# Extending MV-PC Side-Channel Attacks to Higher-Order Masked Implementations of Kyber

Jinnuo Li[1,2], Hao Cheng[2,1,3(✉)], and Chi Cheng[4]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
jinnuo.li@mail.sdu.edu.cn
hao.cheng@sdu.edu.cn
[2] Quan Cheng Laboratory, Jinan, China
[3] State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China
[4] Cyberspace Institute of Advanced Technology, Pei Dingyi Cryptography Research Institute, Guangzhou University, Guangzhou, China
chengchizz@gmail.com

**Abstract.** Multiple-valued plaintext-checking (MV-PC) side-channel attacks are a well-known threat to Kyber. Existing MV-PC oracle-based attacks have mainly targeted unprotected implementations or implementations protected only by first-order masking, leaving higher-order masked implementations insufficiently studied. In this paper, we present the first MV-PC attack against higher-order masked implementations of Kyber targeting the Keccak absorb procedure in the FO transform. Specifically, we propose an efficient message-recovery method that directly extracts the message bits queried by the MV-PC oracle from the Hamming-weight leakage of masked shares, without reconstructing complete byte- or word-level message values. Compared with the prior message-recovery method, our approach significantly reduces the number of traces required for full recovery, while avoiding the rapidly growing computational complexity that limits the application of previous approaches to higher-order masking. We evaluate our attack on two open-source masked Kyber implementations, `mkm4` and `pqm4_masked`. All experiments are conducted on an ARM Cortex-M4 target platform under two compiler optimization levels, `-Os` and `-O3`. Across these, we observe three exploitable storage-related leakage sources. With profiled Hamming-weight classifiers, we instantiate MV-PC oracles and estimate the trace requirements for full secret-key recovery under first-, second-, and third-order masking.

**Keywords:** side-channel attack, post-quantum cryptography, Kyber, higher-order masking.

## 1 Introduction

Kyber, now standardized as ML-KEM in FIPS 203 [1], was selected for its strong balance between performance and security. As a core component of next-generation secure communication systems, it is expected to be deployed not only

on high-performance platforms, but also on embedded and resource-constrained devices. Although the theoretical security of Kyber relies on the hardness of module lattices [13], which offers strong security guarantees, it may be undermined by side-channel attacks [12] against implementations running on physical devices. This issue has also been emphasized throughout the NIST PQC standardization process, where resistance against side-channel and fault attacks was considered an important aspect of practical deployment [2].

In recent years, several side-channel attacks on software and hardware implementations of Kyber have been demonstrated. Among these attacks, Plaintext-Checking (PC) attacks and their variant, Multiple-Valued Plaintext-Checking (MV-PC) attacks [20,23,21,24,14,11,19] deserve particular attention due to their generic applicability and the significant challenges they present in deploying countermeasures. These attacks exploit message-dependent leakage from the re-encryption procedure, including message encoding/decoding and Keccak. In response, several countermeasures have been proposed to protect Kyber implementations against side-channel attacks [10,4], with masking being one of the most widely studied methods.

Existing MV-PC attacks have mainly been evaluated on unprotected Kyber implementations. Some works consider non-ideal or noisy oracle responses and propose techniques to improve key recovery in such settings [23,14,11,19]. Although these techniques are intended to tolerate imperfect oracles and may be applicable to protected implementations, they do not provide a concrete side-channel evaluation against masked Kyber implementations. Closely related to PC and MV-PC attacks, recent Full-Domain (FD) oracle attacks have demonstrated successful key recovery against masked Kyber implementations [9,3], including higher-order masked settings [6,26,5]. However, these attacks mainly leverage leakage from message encoding/decoding rather than Keccak. Therefore, they do not address whether leakage in higher-order masked Keccak can be used to instantiate MV-PC oracles. This motivates us to investigate MV-PC attacks against Kyber through leakage in higher-order masked Keccak.

### 1.1 Contributions

The principal contributions of this study are:

- We propose an efficient message-recovery method for MV-PC oracle-based attacks against higher-order masked Kyber implementations. Instead of reconstructing the complete byte- or word-level message value and then deriving individual bits, our method directly recovers the selected message bits required by the MV-PC oracle from the Hamming-weight leakage of masked shares. This not only reduces full-message recovery to eight traces for Masked-Bytewise-Storage leakage and 32 traces for Masked-Wordwise-Storage leakage, but also avoids the rapidly growing computational cost of prior bit-flipping-based approaches.
- We conduct side-channel analysis of two open-source masked implementations of Kyber: `mkm4` and `pqm4_masked` on the ARM Cortex-M4 microcontroller under different compiler optimization levels. Our analysis identifies

three exploitable storage-related leakage sources: Masked-Bytewise-Storage leakage in `mkm4` compiled with `-Os`, Masked-Wordwise-Storage leakage in `mkm4` compiled with `-O3`, and Masked-Bytewise-Incremental-Storage leakage in `pqm4_masked`. We validate these leakage sources through assembly inspection, power-trace analysis, and Hamming-weight classification experiments.
– We demonstrate full secret-key recovery against Kyber using the proposed message-recovery method and the identified leakage sources. Our evaluation covers first-, second-, and third-order masked settings and estimates the number of side-channel traces required when majority voting is used to handle noisy oracle outputs. We further analyze the tradeoff between leakage parallelism and oracle reliability, and summarize how existing key-recovery techniques affect the overall attack cost.

## 2 Background

### 2.1 Kyber

Kyber, now standardized as ML-KEM [1], is an IND-CCA2 secure KEM based on the Module Learning With Errors problem (MLWE) [13]. Most of its arithmetic operations are performed over the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n+1)$, where the fixed parameters are the prime modulus $q = 3329$ and the polynomial degree $n = 256$. The security level of Kyber is determined by the module rank $l$. Specifically, $l = 2$, 3, and 4 correspond to Kyber-512, Kyber-768, and Kyber-1024, respectively, targeting NIST security levels 1, 3, and 5. Kyber uses two Centered Binomial Distribution (CBD) parameters, $\eta_1$ and $\eta_2$. The secret-key coefficients and the key-generation error coefficients are sampled from $\mathcal{B}_{\eta_1}$, whereas the encryption-error coefficients are sampled from $\mathcal{B}_{\eta_2}$. For a parameter $\eta$, the distribution $\mathcal{B}_\eta$ is defined as $\sum_{i=1}^{\eta}(a_i - b_i)$, where the $a_i$ and $b_i$ are independent uniform samples from $\{0,1\}$. The values of $\eta_1$ and $\eta_2$, together with the other main parameters, are summarized in Table 1. Let $\lfloor x \rfloor$ denote the maximum integer not exceeding $x$, and $\lceil x \rfloor$ denote the rounding function, i.e. $\lceil x \rfloor = \lfloor x + \frac{1}{2} \rfloor$. In Kyber, the compression and decompression functions $\mathrm{Comp}_q(x,d)$ and $\mathrm{DeComp}_q(x,d)$ are defined by

$$\begin{aligned} \mathrm{Comp}_q(x,d) &= \lfloor (2^d/q) \cdot x \rceil \bmod 2^d, \\ \mathrm{DeComp}_q(x,d) &= \lfloor (q/2^d) \cdot x \rceil . \end{aligned}$$

For $\mathrm{Comp}_q(x,d)$, the input $x$ is an element of $\mathbb{Z}_q$, while for $\mathrm{DeComp}_q(x,d)$, the input is an element of $\mathbb{Z}_{2^d}$. When the input is a polynomial, these operations are applied coefficient-wise. Depending on the Kyber security level, the parameter $d$ is instantiated as either $d_u$ or $d_v$. The corresponding values are also listed in Table 1. Moreover, the hash function $\mathcal{G}$, which is the target of the attack considered in this paper, is implemented using SHA3-512. KYBER.CCA KEM consists of key generation, encapsulation and decapsulation. The details of encapsulation and decapsulation are described in Algorithm 1.

**Algorithm 1** KYBER.CCA KEM

∘ KYBER.CCA.Encaps
**Input:** $pk$
**Output:** $ct, K$
1: $\mathbf{m} \xleftarrow{\$} \{0,1\}^{256}$
2: $(\bar{K}, r) = \mathcal{G}(\mathbf{m}, \mathcal{H}(pk))$
3: $ct := \mathsf{KYBER.CPA.Enc}(pk, \mathbf{m}, r)$
4: $K := \mathsf{KDF}(\bar{K}, \mathcal{H}(ct))$

∘ KYBER.CCA.Decaps
**Input:** $sk, ct$
**Output:** $K$
1: $\mathbf{m}' := \mathsf{KYBER.CPA.Dec}(sk, ct)$
2: $(\bar{K}', r') = \mathcal{G}(\mathbf{m}', \mathcal{H}(pk))$
3: $ct' = \mathsf{KYBER.CPA.Enc}(pk, \mathbf{m}', r')$
4: **if** $ct = ct'$ **then**
5: $\quad K := \mathsf{KDF}(\bar{K}', \mathcal{H}(ct))$
6: **else**
7: $\quad K := \mathsf{KDF}(z, \mathcal{H}(ct))$
8: **end if**

**Table 1.** Parameter sets of Kyber.

| Scheme | NIST Security | $n$ | $l$ | $q$ | $\eta_1$ | $\eta_2$ | $(d_{\mathbf{u}}, d_v)$ |
|---|---|---|---|---|---|---|---|
| Kyber-512 | Category 1 | 256 | 2 | 3329 | 3 | 2 | (10,4) |
| Kyber-768 | Category 3 | 256 | 3 | 3329 | 2 | 2 | (10,4) |
| Kyber-1024 | Category 5 | 256 | 4 | 3329 | 2 | 2 | (11,5) |

### 2.2 Multiple-Valued Plaintext-Checking (MV-PC) Oracle

The MV-PC oracle formalizes how leaked information about the decrypted message $\mathbf{m}'$ can be converted into information about the secret key. It was independently proposed by Rajendran et al. [21] and Tanaka et al. [24], enabling efficient side-channel attacks on post-quantum KEMs. The core idea is to construct special ciphertexts such that the bits of the recovered message $\mathbf{m}'$ at $M$ selected positions depend on the $M$ corresponding coefficients of the secret key $\mathbf{s}$. In the following, we take Kyber-1024 as an example to introduce a universal MV-PC oracle-based attack. Assume the secret key of the victim is $\mathbf{s} = (\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$.

Specifically, to target the set $\mathcal{J} = \{j_0, j_1, \ldots, j_{M-1}\}$, corresponding to the $M$ coefficients in $\mathbf{s}_i$, the adversary first chooses the attack parameters $k_1$ and $\{k_{2,j}\}_{j\in\mathcal{J}}$. The adversary then constructs $\mathbf{u} = (\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)$ and $\mathbf{c}_2$ as:

$$\mathbf{u}_x = \begin{cases} (k_1, 0, \ldots, 0), & x = i \\ \mathbf{0}, & x \neq i \end{cases}, \quad \mathbf{c}_2[j] = \begin{cases} k_{2,j}, & j \in \mathcal{J} \\ 0, & j \notin \mathcal{J} \end{cases}$$

The adversary subsequently compresses $\mathbf{u}$ coefficient-wise to obtain

$$\mathbf{c}_1 = \mathrm{Comp}_q(\mathbf{u}, d_u).$$

It then sends $ct := (\mathbf{c}_1, \mathbf{c}_2)$ to the victim. Upon receiving $ct$, the victim decompresses its two components to obtain $\mathbf{u}'$ and $\mathbf{v}'$. Later, the victim computes $\mathbf{m}'$

as follows:

$$\begin{aligned}
\mathbf{m}'[j] &= \mathrm{Comp}_q((\mathbf{v}' - \mathbf{s}^T\mathbf{u}')[j], 1) \\
&= \mathrm{Comp}_q((\mathbf{v}' - \mathbf{s}_0\mathbf{u}'_0 - \mathbf{s}_1\mathbf{u}'_1 - \mathbf{s}_2\mathbf{u}'_2 - \mathbf{s}_3\mathbf{u}'_3)[j], 1) \\
&= \left\lceil \frac{2}{q}(\mathbf{v}'[j] - \mathbf{s}_i[j]\mathbf{u}'_i[0]) \right\rfloor \bmod 2 \\
&= \begin{cases} \left\lceil \frac{2}{q}\left(\left\lceil \frac{q}{32} \right\rfloor \cdot k_{2,j} - \mathbf{s}_i[j] \cdot k_1\right) \right\rfloor \bmod 2, & j \in \mathcal{J} \\ \left\lceil \frac{2}{q}\left(0 - \mathbf{s}_i[j] \cdot k_1\right) \right\rfloor \bmod 2 = 0, & j \notin \mathcal{J} \end{cases}
\end{aligned}$$

The equation above reveals that a specific bit of the message $\mathbf{m}'[j]$ relates to an *adversary-controlled* ciphertext coefficient $k_{2,j}$ and a private-key coefficient $\mathbf{s}_i[j]$. Consequently, a single evaluation of $\mathbf{m}'[j]$ provides a binary decision on $\mathbf{s}_i[j]$, indicating whether $\mathbf{s}_i[j]$ exceeds or falls below a certain value determined by $k_{2,j}$. Repeating this test with different ciphertexts allows us to iteratively narrow the range of $\mathbf{s}_i[j]$ until its exact value is recovered. The MV-PC oracle described above shows that recovering selected bits of $\mathbf{m}'$ is sufficient for extracting information about the secret key. In masked implementations, however, $\mathbf{m}'$ is represented by multiple shares during the processing of $\mathcal{G}$. Therefore, instantiating such an oracle requires a message-recovery procedure that derives the selected bits of $\mathbf{m}'$ from leakage associated with these shares.

## 3 Analysis

### 3.1 Message recovery against (higher-order) masked implementations

Ravi et al. [22] proposed a preliminary message-recovery method exploiting the Masked-Bytewise-Storage vulnerability. Their approach relies on the leakage of Hamming weights from individual masked shares, denoted by $(v_1, v_2)$, to recover the Hamming weight of the original message byte. This is possible because of the linearity of the XOR operation used in Boolean masking. They then introduce a bit-flip at a specific position of the message by manipulating the ciphertext. By repeating the measurement process and observing whether the resulting Hamming weight increases or decreases, the value of the target bit can be inferred. However, this preliminary approach has two major limitations. First, it requires a large number of traces to recover the message bytes, even under an ideal oracle simulation. Second, its computational complexity grows rapidly when extended to higher-order masking or to the Masked-Wordwise-Storage vulnerability. These limitations make the previous method highly inefficient in the above scenarios.

To overcome these limitations, we propose an improved message-recovery framework tailored to MV-PC-oracle-based attacks. Our method is primarily based on the following observation. If a byte or word of the recovered message

$\mathbf{m}'$ has Hamming weight one, then the sum of the Hamming weights of its corresponding masked shares is necessarily odd. If its Hamming weight is zero, then this sum is necessarily even. Therefore, when all non-target bits in the byte or word are fixed to zero, the target bit can be recovered directly by computing the parity of the sum of the Hamming weights of its shares. This observation enables a divide-and-conquer recovery strategy. Unlike previous methods, which first recover the Hamming weight of an entire message byte and then infer individual bit values through iterative bit-flipping, our method isolates and recovers one message bit at a time. We first illustrate the method by considering the recovery of the first bit of the first byte under the Masked-Bytewise-Storage vulnerability. Specifically, we set $\mathbf{c}_2[0] = k_{2,0}$ while all other coefficients corresponding to the first byte are set to zero. After decompression, this gives $\mathbf{v}'[0] = \mathrm{DeComp}_q(k_{2,0}, d_v)$, while the other coefficients of $\mathbf{v}'$ in the first byte remain zero. By the MV-PC construction described above, this further ensures that $\mathbf{m}'[i] = 0$ for $1 \leq i < 8$. Therefore, we obtain:

$$\mathbf{m}'[0] = \sum_{j=1}^{d+1} \mathrm{HW}(\mathrm{share}_j[0:7]) \bmod 2.$$

Thus, by extracting the Hamming weight of the first byte from each share and computing the parity of their sum, the value of $\mathbf{m}'[0]$ can be directly obtained. Moreover, the same ciphertext construction can be applied to the same bit position in all 32 bytes simultaneously. Since this relation holds independently for each byte, a single trace enables the simultaneous recovery of one bit from each byte of $\mathbf{m}'$, yielding 32 message bits in parallel. Recovering all 8 bit positions of each byte consequently requires only 8 traces for full 256-bit message recovery targeting Masked-Bytewise-Storage.

The same principle applies to the Masked-Wordwise-Storage vulnerability. Following the same construction as in the byte-wise case, we consider the recovery of the first bit of the first 32-bit word. Specifically, we again set $\mathbf{c}_2[0] = k_{2,0}$ while all other coefficients corresponding to the first 32-bit word are set to zero. After decompression, this gives $\mathbf{v}'[0] = \mathrm{DeComp}_q(k_{2,0}, d_v)$, while the other coefficients of $\mathbf{v}'$ in the first word remain zero. As in the byte-wise case, the MV-PC construction ensures that $\mathbf{m}'[i] = 0$ for $1 \leq i < 32$. Therefore, we obtain:

$$\mathbf{m}'[0] = \sum_{j=1}^{d+1} \mathrm{HW}(\mathrm{share}_j[0:31]) \bmod 2.$$

Since a 256-bit message consists of eight 32-bit words, one trace recovers one selected bit from each word. Hence, full message recovery under Masked-Wordwise-Storage requires 32 traces.

To recover the remaining bits, let $0 \leq b < 32$ and $0 \leq r < 8$ denote the byte index and the bit offset within a byte, respectively. Similarly, let $0 \leq w < 8$ and $0 \leq r < 32$ denote the word index and the bit offset within a word. We obtain

$$\mathbf{m}'[8b + r] = \sum_{j=1}^{d+1} \mathrm{HW}(\mathrm{share}_j[8b : 8b + 7]) \bmod 2,$$

**Table 2.** Comparisons between this work and [22] in terms of the number of traces required for full message recovery. The numbers are reported for messages of length 256 bits and assuming a perfect single-trace side-channel HW classifier.

| Vulnerability | [22] | | This work |
|---|---|---|---|
| | Traces | Complexity | Traces |
| 1st-order Masked-Bytewise-Storage | 1179 | $\approx 2^{15}$ | 8 |
| 2nd-order Masked-Bytewise-Storage | 98163 | $\approx 2^{21.4}$ | 8 |
| 3rd-order Masked-Bytewise-Storage | - | $\approx 2^{27.4}$ | 8 |
| 1st-order Masked-Wordwise-Storage | - | $\approx 2^{63}$ | 32 |
| 2nd-order Masked-Wordwise-Storage | - | $\approx 2^{93.4}$ | 32 |

and

$$\mathbf{m}'[32w + r] = \sum_{j=1}^{d+1} \mathrm{HW}(\mathrm{share}_j[32w : 32w + 31]) \bmod 2,$$

for the Masked-Bytewise-Storage and Masked-Wordwise-Storage vulnerabilities, respectively.

By enabling independent bit-level recovery, our divide-and-conquer strategy substantially reduces both trace complexity and computational complexity. Table 2 compares our method with that of Ravi et al. [22].[5] Compared with their method, our method is over 100× faster when attacking first-order Masked-Bytewise-Storage vulnerabilities and more than $10,000\times$ faster in the second-order setting. It also avoids the prohibitive computational complexity of prior approaches, making it practical for higher-order masking schemes and the Masked-Wordwise-Storage vulnerability.

### 3.2 Experimental setup

*Target.* From an algorithm perspective, we focus on the internal operation in the KeccakAbsorb function, while as for the practical implementation, we target two different software libraries of masked Kyber, both of which are compiled using `arm-none-eabi-gcc` version 13.2.1 in our experiments:

- `mkm4` library[6], a first-order masked implementation proposed by Heinz et al. [10]. In order to provide a complete exposition of our attack idea, we adapt the `masked_hash_g` procedure to a higher-order masking version. In addition, we note that `mkm4` behaves differently under the two optimization levels -O3 and -Os.[7]

[5] Although Ravi et al. [22] only provide an estimate for the first-order Masked-Bytewise-Storage vulnerability, the results reported in our table were independently derived using our own scripts based on their methodology. Our experimental results are consistent with their estimates.

[6] `https://github.com/masked-kyber-m4/mkm4`

[7] With `-Os`, the compiler generates code that is more optimized for code size, which is a common choice for use in embedded environment.

```
void keccak_absorb_masked(uint64_t *s1, uint64_t *s2,unsigned int r, const unsigned char *m1,
     const unsigned char *m2, unsigned long long int mlen,unsigned char p) {
1: ... Processing ...
2: memxor(s1, m1, mlen);
// padding
3: ((char *) s1)[mlen] ^= p;
4: ((char *) s1)[r-1] ^= 128;
5: memxor(s2, m2, mlen);
6: ... Processing ...
}
void memxor(void *dest, const void *src, size_t len)
{
1: char *d = dest;
2: const char *s = src;
3: while(len--)
4: *d++ ^= *s++;
}
```

**Fig. 1.** The C code of `masked_hash_g()` procedure from `mkm4` library.

– `pqm4_masked` library[8], a higher-order masked implementation proposed by Bronchain and Cassiers [4]. Since the memory load/store behavior of this implementation is consistent under different optimization levels, we consider only `-O3`.

*Device.* Our experimental setup consists of the ChipWhisperer Lite, the CW308 UFO main board, and a CW308T-STM32F4 target board, containing an STM32F415-RGT6 chip based on an ARM Cortex-M4 32-bit RISC core. The operating frequency is set to 24MHz, at a sample rate of 96MHz. We use the Cortex-M4 core for our target because NIST recommended it for the evaluation of post-quantum cryptography standard algorithms.

### 3.3 Leakage analysis of `mkm4` library

In masked implementations of Kyber, the decrypted message under masking protection is typically constructed as follows:

$$\begin{aligned}\mathsf{masked_msg} = (\mathsf{share}_1 \,\|\, \mathcal{H}(pk) \,\|\, \mathsf{share}_2 \,\|\, 32\text{-byte } 0 \\ \|\, \cdots \,\|\, \mathsf{share}_{d+1} \,\|\, 32\text{-byte } 0)\end{aligned}$$

As shown in Figure 1, for `mkm4` library, within the `keccak_absorb_masked` function, the decrypted message is copied from one memory location to another. This operation is performed sequentially for each part of the message. Therefore, the leakage produced by each memory write can be exploited to recover the Hamming weight of a specific byte or word.

*Differences between optimization levels* `-Os` *and* `-O3`. We disassemble the critical code snippets to analyze the register behavior. As shown in Figure 2 (a), when

[8] `https://github.com/simple-crypto/pqm4_masked`

```
    ...
1: /* load */
2: ldr.w r3, [r5], #4 /* load 32-bit */
3: ldr r6, [r4, #0] /* load 32-bit */
4: cmp r5, r7
    /* xor */
5: eor.w r3, r3, r6
6: /* store to memory*/
7: str.w r3, [r4], #4 /* store 32-bit */
    ...
```

(a) With `-O3`.

```
    ...
1: /* load */
2: ldrb.w r3, [r0], #1 /* load 8-bit */
3: ldrb.w r4, [r1], #1 /* load 8-bit */
    /* xor */
4: eors r3, r4
5: /* store to memory*/
6: strb.w r3, [r0, #-1] /* store 8-bit */
    ...
```

(b) With `-Os`.

**Fig. 2.** Disassembled assembly code snippet of KeccakAbsorb function from `mkm4` library under different compile levels using `arm-none-eabi-objdump`.

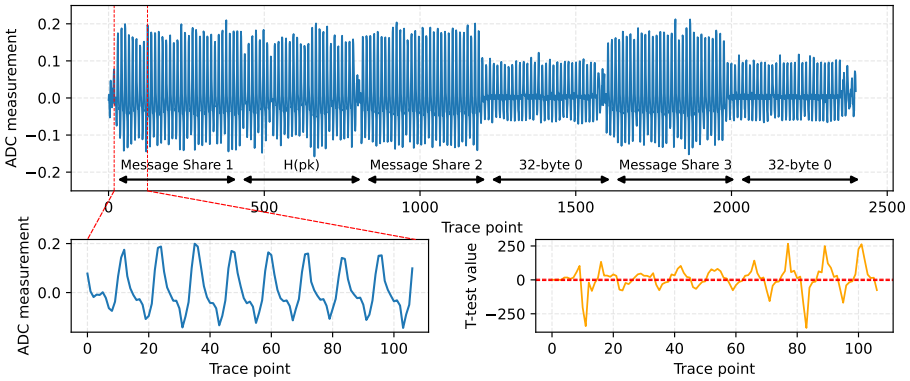


**Fig. 3.** Power trace of KeccakAbsorb function of the second-order masked implementations of `mkm4` under `-Os` (top); More detailed power trace for the first 8 operations (bottom left); T-test results for the first 8 operations (bottom right).

compiled with the `-O3` optimization level, the implementation loads a word at a time (using `ldr`), performs an XOR operation, and writes a word to memory (using `str`). Consequently, the leakage observed in each operation corresponds to the Hamming weight of a word. In contrast, as shown in Figure 2 (b), the `-Os` binary uses the byte-wise load/store instructions `ldrb` and `strb`. In this case, both the load and store operations are performed at byte granularity, so the corresponding leakage reflects the Hamming weight of a byte.

**Compiling with `-Os`** We first analyze a Bytewise-Storage vulnerability: side-channel leakage in `mkm4` with `-Os`. We collect more than 1,000 power side-channel traces using random inputs and compute their average to visualize the overall leakage behavior.

As shown in the top plot of Figure 3, the averaged trace clearly reveals that each message share is processed sequentially, and the processing is performed byte by byte. For each message share, we observe only 32 similar operational

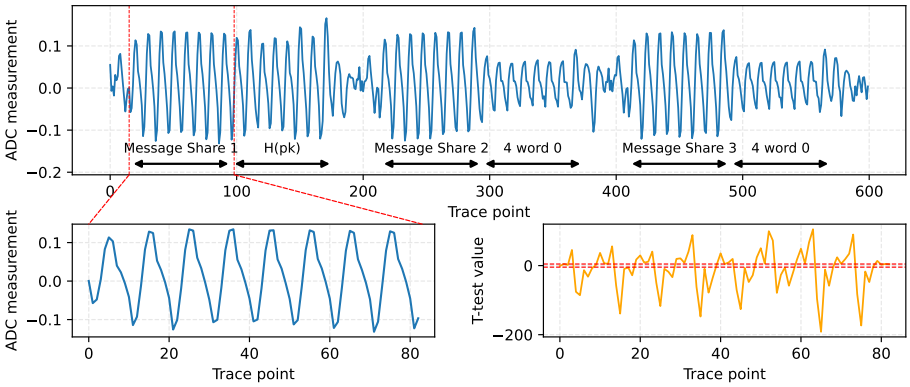


**Fig. 4.** Power trace of KeccakAbsorb function of the second-order masked implementations of `mkm4` under `-O3` (top); More detailed power trace for the first 8 operations (bottom left); T-test results for the first 8 operations (bottom right).

patterns in the averaged traces. We then focus on the processing of the first eight bytes of the message in the following experiments (as shown in the bottom left and bottom right plots of Figure 3).

To this end, we further collect two sets of power traces, denoted as $\mathcal{T}_0$ and $\mathcal{T}_1$, each consisting of 1000 side-channel traces. For $\mathcal{T}_0$, the input message is chosen uniformly at random, except that the Hamming weight of the first eight bytes is fixed to zero. For $\mathcal{T}_1$, the input is generated in the same manner, except that the Hamming weight of the first eight bytes is fixed to eight. Each trace is normalized prior to analysis, and Welch's t-test [7] is applied to highlight statistically significant differences between the two trace sets. As shown in the bottom-right plots of Figure 3, the t-test trace exhibits eight peaks exceeding the threshold of $\pm 4.5$, each corresponding to one of the first eight bytes. These peaks allow precise identification of the narrow time window during which each byte is updated, directly revealing the byte-wise leakage.

**Compiling with `-O3`** We then analyze the side-channel leakage in `mkm4` compiled with `-O3`, using the same methodology as above. As discussed in Section 3.3, under the `-O3` optimization level, the operations are word-wise rather than byte-wise. Consequently, for each message share, we observe only eight similar operational patterns in the averaged traces, corresponding to the eight 32-bit words of the share. Focusing on the first eight words, we also collect two sets of power traces, denoted as $\mathcal{T}_0$ and $\mathcal{T}_1$, where the Hamming weights of these words are fixed to 0 and 32, respectively, while the remaining inputs are random. Even under `-O3`, the first eight words exhibit distinct operational patterns, demonstrating that the word-wise updates remain a clear source of leakage, similar to the byte-wise case under `-Os`.

```
void masked_hash_keccak(uint8_t *out, size_t outlen, size_t out_msk_stride, size_t
     out_data_stride, const uint8_t *in, size_t inlen, size_t in_msk_stride, size_t
     in_data_stride, size_t rate, uint8_t delim) {
1: ... Processing ...
2: for (size_t i = 0; i < inlen; i++) {
3: for (size_t j = 0; j < NSHARES; j++) {
4: XORU64(msk_a, i + j * Plen, in[j * in_msk_stride + i * in_data_stride]);
5: }
6: }
7: ... Processing ...
}

#define XORU64(value, address, byte)
1: do {
2: (value)[(address) >> 3]^= (((uint64_t)(byte)) << 8 * ((address)&0x7));
3: } while (0)
```

**Fig. 5.** The C code of snippet `masked_sha3_512()` procedure from `pqm4_masked` library.

### 3.4 Leakage analysis of `pqm4_masked` library

We now turn to the `pqm4_masked` library. Compared with `mkm4`, this implementation exhibits a different leakage pattern because the order in which the masked data are processed is different. As discussed above, the implementation of the `mkm4` library processes data share by share, completing all bytes of the first share before proceeding to the next. In contrast, as shown in Figure 5, the implementation of the `pqm4_masked` library processes data byte by byte across shares, which gives rise to a Bytewise-Incremental-Storage vulnerability.

The assembly code in Figure 6 confirms this behavior. The compiler emits a `strd` instruction, which stores a pair of 32-bit registers as a 64-bit double-word in memory. However, the loop updates only one byte of this 64-bit value in each iteration. As a result, once a byte is written into the 64-bit word, it remains present in the value written by subsequent store operations until the entire word is completed. Hence, the first inserted byte contributes to the Hamming weight of eight consecutive stores, the second byte contributes to seven stores, and so on. This cumulative effect is the source of the incremental-storage leakage.

We further perform side-channel experiments to validate the incremental-storage behavior identified above. Specifically, we conduct two experiments. In the first experiment, we compare a fully random dataset with another fully random dataset where the first byte of Share 1 is fixed to `0xFF`. In the second experiment, we use the same setup but fix the second byte of Share 1 to `0xFF`. The bottom two plots in Figure 7 provide direct evidence of the incremental storage vulnerability. The bottom-left plot in Figure 7 shows the result when we fix the first byte of Share 1 to `0xFF`. The bottom-right plot shows the result when we fix the second byte of Share 1 to `0xFF`. In both cases, the t-test traces exhibit multiple leakage peaks, indicating that the fixed byte affects several consecutive store operations. More precisely, the bottom-left plot shows eight distinct peaks for the first byte, whereas the bottom-right plot shows seven distinct peaks for the second byte. This observation matches the instruction-level analysis above. Note that in this paper, we only exploit the leakage of the first storage during

```
        ...
 1: /* load */
 2: ldrd r0, lr, [r1] /* Load 64-bit from msk_a */
        ...
 3: ldrb.w r8, [r6, #1]! /* Load 1-byte from in */
        ...
 4: /* shift */
 5: lsr.w r7, r8, r7 /* Shift */
        ...
 6: lsl.w r4, r8, r3 /* Shift */
 7: /* xor */
 8: eors r4, r0 /* XOR lower 32-bit */
        ...
 9: eor.w r0, r1, lr /* XOR higher 32-bit */
10: /* store to memory */
11: strd r4, r0, [ip] /* store 64-bit */
        ...
```

**Fig. 6.** Disassembled assembly code snippet of KeccakAbsorb function from `pqm4_masked` library using `arm-none-eabi-objdump`.

the incremental storage procedure (i.e., the first byte of the 8 consecutive bytes of each share). This choice is motivated by the attack results in the following Section 4, which show that the classifier accuracy decreases significantly as the range of possible Hamming weights increases.

## 4 Attack

### 4.1 Profiling Phase

In this section, we describe how the side-channel oracles used in our attack are constructed. As shown in Section 3, the targeted implementations give rise to three different leakage sources: byte-wise storage leakage in `mkm4` compiled with `-Os`, word-wise storage leakage in `mkm4` compiled with `-O3`, and byte-wise incremental-storage leakage in `pqm4_masked`. For each case, we train neural-network classifiers to predict the Hamming weight of the corresponding intermediate value. The classifiers follow the CNN-based profiling framework of Tanaka et al. [24], with minor modifications to match the length of our trace windows and the number of output classes. The models are trained on a PC with an Intel Core i7-14700KF CPU, an NVIDIA GeForce RTX 4090 GPU, and 64GB of RAM. Rather than training a separate classifier for each share, we only train one classifier for each leakage type and reuse it across positions that are processed by the same code pattern. This is justified by the fact that the corresponding byte or word updates are computed in the same way across different shares and therefore exhibit similar power-trace patterns.

For each classifier, we collect 100,000 labeled traces and split them into training, validation, and test sets with a ratio of $8:1:1$. Each trace window is normalized before classification. The ground-truth Hamming-weight label is computed from the known inputs and masks on the profiling device. For a trace window

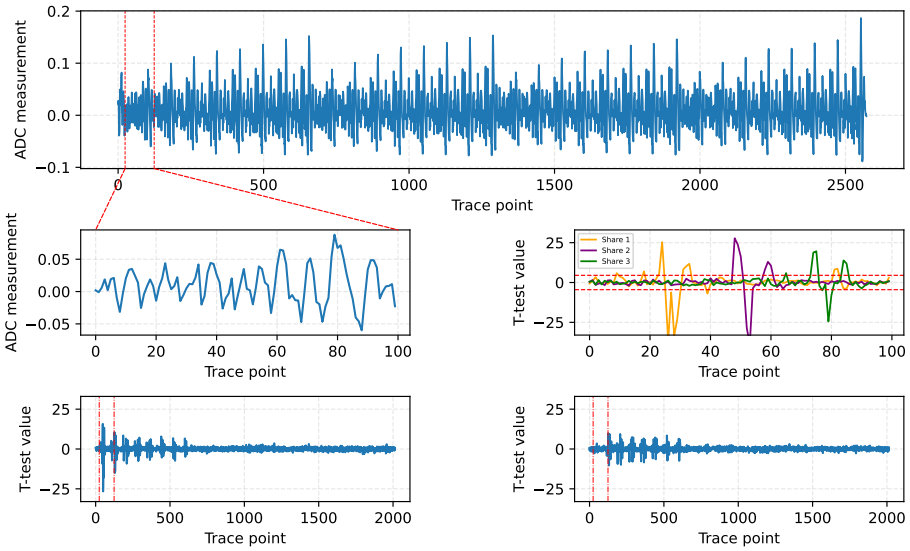


**Fig. 7.** Power trace of KeccakAbsorb function of the second-order masked implementations of `pqm4_masked` (top); More detailed power trace for the first 3 operations (middle left); T-test results for the first 8 operations (middle right); T-test results when the first byte of Share 1 = `0xFF` (bottom left); T-test when the second byte of Share 1 = `0xFF` (bottom right);

**tr**, the trained classifier outputs a prediction for the Hamming weight of the targeted intermediate.

### 4.2 Attacking phase

After the profiling phase, we proceed to the attacking phase. The assumption is that the attacker has access to a target device that is similar to the profiling device. The attacker submits chosen ciphertexts to the decapsulation procedure and collects the corresponding power traces. For each collected trace, the attacker extracts the same local leakage windows as those used in the profiling phase and applies the trained classifier to predict the Hamming weight of the targeted intermediate value.

Experimental results for the three leakage sources are shown in Table 3, Table 4, and Table 5, respectively. The Bytewise-Storage leakage in `mkm4` compiled with `-Os` enables reliable byte-level Hamming-weight classification, achieving an average accuracy of 0.9873 across all shares and Hamming-weight classes. In contrast, the implementation compiled with `-O3` yields a lower average accuracy of 0.8373. This is expected because the classifier must distinguish word-level Hamming weights in the range $\{0, \ldots, 32\}$, which is a harder classification task than byte-level recovery. We also observe that the first word is less accurately recovered than the remaining word positions. This effect can be attributed to

**Table 3.** Empirical probability (mean over 10K tests) to recover the Hamming weight of a single **byte** from a single trace of the extended 2-order masked implementation in `mkm4` library under `-Os`.

| Share | Hamming Weight | | | | | | | | | Avg |
|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | |
| **Share 1** | 0.9924 | 0.9785 | 0.9792 | 0.9778 | 0.9764 | 0.9760 | 0.9858 | 0.9828 | 0.9912 | 0.9822 |
| **Share 2** | 0.9939 | 0.9912 | 0.9835 | 0.9896 | 0.9863 | 0.9875 | 0.9882 | 0.9939 | 0.9995 | 0.9904 |
| **Share 3** | 0.9953 | 0.9870 | 0.9868 | 0.9810 | 0.9891 | 0.9879 | 0.9893 | 0.9894 | 0.9986 | 0.9894 |
| **Avg** | 0.9939 | 0.9856 | 0.9832 | 0.9828 | 0.9839 | 0.9838 | 0.9878 | 0.9887 | 0.9964 | **0.9873** |

**Table 4.** Empirical probability (mean over 10K tests) to recover the Hamming weight of a single **word** from a single trace of the extended 2-order masked implementation in `mkm4` under `-O3`.

| Share | Word Position | | | | | | | | Avg |
|---|---|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | |
| **Share 1** | 0.7735 | 0.8377 | 0.8415 | 0.8495 | 0.8458 | 0.8492 | 0.8474 | 0.8467 | 0.8396 |
| **Share 2** | 0.7759 | 0.8368 | 0.8389 | 0.8408 | 0.8367 | 0.8450 | 0.8421 | 0.8380 | 0.8365 |
| **Share 3** | 0.7805 | 0.8271 | 0.8352 | 0.8479 | 0.8342 | 0.8406 | 0.8405 | 0.8376 | 0.8359 |
| **Avg** | 0.7766 | 0.8339 | 0.8385 | 0.8461 | 0.8389 | 0.8449 | 0.8433 | 0.8408 | **0.8373** |

the three-stage pipeline of the Cortex-M4 processor, where the execution of adjacent instructions overlaps. Table 5 shows that the Bytewise-Incremental-Storage leakage in `pqm4_masked` is also highly exploitable. The average accuracy reaches 0.9913, confirming that the first store in the incremental-storage sequence provides a stable byte-level leakage source.

We have shown that the attacker can recover the Hamming weight of the targeted byte or word of the message $\mathbf{m}'$ from a single trace. We now evaluate how these Hamming weight predictions translate into full secret-key recovery using the attack strategy described in Section 3.1.

Let $M$ denote the number of message positions revealed by one trace, namely $M = 32$, 8, and 4 for the byte-wise, word-wise, and incremental-storage leakage models, respectively. Under an ideal oracle, the number of traces required for full key recovery is:

$$N_{trace} = L \times l \times \lceil n/M \rceil, \tag{1}$$

where $L$ is the number of MV-PC oracle queries required to recover $M$ secret coefficients in parallel, and $l$ and $n$ are the Kyber parameters determined by the selected security level.

In practice, the oracle derived from side-channel classification is noisy. A single-trace prediction may be insufficient for reliable full key recovery, especially for the word-wise leakage in mkm4 compiled with `-O3`. We therefore use majority voting over $t$ independent traces collected under the same attack setting. Since

**Table 5.** Empirical probability (mean over 10K tests) to recover the Hamming weight of a single **byte** from a single trace of 2-order masked implementation in "`pqm4_masked`" library under `-O3`.

| **Share** | **Hamming Weight** | | | | | | | | | **Avg** |
|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | |
| **Share 1** | 0.9963 | 0.9937 | 0.9954 | 0.9813 | 0.9971 | 0.9859 | 0.9901 | 0.9924 | 0.9965 | 0.9921 |
| **Share 2** | 0.9981 | 0.9882 | 0.9944 | 0.9906 | 0.9901 | 0.9918 | 0.9943 | 0.9960 | 1.0000 | 0.9937 |
| **Share 3** | 0.9942 | 0.9875 | 0.9874 | 0.9842 | 0.9834 | 0.9896 | 0.9835 | 0.9870 | 0.9959 | 0.9881 |
| **Avg** | 0.9962 | 0.9898 | 0.9924 | 0.9854 | 0.9902 | 0.9891 | 0.9893 | 0.9918 | 0.9975 | **0.9913** |

**Table 6.** Estimated number of side-channel traces required to recover the secret key of Kyber-1024 (using majority voting to enhance accuracy, where the threshold for enhanced accuracy $p_{\text{bit}}^{(t)}$ is set to be greater than 0.9999). Note that for the first and third targets, we use the average accuracy for evaluation. For `mkm4 (-O3)`, to ensure experimental reliability, we evaluate based on the word position with the lowest accuracy. Specifically, for $M = 7$, this means that the result of the first word is discarded, thereby reducing $t$.

| Target | $M$ | $p_{\text{bit}}^{(t)}$ and $t$ | | | | | | $N'_{\text{traces}}$ |
|---|---|---|---|---|---|---|---|---|
| | | 1st-order | | 2nd-order | | 3rd-order | | |
| `mkm4` (`-Os`) | 32 | 0.99999 | 7 | 0.99994 | 7 | 0.99997 | 9 | $t \times L \times 32$ |
| `mkm4` (`-O3`) | 8 | 0.99990 | 141 | 0.99990 | 477 | 0.99990 | 1571 | $t \times L \times 128$ |
| | 7 | 0.99991 | 57 | 0.99990 | 131 | 0.99990 | 835 | $t \times L \times 148$ |
| `pqm4_masked` | 4 | 0.99999 | 5 | 0.99999 | 7 | 0.99996 | 7 | $t \times L \times 256$ |

each majority vote consumes $t$ traces, the total number of traces becomes:

$$N'_{trace} = t \times \big(L \times l \times \lceil n/M \rceil\big), \tag{2}$$

Other methods that leverage multiple traces can also be evaluated using Equation 2.

Table 6 summarizes the resulting trace requirements for Kyber-1024 under the three leakage sources and for first, second, and third order masking. The results show that our method successfully instantiates MV-PC oracle-based full key recovery for all evaluated leakage sources and masking orders. As the masking order increases, the required repetition factor $t$ also increases, since the recovered message bit depends on Hamming-weight predictions from more masked shares. Nevertheless, the attack remains effective even in the third-order setting. This indicates that increasing the masking order does not necessarily improve resistance to MV-PC oracle-based attacks when memory operations continue to expose exploitable Hamming-weight leakage from the masked shares. Moreover, for `mkm4` compiled with `-O3`, the two settings $M = 8$ and $M = 7$ highlight

**Table 7.** Comparison of different methods used in previous works and their impact on key metrics.

| Method | Previous works | $n$ | $L$ | $t$ | Offline cost |
|---|---|---|---|---|---|
| Cyclic rotation | [6,18,22] | ○ | ○ | ▲ | ○ |
| Huffman Coding | [14,15,20,23] | ○ | ▼ | ○ | ○ |
| Adaptive attack | [8,15,17] | ○ | ▼ | ○ | ○ |
| Soft analysis | [9,11,25] | ○ | ○ | ▼ | ▲ |
| Lattice reduction | [8,16,17] | ▼ | ○ | ○ | ▲ |

▲: increase; ▼: decrease; ○: no significant change

the tradeoff between parallelism and reliability. Using all eight word positions maximizes the number of coefficients tested per trace, but the first word has noticeably lower recovery accuracy and therefore requires many more repetitions. Discarding this unstable word reduces the parallelism from $M = 8$ to $M = 7$, but substantially lowers $t$ and hence the final trace complexity.

Table 7 summarizes how techniques used in prior key-recovery attacks affect the total trace complexity. From Equation 2, the online cost is mainly determined by the number of recovered coefficients $n$, the attack-encoding length $L$, and the repetition factor $t$ required by the noisy oracle. Huffman coding and adaptive attacks reduce $L$, soft-analysis methods reduce $t$ at the cost of higher offline computation, and lattice reduction reduces $n$ with additional offline cost. Cyclic rotation instead improves oracle reliability through repeated observations and therefore increases $t$. Thus, our majority-voting evaluation provides a conservative baseline for the leakage sources studied here, while the techniques in Table 7 can be viewed as complementary optimizations that may further reduce the overall attack cost.

## 5 Conclusion

In this paper, we present the first MV-PC oracle-based side-channel attack against higher-order masked implementations of Kyber. We propose an efficient message-recovery method that directly derives selected bits of the decrypted message from the Hamming-weight leakage of masked shares, thereby significantly reducing the cost of instantiating the MV-PC oracle and making the attack practical for higher-order masking.

We evaluate the proposed attack on two masked Kyber implementations running on an ARM Cortex-M4 platform under different compiler optimization settings. Our analysis identifies three exploitable storage-related leakage sources, and our experiments show that these leakages enable reliable Hamming-weight recovery. Based on the measured classifier accuracies, we estimate the trace requirements for full secret-key recovery under first-, second-, and third-order masking using majority voting. These results indicate that, for the evaluated

implementations, increasing the masking order does not necessarily prevent MV-PC oracle-based attacks when exploitable storage-related leakage remains. They therefore highlight the need for further research on secure and efficient masking implementations and suitable hiding countermeasures.

**Acknowledgments.** This work was supported by the National Natural Science Foundation of China (Grant Nos. 62502275 and 62672136) and Quan Cheng Laboratory (Grant No. QCL20250101).

## References

1. Module-lattice-based key-encapsulation mechanism standard. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 203 (2024), https://doi.org/10.6028/NIST.FIPS.203
2. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., Liu, Y.K.: Status report on the third round of the nist post-quantum cryptography standardization process (2022), https://doi.org/10.6028/NIST.IR.8413-upd1
3. Backlund, L., Ngo, K., Gärtner, J., Dubrova, E.: Secret key recovery attack on masked and shuffled implementations of crystals-kyber and saber. In: Applied Cryptography and Network Security Workshops (ACNS). pp. 159–177 (2023), https://doi.org/10.1007/978-3-031-41181-6_9
4. Bronchain, O., Cassiers, G.: Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based kems. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) **2022**(4), 553–588 (2022), https://doi.org/10.46586/tches.v2022.i4.553-588
5. Du, J., Wang, Z., Yu, A.: Revisiting the masking strategy: A side-channel attack on crystals-kyber. IEEE Transactions on Information Forensics and Security **20**, 3387–3399 (2025), https://doi.org/10.1109/TIFS.2025.3550061
6. Dubrova, E., Ngo, K., Gärtner, J., Wang, R.: Breaking a fifth-order masked implementation of crystals-kyber by copy-paste. In: Fukumitsu, M., Hasegawa, S. (eds.) Asia Public-Key Cryptography Workshop (APKC). pp. 10–20. ACM (2023), https://doi.org/10.1145/3591866.3593072
7. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side-channel resistance validation. In: NIST Non-Invasive Attack Testing Workshop. vol. 7, pp. 115–136 (2011)
8. Guo, Q., Mårtensson, E., Åström, A.: The perils of limited key reuse: Adaptive and parallel mismatch attacks with post-processing against kyber. IACR Communications in Cryptology (IACR CiC) (2024), https://doi.org/10.62056/a3n5qj888
9. Guo, Q., Nabokov, D., Nilsson, A., Johansson, T.: SCA-LDPC: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes. In: ASIACRYPT. pp. 203–236. LNCS 14441, Springer-Verlag (2023), https://doi.org/10.1007/978-981-99-8730-6_7
10. Heinz, D., Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels, A.: First-order masked kyber on ARM cortex-m4 (2022), https://eprint.iacr.org/2022/058
11. Hermelink, J., Mårtensson, E., Tran, M.: Noise-tolerant plaintext-checking oracle attacks - A soft-analytic approach applied to ML-KEM. Cryptology ePrint Archive, Paper 2025/1496 (2025), https://eprint.iacr.org/2025/1496

12. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Advances in Cryptology (CRYPTO). pp. 388–397. LNCS 1666, Springer-Verlag (1999), https://doi.org/10.1007/3-540-48405-1_25
13. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography (DCC) **75**(3), 565–599 (2015), https://doi.org/10.1007/s10623-014-9938-4
14. Li, J., Cheng, C., Shen, M., Chen, P., Guo, Q., Liu, D., Wu, L., Weng, J.: Grafted trees bear better fruit: An improved multiple-valued plaintext-checking side-channel attack against kyber. In: Design, Automation, and Test in Europe (DATE). pp. 1–7. IEEE (2025), https://doi.org/10.23919/DATE64628.2025.10992764
15. Li, Z., Xu, J., Zou, Y., Hu, L.: Key recovery attack on crystals-kyber and saber kems in key reuse scenario. In: European Symposium on Research in Computer Security (ESORICS). pp. 259–278. Springer (2024), https://doi.org/10.1007/978-3-031-70896-1_13
16. Mi, R., Jiang, H., Zhang, Z.: Lattice reduction meets key-mismatch: New misuse attack on lattice-based NIST candidate kems (2022), https://eprint.iacr.org/2022/1064
17. Mondal, P., Kundu, S., Bhattacharya, S., Karmakar, A., Verbauwhede, I.: A practical key-recovery attack on lwe-based key-encapsulation mechanism schemes using rowhammer. In: Applied Cryptography and Network Security Workshops (ACNS). pp. 271–300 (2024), https://doi.org/10.1007/978-3-031-54776-8_11
18. Ngo, K., Dubrova, E., Guo, Q., Johansson, T.: A side-channel attack on a masked IND-CCA secure saber KEM implementation. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) **2021**, 676–707 (2021), https://doi.org/10.46586/tches.v2021.i4.676-707
19. Poilbout, P., Roche, T., Imbert, L.: Handling noisy plaintext checking oracles with spirit - application to kyber. In: International Conference on Post-Quantum Cryptography (PQCrypto). pp. 311–341. LNCS 16492, Springer (2026), https://doi.org/10.1007/978-3-032-22698-3_10
20. Qin, Y., Cheng, C., Zhang, X., Pan, Y., Hu, L., Ding, J.: A systematic approach and analysis of key mismatch attacks on lattice-based NIST candidate kems. In: ASIACRYPT. vol. 13093, pp. 92–121 (2021), https://doi.org/10.1007/978-3-030-92068-5_4
21. Rajendran, G., Ravi, P., D'Anvers, J., Bhasin, S., Chattopadhyay, A.: Pushing the limits of generic side-channel attacks on lwe-based kems: Parallel PC oracle attacks on kyber KEM and beyond. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) **2023**(2), 418–446 (2023), https://doi.org/10.46586/tches.v2023.i2.418-446
22. Ravi, P., Bhasin, S., Roy, S.S., Chattopadhyay, A.: 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), https://doi.org/10.1109/TIFS.2021.3139268
23. Shen, M., Cheng, C., Zhang, X., Guo, Q., Jiang, T.: 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 (TCHES) **2023**(1), 89–112 (2023), https://doi.org/10.46586/tches.v2023.i1.89-112
24. Tanaka, Y., Ueno, R., Xagawa, K., Ito, A., Takahashi, J., Homma, N.: Multiple-valued plaintext-checking side-channel attacks on post-quantum KEMs. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) **2023**(3), 473–503 (2023), https://doi.org/10.46586/tches.v2023.i3.473-503

25. Tong, X., Nabokov, D., Guo, Q.: LDPC strikes again: New key-recovery chosen-ciphertext side-channel attacks on ntru-based kems (2025), https://doi.org/10.36227/techrxiv.176288019.98130379/v1
26. Wang, R., Brisfors, M., Dubrova, E.: A side-channel attack on a higher-order masked crystals-kyber implementation. In: Applied Cryptography and Network Security Workshops (ACNS). vol. 14585, pp. 301–324 (2024), https://doi.org/10.1007/978-3-031-54776-8_12