Simple Power Analysis Assisted Chosen Cipher-Text Attack on ML-KEM

Alexandre Berzati[1], Andersson Calle Viera[1,2], Maya Chartouny[1,3], and David Vigilant[1]

[1] Thales DIS, France
`alexandre.berzati, andersson.calle-viera, maya.saab-chartouni, david.vigilant@thalesgroup.com`
[2] Sorbonne Université, CNRS, Inria, LIP6, F-75005 Paris, France
[3] Université Paris-Saclay, UVSQ, CNRS, Laboratoire de mathématiques de Versailles, 78000, Versailles, France

Abstract. Recent work proposed by Bernstein *et al.* (from EPRINT 2024) identified two timing attacks, KyberSlash1 and KyberSlash2, targeting ML-KEM decryption and encryption algorithms, respectively, enabling efficient recovery of secret keys. To mitigate these vulnerabilities, correctives were promptly applied across implementations. In this paper, we demonstrate a very simple side-channel-assisted power analysis attack on the patched implementations of ML-KEM. Our result showed that original timing leakage can be shifted to power consumption leakage that can be exploited on specific data. We performed a practical validation of this attack on both the standard and a shuffled implementations of ML-KEM on a Cortex-M4 platform, confirming its effectiveness. Our approach enables the recovery of the ML-KEM secret key in just 30 seconds for the standard implementation, and approximately 3 hours for the shuffled implementation, achieving a 100% success rate in both cases.

Keywords: ML-KEM · Kyber · Lattice-based cryptography · Post-quantum cryptography · Side-channel attacks · Simple power analysis

1 Introduction

Quantum computers represent a major threat to current cryptographic systems. Assuming that powerful enough computers will be available in the future, conventional public key algorithms such as RSA [RSA78] and Diffie-Hellman key exchange [DH76] can be broken by Shor's [Sho94] quantum algorithm. This has led to the development of post-quantum cryptography, which aims to create quantum-resistant algorithms. The National Institute of Standards and Technology (NIST) has standardized four post-quantum cryptography (PQC) algorithms. In the key encapsulation method (KEM) category, ML-KEM [NIS23], derived from Kyber [SAB+22], has been selected as the primary algorithm by the NIST. Its security is based on the module learning with errors problem (MLWE) [LS15].

Side-channel attacks (SCA) exploit the physical implementation of cryptographic systems, such as power consumption, electromagnetic emanation, or timing information to infer secrets. In a recent work [BBB$^+$24], Bernstein et al. reported two timing attacks on ML-KEM named "KyberSlash1" and "KyberSlash2". The first variant targets the division step in the decryption (K-PKE.Decrypt, Algorithm 1), which directly reveals information about the secret key. The second variant targets the division operation in the encryption (K-PKE.Encrypt, Algorithm 1), leaking details about the ciphertext and enabling the construction of a plaintext-checking oracle during decapsulation, allowing key recovery. The results demonstrated that the secret keys could be recovered within a few hours using KyberSlash1 and a few minutes using KyberSlash2. In response to these vulnerabilities, the reference implementation of ML-KEM was quickly updated to counter the KyberSlash attacks. Then, a majority of other open-source implementations integrated the patch as well. Even if the reference implementation is not expected to thwart side-channel analysis other than timing, we show in this paper that these modifications enable a simple side-channel-assisted power analysis attack. The attack presented here remains really practical and achieves full key recovery in approximately 30 seconds.

Our contribution. In this article, we will present new results. In fact,

- We analyzed the implementation of a part of the ML-KEM decapsulation procedure following the KyberSlash attacks and confirmed that it results in data-dependent leakage that can be exploited.
- We did a practical demonstration of a chosen ciphertext attack, assisted by simple power analysis, on ML-KEM versions 512, 768, and 1024.
- We also present a detailed attack strategy for a shuffled version of ML-KEM, and its practical demonstration.
- Detailed notebooks are provided. They outline the end-to-end attack that can be reproduced. We also provided a dataset of messages for those who do not have the equipment to perform trace acquisitions for the attack.
- Finally, we showcase how a simple modification to the code of the sensitive function can effectively reduce the leakage without introducing any overhead.

2 Background Information

2.1 Notation

For α an even integer (resp. odd), we define $r' := r \bmod^{\pm}\alpha$ the unique $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ (resp. $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$) such that $r' = r \mod \alpha$.
Let us define a polynomial ring $R = \mathbb{Z}[X]/(X^n + 1)$ with n a power of 2 and $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. We also define R_q^k the module of rank k whose elements are polynomials from R_q.
Polynomials from R_q are denoted by lowercase letters, e.g., $v \in R_q$. We denote matrices and vectors with bold uppercase letters and bold lowercase letters, e.g., $\mathbf{A} \in R_q^{k\times k}$ and $\mathbf{u} \in R_q^k$. Unless otherwise stated, vectors are represented

column-wise. Given a matrix $\mathbf{A}$ (resp. a vector $\mathbf{u}$), we denote by $\mathbf{A}^T$ (resp. $\mathbf{u}^T$) its transpose. We denote $\hat{f}$ the NTT representation of f.
For $i \in [0, k[$, the i-th polynomial of a vector $\mathbf{u} \in R_q^k$ will be denoted by $\mathbf{u}_i$.
For $j \in [0, n[$, the j-th coefficient of a polynomial $v \in R_q$ will be denoted by $v[j]$.
$\lceil \cdot \rfloor$ denotes the function that rounds to the nearest integer.

2.2 ML-KEM

ML-KEM [NIS23] is a post-quantum key encapsulation mechanism currently being standardized by NIST. It is derived from CRYSTALS-Kyber [SAB+22], which was selected at the end of the third round of the NIST competition. This scheme is based on lattice cryptography, with its security relying on the difficulty of solving the module learning with errors (MLWE) problem [LS15]. ML-KEM offers three security levels: ML-KEM-512 (corresponding to NIST security level 1), ML-KEM-768 (level 3), and ML-KEM-1024 (level 5).

Compress and Decompress. ML-KEM uses several mechanisms to compress and decompress the size of ciphertexts. The same methods are used to map bits elements to coefficients from $\mathbb{Z}_q$ which allow to recover a message bit even after small noise is added to it. Conversely, we can also map elements from $\mathbb{Z}_q$ to bits using the same methods. Let us describe how they work.
$\texttt{Compress}_d$ lossily compresses an element from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^d}$ with $2^d < q$:

$$\begin{aligned} \texttt{Compress}_d : \mathbb{Z}_q &\longrightarrow \mathbb{Z}_{2^d} \\ x &\longmapsto \left\lceil \frac{2^d}{q} \cdot x \right\rfloor \mod 2^d. \end{aligned}$$

$\texttt{Decompress}_d$ takes an element in $\mathbb{Z}_{2^d}$ and maps it to an element in $\mathbb{Z}_q$.

$$\begin{aligned} \texttt{Decompress}_d : \mathbb{Z}_{2^d} &\longrightarrow \mathbb{Z}_q \\ y &\longmapsto \left\lceil \frac{q}{2^d} \cdot y \right\rfloor. \end{aligned}$$

Algorithm 1 outlines a simplified view of the key-generation, encryption, and decryption processes of the internal public key encryption algorithm (PKE). `SampleA` is the function generating a uniformly random matrix $\hat{\mathbf{A}}$ in the NTT domain. `SampleB` samples coefficients from the centered binomial distribution.

KeyGen. In the `KeyGen` process, an LWE instance $(\hat{\mathbf{A}}, \hat{\mathbf{t}} = \hat{\mathbf{A}} \cdot \hat{\mathbf{s}} + \hat{\mathbf{e}} \mod q)$ is computed, where $\hat{\mathbf{A}}$ is uniformly generated using a seed ρ, the secret $\hat{\mathbf{s}}$ and the error $\hat{\mathbf{e}}$ are sampled from the same centered binomial distribution. The public key consists of $(\rho, \hat{\mathbf{t}})$, while the secret key is $\hat{\mathbf{s}}$.

Encryption. To encrypt a message m, a vector $\mathbf{y}$ and errors $\mathbf{e}_1$ and e_2 are generated, all drawn from a centered binomial distribution. Two LWE instances are then constructed, which (up to some details) are represented as $(\mathbf{A}^T\mathbf{y} +$

$\mathbf{e}_1, \mathbf{t}^T\mathbf{y} + e_2)$. Next, μ is computed as $\lceil \frac{q}{2} m \rfloor$ and added to the right-hand side of the pair. Finally, the resulting pair $(\mathbf{u}, v)$ is compressed into $(\mathbf{c}_1, c_2)$.

Algorithm 1 K-PKE

```
 1: K-PKE.KeyGen(d)
 2:     $(\rho, \sigma) \leftarrow G(d||k)$
 3:     $\hat{\mathbf{A}} \leftarrow \texttt{SampleA}(\rho)$
 4:     $\mathbf{s} \leftarrow \texttt{SampleB}(\sigma, coins_0)$
 5:     $\mathbf{e} \leftarrow \texttt{SampleB}(\sigma, coins_1)$
 6:     $\hat{\mathbf{s}} \leftarrow \mathrm{NTT}(\mathbf{s})$
 7:     $\hat{\mathbf{e}} \leftarrow \mathrm{NTT}(\mathbf{e})$
 8:     $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
 9: return $\big(\mathrm{ek}_{\mathrm{PKE}} = (\hat{\mathbf{t}}||\rho), \mathrm{dk}_{\mathrm{PKE}} = (\hat{\mathbf{s}})\big)$
10: K-PKE.Encrypt($\mathrm{ek}_{\mathrm{PKE}}, m, r$)
11:     $\hat{\mathbf{A}} \leftarrow \texttt{ExpandA}(\rho)$
12:     $\mathbf{y} \leftarrow \texttt{SampleB}(r, coins_2)$
13:     $\mathbf{e}_1 \leftarrow \texttt{SampleB}(r, coins_3)$
14:     $e_2 \leftarrow \texttt{SampleB}(r, coins_4)$
15:     $\hat{\mathbf{y}} \leftarrow \mathrm{NTT}(\mathbf{y})$
16:     $\mathbf{u} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{y}}) + \mathbf{e}_1$
17:     $\mu \leftarrow \texttt{Decompress}_1(\texttt{ByteDecode}_1(m)) = \lceil \frac{q}{2} m \rfloor$
18:     $v \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{y}}) + e_2 + \mu$
19:     $\mathbf{c}_1 \leftarrow \texttt{Compress}_{d_u}(\mathbf{u})$
20:     $c_2 \leftarrow \texttt{Compress}_{d_v}(v)$
21: return $\mathbf{c} = (\mathbf{c}_1||c_2)$
22: K-PKE.Decrypt($\mathrm{dk}_{\mathrm{PKE}}, \mathbf{c}$)
23:     $\mathbf{u}' \leftarrow \texttt{Decompress}_{d_u}(\mathbf{c}_1)$
24:     $v' \leftarrow \texttt{Decompress}_{d_v}(c_2)$
25:     $w \leftarrow v' - \mathrm{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathrm{NTT}(\mathbf{u}'))$
26:     $m \leftarrow \texttt{Compress}_1(w)$
27: return $m$
```

Decryption. To decrypt, the values $(\mathbf{u}', v')$ are first recovered from $(\mathbf{c}_1, c_2)$, with $\Delta_{\mathbf{u}} = \mathbf{u}' - \mathbf{u}$ and $\Delta_v = v' - v$ representing the differences between the received and original values. The decryption procedure then computes $w = v' - s^T\mathbf{u}'$, serving as an approximation of the message polynomial. Specifically, we have:

$$\begin{aligned}
w &= v' - \mathbf{s}^T\mathbf{u}' \\
&= v + \Delta_v - \mathbf{s}^T(\mathbf{u} + \Delta_{\mathbf{u}}) \\
&= \mathbf{t}^T\mathbf{y} + e_2 + \mu + \Delta_v - \mathbf{s}^T(\mathbf{A}^T\mathbf{y} + \mathbf{e}_1 + \Delta_{\mathbf{u}}) \\
&= (\mathbf{A}\mathbf{s} + \mathbf{e})^T\mathbf{y} + e_2 + \mu + \Delta_v - \mathbf{s}^T(\mathbf{A}^T\mathbf{y} + \mathbf{e}_1 + \Delta_{\mathbf{u}}) \\
&= \mu + \big(\mathbf{e}^T\mathbf{y} + e_2 + \Delta_v - \mathbf{s}^T\mathbf{e}_1 - \mathbf{s}^T\Delta_{\mathbf{u}}\big) \\
&= \left\lceil \frac{q}{2} m \right\rfloor + \varepsilon
\end{aligned} \tag{1}$$

where $\varepsilon = \mathbf{e}^T\mathbf{y} + e_2 + \Delta_v - \mathbf{s}^T\mathbf{e}_1 - \mathbf{s}^T\Delta_{\mathbf{u}}$ is the noise. The approximate polynomial w is decoded into the message m, one bit at a time using the function $\texttt{Compress}_1$. Informally, it can be described as follows: if a given coefficient of the polynomial $w[i]$ is within the range $[q/4, 3q/4[$, then $m_i = 1$; otherwise, $m_i = 0$.

KEM. The CPA-secure PKE is converted into a CCA-secure KEM using the well-known Fujisaki-Okamoto (FO) transformation [FO99] and is described in Algorithm 2.

Algorithm 2 ML-KEM

```
1: ML-KEM.KeyGen($d$,$z$)
2:     $(\mathrm{ek_{PKE}}, \mathrm{dk_{PKE}}) \leftarrow$ K-PKE.KeyGen($d$)
3:     $\mathrm{ek} \leftarrow \mathrm{ek_{PKE}}$
4:     $\mathrm{dk} \leftarrow (\mathrm{dk_{PKE}} \parallel \mathrm{ek} \parallel H(\mathrm{ek}) \parallel z)$
5: return $(\mathrm{ek}, \mathrm{dk})$
6: ML-KEM.Encaps(ek,$m$)
7:     $(K, r) \leftarrow G(m \parallel H(\mathrm{ek}))$
8:     $\mathbf{c} \leftarrow$ K-PKE.Encrypt($\mathrm{ek}, m, r$)
9: return $(K, \mathbf{c})$
10: ML-KEM.Decaps(dk,$\mathbf{c}$)
11:     $m' \leftarrow$ K-PKE.Decrypt($\mathrm{dk_{PKE}}, \mathbf{c}$)
12:     $(K', r') \leftarrow$ SHA3($m' \parallel h$)
13:     $\bar{K} \leftarrow$ SHAKE256($z \parallel \mathbf{c}$)
14:     $\mathbf{c}' \leftarrow$ K-PKE.Encrypt($\mathrm{ek_{PKE}}, m', r'$)
15:     if $\mathbf{c} \neq \mathbf{c}'$ then
16:         $K' \leftarrow \bar{K}$
17: return $K'$
```

2.3 Simple Power Analysis

Side-channel attacks exploit unintended information emitted from devices activity, such as timing, electromagnetic (EM) emissions, or power consumption, to infer secret information. One of the most accessible forms of SCA, and certainly the most visually intuitive, is Simple Power Analysis (SPA). SPA leverages distinct fluctuations in power consumption, or EM traces to differentiate the operations performed by a device. Compared to other types of SCA, an SPA requires minimal equipment, is relatively little invasive, and can necessitate only a few traces to perform the attack. When dealing with cryptographic operations SPA focuses on analyzing the traces of a device to extract information about secret data.

A well-known example of SPA is the attack on the square-and-multiply algorithm, commonly used in modular exponentiation. This algorithm's sequence of squaring (bit to 0 and bit to 1) and conditional multiplication (only bit to 1) reveals patterns in the power traces depending on the bit values of the exponent. By observing these power variations, an attacker can deduce the entire secret exponent bit by bit.

Such an example demonstrates that even the simplest SCA can be remarkably powerful in recovering secret information. In the following, we will show how to use an SPA to recover the full secret ML-KEM secret key with only three traces.

2.4 Prior Work

CCA assisted with SCA. The FO transform offers guaranteed security against chosen-ciphertext attacks (CCA). However, for real-life implementations, an attacker can access side-channel information during the process, which can be used as a distinguisher to mount hybrid SCA assisted CCA. Without loss of generality, we will explain the general methodology when applied to ML-KEM-512, but the type of attack applies to the other two security levels as well.

Let us recall that the secret key vector $\mathbf{s} \in R_q^k$ is of the form:

$$\forall\, i \in [0, k[,\ \mathbf{s}_i = \mathbf{s}_i[0]X^0 + \mathbf{s}_i[1]X^1 + \cdots + \mathbf{s}_i[n-1]X^{n-1}.$$

For ML-KEM-512, we have $\eta_1 = 3$, so $\forall\, j \in [0, n[,\ \mathbf{s}_i[j] \in [-3, 3]$.

The ciphertext is of the form $c = (c_1,\, c_2)$ and in the decryption procedure we have $\mathbf{u}' = \texttt{Decompress}_{d_u}(c_1) \in R_q^k$ and $v' = \texttt{Decompress}_{d_v}(c_2) \in R_q$ where:

$$\forall i \in [0, k[,\ \mathbf{u}'_i = \mathbf{u}'_i[0]X^0 + \mathbf{u}'_i[1]X^1 + \cdots + \mathbf{u}'_i[n-1]X^{n-1}, \text{ and}$$
$$v' = v'[0]X^0 + \cdots + v'[n-1]X^{n-1}.$$

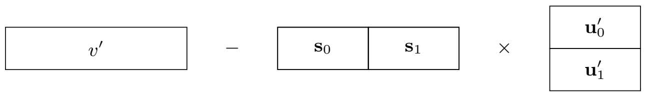
For ML-KEM-512, we have $k = 2$, i.e., $\mathbf{s} = (\mathbf{s}_0,\, \mathbf{s}_1)$ and $\mathbf{u}' = (\mathbf{u}'_0,\, \mathbf{u}'_1)$.

In the decryption procedure inside the decapsulation, we compute:

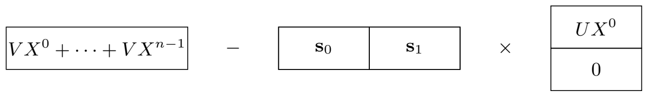
$$w = v' - \mathbf{s}^T \mathbf{u}'. \tag{2}$$

Let us denote by $V \in \mathbb{Z}_q$ and $U \in \mathbb{Z}_q$ two carefully chosen values[4] such that $\mathbf{u}' = (UX^0, 0)$ and $v' = VX^0 + \cdots + VX^{n-1}$.

Figure 1a shows a visualization of the computation of w for a normal ciphertext while Figure 1b shows the same computation for a chosen ciphertext.



(a) A normal valid ciphertext.



(b) A crafted ciphertext.

Fig. 1: Decryption equation computed when different type of ciphertext are used.

[4] U is chosen to scale the possible values of $\mathbf{s}$ just enough to be at the border $\frac{q}{4}$ or $\frac{3 \times q}{4}$. Then, adding different V allows us to distinguish the correct value.

For the second case, which typically corresponds to some type of ciphertext chosen by an attacker, we can further develop equation (2) as follows:

$$\begin{aligned} w = v' - \mathbf{s}^T\mathbf{u}' &= VX^0 + \cdots + VX^{n-1} - \mathbf{s}_0 \times UX^0 \\ &= VX^0 + \cdots + VX^{n-1} - \big(U\mathbf{s}_0[0]X^0 + \cdots + U\mathbf{s}_0[n-1]X^{n-1}\big) \\ &= (V - U\mathbf{s}_0[0])X^0 + \cdots + (V - U\mathbf{s}_0[n-1])X^{n-1}. \end{aligned} \tag{3}$$

After decoding this quantity onto the message bits, each coefficient can only take one of two values, 0 or 1, depending on the values of the message polynomial. More precisely, for $j \in [0, n[$, the message coefficient $w[j] = V - U\mathbf{s}_0[j]$ is decoded into the message bit $m[j] = \texttt{Compress}_1(w[j])$. Based on its possible values, we are able to infer two conditions on the coefficients of the secret polynomial $\mathbf{s}_0$:

– If $m[j] = 1$ we know that $(V - U\mathbf{s}_0[j]) \in [q/4, 3q/4[$.
– If $m[j] = 0$ we know that $(V - U\mathbf{s}_0[j]) \in [0,\ q/4[\cup[3q/4,\ q[$.

Thus, the knowledge of the value of the bit $m[j]$ for different tuples U, V can be used as a binary distinguisher for the possible values of $\mathbf{s}_0[j] \in [-3, 3]$. For successive carefully chosen ciphertext, this method allows to recover the entire secret key. Different variants of this attack exist in the literature [BDH+19, QCZ+21, XPR+22, RBRC22, SCZ+23, RRD+23, TUX+23, RRCB20]. The main difference being the number of bits of the message that can be recovered per query to the decapsulation procedure and the operation targeted by the attack.

KyberSlash. In a recent work [BBB+24], Bernstein et al. proposed a timing attack on ML-KEM. The core idea is to exploit non constant time divisions on certain platforms. They proposed two variants, “KyberSlash1” and “KyberSlash2”.

The first variant targets the division step in the decryption (K-PKE.Decrypt 1):

```
t = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
```

Since `t` can be a coefficient of the message m, leakage on this value can be exploited by an attacker to gain sensitive information. By crafting specific ciphertexts, this variable execution time can be used to recover the entire secret key.

The second variant targets the division step in the encryption (K-PKE.Encrypt1):

```
t[j] = ((((uint16_t)u << 4) + KYBER_Q/2)/KYBER_Q) & 15;
```

For this attack, timing leakages reveal details about the ciphertext and enable the construction of a plaintext-checking oracle during decapsulation, allowing full key recovery.

The results demonstrated that the secret keys could be recovered reliably within a few hours using KyberSlash1 and a few minutes using KyberSlash2. To counter this attack, dedicated modifications were made to the reference implementation of the ML-KEM. However, as we will show later, these modifications enable a simple side-channel attack assisted with chosen ciphertext attack.

3 New Theoretical Leakage

In this section, we highlight a new potential leakage arising from the new implementation of the polynomial compression in the decryption after “KyberSlash1”. We briefly start by explaining the differences between the specification of the compression routine and its implementation. Then, we present the post-”KyberSlash1” implementation, followed by an analysis of the newly generated leakage.

3.1 Difference Between Specification and Implementation

The ML-KEM specification, Algorithm 2, uses a generic compression function as defined in Section 2.2. From an algorithmic perspective, this allows to set parameters d_u, d_v, and 1 to compress $\mathbf{u}$, v, and the message polynomial w, respectively. However, in practice, the implementation separates these cases into: `polyvec_compress` for compressing the vector $\mathbf{u}$, `poly_compress` for compressing the polynomial v, and, `poly_tomsg` for compressing the message polynomial w. Furthermore, rounding is performed on coefficients in $]-1664, 1664]$, meaning:

- $m[j] = 1$ if $w[j] \in \,]-1664, -832[\, \cup \,[832, 1664[\,$.
- $m[j] = 0$ if $w[j] \in [-832, 832[\,$.

In the next section, we will focus on the `poly_tomsg` implementation to describe the new attack path.

3.2 Post-KyberSlash `poly_tomsg` Implementation

```c
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES],
                    const poly *a){
  unsigned int i,j;
  uint32_t t;
  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t  = a->coeffs[8*i+j];
      t <<= 1;
      t += 1665;
      t *= 80635;
      t >>= 28;
      t &= 1;
      msg[i] |= t << j;
    }
  }
}
```

Fig. 2: C Code of the new `poly_tomsg` implementation.

The KyberSlash attacks [BBB+24] led the authors of the reference implementation to release a patched version[5] which became the new standard across multiple implementations such as [KSSW22, KPR+, BDK+, KLJJ]. Figure 2 presents the updated implementation of the `poly_tomsg` function.

One of the modifications addressed the critical division by q, which on some platforms, was not constant-time and therefore vulnerable to timing attacks, as highlighted in ”KyberSlash1”. To resolve this, the updated implementation adopts an approach inspired by Barrett reduction, i.e., approximating the division by q using a precomputed constant. Specifically, for a given a, the division $\frac{a}{q}$ is approximated by computing $\frac{(a \times x)}{2^s}$, where $x = \left\lceil \frac{2^s}{q} \right\rfloor$. If $\frac{x}{2^s}$is a sufficiently accurate approximation of $\frac{1}{q}$, the results remain identical. This method offers two significant advantages. First, it avoids the need to normalize values back to the positive range $\mathbb{Z}_q$.Second, it replaces costly and potentially insecure division operations with efficient multiplications and bit-shifts, which are constant-time and thus secure.

In the patched ML-KEM implementation, the authors chose $s = 28$ resulting in $x = \left\lceil \frac{2^{28}}{q} \right\rfloor = 80635$, as shown line 13 of Figure 2.

3.3 Newly Generated Leakage

Our analysis started by observing that from Figure 2, the decoding of the message involves a lot of arithmetic operations involving signed and unsigned data. Since q is relatively small, we decided to enumerate, for all the possible values of a message, the Hamming Weights (HW) resulting from each C step of `poly_tomsg`. This result is illustrated in Figure 3.

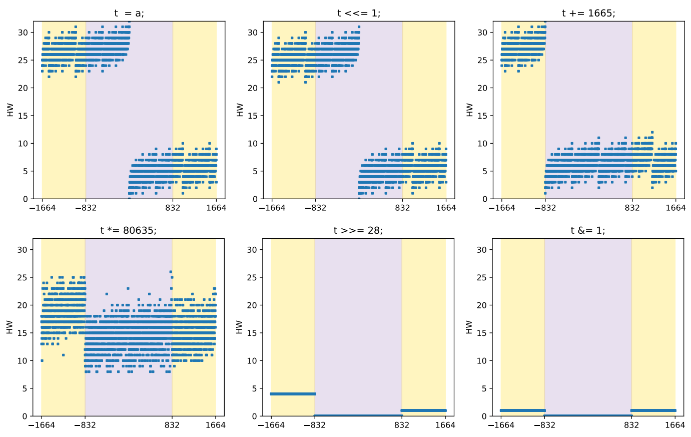


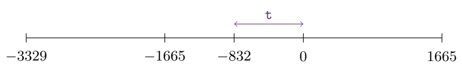
Fig. 3: HW evolution given the steps of the `poly_tomsg` function, (🟪: coefficients rounded to 0, 🟨: coefficients rounded to 1).

[5] https://github.com/pq-crystals/kyber/commit/bc8e640727b5178eb1c65867d6ba6599b3ad88e5

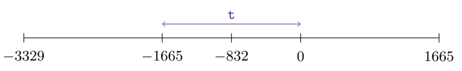
First, we observe two distinct classes of HW based on whether the coefficient is negative or positive, which is expected due to the two's complement representation of negative values. Then, shifting right by 1 doesn't produce significative change in the HW. Next, when 1665 is added to the coefficient, a noticeable difference in HW is visible between negative coefficients in the range of $]-1664, -832[$ and all the remaining coefficients. After multiplying each coefficient by 80635, a slight difference remains visible between these two classes. Finally, extracting the bit 28 also produces difference in HW which is expected since we want to recover a bit sign from coefficients rounded to 0 and 1.

In the following, we have chosen to focus the analysis on negative messages since it allows to exhibit two classes with strong difference in HW. The first class contains coefficients of the message in the range $[-832, 0[$ and the second class contains coefficients in the range $]-1664, -832[$.

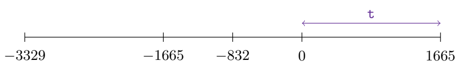
Figure 4 illustrates the effect of shifting by 1 and adding 1665 on the coefficient `t` for the first class. Similarly, Figure 5 shows this effect for the second class.



(a) Initial value of `t`.

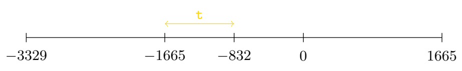


(b) Result of `t` after shift by 1.

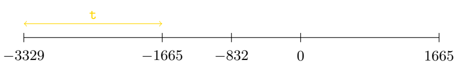


(c) Result of `t` after addition by 1665.

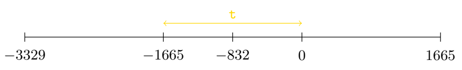
Fig. 4: Representation of steps in lines 11 and 12 from Fig. 2 for $\texttt{t} \in [-832, 0[$.



(a) Initial value of `t`.



(b) Result of `t` after shift by 1.



(c) Result of `t` after addition by 1665.

Fig. 5: Representation of steps in lines 11 and 12 from Fig. 2 for $\texttt{t} \in]-1664, -832[$.

For the first case, the coefficients initially in the $[-832, 0[$ are transformed into positive values within the range $]0, 1664[$ at the end of the addition by 1665. However, for the second case, after the addition, the coefficients in the range $]-1664, -832[$ are transformed into negative values within the range $]-1664, 0[$.

Consequently, the clear HW difference between the two classes is explained because in the first class the negative coefficients become positive and in the second class the negative coefficients remain negative. This clear difference in HW from negative values rounded to 0 and rounded to 1 could potentially be observable on side channel traces, where either power consumption or electromagnetic emissions are directly linked to the HW of values.

This work builds upon the idea presented by Tosun et al. [TMS24], who first investigated leakage vulnerabilities arising from central reduction operations in lattice-based cryptographic schemes. Specifically, they exploit the differences in Hamming weight between coefficients in the range $\left[-\frac{q}{2}, 0\right[$ and those in the range $\left[0, \frac{q}{2}\right]$, which produces two distinct classes of coefficients. We extend this observation on a specific implementation. We highlight the leakage amplification caused by the transition from negative values in the range $\left[-\frac{q}{2}, 0\right[$ into a positive value in the range $\left[0, \frac{q}{2}\right]$ over a single instruction. As we will see in the sequel, this unique targeting leads to a simple and efficient attack.

4 Exploiting the New Leakage

Having established a clear HW difference between the two classes, in Section 4.3 we first perform a t-test to confirm the difference between negative coefficients rounded to 0 and negative coefficients rounded to 1. We then demonstrate its practical exploitation on the `poly_tomsg` function. As seen in Section 2.4, side-channel information can be used as an oracle to mount hybrid SCA assisted CCA targeting the decapsulation procedure.

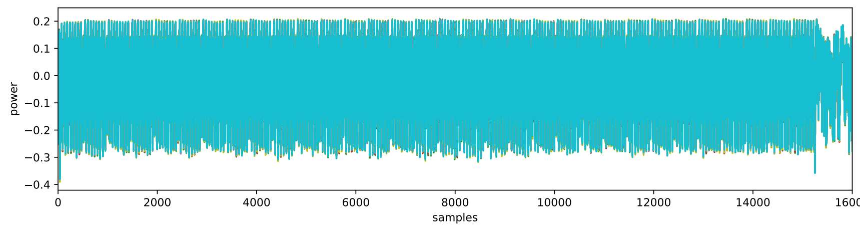
4.1 Experimental Setup

The evaluation was done on a ChipWhisperer-Lite 32-bit [OC14], with an STM32F3 micro-controller. We chose this target as it is often used in academic research targeting embedded systems. We used the ChipWhisperer built-in FPGA to perform the acquisition of the power consumption traces at 4 samples per cycle. We targeted the C reference implementation from [BDK$^+$] as it is the default one that serves as a base layer for most of the other implementations. We compiled the ML-KEM-512 version using gcc-arm cross-compiler arm-none-eabi-gcc 12.2.1, with the option -Os. The trace analysis and statistical testing is done in Python, using the "numpy" [HMvdW$^+$20] and "scipy" [VGO$^+$20] packages. All the materials used to capture and process the traces, as well as the complete attack, will soon be available as Jupyter notebooks.

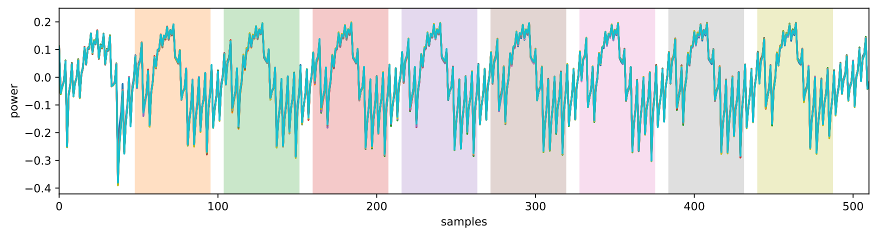
4.2 Identifying Operation

The first step is to precisely identify the location where the 256 coefficients are processed within the `poly_tomsg` function. In a non-profiled setting, detecting clear repeating patterns in the power consumption trace is helpful for building an attack as straightforward as possible.

In Figure 6a, we can observe the same repeating pattern occurring 32 times, each one spanning around 500 samples. Those patterns should correspond to the `for` loop line 5 of the C code in Figure 2. Figure 6b corresponds to a zoomed view on the first 510 samples. We can see repeating patterns, specifically 8 of them, which have been highlighted for better visibility. Looking at the C code in Figure 2 allows to identify those patterns to the second `for` loop line 7.



(a) 10 traces for complete `poly_tomsg` execution on 16000 samples.



(b) 10 traces for `poly_tomsg` execution zoomed on the 8 first coefficients.

Fig. 6: 10 power consumption traces for random inputs of `poly_tomsg`.

Each of the identified colored patterns correspond to the execution of the assembly code snippet shown in Figure 7.

```
...
ldrsh.w  r3, [r4], #2             ; t  = a->coeffs[8*i+j];
ldrb.w   ip, [r0]
lsls     r3, r3, #1               ; t <<= 1;
addw     r3, r3, #1665            ; t += 1665;
muls     r3, r7                   ; t *= 80635;
ubfx     r3, r3, #28, #1          ; t >>= 28; and t &= 1;
lsls     r3, r1
adds     r1, #1
orr.w    r3, r3, ip
cmp      r1, #8
strb     r3, [r0, #0]
...
```

Fig. 7: Assembly code on Cortex-M4.

According to our analysis in Section 3.3, the instructions that should provide side-channel information should be the `addw` and perhaps the `muls`. Prior work often exploits the leakage produced from the right shift by 28 but in our case due to the the compiler optimization flag -Os, this shift and the next and are actually merged into an `ubfx` instruction. The direct extraction of the bit 28 could result in less visible leakage in this case. Having clearly identified the targeted operation, we can now start the attack.

4.3 Building Reference Means

In all the rest, we consider $i \in [0, k[$ and $j \in [0, n[$. Following our previous notations, during the decapsulation algorithm, within the `poly_tomsg`, we compute $w = v' - \mathbf{s}^T\mathbf{u}'$ as described in Equation 3.
Our objective is to determine two sets of U and V coefficients such that $w_j = V - U\mathbf{s}_i[j]$ is negative and has a specific rounding regardless of the value of the secret in $[\![-\eta_1, \eta_1]\!]$. For the first set we want all the message's coefficients to be rounded to 0, which we will refer to as $\mathcal{D}_0$. For the other set, we want all the message's coefficients to be rounded to 1 and it will be denoted by $\mathcal{D}_1$.

For $\mathcal{D}_0$, we set V as the center of the interval $[-832, 0[$, i.e., $V = -416$. For U, we must select a value that ensures w_i remains within $[-832, 0[$ regardless of the secret's value. Therefore, we choose U such that $-832 < -416 + \mathbf{s}_i[j]U < 0$, i.e., $|U| < 416 \cdot \dfrac{1}{\eta_1}$. For our attack we restrict the values to $0 < U < 416 \cdot \frac{1}{\eta_1}$.

Similarly, for $\mathcal{D}_1$, we set $V = -1248$ as the center of the interval $]-1664, -832[$, and we also choose U such that $0 < U < 416 \cdot \frac{1}{\eta_1}$.

Note that the chosen values of U and V for $\mathcal{D}_0$ and $\mathcal{D}_1$ correspond to their decompressed forms. However, the compressed representations of U and V must be sent to the decapsulation algorithm, where they will be decompressed to match the specified values. In other words, we are looking for a y such that:

$$0 < \texttt{Decompress}_{d_u}(y) = U < 416 \cdot \frac{1}{\eta_1}.$$

Table 1 summarizes the possible ciphertexts to build $\mathcal{D}_0$ and $\mathcal{D}_1$ depending on the security version of ML-KEM.

	$\mathcal{D}_0$		$\mathcal{D}_1$	
	U max	V	U max	V
ML-KEM-512	137	-416	137	-1248
ML-KEM-768	205	-416	205	-1248
ML-KEM-1024	206	-416	206	-1248

Table 1: Ciphertexts used for the attack on ML-KEM.

Based on Table 1, for ML-KEM-512, we have $137 = \texttt{Decompress}_{10}(42)$; for ML-KEM-768, we have $205 = \texttt{Decompress}_{10}(63)$; and for ML-KEM-1024, we have $206 = \texttt{Decompress}_{11}(127)$.

T-test between values. To determine if there is a difference between the class $\mathcal{D}_0$ and $\mathcal{D}_1$ we performed a *specific t-test* as specified in [GJJR11, SM15]. We collected 42 power consumption traces on the ChipWhisperer for both classes. Figure 8 represents 10 traces among the 42 for each class, together with the corresponding t-test result.

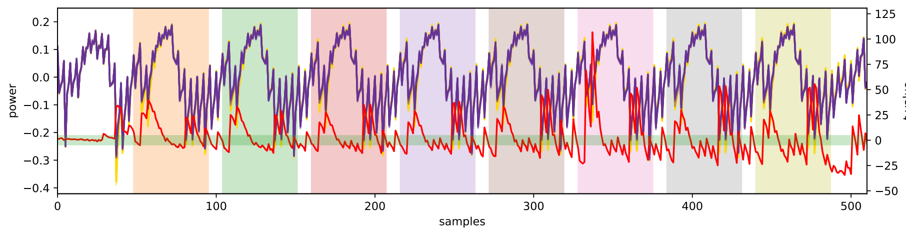


Fig. 8: 10 traces for each class and t-test (▨: coefficients from $\mathcal{D}_0$, ▨: coefficients from $\mathcal{D}_1$, 🟥: t-test result).

Here, we can see that each pattern shows a significant t-value at the beginning and at the end of each window, indicating potentially exploitable leakage.

Figure 9 highlights the 57 samples corresponding to the 256 coefficients from the `poly_tomsg` function, superposed in the same figure. In addition to the t-test results, this confirms clear differences between the traces of coefficients from $\mathcal{D}_0$ and coefficients from $\mathcal{D}_1$, specifically between the samples 5 and 12.

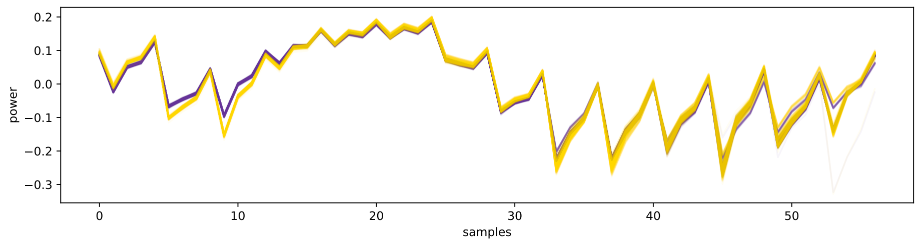


Fig. 9: Samples of the 256 coefficients of `poly_tomsg` superposed (▨: coefficients from $\mathcal{D}_0$, ▨: coefficients from $\mathcal{D}_1$).

Therefore, for each dataset we compute the mean between the samples 5 and 12, denoted as $\mathcal{M}_0$ and $\mathcal{M}_1$. These respective means will be used as references to test for coefficients rounded to 0 and for coefficients rounded to 1 respectively.

4.4 Recovering the Secret Key

With the reference means $\mathcal{M}_0$ and $\mathcal{M}_1$ established for the datasets $\mathcal{D}_0$ and $\mathcal{D}_1$, respectively, we can now exploit crafted ciphertexts to determine whether each coefficient of the message will be rounded to 0 or 1. This distinguisher provides direct information on the secret key. In fact, we can adaptively select malicious ciphertexts to partition the set of possible values of the secret key coefficient. Accumulating a sufficient number of these queries allows to recover the exact secret key coefficient.

Choice of U and V. We need to find malicious ciphertexts that provide information on the secret key. To achieve this, we use the strategy from [RRD+23] and adapt it to our use case. Note that, as we have seen on Figure 9, we can distinguish between negative coefficients rounded to 0 and negative coefficients rounded to 1.
Figure 10 and Figure 14 depicts the successive queries exploiting this difference, depending on the level of ML-KEM targeted, the attack step and the rounded values previously detected.

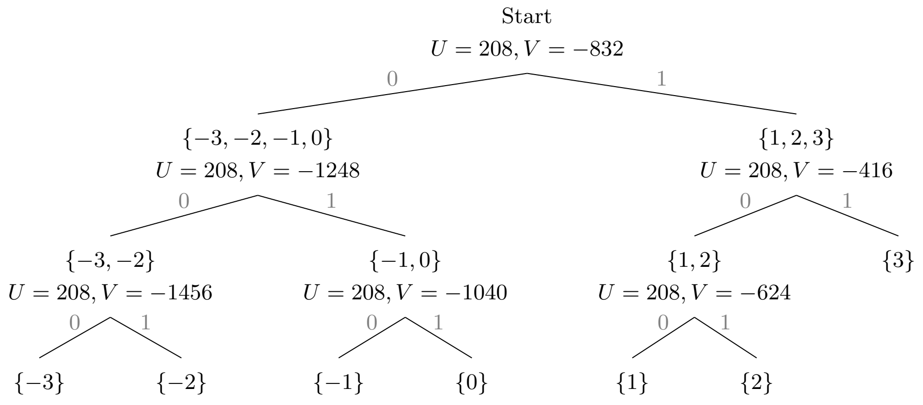


Fig. 10: Query tree for ML-KEM-512 with $\eta_1 = 3$.

For instance, in the case of ML-KEM-512, at the beginning of the attack we know that $s_j[i] \in \{-3,-2,-1,0,1,2,3\}$. We start by sending the crafted ciphertext corresponding to $U = 208$ and $V = -832$ to the decapsulation procedure. Then, in the decapsulation we compute $w[i] = V - Us_j[i] = -832 - 208s_j[i]$. If $s_j[i] \in \{-3,-2,-1,0\}$ then $-832 \leq w[i] < 0$ and therefore will be rounded to 0. If $s_j[i] \in \{1,2,3\}$ then $-1664 < w[i] < -832$ and therefore will be rounded to 1. So now we have partitioned the possible values of $s_j[i]$ depending on the result given from our distinguisher. We continue in the same manner until we have only one possibility for the secret coefficient targeted. We can repeat this procedure until we find all the coefficients of the secret key. In practice, since the coefficients are independent and because the coefficient U stays the same throughout the whole process, we can recover all 256 coefficients of the secret in parallel by taking $v = V + VX^1 + \ldots + VX^{n-1}$, as detailed in Equation 3.

4.5 Attack Validation

We compute the euclidean distance with respect to $\mathcal{M}_0$ and $\mathcal{M}_1$ and we choose the minimal value as a metric to determine whether a new trace corresponds to a coefficient rounded to 0 or to 1.

Number of traces. To construct the means, we can perform $\texttt{Decompress}_{d_u}(416 \cdot \frac{1}{\eta_1})$ possible queries for each dataset, $\mathcal{D}_0$ and $\mathcal{D}_1$. For instance, for ML-KEM-512, it takes around 4 minutes on our setup to collect the traces for each dataset. After that, since we are targeting the 256 coefficients of the secret key in parallel, we will use the windows from the 256 coefficients per query resulting in $256 \times \texttt{Decompress}_{d_u}(416 \cdot \frac{1}{\eta_1})$ sub-traces to construct each mean.

Moreover, we need 3 traces to recover all the 256 coefficients of a polynomial of the secret key,. Therefore, we need $3 \times k$ traces to recover the entire secret key. For instance, for ML-KEM-512,it takes around 30s on our setup.

Success rate. To assess the performance of our attack, we decided to test it on the 10 first keys of the KAT files for each security level of ML-KEM. We have found a 100% success rate each time. Table 2 summarizes the attack performance for each security level of ML-KEM.

	Nb of traces $\mathcal{M}_0$	Nb of traces $\mathcal{M}_1$	Nb of attack traces	Success rate
ML-KEM-512	42	42	6	100%
ML-KEM-768	63	63	9	100%
ML-KEM-1024	127	127	12	100%

Table 2: Summary of our results.

5 Application to Shuffled Implementation of `poly_tomsg`

Shuffling is a simple and low cost countermeasure that increases the number of traces required to perform a side-channel attack. Practical implementations often use the Fisher-Yates algorithm [Dur64] which is an efficient and robust method to create a random permutation.

5.1 Adapting the Attack Strategy

Figure 15 shows a proof of concept implementation of a shuffled `poly_tomsg` function using the Fisher-Yates algorithm to produce a permutation denoted `index`. This permutation is used as the index of the polynomial `a` being compressed. Even though the order in which we store the message bits is random, we can still exploit the same leakage as before by making some adjustments to our attack. However, in this scenario, we can no longer perform the attack in parallel due to the shuffling and so each coefficient must be targeted individually.

The attack proceeds as before, by first constructing the reference means $\mathcal{M}_0$ and $\mathcal{M}_1$, as detailed in Subsection 4.3. For each coefficient of the secret key, the attacker performs queries to recover its value. This time, initial values U and V are sent to the decapsulation algorithm to count the total number of coefficients rounded to 1, establishing a baseline reference denoted $\mathcal{N}_0$. The attacker then modifies U and V, repeats the query, and observes the updated count 1 denoted $\mathcal{N}_1$. Based on these results, the attacker can compare $\mathcal{N}_0$ and $\mathcal{N}_1$ and determine the current position in the corresponding attack tree Figure 11 for ML-KEM-512 or Figure 16 for the other security level. At each step, this allows to refine the possible values of the secret coefficient and allows to select the next U and V values to send with a new query. This process is iterated until a leaf is reached and only one possible secret coefficient is left. An attacker has to repeat this search for all the $256 \times k$ coefficients, to recover the secret key.
Figure 11 and Figure 16 depict the successive queries to perform to recover one secret coefficient depending on the ML-KEM security level targeted, the attack step and the rounded values previously detected.

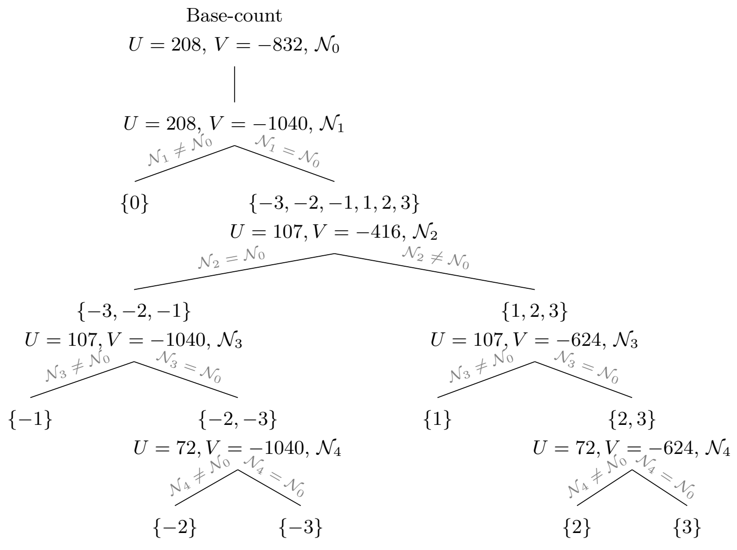

Base-count
$U = 208, V = -832, \mathcal{N}_0$
$U = 208, V = -1040, \mathcal{N}_1$
$\mathcal{N}_1 \neq \mathcal{N}_0$
$\mathcal{N}_1 = \mathcal{N}_0$
$\{0\}$
$\{-3,-2,-1,1,2,3\}$
$U = 107, V = -416, \mathcal{N}_2$
$\mathcal{N}_2 = \mathcal{N}_0$
$\mathcal{N}_2 \neq \mathcal{N}_0$
$\{-3,-2,-1\}$
$U = 107, V = -1040, \mathcal{N}_3$
$\{1,2,3\}$
$U = 107, V = -624, \mathcal{N}_3$
$\mathcal{N}_3 \neq \mathcal{N}_0$
$\mathcal{N}_3 = \mathcal{N}_0$
$\{-1\}$
$\{-2,-3\}$
$U = 72, V = -1040, \mathcal{N}_4$
$\{1\}$
$\{2,3\}$
$U = 72, V = -624, \mathcal{N}_4$
$\mathcal{N}_4 \neq \mathcal{N}_0$
$\mathcal{N}_4 = \mathcal{N}_0$
$\{-2\}$
$\{-3\}$
$\{2\}$
$\{3\}$

Fig. 11: Query tree for ML-KEM-512 with $\eta_1 = 3$, $\mathcal{N}_i$ denotes the number of coefficients of w rounded to 1.

5.2 Attack Validation

Again, we compute the euclidean distance with respect to $\mathcal{M}_0$ and $\mathcal{M}_1$ and we choose the minimal value to determine wether a new trace corresponds to a coefficient rounded to 0 or to 1.

Number of traces. Following the same procedure described in Section 4.5, we will also need $\texttt{Decompress}_{d_u}(416 \cdot \frac{1}{\eta_1})$ traces to construct the means. However, to recover the entire secret key, we need at most 5 queries per coefficient of each polynomials. Therefore, we need at most $5 \times 256 \times k$ queries for ML-KEM-512 and at most $4 \times 256 \times k$ queries for ML-KEM-768 and ML-KEM-1024, to recover the entire secret key. However, in practice the number of queries is significantly reduced since the most frequently occurring value in the secret key is 0, which only needs 2 queries for each security level of ML-KEM. In practice, for ML-KEM-512, it took around 2 hours and 30 minutes on oursetup to recover the secret key.

Success rate. To assess the performance of our attack, we decided to test it on the first key of the KAT files for each security level of ML-KEM. We guessed the entire secret key each time with no error. Table 3 summarizes the attack performance for each security level of ML-KEM.

	Nb of traces $\mathcal{M}_0$	Nb of traces $\mathcal{M}_1$	Nb of attack traces (worst case)	Success rate
ML-KEM-512	42	42	2 560	100%
ML-KEM-768	63	63	3 072	100%
ML-KEM-1024	127	127	4 096	100%

Table 3: Summary of our results on the shuffled version.

6 Reducing the Leakage Without Overhead (for Cortex M4)

From our observation of Figure 2, one main source of leakage comes from the sign change involved by the addition at line 12. One simple way to minimize it is to invert order of the addition and multiplication (respectively lines 12 and 13) so that the multiplication spreads on the most significant bits and reduces the impact of the sign change due to the addition. We intuited that this simple switch could lead to a more gradual differentiation between the two datasets, $\mathcal{D}_0$ and $\mathcal{D}_1$, rather than the abrupt change currently observed.

Figure 12 provides a visual representation of how this modification affects the evolution of the HW across all potential values of a message coefficient.

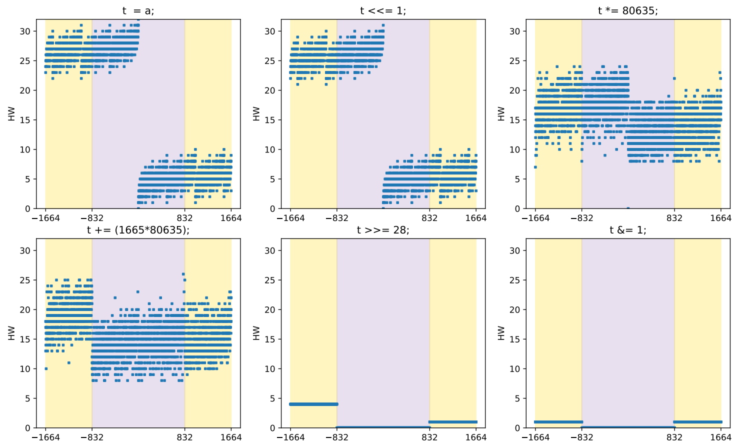


Fig. 12: Coefficient's HW evolution given the computation steps , (🟪: coefficients rounded to 0, 🟨: coefficients rounded to 1).

The difference between the two datasets has been reduced but it is important to note that this trick is not ultimate. Some remaining slight difference still exists and might be enough to mount an SCA. To ensure formal resistance to SCA, only a rigorous masked implementation must be used. When considering the low cost proposed countermeasure, the bit we want to extract, encoding the rounding of the coefficient, is located at the position 28. This positioning leaves us with sufficient space to introduce a constant to change the HW of the value, either into the three most significant bits or into some of the least significant bits, as long as this modification does not cause an overflow (carry) or an underflow (borrow).

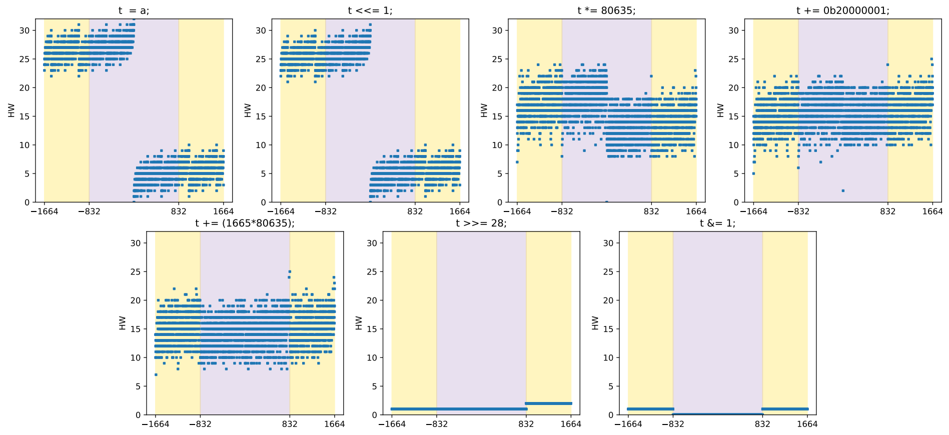


Fig. 13: HW evolution given the steps of the `poly_tomsg` function, (🟪: coefficients rounded to 0, 🟨: coefficients rounded to 1).

7 Conclusion

Recently, the ML-KEM reference implementation has been updated to thwart the KyberSlash threat, and this source code has been rapidly integrated into various open-source libraries. We have shown in this paper that special care must be taken with this piece of code if side channels are applicable. We have presented a straightforward Simple Power Analysis that can recover the private key in a few seconds, and a few minutes in the presence of shuffling. We describe a simple and efficient strategy since only valid ciphertext is needed, and no clone open device is required. From a study of the Hamming weight distributions, we have highlighted the origin of the leakage. We exploit a sign change during the instruction flow within the KyberSlash updated code. The advantage is that if the right dedicated ciphertexts are used, averaging classes can be realized without knowing the device's private key. We showed that a simple adjustment to the source code can significantly reduce this leakage, all while maintaining efficiency with zero overhead. However, the proposed low-cost update does not, in theory, completely defeat side-channel analysis. A rigorous masked implementation must be used to achieve formal resistance when side channels are applicable.

8 Acknowledgements

This work has been funded by the French government (France 2030) and the European union - Next Generation EU (France Relance) in the context of the Resque project. We also thank the reviewers for their time and feedback.

References

BBB$^+$24. Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. Kyberslash: Exploiting secret-dependent division timings in kyber implementations. *IACR Cryptol. ePrint Arch.*, page 1049, 2024.

BDH$^+$19. Ciprian Băetu, F. Betül Durak, Loïs Huguenin-Dumittan, Abdullah Talayhan, and Serge Vaudenay. Misuse attacks on post-quantum cryptosystems. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 747–776, Darmstadt, Germany, May 19–23, 2019. Springer, Cham, Switzerland.

BDK$^+$. Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Kyber official implementation. `https://github.com/pq-crystals/kyber`.

DH76. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

Dur64. Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, July 1964.

FO99. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Berlin, Heidelberg, Germany.

GJJR11. Gilbert Goodwill, Benjamin Jun, Joshua Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*. National Institute of Standards and Technology, 2011.

HMvdW$^+$20. Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

KLJJ. Matthias J. Kannwischer, Thing-han Lim, Ry Jones, and Nigel Jones. MLKEM-C-EMBEDDED optimized for embedded microcontrollers. `https://github.com/pq-code-package/mlkem-c-embedded`.

KPR$^+$. Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.

KSSW22. Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

LS15. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

NIS23. NIST. FIPS 203: Module-lattice-based key-encapsulation mechanism standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2023. `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf`.

OC14. Colin O'Flynn and Zhizhang (David) Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 243–260, Cham, 2014. Springer International Publishing.

QCZ$^+$21. Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based NIST candidate KEMs. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part IV*, volume 13093 of *Lecture Notes in Computer Science*, pages 92–121, Singapore, December 6–10, 2021. Springer, Cham, Switzerland.

RBRC22. Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) nist pqc candidates for practical message recovery attacks. *IEEE Transactions on Information Forensics and Security*, 17:684–699, 2022.

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

RRD$^+$23. Gokulnath Rajendran, Prasanna Ravi, Jan-Pieter D'Anvers, Shivam Bhasin, and Anupam Chattopadhyay. Pushing the limits of generic side-channel attacks on LWE-based KEMs - parallel PC oracle attacks on Kyber KEM and beyond. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):418–446, 2023.

RSA78. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

SAB$^+$22. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`.

SCZ$^+$23. Muyan Shen, Chi Cheng, Xiaohan Zhang, Qian Guo, and Tao Jiang. Find the bad apples: An efficient method for perfect key recovery under imperfect SCA oracles - A case study of Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):89–112, 2023.

Sho94. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Santa Fe, NM, USA, November 20–22, 1994. IEEE Computer Society Press.

SM15. Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513, Saint-Malo, France, September 13–16, 2015. Springer, Berlin, Heidelberg, Germany.

TMS24. Tolun Tosun, Amir Moradi, and Erkay Savas. Exploiting the central reduction in lattice-based cryptography. *IEEE Access*, 12:166814–166833, 2024.

TUX$^+$23. Yutaro Tanaka, Rei Ueno, Keita Xagawa, Akira Ito, Junko Takahashi, and Naofumi Homma. Multiple-valued plaintext-checking side-channel attacks on post-quantum KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):473–503, 2023.

VGO$^+$20. Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

XPR+22. Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Transactions on Computers*, 71(9):2163–2176, 2022.

A Other security levels query tree for attack on standard implementation

Figure 14 details all the successive queries used to mount our attack on ML-KEM-768 and ML-KEM-1024.

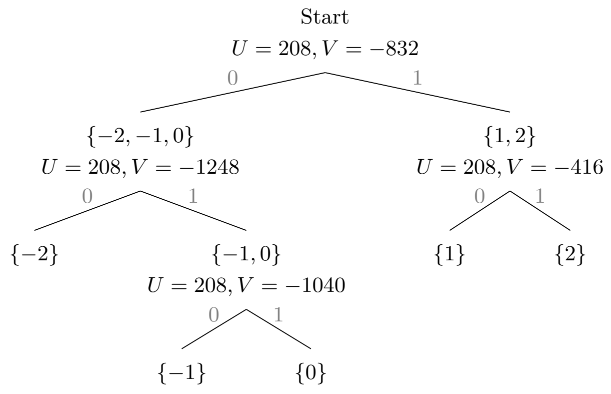


Fig. 14: Query tree for ML-KEM-768 and ML-KEM-1024 with $\eta_1 = 2$.

B Shuffled Implementation of `poly_tomsg`

We give in figure 15 the shuffled implementation of the `poly_tomsg` that we targeted in Section 5.

```c
static uint8_t index[KYBER_N] = {0, 1, ..., 254, 255}

void shufflepoly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES],
                       const poly *a){
  unsigned int i,j;
  uint32_t t;
  unsigned int temp, random_index, s_i, s_j;
  for(i = KYBER_N - 1; i>=1; --i) {
    randombytes(&random_index, 1);
    random_index = random_index%(i + 1);
    temp = index[i];
    index[i] = index[random_index];
    index[random_index] = temp;
  }
  for(i = 0; i < KYBER_INDCPA_MSGBYTES; i++){
    msg[i] = 0;
  }
  for(i=0;i<KYBER_N/8;i++) {
    for(j=0;j<8;j++) {
      temp = index[8*i+j];
      s_i = temp>>3;
      s_j = temp&0x7;
      t  = a->coeffs[temp];
      t <<= 1;
      t += 1665;
      t *= 80635;
      t >>= 28;
      t &= 1;
      msg[s_i] |= t << s_j;
    }
  }
}
```

Fig. 15: C Code of the shuffled `poly_tomsg` implementation.

C Other security levels query tree for attack on shuffled implementation

Figure 16 details all the successive queries used to mount our attack on the shuffled versions of ML-KEM-768 and ML-KEM-1024.

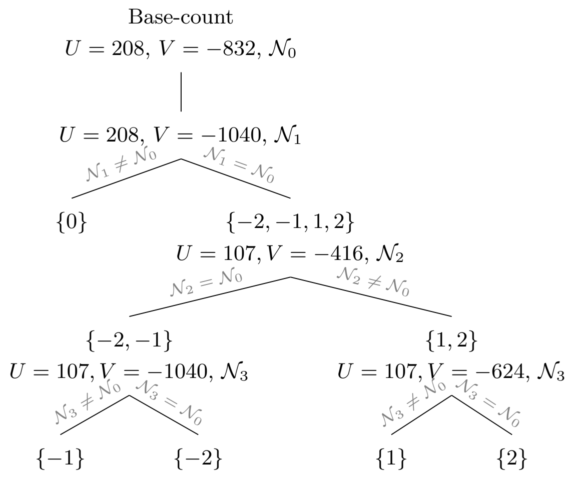

Base-count
$U = 208,\ V = -832,\ \mathcal{N}_0$
$U = 208,\ V = -1040,\ \mathcal{N}_1$
$\mathcal{N}_1 \neq \mathcal{N}_0$
$\mathcal{N}_1 = \mathcal{N}_0$
$\{0\}$
$\{-2, -1, 1, 2\}$
$U = 107, V = -416,\ \mathcal{N}_2$
$\mathcal{N}_2 = \mathcal{N}_0$
$\mathcal{N}_2 \neq \mathcal{N}_0$
$\{-2, -1\}$
$U = 107, V = -1040,\ \mathcal{N}_3$
$\{1, 2\}$
$U = 107, V = -624,\ \mathcal{N}_3$
$\mathcal{N}_3 \neq \mathcal{N}_0$
$\mathcal{N}_3 = \mathcal{N}_0$
$\mathcal{N}_3 \neq \mathcal{N}_0$
$\mathcal{N}_3 = \mathcal{N}_0$
$\{-1\}$
$\{-2\}$
$\{1\}$
$\{2\}$

Fig. 16: Query tree for ML-KEM-768 and ML-KEM-1024 with $\eta_1 = 2$, $\mathcal{N}_i$ denotes the number of coefficients of w rounded to 1.