We remark that the definition of these likelihood vectors is very 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. In particular, when the algorithm is applied to a code whose parity-check matrix is $\left[\text{rot}\left(\mathbf{r}_2^k\right) \mid \text{rot}\left(\mathbf{r}_1^k\right) \mid \mathbf{I}\right]$ and when the target syndrome is $\hat{\mathbf{c}}^k - \mathbf{m}^k\mathbf{G}$. However, for each of the η queries, we get a new pair of code and syndrome, while the error vectors $\left[\mathbf{x} \mid \mathbf{y} \mid \mathbf{e}^k + (\mathbf{e}^{\star})^k\right]$ generating the syndromes all share its first part $(\mathbf{x}, \mathbf{y})$. This artificial case does not seem to have any connection to the real-world noise models used in coding theory.

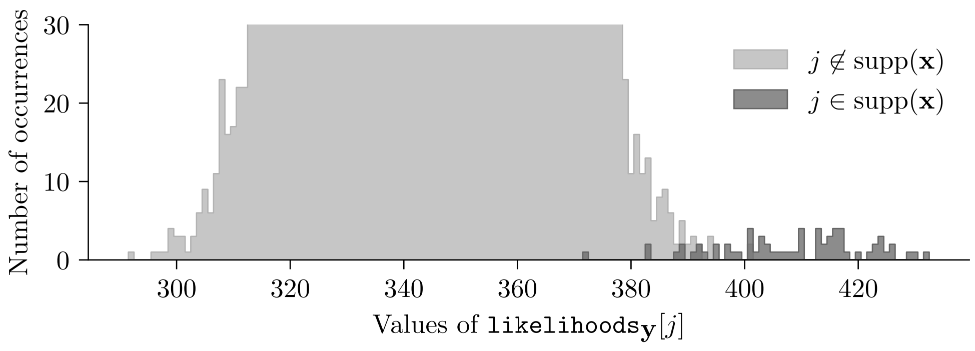


Figure 7: The distribution of the likelihoods $\texttt{likelihoods}_{\mathbf{x}}[j]$ when j is inside and outside the support of secret vector $\mathbf{x}$, using $\eta = 15$ queries to $\mathcal{O}^{\varepsilon}_{\mathrm{ES}}$ for $\varepsilon = 0.25$, considering parameters for 128 bits of security.

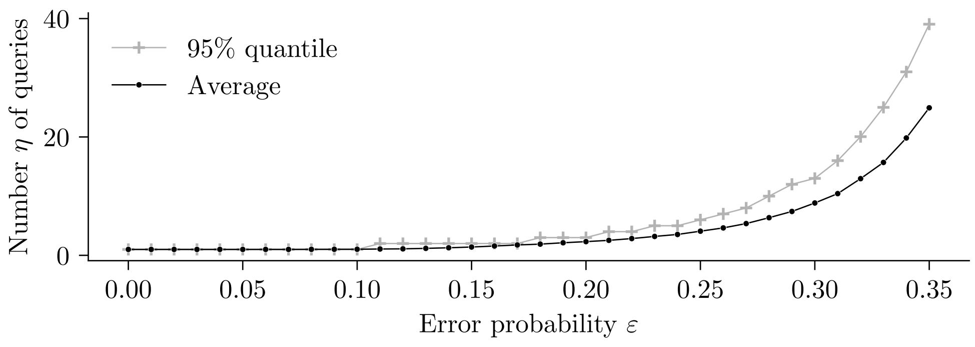


Figure 8: The number of queries to $\mathcal{O}^{\varepsilon}_{\mathrm{ES}}$ needed to recover the key for different values of ε, considering parameters for 128 bits of security. For each value of ε, we use data from the simulation of key recovery of 1000 freshly generated keys.

Figure 9 illustrates the real number of challenged needed for a full key recovery, for different number of repeated challenges. For each number ρ of repeated challenges in $\{1, 5, 9, 13, 17, 21, 25\}$, we computed the weighted average error rate by considering both the store and load operations. Notice that the average has to be weighted because store operation is used once for each bit in the first copy of each RM block, while the load is used for the other copies. Then for this average error rate, we used the number of queries needed for the attack in Figure 8 and multiplied it by ρ to get the true number of challenges for a real-world attack[2]. We can see that, for attacking ExpandAndSum under our target devices, it does not seem to be useful to average the result for multiple challenges. The reason is that, from each different challenge, we get a lot more information to compute the likelihoods than we get from repeating the challenge to get rid of noise.

5 SCA of the FindPeaks Operation

In this section, we show that side-channel leakage from the FindPeaks operation within the Reed-Muller decoding procedure can be used to deduce the output of the Reed-Muller decoding procedure. This information can be used by an attacker for full key recovery using valid chosen ciphertexts.

[2]It is useful to think as if the averaged result of the SCA constitute one query to the abstract oracle.

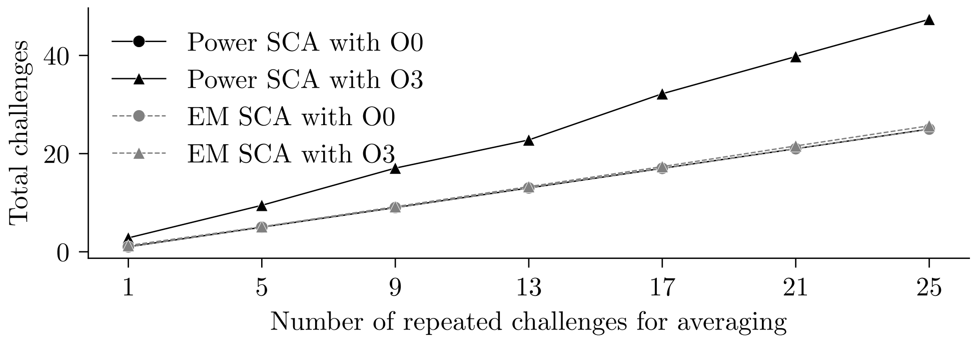


Figure 9: The real number of challenges to a target device needed to recover the key when using different number of repeated challenges for averaging, considering parameters for 128 bits of security.

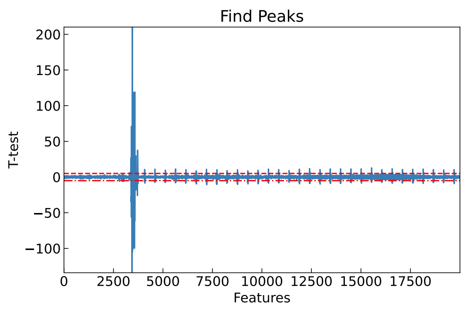
5.1 Side-Channel Vulnerability

The FindPeaks operation is the final operation in the Reed-Muller decoding procedure. Refer to Alg.4 for the pseudo-code of the FindPeaks procedure and Fig.11 for the C code snippet of the FindPeaks operation. This operation takes as input an array in with 128 elements and involves finding the index denoted as pos (7 bits), that contains the absolute maximum value. This implementation of the peak finding operation is shown in Lines 7 to 17 of Fig.11. The operation iterates over every element of the array in (i.e.) in$[i]$ for $i \in [0, 127]$, and the operation in each iteration is given as follows: For element in$[i]$, it computes its absolute value denoted as abs in Line 10.

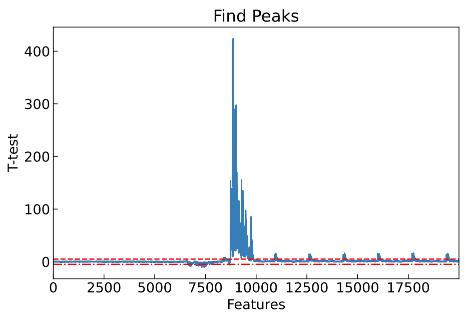
Then, it computes a variable denoted as mask (Line 12) based on the absolute value of the previous peak (peak_abs) and abs. The mask variable can only take two possible values - 0x0000/0xFFFF where mask = 0x0000, when peak_abs > abs, and mask = 0xFFFF when peak_abs ≤ abs. The mask variable is then used to update the pos variable, with the index of the newly identified peak (Line 15). Similarly, it is also used to update the peak variable, which contains the value of the current peak (Line 14), and the peak_abs variable, which contains the absolute value of the current peak.

Thus, we observe that the mask variable takes the value of 0xFFFF, only when a new peak is encountered. On the other hand, mask = 0x0000, whenever no new peak is encountered. Thus, identifying the mask variable for every iteration of the FindPeaks operation, can enable an attacker to recover the index of the peak. The last occurrence of mask = 0xFFFF in the 128 iterations of the FindPeaks operation provides the position of the peak variable (pos in Line 15. This comprises of the 7 least significant bits of the output of the FindPeaks operation.

The mask variable can only take two values: 0x0000/0xFFFF, which have a Hamming Weight (HW) difference of 16 bits. Thus, these two values should be easily distinguishable through the power/EM side-channel. In order to distinguish between mask = 0x0000/0xFFFF, we utilize the same t-test based template technique that we utilized to recover single bits of the RM codeword using leakage from the ExpandAndSum operation (Refer to Sec.4.1.2). While leakage from the ExpandAndSum operation involved distinguishing 1 bit of information (i.e.) 0 vs 1, leakage due to the mask variable in the FindPeaks operation is much more pronounced since the difference between the two possible mask values is 16 bits. We propose to build a template for the mask = 0x0000/0xFFFF, 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.



(a) EM side-channel



(b) Power side-channel

Figure 10: t-test leakage corresponding to Mask variable in the second iteration of the FindPeaks operation.

5.2 Experimental Evaluation

Refer to Fig.10 which shows the t-test leakage due to the mask variable in the second index of the FindPeaks operation (as a representative), for O3 optimized implementation, using both the EM and power side-channel. We can observe a very high peak in the t-test plot, and this leakage can be used to build template for the mask = 0x0000/0xFFFF for the second iteration. Similarly, leakage of the mask variable in other iterations can be obtained to build side-channel templates. For the O3 optimized implementation, we obtained a 100% success rate in recovering the mask variable even with single traces, using both the power and EM side-channel. Since the leakage is very dominant, averaging of side-channel traces is not required to amplify success rate.

Thus, we observe that this leakage can be used to easily recover the least 7 significant bits of the output of the FindPeaks operation. In the following, we show how this information can be used to recover the key, when a sufficiently large number of challenges is analyzed.

```c
uint8_t find_peaks(uint16_t in[128])
{
    uint16_t peak_abs = 0;
    uint16_t peak = 0;
    uint16_t pos = 0;
    uint16_t t, abs, mask;
    for (uint16_t i = 0; i < 128; i++)
    {
        t = in[i];
        abs = t ^ ((-(t >> 15)) & (t ^ -t));
        /* Computation of Mask Variable */
        mask = -(((uint16_t)(peak_abs - abs)) >> 15);
        /* Utilization of Mask Variable */
        peak ^= mask & (peak ^ t);
        pos ^= mask & (pos ^ i);
        peak_abs ^= mask & (peak_abs ^ abs);
    }
    pos |= 128 & ((peak >> 15) - 1);
    return (uint8_t) pos;
}
```

Figure 11: C code snippet of FindPeaks operation in RM decoder of HQC KEM. The target operations/variables for SCA are highlighted in red.

5.3 Key Recovery Attack

We emphasize that the key recovery algorithm using SCA on FindPeaks is more complex than the one used for attacking ExpandAndSum. However, the algorithm we explain in this section can be used to attack even the shuffled implementation of FindPeaks. Therefore, in this section, we take some time to carefully explain it in detail.

5.3.1 The oracle

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 j-th 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 be confident that the decoding of the j-th RM block did not corrected all the errors whenever there is a difference in the 7 least significant bits of $\mathbf{c}_j^{\text{FP}}$ and $\mathbf{c}_j^{\text{RS}}$. This motivates us to define the oracle $\mathcal{O}_{\text{FP}}$ that, when called with input pk, returns the tuple $(\mathbf{d}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$, whose elements are explained next.

As usual, $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{e}$ are are honestly chosen following the encryption of vector $\mathbf{m}$ using public key pk, as described in Algorithm 1 using public key pk. The side-channel information is represented by vector $\mathbf{d} \in \mathbb{F}_2^{n_1}$, such that, each of its entries j is

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

We can use oracle $\mathcal{O}_{\text{FP}}$ to define its noisy variant $\mathcal{O}_{\text{FP}}^{\varepsilon}$ as follows. On input pk, oracle $\mathcal{O}_{\text{FP}}^{\varepsilon}$ first calls its parent oracle obtaining $(\mathbf{d}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e}) \leftarrow \mathcal{O}_{\text{FP}}(\mathsf{pk})$. It then generates a noise vector $\mathbf{e}^{\star} \in \mathbb{F}_2^{n_1}$ in which each coordinate has probability ε of being 1. Then compute vector $\mathbf{d} \in \mathbb{F}_2^{n_1}$ as $\mathbf{d} \leftarrow \mathbf{d}' + \mathbf{e}^{\star}$. Finally, oracle $\mathcal{O}_{\text{FP}}^{\varepsilon}$ returns the tuple $(\mathbf{d}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$.

5.3.2 Using the oracle to get information on the key

In this section, we investigate what kind of information on the secret key we can learn from querying the oracle $\mathcal{O}_{\text{FP}}^{\varepsilon}$. Remember that the input for the decoding process is $\mathbf{c}' = \mathbf{m}\mathbf{G} + \hat{\mathbf{e}}$, where $\hat{\mathbf{e}} = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$. First, the last $n - n_1 n_2$ bits of $\mathbf{c}'$ are dropped, 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 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 $\mathrm{w}(\hat{\mathbf{e}}_j) \le \lfloor (n_2/2 - 1)/2 \rfloor$. Therefore, if the j-th bit of the oracle output $\mathbf{d}$ is equal to 1, then, with probability $1 - \varepsilon$, we know that the weight of block $\hat{\mathbf{e}}_j$ must be larger than $\lfloor (n_2/2 - 1)/2 \rfloor$.

Now $\hat{\mathbf{e}}$ is related to $\mathbf{x}$, $\mathbf{y}$, $\mathbf{r}_1$ and $\mathbf{r}_2$ by the following equation

$$\hat{\mathbf{e}} = \sum_{s \in \text{supp}(\mathbf{r}_2)} (\mathbf{x} \gg s) + \sum_{s \in \text{supp}(\mathbf{r}_1)} (\mathbf{y} \gg s) + \mathbf{e}.$$

Since $\mathbf{e}$ 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 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 related vectors $\mathbf{y}$ and $\mathbf{r}_1$, but notice that we could have chosen $\mathbf{x}$ and $\mathbf{r}_2$ without loss of generality. Let us consider the j-th RM block $\hat{\mathbf{e}}_j$, and notice that it can be written as

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

Therefore, from $\mathbf{d}[j]$ and $\mathbf{r}_1$, we can learn a small amount of information on the weight of $\operatorname{slice}_{(jn_2-s)}^{n_2}(\mathbf{y})$ for each s in $\operatorname{supp}(\mathbf{r}_1)$. For example, if $\mathbf{d}[j] = 1$, then, for each s in $\operatorname{supp}(\mathbf{r}_1)$, we increase our belief that the weight of $\operatorname{slice}_{(jn_2-s)}^{n_2}(\mathbf{y})$ 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 $\operatorname{slice}_i^{n_2}(\mathbf{y})$ for each possible $i \in \{0, \ldots, n-1\}$.

This motivates us to define two vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ in $\mathbb{R}^n$. Each entry $\mathbf{w}_1[i]$ represents the average number of failures when $\operatorname{slice}_i^{n_2}(\mathbf{y})$ was selected by an entry of $\mathbf{r}_1$, while, analogously, each entry $\mathbf{w}_2[i]$ represents the same quantity but for $\mathbf{x}$ and $\mathbf{r}_2$. Algorithm 5 shows how to build these vectors using outputs from $\mathcal{O}_{\text{FP}}^{\varepsilon}$. Suppose we made η queries to $\mathcal{O}_{\text{FP}}^{\varepsilon}$, obtaining $\left(\mathbf{d}^k, \mathbf{r}_1^k, \mathbf{r}_2^k\right)$, for $k = 1$ to η. Then we build $\mathbf{w}_1$ and $\mathbf{w}_2$ as

$$\mathbf{w}_\alpha = \textsc{Build}_{\mathbf{w}}\left(\left(\mathbf{r}_\alpha^k, \mathbf{d}^k\right)\right), \text{ for } \alpha = 1 \text{ and } 2.$$

Algorithm 5 Building vectors $\mathbf{w}_\alpha$ using the corresponding answers from oracle $\mathcal{O}_{\text{FP}}^{\varepsilon}$.

```
1: procedure Build_w(Pairs $(\mathbf{r}_\alpha^k, \mathbf{d}^k)$ for $k = 1$ to $\eta$)
2:     $(\texttt{counts}, \texttt{sum}) \leftarrow (\mathbf{0}, \mathbf{0}) \in \mathbb{R}^n \times \mathbb{R}^n$
3:     for $k = 1$ to $\eta$ do                                ▷ Iterate over oracle answers
4:         for $s$ in supp $(\mathbf{r}_\alpha^k)$ do               ▷ Iterate over the rows selected by $\mathbf{r}^k$
5:             for $j = 1$ to $n_1$ do                            ▷ Iterate over RM blocks
6:                 $i \leftarrow jn_2 - s \bmod n$          ▷ Block $j$ in row $s$ corresponds to slice $[i : i + n_2]$
7:                 $\texttt{counts}[i] \leftarrow \texttt{counts}[i] + 1$
8:                 $\texttt{sum}[i] \leftarrow \texttt{sum}[i] + \mathbf{d}^k[j]$
9:             end for
10:        end for
11:    end for
12:    $\mathbf{w}_\alpha \leftarrow \mathbf{0} \in \mathbb{R}^n$                                    ▷ Initialize $\mathbf{w}_\alpha$
13:    for $i = 1$ to $n$ do
14:        $\mathbf{w}_\alpha[i] \leftarrow \texttt{sum}[i]/\texttt{counts}[i]$     ▷ Average number of failures related to slice $[i : i + n_2]$
15:    end for
16:    return $\mathbf{w}_\alpha$
17: end procedure
```

Figure 12 illustrates how, for a sufficiently large number η of oracle queries, $\mathbf{w}_1$ and $\mathbf{w}_2$ can be seen as approximations of the weights each possible slice 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 queries. We can clearly see that the shapes of both graphs are very similar, except that there is considerable noise in $\mathbf{w}_1$, despite the large number of queries.

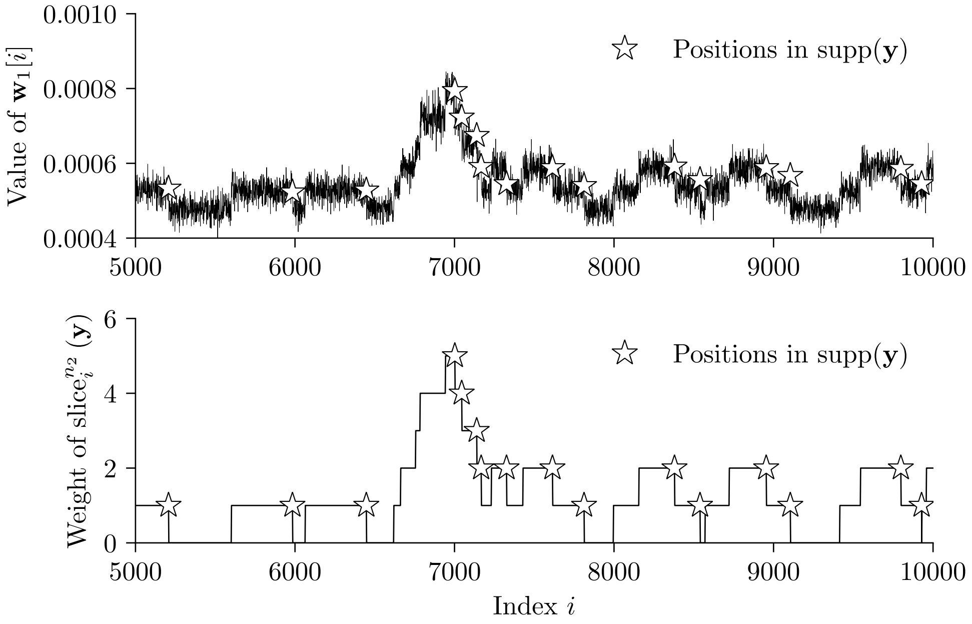


Figure 12: A comparison between $\mathbf{w}_1$ and the weight of each slice of n_2 bits of $\mathbf{y}$, considering η queries to $\mathcal{O}_{\text{FP}}^{\varepsilon}$ with $\varepsilon = 0$ and parameters for 128 bits of security.

5.4 Recovering the key

We now discuss how to use the information from $\mathcal{O}_{\text{FP}}^{\varepsilon}$ to effectively recover the key. In particular, we define how to compute likelihood vectors from the information provided by the oracle that are summarized by the $\mathbf{w}_1$ and $\mathbf{w}_2$.

For concreteness, let us focus on vector $\mathbf{y}$, but notice that our discussion 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 12, this definition implies that vector $\mathbf{w}_1$ can be seen as a noisy representation of $\Delta_{\mathbf{y}}$.

We can point the following 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 one in $\mathrm{slice}_i^{n_2}(\mathbf{y})$ but this one 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 $\mathbf{y}[i - n_2] = 0$.

These two patterns can be seen in Figure 12, by noticing that the non-null entries of of $\mathbf{y}$ are typically located on the right edges of the graphs, which are distant by n_2 positions of one left edge. We point that, when looking at Figure 12, it is tempting to consider that the associated edge is the closest left edge at the same level, but when there are close non-null entries in $\mathbf{y}$, this is not true. Consider, for example, the entries located between $i = 8000$ and $i = 8500$, where the corresponding right and left edges are not at the same level.

While these two cases can help us find most of the ones, there is an important caveat: when $\mathbf{y}[i] = \mathbf{y}[i+n_2] = \mathbf{y}[i+2n_2] = 1$, the middle non-null entry at $\mathbf{y}[i+n_2]$ cannot be detected by any of the cases. Additionally, this case could be extended for 4 or more non-null entries separated by n_2 bits in a sequence. To see this, suppose, as a toy example, that $\operatorname{supp}(\mathbf{y}) = \{n_2, 2n_2, 3n_2\}$. Then $0 = \Delta_{\mathbf{y}}(0) < \Delta_{\mathbf{y}}(1) = 1$, and therefore we detect that $\mathbf{y}[n_2] = 1$. Additionally, $1 = \Delta_{\mathbf{y}}(3n_2) > \Delta_{\mathbf{y}}(3n_2+1) = 0$, from which we conclude that $\mathbf{y}[3n_2] = 1$. But there is no transition in $\Delta_{\mathbf{y}}$ that allows us to detect that $\mathbf{y}[2n_2] = 1$, since both $\Delta_{\mathbf{y}}(n_2) = \Delta_{\mathbf{y}}(n_2+1) = 1$ and $\Delta_{\mathbf{y}}(2n_2) = \Delta_{\mathbf{y}}(2n_2+1) = 1$.

However, a careful consideration of the perfect information from $\Delta_{\mathbf{y}}$ allows us to detect this cases. One could detect these problematic cases by noticing a place where there is a left edge but no corresponding right edge at n_2 positions after, and flagging it as a 1. The problem is that, when we try to build the likelihood functions with the noisy representation of $\Delta_{\mathbf{y}}$ provided by $\mathbf{w}_1$, such techniques are difficult to code and they do not seem to be very robust.[3] As we demonstrate next, we use a simpler technique that appears to be effective when recovering the key.

These observations form the basis of how we can compute likelihoods based on $\mathbf{w}_1$ and $\mathbf{w}_2$, which, as we discussed in the previous section, can be seen as an approximation of $\Delta_{\mathbf{y}}$ and $\Delta_{\mathbf{x}}$, respectively. To compute the likelihoods, we propose Algorithm 6. Together with the vector $\mathbf{w}_\alpha$, the algorithm takes two parameters: integers ℓ and ν. Parameter ℓ serves as the window size that is used to smooth out the noisy vector $\mathbf{w}_\alpha$. Parameter ν is important in corner cases, such as the one mentioned in the last paragraph, and its role is explained next.

Algorithm 6, on input $\mathbf{w}$, ν, and ℓ computes the likelihoods in two phases. First it computes the likelihoods of each index by considering the sizes of the steps around it, considering the difference between the averages of two window of length ℓ, one before and one after i. Higher differences moving down (positive `diff`), increase the likelihood of index i, high differences moving up (negative `diff`), increase the likelihood of index $i + n_2$. Then, if $\nu > 0$, comes the vetting of indexes that are at a distance multiple of n_2 from points of high likelihood. If this is the case, then for the $2w$ points of highest likelihood, the likelihoods of its neighbors at cyclic distance jn_2 is set to ∞, for $j = 1$ to ν, which means that they will not be considered as a non-null entry when trying to solve the key equation.

Now that we defined the likelihoods function we are ready to define the full recovery algorithm as Algorithm 7. Essentially, this algorithm combines calls to the key equation solving from Algorithm 3 with the iteration over possible values of η and window size ℓ, from 0 to $\eta_{\max}$ and 1 to $\ell_{\max}$, respectively.

The choice of iterating over parameters ℓ and ν before computing the likelihood dramatically reduces the number of oracle queries needed to recover the key, at the cost of more processing power. The reason why we iterate over them is that it seems to be difficult to give a fixed set of parameters (ℓ, ν) that at the same time fit all public keys and attacking setups. On the one hand, a large ℓ is better to get rid of the noise when the number η of queries to $\mathcal{O}^{\varepsilon}_{\text{FP}}$ is small or when the oracle error ε is high. On the other hand, in cases when there are non-null entries in $\mathbf{x}$ or $\mathbf{y}$ that are close to each other, a smaller ℓ is better for distinguishing them.

Parameter η clearly is important when the public key falls into the problematic case of more than 2 ones separated by n_2 positions. However, even if there are only 2 ones separated by n_2 positions, we observed that $\eta = 1$ ends up being very useful as these cases are also harder to detect because only one of the edges will be apparent in the noisy vectors $\mathbf{w}_{\mathbf{x}}$ or $\mathbf{w}_{\mathbf{y}}$.

[3]We tried implementing these detection methods, but we faced the problem that these events are rare and there is too much noise to accurately detect them. Furthermore, if these cases are not detected, the key recovery becomes very difficult.

Algorithm 6 Computing the likelihoods for vector $\mathbf{w}$ using moving averages of length ℓ.

```
procedure LIKELIHOODS($\mathbf{w}, \nu, \ell$)
    likelihoods $\leftarrow \mathbf{0} \in \mathbb{R}^n$
    for $i = 0$ to $n - 1$ do
        mean_before $\leftarrow$ Mean value of $\text{slice}^{\ell}_{i-\ell+1}(\mathbf{w})$    ▷ Includes i
        mean_after $\leftarrow$ Mean value of $\text{slice}^{\ell}_{i+1}(\mathbf{w})$    ▷ Does not include i
        diff $\leftarrow$ mean_before $-$ mean_after    ▷ Higher values on right peaks
        likelihoods[$i$] $\leftarrow$ likelihoods[$i$] $+$ diff
        likelihoods[$i + n_2$] $\leftarrow$ likelihoods[$i + n_2$] $-$ diff
    end for
    if $\nu > 0$ then    ▷ If $\nu > 0$ then comes the vetting of indexes
        sorted_likelihoods_indexes $\leftarrow$ Indexes of likelihoods in decreasing order
        for $k = 0$ to $2w - 1$ do
            $i \leftarrow$ sorted_likelihoods_indexes[$k$]
            for $v = 1$ to $\nu$ do
                likelihoods[$i + \nu n_2$] $\leftarrow \infty$
                likelihoods[$i - \nu n_2$] $\leftarrow \infty$
            end for
        end for
    end if
    return likelihoods
end procedure
```

Algorithm 7 Recovering the key using the $\mathcal{O}^{\varepsilon}_{\text{FP}}$.

```
procedure KEYRECOVERY($\mathbf{w}_1, \mathbf{w}_2$)
    for $\nu = 0$ to $\nu_{\max}$ do
        for $\ell = 1$ to $\ell_{\max}$ do
            likelihoods$_{\mathbf{x}}$ $\leftarrow$ LIKELIHOODS($\mathbf{w}_2, \ell, \nu$)
            likelihoods$_{\mathbf{y}}$ $\leftarrow$ LIKELIHOODS($\mathbf{w}_1, \ell, \nu$)
            $\mathbf{y} \leftarrow$ TRYTOSOLVEKEYEQUATION(likelihoods$_{\mathbf{x}}$, likelihoods$_{\mathbf{y}}$)
            if $\mathbf{y} \neq \perp$ then
                return $\mathbf{y}$
            end if
        end for
    end for
    return $\perp$
end procedure
```

Figure 13 shows the number of queries needed for a successful attack against 128 bits of security parameters using oracle $\mathcal{O}^{\varepsilon}_{\text{FP}}$. For each value of ε, we use data from the simulation of key recovery of 30 freshly generated keys, using parameters $\nu_{\max} = 4$ and $\ell_{\max} = 50$. Notice that even a small increase in ε from 0 to 0.0025 has a significant impact on the number of queries needed, which goes from about 200000 to more than a million. This happens because RM errors rarely occur and even a small error $\varepsilon = 0.0025$ causes a lot of noise.

We conclude this section by pointing that, differently from what we did when attacking ExpandAndSum, we do not need to consider the effect of averaging. This is because, as mentioned in the beginning of the section, we observed a 100% accuracy for our classifier of the 7 least significant bits of the output of FindPeaks.

6 Attacking Protected Implementations

In the previous sections, we demonstrated the ability of our attack to perform full key recovery on unprotected implementations of the ExpandAndSum and FindPeaks opera-

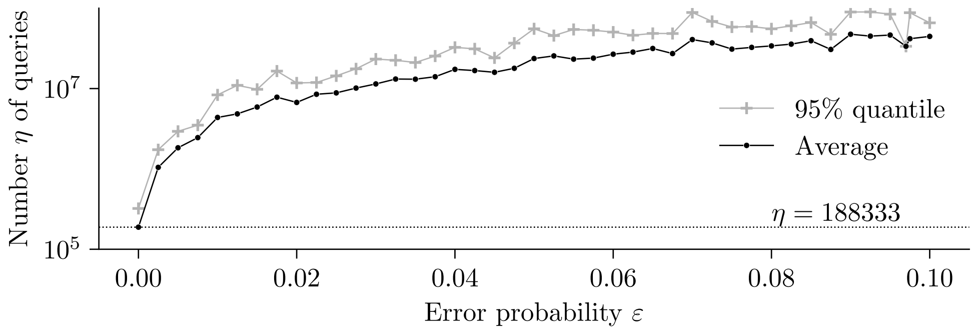


Figure 13: The number of queries to $\mathcal{O}^{\varepsilon}_{\text{FP}}$ needed to recover the key for different values of ε, considering parameters for 128 bits of security.

tion within the Reed-Muller decoder. We showed that our attacks are robust to noise in the side-channel measurements and are capable of full key recovery, even in the presence of non-negligible error in the information extracted from the side-channel measurements. In the following, we study the applicability of our attack to two well-known SCA countermeasures - (1) Shuffling and (2) Masking.

In Sec. 4 and Sec. 5, we demonstrated the profiling leakage of the ExpandAndSum and FindPeaks operation for known codewords, can be used to build templates, which can be used to recover information about the codeword, leading to full key recovery. However, both the shuffling or masking countermeasure randomize execution of operations on the codeword, which ensures that an attacker cannot profile leakage for known codewords. This disables the attacker from using leakage detection techniques such as TVLA to build templates for side-channel analysis. In this respect, we assume two scenarios.

1. Access to Clone Device: In the presence of a clone device, the attacker can build templates using leakage from the clone device, where the attacker can control of have knowledge of the shuffling order, in case of the shuffling countermeasure, or the knowledge of the shares in case of the masking countermeasure. This will allow the attacker to still build templates for target operations corresponding to known values of the codeword.

2. No Access to Clone Device: In this scenario, the attacker has to adopt unsupervised approaches towards building templates, as he cannot control the shuffling/masking countermeasure on the target device. In this respect, we present novel methodologies on how to detect leakage from the target operations directly using leakage from the target device.

We first study the applicability of our attack to the shuffling countermeasure, followed by the masking countermeasure.

6.1 Shuffled Implementation of ExpandAndSum Operation

In the shuffled implementation of the ExpandAndSum Operation, we assume that the bits are extracted and processed in a random order. We first explain our methodology to build templates using leakage directly from the shuffled implementation of the ExpandAndSum operation, followed by our key recovery attack.

6.1.1 Building Templates directly on Target Device

For the attack on shuffling countermeasure, we assume that the attacker has (partial) knowledge on when the bit of the codeword is being processed during the execution, but could not know the exact position of the bit on the trace. For the attack, we adopt the approach from unsupervised machine learning, namely, using k-means clustering [Mac67]. In our case, since our target is bit value, $k = 2$. The general idea of the attack can be found in Algorithm 8.

Algorithm 8 k-means clustering for attacking shuffling protected implementation

```
 1: procedure $K$-MEANS(T, $repeat$)
 2:     L $\leftarrow \mathbf{0} \in \mathbb{R}^N$
 3:     TVLA $\leftarrow 0$
 4:     idx $\leftarrow 0$
 5:     for $i = 0$ to $n - 1$ do                          ▷ Run the clustering for each time samples
 6:         $\mu_0, \mu_1 \leftarrow$ RandomAssign($T^i$)          ▷ Assign 2 traces randomly as mean cluster
 7:         $temp\_label \leftarrow$ ShortestDistance($T^i, \mu_0, \mu_1$)     ▷ Assign label to each trace
 8:         $\mu_0, \mu_1 \leftarrow$ UpdateLabel($temp\_label, \mu_0, \mu_1$)       ▷ Update cluster mean
 9:         for $j = 1$ to repeat-1 do                            ▷ Repeat the procedure
10:             $temp\_label \leftarrow$ ShortestDistance($T^i, \mu_0, \mu_1$)
11:             $\mu_0, \mu_1 \leftarrow$ UpdateLabel($temp\_label, \mu_0, \mu_1$)
12:         end for
13:         temp_TVLA $\leftarrow$ CalcTVLA($temp\_label$)          ▷ Calculate absolute TVLA value
14:         if temp_TVLA $>$ TVLA then
15:             TVLA $\leftarrow$ temp_TVLA
16:             idx $\leftarrow i$
17:             L $\leftarrow temp\_label$
18:         end if
19:     end for
20:     return L
21: end procedure
```

The general intuition for the approach is: for given time samples, if there is high leakage (as indicated by TVLA value), the trace should be easily clustered and this should be captured through k-means algorithm. Hence, this approach is related to the quality of the leakage. We would like to emphasize that this is a heuristic approach.

As a proof of concept, we performed experiments on EM side-channel leakage obtained from the shuffled ExpandAndSum operation on O0 optimized implementation. We collected 50,000 raw non-averaged traces with shuffling implementation and refer to Figure 14, which shows the TVLA calculated using the correct label, as well as TVLA calculated using predicted label obtained using k-means clustering. As observed, that within the region of interest, the predicted label achieve similar TVLA as the actual one. Indeed, when we compute the similarity between the predicted label and actual value, we achieve 100% similarity. However, note that there are other TVLA peaks as well that are not in the region of interest. Nevertheless, if the attacker has approximate knowledge of the location of the processed bits, then he/she should be able to differentiate between the true peaks from the false peaks.

The results for our experiments are then shown in Figure 15. For O0, we observe that it could achieve low error rate (< 0.05) for all bits. A more sophisticated attacker with a better SCA setup can potentially perform perfect recovery of all the bits. One can adopt a similar approach for O3 optimized implementation as well, but it is expected that the success rate in recovery will be impacted, given that leakage from O3 optimized implementation is less pronounced than O0 optimized implementation. Nevertheless, we have shown that it is indeed possible to build side-channel templates directly from leakage of the shuffled ExpandAndSum operation.

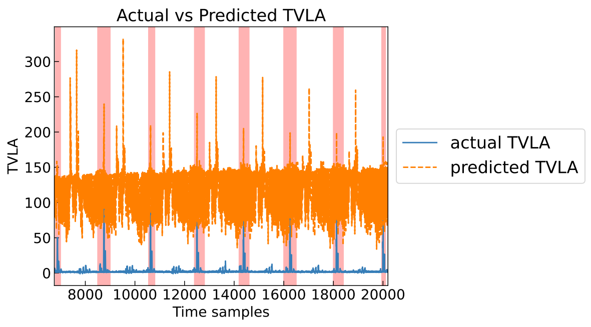


Figure 14: Actual vs Predicted TVLA of O0 implementation (shaded areas indicate different targeted bits)

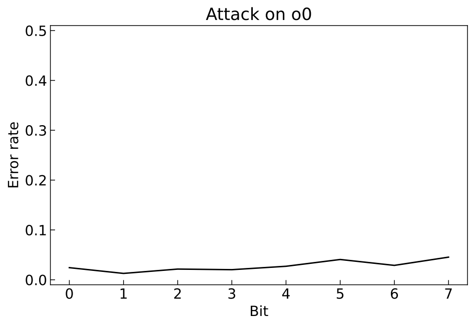


Figure 15: Error Rate for O0 implementation

6.1.2 Key Recovery Attack

In a shuffled implementation of ExpandAndSum, while we still can accurately recover each bit of $\mathbf{c}' = \mathbf{mG} + \hat{\mathbf{e}}$, the problem is that we cannot know its position, and therefore cannot use the same key recovery procedure.

Therefore, side-channel analysis on the shuffled implementation of ExpandAndSum allows us to recover the hamming weight of each of the n_2-bit blocks where the shuffling occurs. In this section, we show that this information is enough to recover the key using an idea similar to the one used in Section 4.

The Oracle. Let $\mathcal{O}_{\text{S(ES)}}$ denote the oracle that models the information given by the side-channel information on the shuffled implementation of ExpandAndSum. On input pk, it returns a freshly generated tuple $(\mathbf{w}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2)$, where $\mathbf{r}_1$ and $\mathbf{r}_2$ are the random sparse vectors used for the encryption of $\mathbf{m}$, and $\mathbf{w} \in \mathbb{Z}^{n_1}$ is the list of weights of each block of n_2 bits of the noisy codeword $\mathbf{c}' = \mathbf{mG} + \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$.

To account for possible errors during the side-channel analysis, we define a generalized oracle $\mathcal{O}^{\varepsilon}_{\text{S(ES)}}$ that also returns a tuple $(\mathbf{w}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2)$, but the output is computed as follows. First, $\mathcal{O}^{\varepsilon}_{\text{S(ES)}}$ calls the perfect oracle $(\mathbf{w}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2) \leftarrow \mathcal{O}_{\text{S(ES)}}(\mathsf{pk})$. Then it computes

$$\mathbf{w}[j] = \sum_{i=1}^{\mathbf{w}'[j]} (1 - B_i) + \sum_{i=\mathbf{w}'[j]+1}^{n_2} B_i,$$

where each B_i is a Bernoulli random variable with ε probability of being 1. Intuitively, the idea is that the B_i variables represent a possible bit error when using SCA to recover a bit.

The Likelihoods Vectors. Now let us analyze how the outputs of oracles $\mathcal{O}_{\text{S(ES)}}$ and $\mathcal{O}^{\varepsilon}_{\text{S(ES)}}$ give us information on the secret key. Suppose we are given an output $(\mathbf{w}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2)$ of $\mathcal{O}_{\text{S(ES)}}$. Notice that, since we know $\mathbf{m}$, we can compute $\mathbf{c}^{\text{RMRS}} = \mathbf{mG}$, and let $\mathbf{c}_j^{\text{RMRS}}$ denote the j-th RM block of n_2 bits of the encoded message, where j goes from 0 to $n-1$.

Now, compare the weight of each $\mathbf{c}_j^{\text{RMRS}}$ with $\mathbf{w}[j]$. We know that, by definition $\mathbf{w}[j] = \mathrm{w}\left(\mathbf{c}_j^{\text{RMRS}} + \hat{\mathbf{e}}_j\right)$, where $\hat{\mathbf{e}}_j$ is the j-th block of n_2 bits of vector $\hat{\mathbf{e}} = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$. As in the other attacks, our intention is to exploit the fact that $\hat{\mathbf{e}}$ is directly dependent on the secret key, but we do not have access to its entries directly.

Consider vectors $\mathbf{w}[j]$ and $\mathbf{c}_j^{\text{RMRS}}$ that we know, since we have access to oracle $\mathcal{O}_{\text{S(ES)}}$. We can see that there are two cases of interest.

1. When $\mathbf{w}[j] > \mathrm{w}\left(\mathbf{c}_j^{\text{RMRS}}\right)$, 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 $\mathbf{w}[j] < \mathrm{w}\left(\mathbf{c}_j^{\text{RMRS}}\right)$, 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 `flip_candidates` $\in \{0,1\}^{n_1 n_2}$ such that

$$\texttt{flip\_candidates}[i] = \begin{cases} 1 - \mathbf{c}^{\text{RMRS}}[i], \text{ if } \mathrm{w}\left(\mathbf{c}^{\text{RMRS}}_{\lfloor i/n_2 \rfloor}\right) > \mathbf{w}\left[\lfloor i/n_2 \rfloor\right] \\ \mathbf{c}^{\text{RMRS}}[i], \text{ if } \mathrm{w}\left(\mathbf{c}^{\text{RMRS}}_{\lfloor i/n_2 \rfloor}\right) < \mathbf{w}\left[\lfloor i/n_2 \rfloor\right] \\ 0, \text{otherwise.} \end{cases} \tag{2}$$

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 block j of n_2 bits increased when going from $\mathbf{c}^{\text{RMRS}}$ to $\mathbf{c}'$, then all of the null entries in the block j are marked as 1. Alternatively, if the weight of the block j decreased, then all of the null, entries in the block j are marked as 1.

We are now ready to define a likelihood function that can be seen as a variant of the one used for attacking the non-shuffled implementation of ExpandAndSum. Suppose we make a number η of queries to oracle $\mathcal{O}^{\varepsilon}_{\text{S(ES)}}$, and obtain a set of tuples $\left\{\left(\mathbf{w}^k, \mathbf{m}^k, \mathbf{r}_1^k, \mathbf{r}_2^k\right) \text{ for } k = 1 \text{ to } \eta\right\}$. Let $\texttt{flip\_candidates}^k$ denote the vector defined in Equation 2, by using the results of each query k make the substitution of variables as $\mathbf{w} \leftarrow \mathbf{w}^k$ and $\mathbf{c}^{\text{RMRS}} \leftarrow \mathbf{m}^k\mathbf{G}$.

Then we compute the likelihood vectors $\texttt{likelihoods}_{\mathbf{x}}$ and $\texttt{likelihoods}_{\mathbf{y}}$ as

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

$$\texttt{likelihoods}_{\mathbf{y}}[j] = \sum_{k=1}^{\eta} \left|\mathrm{supp}\left(\texttt{flip\_candidates}^k\right) \cap \mathrm{supp}\left(\mathbf{r}_1^k \gg i\right)\right|.$$

We invite the reader to compare these likelihood vectors with the ones used to attack the non-shuffled implementation of ExpandAndSum in Section 4.4. In particular, it is interesting to notice that the ones used for the non-shuffled implementation can be seen as a particularization of these new ones 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 the Algorithm 3 to try to solve the key equation. Figure 16 shows the number of queries needed for a successful attack against 128 bits of security. For each value of ε, we simulated the attack against 100 random keys. We can see that the attack is very efficient, allowing

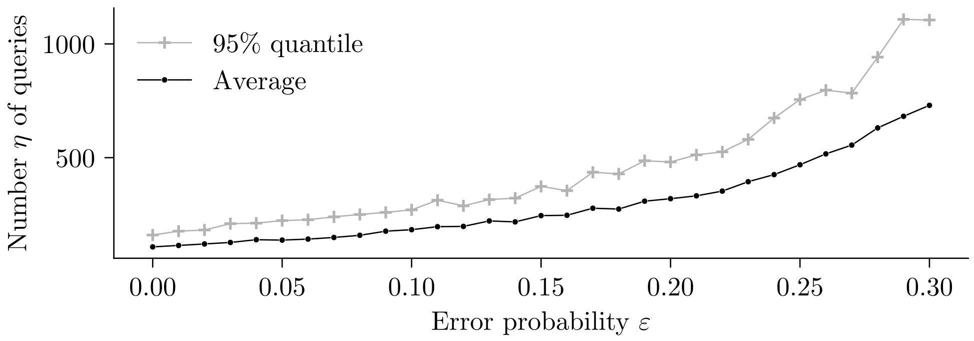


Figure 16: The number of queries to $\mathcal{O}^{\varepsilon}_{\text{S(ES)}}$ needed to recover the key for different values of ε, considering parameters for 128 bits of security.

us to recover the secret key with less than 200 queries, on average, even for bit error rates close to $\varepsilon = 0.10$.

Remember that, differently from our SCA against FindPeaks, that there is also some possible bit error rates when trying to recover the hamming weight of the blocks of $\mathbf{c}'$. Therefore, just like what was done in Section 4.4, it is useful to consider the effect of averaging multiple traces in the key recovery. Figure 17 shows the results on the true number of challenges needed for the recovery attack. Again, the best results were observed when using only 1 trace for the same challenge, instead of multiple ones. In particular, our results suggest that about 300 traces are enough, on average, to recover the secret key under the two setups (EM and power) and optimization levels.

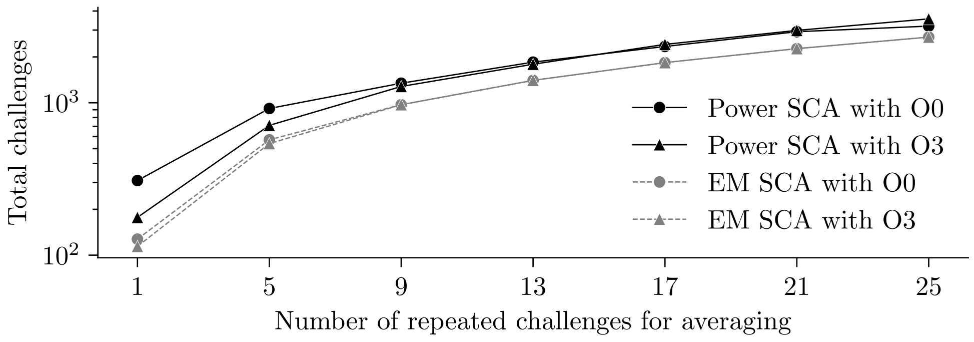


Figure 17: The real number of challenges to a target device needed to recover the key when using different number of repeated challenges for averaging, considering parameters for 128 bits of security.

6.2 Shuffled Implementation of FindPeaks Operation

In the shuffled implementation of the FindPeaks operation, we assume that the elements of the input array are processed in a random order. In this respect, we can adopt the same approach that used for building templates directly on the shuffled ExpandAndSum operation, to build templates for the `mask` variable, for every iteration of the FindPeaks operation.

Refer to Section 6.1.1 for a detailed description of the same. From Figure 18, we plotted the TVLA calculated using actual label. We then calculated the TVLA using predicted

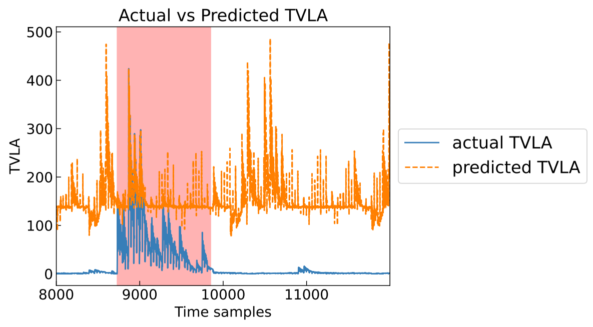


Figure 18: Actual TVLA vs TVLA from k-means prediction for Find Peaks CW O3 implementation (shaded area indicates region of interest)

label obtained using k-means clustering. As observed, that within the region of interest, the predicted label achieve similar TVLA as the actual one. Indeed, when we compute the similarity between the predicted label and actual value, we achieve 100% similarity. Thus, we can see that it is indeed possible to build side-channel templates directly from leakage of the shuffled FindPeaks operation.

6.2.1 Key Recovery Attack

The Oracle. To model the information we get from side-channel analysis of the FindPeaks operation, first we define two oracles: $\mathcal{O}_{\mathrm{S(FP)}}$ and its noisy generalization $\mathcal{O}^{\varepsilon}_{\mathrm{S(FP)}}$. Given a public key pk, the oracle $\mathcal{O}_{\mathrm{S(FP)}}$ outputs a tuple $(\mathbf{p}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2)$. As usual, $\mathbf{r}_1$ and $\mathbf{r}_2$ are the random sparse vectors used for the encryption of $\mathbf{m}$ using pk. The side-channel information is contained in vector $\mathbf{p} \in \mathbb{Z}^{n_2}$, in which $\mathbf{p}[j]$ denotes the number of times there is a change of value in the variable containing the current peak when iterating over the shuffled $\mathbf{a}''$ array. Formally, this value is:

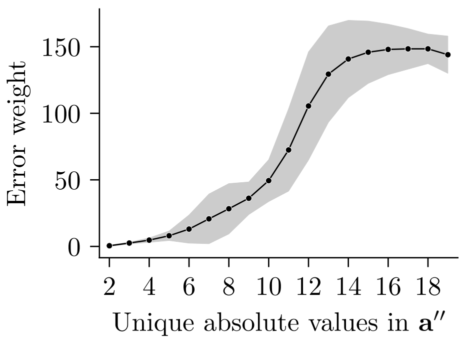
$$\mathbf{p}[i] = \left|\{i \in \{0, \ldots, 127\} : |\mathbf{a}''[i]| > |\mathbf{a}''[j]|\,, \text{ for all } 0 \le j < i\}\right|.$$

Similarly, on input pk, the noisy oracle $\mathcal{O}^{\varepsilon}_{\mathrm{S(FP)}}$ also outputs a tuple $(\mathbf{p}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2)$, where $\mathbf{p}$ is the number of times the peak changed but with some noise dependent on ε. Therefore, the noisy oracle $\mathcal{O}^{\varepsilon}_{\mathrm{S(FP)}}$ behaves as follows. First call the non-noisy oracle $(\mathbf{p}', \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2) \leftarrow \mathcal{O}_{\mathrm{S(FP)}}$. Then compute

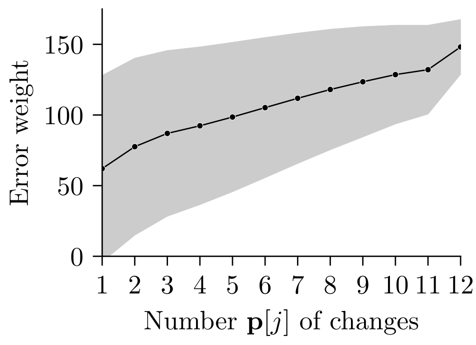
$$\mathbf{p}[j] = \sum_{i=1}^{\mathbf{p}'[j]} (1 - B_i) + \sum_{i=\mathbf{p}'[j]+1}^{n_2} B_i,$$

where each B_i is a Bernoulli random variable with ε probability of being 1. That is, each B_i introduces an error of an entry being wrongly classified as a change in the variable containing the current peak. Oracle $\mathcal{O}^{\varepsilon}_{\mathrm{S(FP)}}$ then returns $(\mathbf{p}, \mathbf{m}, \mathbf{r}_1, \mathbf{r}_2)$.

The Likelihoods Vectors. Remember that, in Section 5.4, we developed a key recovering algorithm that uses the information from oracle $\mathcal{O}^{\varepsilon}_{\mathrm{FP}}$, which outputs whether each block of RM codewords was successfully decoded or not. Furthermore, remember that the main argument for the algorithm to work is: when there is a decoding failure in the j-th RM block, it means that the weight of $\hat{\mathbf{e}}_j$ must be high. The difficulty when attacking the shuffled implementation of FindPeaks is that we cannot accurately detect when there is a RM error in the block. In particular, the shuffling makes does not let us compute the FindPeaks outputs, and therefore we cannot compare their 7 least significant bits with what is expected when the RM decoding successfully decoded all errors in each block.



(a) Correlation between error weight and the number of unique absolute values in $\mathbf{a}''$.



(b) The correlation between $\mathbf{p}[j]$ and the weight of $\hat{\mathbf{e}}_j$.

Figure 19: The correlations observed that allow key recovery, considering parameters for 128 bits of security. The grey error bands illustrate the standard deviation.

Let us then work with the information given by $\mathcal{O}^{\varepsilon}_{\text{S(FP)}}$, which is the number of times the variable computing the peak changes in a shuffled implementation of FindPeaks. We can point the following two conditions for one such number $\mathbf{p}[j]$ to be high.

1. The first one is when the peak is located in the end of the vector $\mathbf{a}''$, so there is more opportunity for the current peak variable to be changed during the computation.

2. The second condition is when there is a large number of unique entries in the array $\mathbf{a}''$. This makes it more likely that the variable with the current peak will take different values before finding the peak.

Notice that, because of the shuffling, the position of the peak is independent of the error weight, which means the first condition is not useful for us. However, the second condition can be exploited, as we explain next.

Remember that the $\mathbf{a}''$ vector, which is the output of the Hadamard transform, can be seen as encoding the distance from the corrupted RM codeword and each possible valid codeword. Intuitively, then, when the noise is high, the variance within the absolute value of the entries in the $\mathbf{a}''$ vector should also be higher. Figure 19a shows that this appears to indeed be the case, that is, the number of unique values in $\mathbf{a}''$ is positively correlated the error weight.

Now, to attack the shuffled implementation of FindPeaks using oracle $\mathcal{O}^{\varepsilon}_{\text{S(FP)}}$, we need to verify that the following information flow is valid. From $\mathbf{p}[j]$ we expect to get information on the number of unique values in $\mathbf{a}''$ of the 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 19b demonstrates this correlation, where we can see that larger values of $\mathbf{p}[j]$, on average, corresponds 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 suggests that, if we want to use this information for key recovery, we may need a large number of queries to get rid of the noise.

With this correlation, we conclude 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 notice that this is exactly the same property that the output $\mathbf{d}$ of the oracle $\mathcal{O}^{\varepsilon}_{\text{FP}}$, for the non-shuffled FindPeaks, also exhibited. Therefore, we can compute the likelihoods vectors in the exact same way we did in Section 5.4.

Formally, we first make a number η queries to $\mathcal{O}^{\varepsilon}_{\text{S(FP)}}$, obtaining $\left(\mathbf{p}^k, \mathbf{r}_1^k, \mathbf{r}_2^k\right)$, for $k = 1$

to η. Then we build the auxiliary vectors $\mathbf{w}_1$ and $\mathbf{w}_2$

$$\mathbf{w}_\alpha = \textsc{Build}_{\mathbf{w}}\Big(\big(\mathbf{r}_\alpha^k, \mathbf{p}^k\big)_{k=1}^{\eta}\Big), \text{ for } \alpha = 1 \text{ and } 2,$$

where $\textsc{Build}_{\mathbf{w}}$ is the procedure described in Algorithm 5. Finally, we compute the likelihoods vectors as described in the special key recovery procedure described in Algorithm 7, which iterates over parameters ν and ℓ just as we do for attacking the non-shuffled implementation of FindPeaks.

Key Recovery. Figure 20 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 queries needed for the attack is about a hundred times larger than what is needed to attack the non-shuffled implementation of FindPeaks. Furthermore, we can see that even a very small amount of noise 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. Thus, we observe that the shuffling countermeasure for the FindPeaks operation significantly increases the number of queries compared to shuffling for the ExpandAndSum operation, which can be defeated with only a few thousand queries.

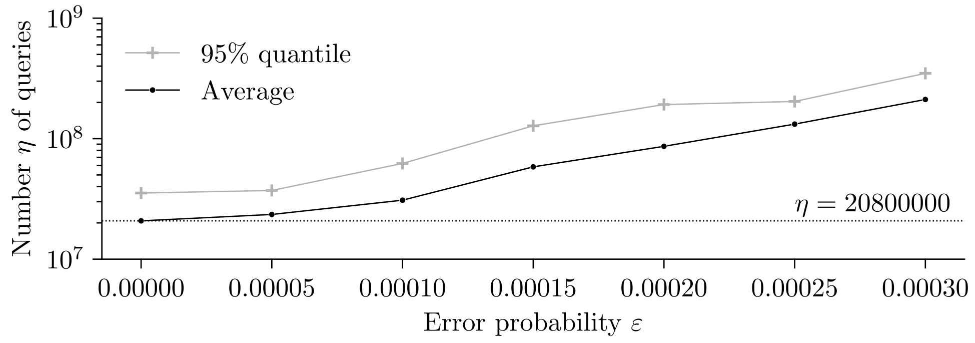


Figure 20: The number of queries to $\mathcal{O}^{\varepsilon}_{\text{S(FP)}}$ needed to recover the key for different values of ε, considering parameters for 128 bits of security.

6.3 On Attacking Masked Implementations

To the best of knowledge, we are not aware of masking schemes for HQC KEM. Thus, we will not be able to provide concrete details on how our attack applies to the masked implementations. Nevertheless, we consider certain possibilities and assumptions on how the targeted operations of our attack will be implemented in a probable masked implementation. We assume that the codeword $\mathbf{c}'$ is computed as boolean shares, where each bit of the codeword is split into multiple boolean shares. We assume that the ExpandAndSum operation is performed over the boolean shares of $\mathbf{c}'$, and still involves bitwise manipultion of the individual shares.

If the attacker has access to a clone device, the attacker can train on the individual shares with knowledge of the shares, and can simply recover all the shares of the codeword one bit at a time, to perform full key recovery. However, in absence of a clone device, the attacker has to adopt unsupervised approaches to build distinguishers for the single bits of the codeword.

6.4 Building Templates directly on Target Device (Masked ExpandAndSum Operation)

For the attacks on masking countermeasure, one of the common approach is to combine the time samples in which the mask and masked value are being processed. We assume that the attacker has partial knowledge on the time window where the shares are being manipulated. One of the main issue with combining time samples is to identify the precise time samples, and could be further complicated by desynchronization of the trace.

Recently, it has been reported that the approach based on Deep Learning (DL) allows to combine these samples automatically during the training and can be used to target masking implementation directly [MDP19]. Thus, in this work, we adopt similar approach, namely, we used Convolutional Neural Networks (CNNs), commonly used learning paradigm in DL.

CNNs can be commonly considered to be consisting of three types of layers: convolutional, pooling and fully connected layers. The convolution layer computes the output of neurons connected to local regions in the input, where each is computing a dot product between the weights and input within a small region (convolution kernel). Next is the pooling layer, in which number of features is downsampled by combining the outputs from a group of neurons at the previous layer into a single neuron in the next layer (usually by taking the max or the average). Lastly, in fully connected layers, every neuron in one layer will be connected to every neuron in another layer.

Since it is not the aim of this study, we did not investigate further into optimization of the CNN model. For evaluating the attack, we use the modified version of CNN model from [PSB+18], which can be observed in Figure 21. The size of the batch is equal to 200 and we use RMSprop optimizer with a initial learning rate of 10^{-5} and 200 epochs for training (in some cases, 20 epochs is sufficient, however, for bits with low leakage, more epochs are required and fine tuning is required to avoid overfitting).

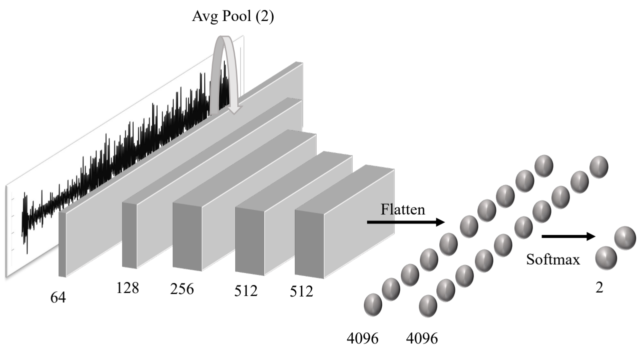
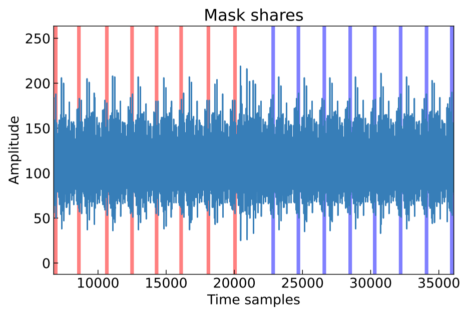


Figure 21: Architecture of the modified CNN used for the attack on masking experiment

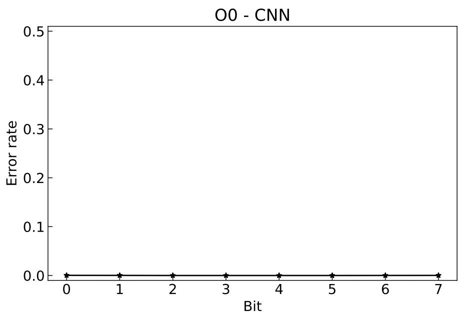
For O0 implementation, we collected $50,000$ raw non-averaged traces with simple masking enabled. For training, validation and testing, we further split the trace into $30,000$, $10,000$ and $10,000$ traces respectively. In Figure 22a, we highlight the regions where the mask and masked share are being processed for one byte.

From the figure, we could observe that for O0 implementation, the pattern is quite regular and as such shares recombining might be easier. Thus, we use the knowledge of these regions when doing the training and the attack on each bit (i.e., combining both region for masked bit and mask value for bit i, $i \in \{0, ..., 7\}$.

In Figure 22b, we plot the results on CNN targeting masking implementation on O0 implementation. We observe that we can recover all the bits with almost 100% accuracy.



(a) Regions of Interest for O0 (red regions indicate different masked bits and blue regions indicate different mask values)



(b) Error rate for O0 masked implementation on different target bits

Figure 22: Performance of CNN when attacking masked O0 implementation

This makes sense, since the pattern are regular and they have high leakage. For CNN, we requires less than 20 epoch for training.

As such for this experiment targeting masking implementation, we have shown that the error rates are low enough. Thus, we show that even for masking implementation, we could achieve sufficient error rate required for the recovery for most of the bits. We do not present an analysis for the masked FindPeaks operation, as we do not have sufficient information on how the masked FindPeaks operation will be implemented.

7 Countermeasures and Concluding Remarks

We start by discussing potential countermeasures against our proposed attacks. Firstly, we show in Section 6 that shuffling the ExpandAndSum operation is not sufficient to prevent the attack, as knowledge of the Hammming Weight (HW) of the individual RM codewords is sufficient for key recovery. However, we observe that performing a high-level shuffle in the order of processing of the n_1 RM codewords, ensures that an attacker cannot obtain targeted information about any of the n_1 RM codewords, from both the ExpandAndSum and FindPeaks operations. While this appears to prevent key recovery, we leave concrete analysis of the same for future work. Moreover, a combination of masking and shuffling could also offer protection against key recovery, however a concrete analysis of the same is also left for future work.

In this paper, we demonstrate the first chosen-ciphertext attack on HQC KEM that is conducted using valid ciphertexts. We demonstrate how side-channel analysis of two operations of the Reed-Muller decoder, namely the ExpandAndSum and the FindPeaks operations, allows us to recover the HQC secret key using only valid ciphertexts. We present novel key recovery attacks exploiting leakage from both the operations to recover the key with 100% success rate, albeit with varying number of traces depending upon the type of exploited leakage. All our experiments are performed on the open-source implementation of HQC KEM taken from the *pqm4* library, with our attacks validated using both the power and EM side-channel. We also demonstrate novel key recovery attacks which also work on shuffled implementations, thereby demonstrating that shuffling increases the attacker's complexity for key recovery, but does not concretely prevent the attack. Similarly, we also discuss the applicability of our attack to masking countermeasures. To the best of our knowledge, we are not aware of a side-channel protected design for HQC KEM, and thus we believe our work stresses the need towards more research on secure and efficient masking and hiding countermeasures for HQC KEM.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology, 2022.

[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.

[ABB+22] 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, 2022. https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.

[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.

[Dum91] Ilya Dumer. On minimum distance decoding of linear codes. In *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pages 50–52, 1991.

[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.

[GJL15] Qian Guo, Thomas Johansson, and Carl Löndahl. A new algorithm for solving Ring-LPN with a reducible polynomial. *IEEE Transactions on Information Theory*, 61(11):6204–6212, 2015.

[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.

[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.

[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.

[LJS$^+$16] Carl Löndahl, Thomas Johansson, Masoumeh Koochak Shooshtari, Mahmoud Ahmadian-Attari, and Mohammad Reza Aref. Squaring attacks on McEliece public-key cryptosystems using quasi-cyclic codes of even dimension. *Designs, Codes and Cryptography*, 80(2):359–377, 2016.

[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.

[Mac67] J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pages 281–297. University of California Los Angeles LA USA, 1967.

[MDP19] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. A comprehensive study of deep learning for side-channel analysis. *IACR Cryptol. ePrint Arch.*, page 439, 2019.

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[PSB$^+$18] Emmanuel Prouff, Rémi Strullu, Ryad Benadjila, Eleonora Cagli, and Cécile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. *IACR Cryptol. ePrint Arch.*, page 53, 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.

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

[Sen11] Nicolas Sendrier. Decoding one out of many. In *International Workshop on Post-Quantum Cryptography*, pages 51–67. Springer, 2011.

[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.

[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.

[Ste88] Jacques Stern. A method for finding codewords of small weight. In *International Colloquium on Coding Theory and Applications*, pages 106–113. Springer, 1988.

[TS16] Rodolfo Canto Torres and Nicolas Sendrier. Analysis of information set decoding for a sub-linear error weight. In *Post-Quantum Cryptography*, pages 144–161. Springer, 2016.

[WTBBG19] Guillaume Wafo-Tapa, Slim Bettaieb, Loic Bidoux, and Philippe Gaborit. A practicable timing attack against HQC and its countermeasure. Cryptology ePrint Archive, Report 2019/909, 2019. https://eprint.iacr.org/2019/909.