A Side-Channel Attack on a Bitsliced Higher-Order Masked CRYSTALS-Kyber Implementation

Ruize Wang, Martin Brisfors and Elena Dubrova

KTH Royal Institute of Technology, Stockholm, Sweden
{ruize,brisfors,dubrova}@kth.se

Abstract. In response to side-channel attacks on masked implementations of post-quantum cryptographic algorithms, a new bitsliced higher-order masked implementation of CRYSTALS-Kyber has been presented at CHES'2022. The bitsliced implementations are typically more difficult to break by side-channel analysis because they execute a single instruction across multiple bits in parallel. However, in this paper, we reveal new vulnerabilities in the masked Boolean to arithmetic conversion procedure of this implementation that make the shared and secret key recovery possible. We also present a new chosen ciphertext construction method which maximizes secret key recovery probability for a given message bit recovery probability. We demonstrate practical shared and secret key recovery attacks on the first-, second- and third-order masked implementations of Kyber-768 in ARM Cortex-M4 using profiled deep learning-based power analysis.

Keywords: Public-key cryptography · Post-quantum cryptography · Kyber · LWE/LWR-based KEM · Side-channel attack

1 Introduction

CRYSTALS-Kyber is a key encapsulation mechanism (KEM) which is indistinguishable under an adaptive chosen-ciphertext attack (IND-CCA2-secure) in the classical and quantum random oracle models [3]. The security of Kyber relies on the hardness of the module learning with errors (M-LWE) problem that comes from inserting unknown noise into otherwise linear equations. Kyber has recently been selected for standardization by the National Institute of Standards and Technology (NIST) [24] and included in the National Security Agency (NSA) suite of cryptographic algorithms recommended for national security systems [1].

However, the theoretical IND-CCA2 security of Kyber KEM can potentially be bypassed by a side-channel attack of its implementation executed on a physical device. Side-channel attacks on software [5, 6, 29, 33, 34, 36, 38, 39, 42, 43] and hardware [21, 31] implementations of Kyber have been demonstrated. The discovered vulnerabilities promoted stronger mitigation techniques against side-channel attacks, e.g. [4, 20, 35] and helped strengthen Kyber implementations that were released later [7, 11]. In these implementations, the known vulnerabilities are typically patched. Indeed, the experiments presented in this paper show that side-channel information extracted from the higher-order masked implementation of Kyber by Bronchain et al. in [9] is more difficult to exploit using previous methods.

Contributions: We discovered new vulnerabilities in a higher-order masked implementation of Kyber [9] that result in an effective message/shared key recovery attack. These vulnerabilities are located in the masked Boolean to arithmetic conversion procedure which is carried out during the re-encryption step of decapsulation.

We also present a new chosen ciphertext construction method which maximizes secret key recovery probability for a given message bit recovery probability. This method uses $3 \times k$ chosen ciphertexts to extract the secret key of Kyber from a masked implementation, where k is the module rank. While this number is the same as in the chosen ciphertext construction method of Ravi et al. [29], the presented way of mapping message bits into the secret key coefficients can raise the likelihood of successfully recovering the full secret key by up to 39% compared to the worst case.

We demonstrate practical shared and secret key recovery attack on ω-order masked implementations of Kyber-768 in ARM Cortex-M4 using profiled deep learning-based power analysis, for $\omega \in \{1, 2, 3\}$. The training of neural networks is performed on traces captured from five profiling devices, different from the device under attack (DUA). The message recovery is carried out using the single-step method of Ngo et al. [25] which extracts the message without extracting each share explicitly. For $\omega = 3$, we use the recursive learning method of [13]; otherwise neural networks do not learn. Our experimental results show that, for the first-order masked implementation run on a DUA, we can recover the shared key from three traces with a close to 1 probability, and the secret key from 18 traces with 0.94 probability.

The rest of this paper is organized as follows. Section 2 reviews previous work on side-channel analysis of Kyber implementations. Section 3 gives a background on Kyber algorithm, shared key establishment protocol and masking. Section 4 defines the adversary model. Section 5 describes the profiling and attack stages. Section 6 presents the equipment used in the experiments. Section 7 analyses side-channel leakage of three different implementations of Kyber and describes new vulnerabilities in the implementation of [9]. Section 8 presents neural network training strategy. Section 9 introduces the new chosen ciphertext construction attack method. Section 10 summarizes experimental results. Section 11 discusses possible countermeasures. Section 12 concludes the paper.

2 Previous work

Since the beginning of the NIST post-quantum cryptography standardization process in 2016, many different side-channel attacks on software [6,34,36,38] and hardware [21,31] implementations of Kyber have been presented. In response to these attacks, protected implementations have been developed such as [7, 9, 14, 19, 32].

In [29], near field EM based secret key recovery attacks on unprotected and protected implementations of Kyber are described. In these attacks, Hamming weight-based templates are constructed for the message decoding operation to recover the message bits. For secret key recovery, $3 \times k$ chosen ciphertexts are required to identify each key coefficient uniquely, where k is the rank of the module. It is also shown how a first-order masked implementation can be broken in two steps, by attacking each share individually.

In [18], a chosen ciphertext side-channel attack on a first-order masked software implementation Kyber combined with belief propagation is presented. The attack can recover the secret key from k traces from the inverse NTT step of decryption, where k is the module rank, for a noise tolerance level $\sigma \leq 1.2$ in the HW leakage on simulated data. Furthermore, the attack can recover the secret key from a single trace if the noise is in the range $\sigma \leq 0.5$ to $\sigma \leq 0.7$, depending on the Kyber parameter set.

In [33] a chosen ciphertext side-channel attack is presented which uses codes to detect faulty positions in the initially recovered secret key. These positions are further corrected with additional traces. An EM-based template attack on an unprotected software implementation of Kyber-512 is demonstrated which can recover a full secret key using 1619 traces on average with 0.4 out of 512 faulty coefficients on average. Another code-based chosen ciphertext construction method for CRYSTALS-Kyber, low density paritiy check codes, targeting masked message encoding using the implementation by Heinz et al [19] is described in [16].

Yet another interesting chosen ciphertext construction method is presented in [28]. The total number of queries required for the secret key recovery is reduced by using binary decision trees. However, the downside of this method is that it relies on an unbalanced distribution of the coefficients of the secret key. In contract, the chosen ciphertext construction method introduced in this paper is equally applicable to algorithms with a uniform distribution of secret key coefficients. Furthermore, it is applicable to masked implementations, whereas the method of [28] is not.

In [5], a chosen ciphertext side-channel attack on a first-order masked and shuffled software implementation of Kyber-768 on an ARM Cortex-M4 is demonstrated which can extract the secret key from 38,016 power traces. The main idea is to recover shuffling indices 0 and 255, extract the corresponding two message bits, and then cyclically rotate the message by modifying the ciphertext. In this way, all message bits are extracted using 128 rotations.

In [38], a side-channel attack on the first-order masked implementation of CRYSTALS-Kyber targeting the message encoding vulnerability found in [34] is presented. In [6], side-channel attacks on two implementations of masked polynomial comparison are demonstrated on the example of CRYSTALS-Kyber.

In [13], a message recovery attack on the higher-order masked implementation of CRYSTALS-Kyber by Heinz et al. in [19] is presented. A new neural network

```
KYBER.CPAPKE.KeyGen()
 1: $(\rho, \sigma) \leftarrow \mathcal{U}(\{0,1\}^{256})$
 2: $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{k \times k}; \rho)$
 3: $\boldsymbol{s}, \boldsymbol{e} \leftarrow \beta_{\eta_1}(R_q^{k \times 1}; \sigma)$
 4: $\boldsymbol{t} = \mathsf{Encode}_{12}(\boldsymbol{A}\boldsymbol{s} + \boldsymbol{e})$
 5: $\boldsymbol{s} = \mathsf{Encode}_{12}(\boldsymbol{s})$
 6: return $(pk = (\boldsymbol{t}, \rho), sk = \boldsymbol{s})$
```

```
KYBER.CPAPKE.Dec($\boldsymbol{s}, c$)
 1: $\boldsymbol{u} = \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c_1), d_u)$
 2: $v = \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c_2), d_v)$
 3: $\boldsymbol{s} = \mathsf{Decode}_{12}(\boldsymbol{s})$
 4: $m = \mathsf{Encode}_1(\mathsf{Compress}_q(v - \boldsymbol{s} \cdot \boldsymbol{u}, 1))$
 5: return $m$
```

```
KYBER.CPAPKE.Enc($pk = (\boldsymbol{t}, \rho), m, r$)
 1: $\boldsymbol{t} = \mathsf{Decode}_{12}(\boldsymbol{t})$
 2: $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{k \times k}; \rho)$
 3: $\boldsymbol{r} \leftarrow \beta_{\eta_1}(R_q^{k \times 1}; r)$
 4: $\boldsymbol{e}_1 \leftarrow \beta_{\eta_2}(R_q^{k \times 1}; r)$
 5: $e_2 \leftarrow \beta_{\eta_2}(R_q^{1 \times 1}; r)$
 6: $\boldsymbol{u} = \boldsymbol{A}^T \boldsymbol{r} + \boldsymbol{e}_1$
 7: $v = \boldsymbol{t}^T \boldsymbol{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
 8: $c_1 = \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\boldsymbol{u}, d_u))$
 9: $c_2 = \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
10: return $c = (c_1, c_2)$
```

Fig. 1. KYBER.CPAPKE algorithms from [3] (simplified).

```
KYBER.CCAKEM.KeyGen()
 1: $z \leftarrow \mathcal{U}(\{0,1\}^{256})$
 2: $(pk, \boldsymbol{s}) =$
    KYBER.CPAPKE.KeyGen()
 3: $sk = (\boldsymbol{s}, pk, \mathcal{H}(pk), z)$
 4: return $(pk, sk)$
```

```
KYBER.CCAKEM.Encaps($pk$)
 1: $m \leftarrow \mathcal{U}(\{0,1\}^{256})$
 2: $m = \mathcal{H}(m)$
 3: $(\hat{K}, r) = \mathcal{G}(m, \mathcal{H}(pk))$
 4: $c =$ KYBER.CPAPKE.Enc($pk, m, r$)
 5: $K = \mathsf{KDF}(\hat{K}, \mathcal{H}(c))$
 6: return $(c, K)$
```

```
KYBER.CCAKEM.Decaps($sk$,c)
 1: $m' =$ KYBER.CPAPKE.Dec($\boldsymbol{s}, c$)
 2: $(\hat{K}', r') = \mathcal{G}(m', \mathcal{H}(pk))$
 3: $c' =$ KYBER.CPAPKE.Enc($pk, m', r'$)
 4: if $c = c'$ then
 5:     return $K = \mathsf{KDF}(\hat{K}', \mathcal{H}(c))$
 6: else
 7:     return $K = \mathsf{KDF}(z, \mathcal{H}(c))$
 8: end if
```

Fig. 2. KYBER.CCAKEM algorithms from [3] (simplified).

training method called *recursive learning* is introduced which constructs the initial state for a neural network model based on the states of models trained for the attack on a lower-order masked implementation.

3 Background

In this section, we describe notation used in the paper, Kyber algorithm specification [3], shared key establishment protocol and masking countermeasure.

3.1 Notation

Let $\mathbb{Z}_q$ be the ring of integers modulo a prime q and R_q be the quotient ring $\mathbb{Z}_q[X]/(X^n + 1)$. We use regular font letters to denote elements in R_q, bold

Table 1. Parameters of different versions of Kyber.

Version	n	k	q	η_1	η_2	(d_u, d_v)
Kyber-512	256	2	3329	3	2	(10, 4)
Kyber-768	256	3	3329	2	2	(10, 4)
Kyber-1024	256	4	3329	2	2	(11, 5)

lower-case letters to represent vectors with coefficients in R_q, and bold upper-case letters to represent matrices. The transpose of a vector $\boldsymbol{v}$ (or matrix $\boldsymbol{A}$) is denoted by $\boldsymbol{v}^T$ (or $\boldsymbol{A}^T$). The ith entry of a vector $\boldsymbol{v}$ is denoted by $\boldsymbol{v}[i]$. The polynomial multiplication is denoted by the sign "$\cdot$". The Boolean XOR is denoted by the sign "$\oplus$". The term $\lceil x \rfloor$ denotes rounding of x to the closest integer with ties being rounded up.

The term $x \leftarrow \mathcal{D}(S; r)$ stands for sampling x from a probability distribution $\mathcal{D}$ over a set S using seed r. The uniform distribution is denoted by $\mathcal{U}$. The centered binomial distribution with parameter μ is denoted by B_μ.

3.2 Kyber algorithm

Kyber [3] consists of a chosen-plaintext attack (CPA)-secure PKE scheme, KYBER.CPAPKE, and a CCA-secure KEM scheme, KYBER.CCAKEM, which is built on the top of KYBER.CPAPKE using a version of the Fujisaki-Okamoto (FO) transform [15]. These schemes are described in Fig. 1 and Fig. 2 respectively.

Inputs and outputs to all API functions of Kyber are byte arrays. Kyber works with vectors of ring elements in R_q^k, where k is the rank of the module defining the security level. There are three versions of Kyber: Kyber-512, Kyber-768 and Kyber-1024, for $k = 2, 3$ and 4, respectively, see Table 1 for details. In this paper, we focus on Kyber-768.

Kyber uses the number-theoretic transform (NTT) to perform multiplications in R_q efficiently. The NTT details are ommitted from Fig. 1 and Fig. 2 to simplify the pseudocode.

The Decode_l function decodes an array of $32l$ bytes into a polynomial with n coefficients in the range $\{0, 1, \cdots, 2^l - 1\}$. The Encode_l function is the inverse of Decode_l. It first encodes each polynomial coefficient individually and then concatenates the output byte arrays.

The $\mathsf{Compress}_q(x, d)$ and $\mathsf{Decompress}_q(x, d)$ functions, for $x \in \mathbb{Z}_q$ and $d < \lceil \log_2(q) \rceil$, are defined by:

$$\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod^+ 2^d,$$
$$\mathsf{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor.$$

If $\mathsf{Compress}_g$ or $\mathsf{Decompress}_q$ is applied to $x \in \mathbb{R}_q$ or $x \in \mathbb{R}_q^k$, the function is applied to each coefficient individually. These functions enable the removal of

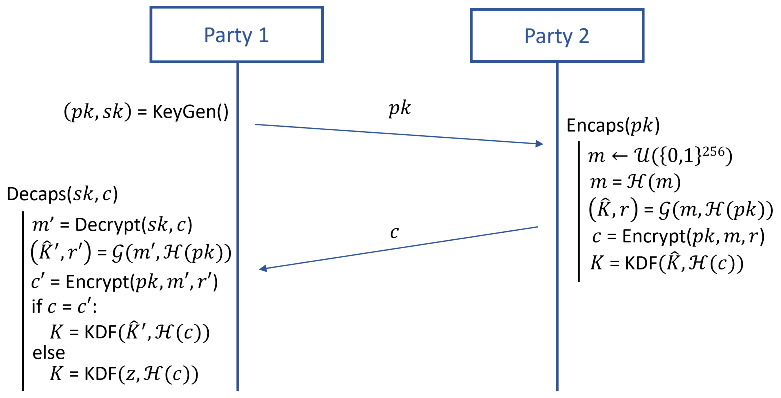
Party 1
Party 2
(pk, sk) = KeyGen()
pk
Encaps(pk)
$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$m = \mathcal{H}(m)$
$(\hat{K}, r) = \mathcal{G}(m, \mathcal{H}(pk))$
c = Encrypt(pk, m, r)
K = KDF($\hat{K}, \mathcal{H}(c)$)
c
Decaps(sk, c)
m' = Decrypt(sk, c)
$(\hat{K}', r') = \mathcal{G}(m', \mathcal{H}(pk))$
c' = Encrypt(pk, m', r')
if $c = c'$:
K = KDF($\hat{K}', \mathcal{H}(c)$)
else
K = KDF($z, \mathcal{H}(c)$)

Fig. 3. A shared key establishment protocol based on Kyber KEM.

some low-order bits in the ciphertext without significantly affecting the correctness probability of decryption.

The functions $\mathcal{G}$ and $\mathcal{H}$ represent the SHA3-512 and SHA3-256 hash functions, respectively. The KDF is a key derivation function. It is realized by SHAKE-256.

3.3 Shared key establishment protocol

Figure 3 depicts a possible shared (session) key establishment protocol based on Kyber KEM. Two parties, Party 1 and Party 2, are willing to establish a shared key. To accomplish this, Party 1 employs the key generation algorithm KYBER.CCAKEM.KeyGen() to generate a (public, secret) key pair (pk, sk) and sends pk to Party 2. Party 2 employs the encapsulation algorithm KYBER.CCAKEM.Encaps() to generate a ciphertext c encapsulating a shared key K and transmits c to Party 1. Finally, Party 1 obtains K from c using the decapsulation algorithm KYBER.CCAKEM.Decaps().

3.4 Masking countermeasure

Masking is a popular method to protect implementations of cryptographic algorithms from power and EM side-channel attacks [10]. A ω-order mask splits a sensitive variable x into $\omega + 1$ shares: $x_0, x_1, \ldots, x_\omega$, where x can be reconstructed from the shares as $x = x_0 \circ x_1 \circ \ldots \circ x_\omega$. The type of the operation "$\circ$" depends on the masking method being used, for instance, in arithmetic masking "$\circ$" is the arithmetic addition, while in Boolean masking it is the XOR.

By performing operations separately on shares, the computations do not involve x directly, thereby preventing leakage of side-channel information about x in theory. The shares are randomized at each execution. Typically, the randomization is performed by assigning random masks $x_0, x_1, \ldots, x_{\omega-1}$ to ω shares and deriving the last share as $x_\omega = x - (x_0 + x_1 + \ldots + x_{\omega-1})$ for arithmetic masking or $x_\omega = x \oplus x_0 \oplus x_1 \oplus \ldots \oplus x_\omega$ for Boolean masking.

4 Adversary model

An adversary model is typically defined using three components: assumptions, goals and capabilities [12].
Assumptions: We assume that an adversary has a physical access to the DUA which runs the Kyber KEM decapsulation algorithm. We also assume that the adversary possesses fully controllable profiling devices that are similar to the DUA. In addition, we assume that the keys (pk, sk) are static.
Capabilities: The adversary is a clever outsider who has equipment and tools for power analysis, as well as expertise in side-channel attacks, Kyber KEM, and deep learning. The adversary is capable of eavesdropping on the channel between the DUA and the server and query the DUA with chosen ciphertexts.
Goals: The goal of the adversary is to extract the shared key K and/or the long-term secret key sk of Kyber from its implementation running on the DUA. Note that the long-term secret key recovery implies the shared key recovery, but not vice versa.

5 Attack description

Fig. 4 illustrates the main steps of the attack.

5.1 Profiling stage

At the profiling stage, the adversary first uses $\mathsf{KYBER.CCAKEM.KeyGen}()$ to generate a key pair (pk_p, sk_p). Then he/she selects uniformly at random a message $m_p \in \{0,1\}^{256}$ and uses $\mathsf{KYBER.CCAPKE.Enc}()$ to compute a ciphertext c_p encrypting m_p. Knowing the message contained in c_p is necessary for creating a labeled dataset for neural network training.

These steps are repeated multiple times until a profiling dataset of the desired size is gathered. Note that a labeled dataset can be created either using a static key pair, or a set of key pairs which is re-generated for each message m_p. This does not affect the success probability of the presented attack.

Then, the adversary runs $\mathsf{KYBER.CCAKEM.Decaps}()$ on a profiling device to decapsulate each c_p in the dataset and measure the total power consumption of the device during the execution of the algorithm. The resulting power trace T_p is recorded.

Finally, the adversary uses the resulting labeled data set to train a neural network $\mathcal{M}$ which learns the leakage profile of $\mathsf{KYBER.CCAKEM.Decaps}()$ in order to predict message bit values from power traces.

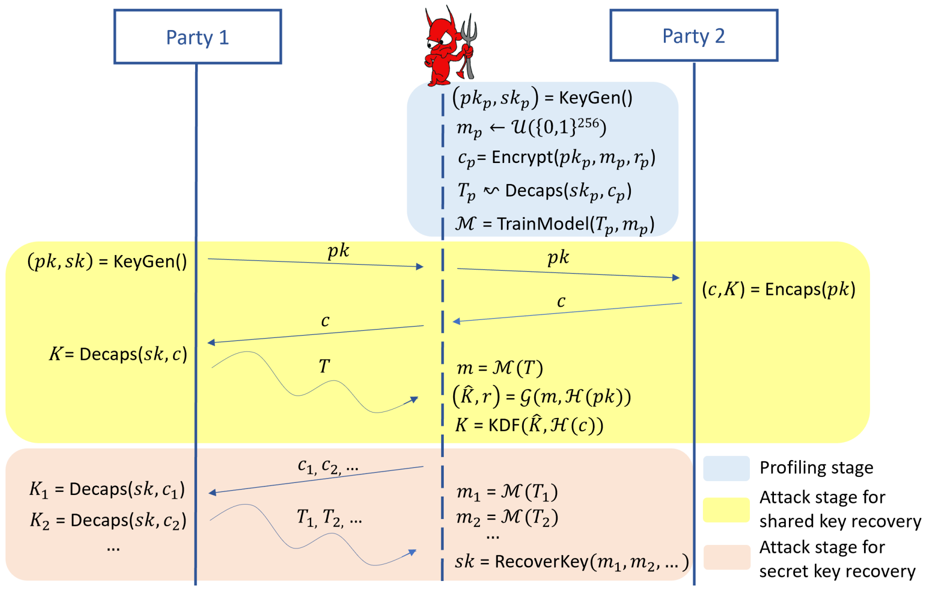
Party 1
Party 2
(pk_p, sk_p) = KeyGen()
$m_p \leftarrow \mathcal{U}(\{0,1\}^{256})$
c_p= Encrypt(pk_p, m_p, r_p)
$T_p \rightsquigarrow$ Decaps(sk_p, c_p)
$\mathcal{M}$ = TrainModel(T_p, m_p)
(pk, sk) = KeyGen()
pk
pk
(c,K) = Encaps(pk)
c
c
K= Decaps(sk, c)
T
$m = \mathcal{M}(T)$
$(\hat{K}, r) = \mathcal{G}(m, \mathcal{H}(pk))$
K = KDF($\hat{K}, \mathcal{H}(c)$)
$c_1, c_2, \ldots$
K_1 = Decaps(sk, c_1)
K_2 = Decaps(sk, c_2)
...
$T_1, T_2, \ldots$
$m_1 = \mathcal{M}(T_1)$
$m_2 = \mathcal{M}(T_2)$
...
sk = RecoverKey($m_1, m_2, \ldots$)
Profiling stage
Attack stage for shared key recovery
Attack stage for secret key recovery

Fig. 4. Attack scenario.

5.2 Attack stage

To recover the shared key K, the adversary eavesdrops on the communication channel between the two parties to obtain the public key pk and the ciphertext c containing the encapsulated K. The adversary also measures the power consumption of the DUA during the execution of the decapsulation algorithm with c as input. The segments of the resulting power trace T corresponding to the processing of the message m encrypted in c are extracted. These segments are given as input to the model $\mathcal{M}$ trained at the profiling stage to predict the bits of m. Once m is recovered, the pre-key $\hat{K}$ is derived as $(\hat{K}, r) = \mathcal{G}(m, \mathcal{H}(pk))$ and then the shared key K is computed as $K = \mathsf{KDF}(\hat{K}, \mathcal{H}(c))$.

To recover the secret key sk, the DUA is queried with chosen ciphertexts $c_1, c_2, \ldots$ and the power consumption of the DUA during the execution of the decapsulation algorithm is measured. We describe the method for constructing chosen ciphertexts in Section 9. The messages $m_1, m_2, \ldots$ are extracted from the recorded power traces $T_1, T_2, \ldots$ similarly to the case of shared key recovery attack. A difference between the two attacks is that chosen ciphertexts are malformed and do not pass the FO transform. This is however not important for the presented attack since it targets an earlier step of the decapsulation algorithm (re-encryption). Finally, the extracted messages $m_1, m_2, \ldots$ are mapped into the coefficients of the secret key according to the mapping table of the chosen ciphertexts construction method.

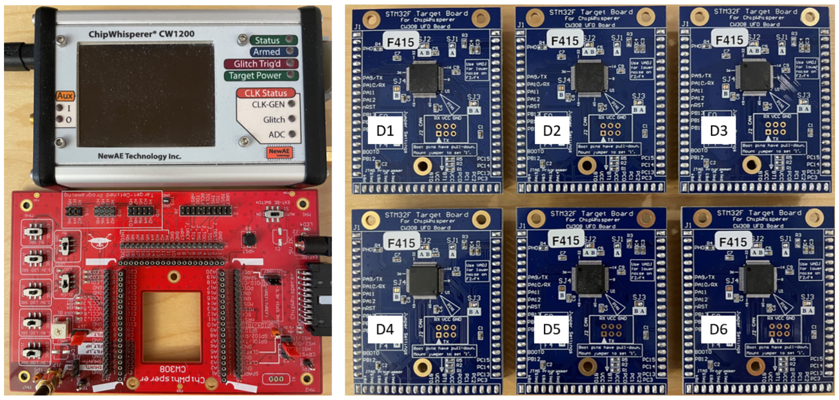

Fig. 5. The equipment used in the experiments.The devices $D_1 - D_5$ are used for the profiling and D_6 is used for the attack.

6 Experimental setup

The equipment used in our experiments is shown in Fig. 5. It consists of the ChipWhisperer-Pro, the CW308 UFO main board and six CW308T-STM32F4 target boards (see Fig. 5 in the Appendix). Each target board contains a STM32F415-RGT6 chip based on ARM Cortex-M4 32-bit RISC core operating at a frequency of 24Mhz. The traces are acquired with the sampling rate of 96MS/s.

Three C implementations of Kyber are used in the experiments:

1. The unprotected implementation by Kannwischer et al. [22].
2. The first-order masked implementation by Heinz et al. [19].
3. The higher-order masked implementation by Bronchain et al. [9].

All implementations are compiled using `arm-none-eabi-gcc` with the highest optimization level `-O3` (recommended default).

7 Leakage analysis

The presented attack targets the message encoding operation at the re-encryption step of the FO transform. In this section, we first analyse the unprotected implementation of the message encoding operation from the post-quantum crypto library for the ARM Cortex-M4 developed by Kannwischer et al. [22]. Then, we compare the realizations of message encoding in the implementations of Heinz et al. [19] and Bronchain et al. [9]. We show that the leakage of the implementation in [9] is significantly weaker than the one in [19]. Thus, the implementation of [9] is more difficult to break. The presented attack would not be as effective if we would use only previously known leakage points. However, we discovered two new leakage points in the masked Boolean to arithmetic conversion procedure of the implementation in [9] which results in an effective attack.

```
void poly_frommsg(poly *r, unsigned char msg[32])
int i,j;
uint16_t mask;
 1: for (i = 0; i < 32; i++) do
 2:   for (j = 0; j < 8; j++) do
 3:     mask = -((msg[i]>>j) & 1);
 4:     r->coeffs[8 * i + j] = mask & ((KYBER_Q+1)/2);
 5:   end for
 6: end for
```

Fig. 6. The C code of `poly_frommsg()` procedure from [22].

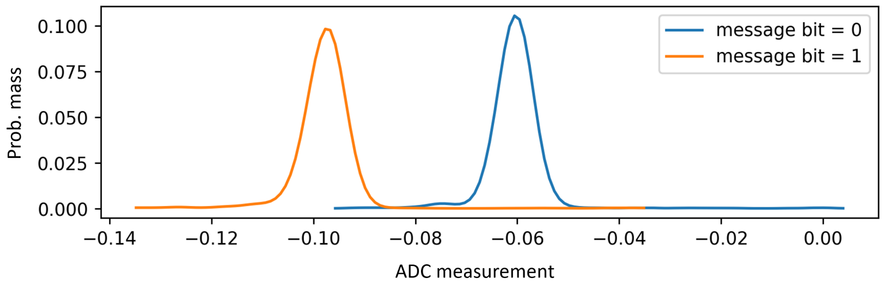

Fig. 7. Distributions of power consumption during the processing of a single message bit by `poly_frommsg()` procedure of the unmasked implementation in [22].

7.1 Unprotected message encoding

The message encoding operation converts an array of $n/8$ bytes representing a message m into a polynomial f in which each of the n coefficients, $f[j]$, is equal to $f[j] = \lceil q/2 \rfloor \cdot m[j]$, where $m[j]$ is the jth bit of m for $j \in \{0, 1, \cdots, 255\}$, see $\mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ at line 7 of KYBER.CPAPKE.Enc() in Fig. 1.

In the unprotected implementation of Kyber by Kannwischer et al. in [22], the message encoding is realized by the procedure called `poly_frommsg()` shown in Fig. 6. It contains two nested **for**-loops in which each polynomial coefficient is computed individually. The intermediate variable *mask* is used to replace the **if-then-else**-statement in order to guarantee a constant processing time regardless of the message bit value. Otherwise, a timing attack can be mounted to extract the message bit.

However, in a software implementation, the power consumption may differ if the Hamming weights of the two processed values differ. The intermediate variable *mask* is computed based on the message bit, see line 3 in Fig. 6. Its value is either 0 (0x0000) or -1 (0xFFFF). The corresponding polynomial coefficient computed from the *mask* in the next line takes values either 0 or $(q+1)/2$. Since in both cases the difference in the Hamming weights of two values is large, one can

```
void masked_poly_frommsg(masked_poly *r, masked_u8_msgbytes *msg)
int i,j;
uint16_t mask;
for (i = 0; i < 32; i++) do
  for (j = 0; j < 8; j++) do
    mask = -((msg->share[0].u8[i] >> j) & 1);
    r->poly[0].coeffs[8*i+j] += (mask & ((KYBER_Q + 1)/2));
  end for
end for
for (i = 0; i < 32; i++) do
  for (j = 0; j < 8; j++) do
    mask = -((msg->share[1].u8[i] >> j) & 1);
    r->poly[1].coeffs[8*i+j] += (mask & ((KYBER_Q + 1)/2));
  end for
end for
...Further processing ...
```

Fig. 8. The C code of `masked_poly_frommsg()` procedure from [19].

recover the message bits by analysing power consumption [2,29,34]. Such type of leakage is referred to as *determiner* leakage [34], because the *mask*/polynomial values are determined by the corresponding message bit.

Fig. 7 shows the distributions of power consumption during the processing of a single message bit by `poly_frommsg()` procedure. The distributions are plotted based on 10K traces at the trace point with the maximum absolute t-test score. The overlap in the plots of message bits with values 0 and 1 determines the difficulty of distinguishing between these values. We can see that there is almost no overlap. This means that two values can be distinguished easily.

7.2 Masked message encoding

A common way to decorrelate a sensitive variable from the power consumption is to split the variable into multiple shares [7, 14, 26]. For the ω-order masked message encoding, the message m is split into $\omega + 1$ Boolean shares $\{m_0, m_1, \cdots, m_\omega\}$, such that $m = m_0 \oplus m_1 \cdots \oplus m_\omega$. For each Boolean share $m_i, i \in \{0, 1, \cdots, \omega\}$, the corresponding arithmetic share f_i is computed so that the jth coefficient of the polynomial f satisfies:

$$f[j] = \sum_{i=0}^{\omega} f_i[j] \bmod q = \lceil q/2 \rfloor \cdot m[j], \tag{1}$$

for all $j \in \{0, 1, \cdots, 255\}$, where "$\sum$" is the arithmetic addition.

Implementation of masked message encoding in [19] The first-order masked implementation of Heinz et al. in [19] adopts the masking strategy of [26].

The message m is split into two Boolean shares $\{m_0, m_1\}$ and the corresponding arithmetic shares $\{f_0, f_1\}$ are computed separately. If both Boolean shares have value 1, then both resulting polynomial coefficients have value $\lceil q/2 \rfloor$, which does not satisfy eq. (1) due to the rounding. To fix this, an extra term is added[1].

The C code of the procedure `masked_poly_frommsg()` realizing the message encoding in the implementation of [19] is shown in Fig. 8. We can see that two nested **for**-loops of the procedure `poly_frommsg()` in Fig. 6 are repeated twice to compute the arithmetic shares $\{f_0, f_1\}$. Therefore, the leakage of each share is similar to the one of the unprotected version. Several attacks exploiting this leakage have been demonstrated recently [13, 16].

Implementation of masked message encoding in [9] The higher-order masked implementation of Bronchain et al. in [9] employs the masking strategy of [32]. It uses a masked Boolean to arithmetic conversion algorithm to transform the Boolean shares $\{m_0, m_1, \cdots, m_\omega\}$ into the arithmetic shares $\{f_0, f_1, \cdots, f_\omega\}$ such that $\sum_{i=0}^{\omega} f_i[j] \bmod q = m[j]$, for all $j \in \{0, 1, \cdots, 255\}$.

Fig. 9 shows the C code of the procedure `masked_poly_frommsg()` realizing the message encoding in the implementation of [9]. First, each Boolean share bit is extracted from the corresponding byte, see line 4 of `masked_poly_frommsg()` in Fig. 9. This is similar to the lines 3 and 9 of `masked_poly_frommsg()` procedure from [19] in Fig. 8. However, an essential difference is that, instead of computing the variable *mask*, the implementation in [9] only extracts each Boolean share bit and performs a masked Boolean to arithmetic conversion latter on. Hence, the difference in the Hamming weight of values computed in line 4 of Fig. 9 is only one, while the difference in the Hamming weight of *mask* values computed in lines 3 and 9 of Fig. 8 is 16. Consequently, it is more difficult to extract the Boolean share bits from the implementation in [9] by power analysis using the leakage related to the line 4 of Fig. 9.

The plots at the bottom of Fig. 10 show the distributions of power consumption during the processing of a single bit of Boolean shares by `masked_poly_frommsg()` procedure of the implementation in [9]. The distributions are plotted based on 10K traces captured from a profiling device running a first-order masked implementation with known masks at the trace point with the maximum absolute t-test score. Note that these traces are used for leakage analysis only. We do not use them for profiling or in the attack.

The plots at the top of Fig. 10 show similar distributions for the implementation in [19]. One can see the significant difference in the overlapping areas of the plots of Boolean share bits with values 0 and 1. In the implementation of [19] there is almost no overlap, while in the implementation of [9] the overlap is large.

7.3 Finding new leakage points

The C code of masked Boolean to arithmetic conversion procedure `secb2a_1bit()` in Fig. 9 contains two operations that are directly related to the individual bits

[1] We refer to [19, 26] for details since our leakage analysis does not rely on that.

```
void masked_poly_frommsg(StrAPoly y, uint8 m[32 * NSHARES], size_t
stride) /* stride is the byte distance between each share */
int i,j,k;
uint32_t t1[NSHARES]; /* Boolean shares */
int16_t t2[NSHARES]; /* Arithmetic shares */
 1: for (i = 0; i < 32; i++) do
 2:   for (j = 0; j < 8; j++) do
 3:     for (k = 0; k < NSHARES; K++) do
 4:       t1[k] = (m[i+k*stride]>>j) & 1; /* Bit extraction from byte */
 5:     end for
 6:     secb2a_1bit(NSHARES, t2, t1); /* Masked B2A */
 7:     for (k = 0; k < NSHARES; k++) do
 8:       y[k][i*8+j] = (t2[k] * (KYBER_Q+1)/2) % KYBER_Q;
 9:     end for
10:   end for
11: end for
void secb2a_1bit(size_t nshares, int16_t *a, uint32_t *x)
 1: b2a_qbit(nshares, a, x);
 2: refresh_add(nshares, a);
void b2a_qbit(size_t nshares, int16_t *a, uint32_t *x)
int i;
 1: a[0] = x[0]; /* Copy the bit of the first share in x */
 2: for (i = 1; i < nshares; i++) do
 3:   secb2a_qbit_n(i+1, a, a, x[i]); /* Use the bit of share i in x */
 4: end for
void secb2a_qbit_n(size_t n, int16_t *c, int16_t *a, uint32_t x)
int j;
int16_t b[n];
int16_t r[2];
 1: ... Processing ...
 2: for (j = 0; j < n; j++) do
 3:   c[j] = b[j] + 2 * KYBER_Q;
 4:   c[j] -= 2 * b[j] * x; /* Compute polynomial value from bit value */
 5:   c[j] = c[j] % KYBER_Q;
 6: end for
 7: c[0] = (c[0] + x) % KYBER_Q;
```

Fig. 9. The C code of `masked_poly_frommsg()` procedure from [9].

of each share. One is located in `b2a_qbit()` procedure, see lines marked in red. In line 1, the bit value of the first Boolean share `x[0]` is copied to the first arithmetic share `a[0]`. In line 3, all bit values of the other shares `x[i]`, $i \in \{1, \ldots, nshares\}$, are given as input to `secb2a_qbit_n()`, one by one, where $nshares$ is the number of shares.

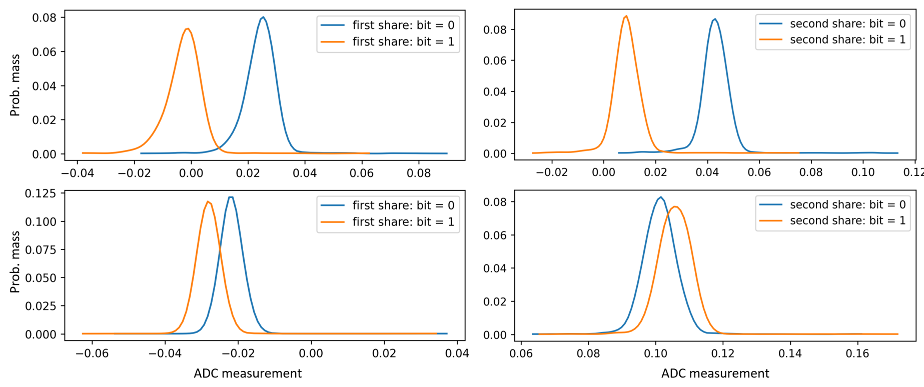

Fig. 10. Distributions of power consumption during the processing of a single bit of Boolean shares by `masked_poly_frommsg()` in the first-order masked implementations of [19] (top) and [9] (bottom).

```
 1: LDRSH r3, [r5, 2]! /* load b[j] to r3 */
 2: ADD.w r2, ip, r3 /* b[j] + 2 * KYBER_Q */
 3: SMULBB r3, r3, r6 /* 2*b[j]*x */
 4: SUBS r3, r2, r3 /* c[j] = b[j] + 2 * KYBER_Q - 2*b[j]*x */
 5: /* calculate c[j] % KYBER_Q and store the value to r3 */
 6: SXTH r3, r3
 7: SMULL fp, r2, sl, r3
 8: ADD.W fp, r2, r3
 9: ASRS r2, r3, 0x1f
10: RSB r2, r2, fp, asr 11
11: ADDS r0, 1
12: MLS r3, lr, r2, r3
13: CMP r0, sb
14: STRH r3, [r4]
```

Fig. 11. Assembly code snippet of a single iteration of additive leakage vulnerability in `secb2a_qbit_n()`.

Another operation related to individual bits of each share is located in `secb2a_qbit_n()` procedure. The bit value of the Boolean share `x[i]` is processed $i+1$ times in the **for**-loop, see lines 2-6 of `secb2a_qbit_n()` in Fig. 9. In line 4, the value of `2*b[j]*x` is computed and subtracted from the intermediate value `c[j]`. Therefore, the Hamming weight of `c[j]` does not change if $\mathtt{x} = 0$. Otherwise, for $\mathtt{x} = 1$, it is likely to change[2]. The corresponding assembly code is shown in Fig. 11.

[2] The Hamming weight of an arithmetic share may remain the same if a non-zero value is subtracted.

To see if the two above-mentioned operations leak side-channel information, we performed t-test of `masked_poly_frommsg()` procedure for the first-, second- and third-order masked implementations with known masks captured from a profiling device. Fig. 12 shows t-test results for a single bit of each Boolean share on 10K traces. We can see multiple leakage points which can be grouped into three types. Peaks on the left-hand side of the black vertical line are related to the extraction of a single Boolean share bit from a byte (line 4 of `masked_poly_frommsg()` in Fig. 9). Such a leakage is also present in the unprotected implementation of [22] (line 3 in Fig. 6) and the masked implementation of [19] (line 3 and 9 in Fig. 8).

Peaks on the right-hand side of the black vertical line, marked by "①" and "②", are related to the processing of individual Boolean share bits by the two above-mentioned operations (lines 1 and 3 of `b2a_qbit()` and line 4 of `secb2a_qbit_n()`). We call them the *direct-copy* leakage and the *additive* leakage, respectively. They are specific for the implementation of arbitrary-order masked Boolean to arithmetic conversion introduced in [9]. To the best of our knowledge, until now nobody has reported that these leakages are exploitable.

For direct-copy leakage, since the first Boolean share `x[0]` is assigned to the arithmetic share `a[0]` directly (line 1 of `b2a_qbit()`), the distance between the peaks corresponding to `x[0]` and `x[1]` is smaller than the distance between the peaks corresponding to `x[i]` and `x[i+1]`, for any $i > 0$.

For additive leakage, the number of peaks is equal to the number of times the share is processed in the **for**-loop. For example, at the bottom plot of Fig. 12 representing the third-order masked implementation, there are two orange peaks (second share), three green peaks (third share), four red peaks (forth share) marked by '②". There is no peak for the first share since `secb2a_qbit_n()`) is not called to process it. The fact that a change of the Hamming weight of a Boolean share does not always lead to the change of the Hamming weight of the arithmetic share may explain why in Fig. 12 additive leakage is weaker than direct-copy leakage.

Next we compare distributions of power consumption of the direct-copy and additive leakages on the example of the first-order masked implementation, see Fig. 13. The distributions are plotted based on 10K traces at the trace point with the maximum absolute t-test score. These traces are acquired from a profiling device running the implementation with known masks during the execution of masked Boolean to arithmetic conversion. We can see that, for direct-copy leakage, the overlap in the plots is larger than the one for additive leakage. This is consistent with t-test results in Fig. 12. Since the overlap is not complete, both types can be exploited for message recovery.

In the message recovery attack presented in Section 10.1, we use both direct-copy and additive leakages as well as the leakage in the bit extraction part.

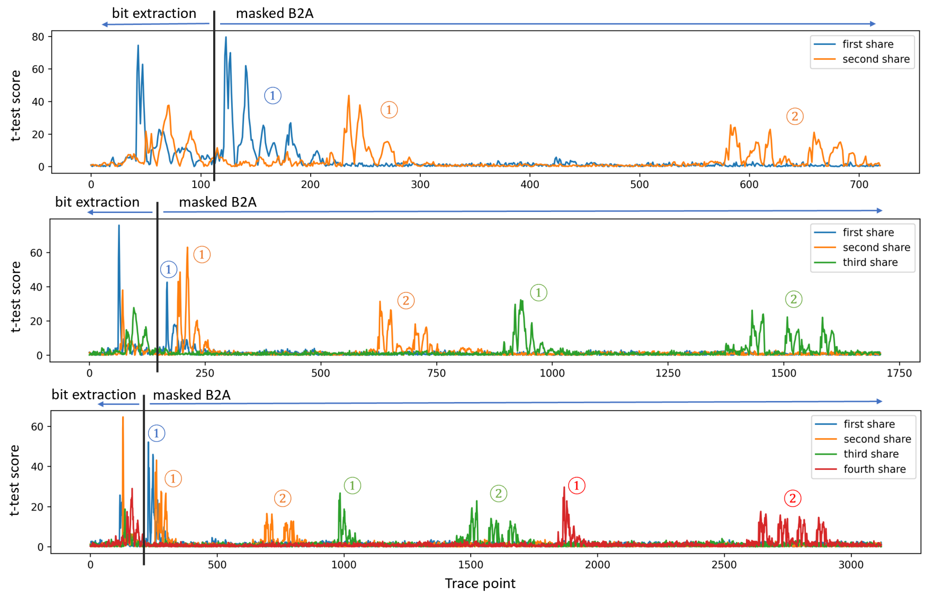

Fig. 12. T-test results for a single bit of each Boolean share of `masked_poly_frommsg()` procedure of the first- (top), second- (middle), third-order (bottom) masked implementations of [9]. Traces are acquired from a profiling device running implementations with known masks (used for leakage analysis only).

8 Neural network training

This section describes our neural network training strategy. It is a combination of techniques employed in previous profiling deep learning-based side-channel attacks on PQC and symmetric cryptographic algorithms, with some differences which we highlight.

Following [13], we train a single universal neural network model for message bit prediction on cut-and-joined and standardized traces. A multilayer perceptron (MLP) with an archtecture similar to the one in [13] is used. A difference from [13] is that we use traces from five profiling devices in the training set. Such an approach is used in the side-channel attack on AES presented in [40]. Another difference from [13] is that we use three leakage points, so cut-and-join is a bit more tedious to perform.

8.1 Trace acquisition and pre-processing

Since `masked_poly_frommsg()` procedure processes the Boolean share bits one-by-one, it is possible to train a universal model for predicting all bits except the

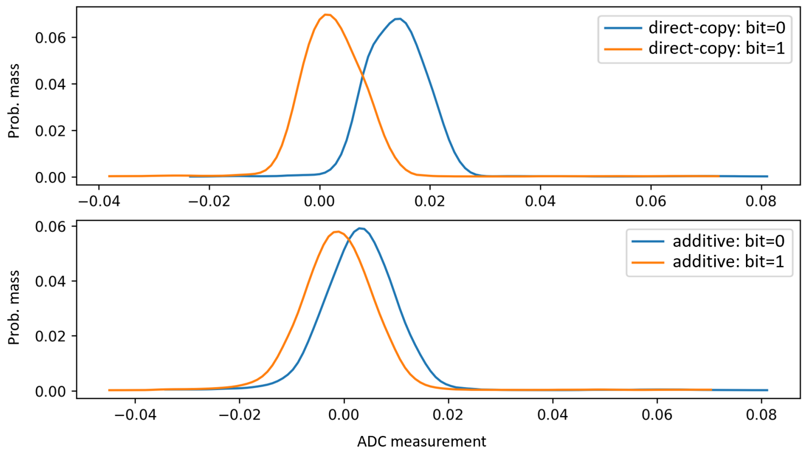

Fig. 13. Distributions of power consumption during the processing of a single bit of Boolean shares by `secb2a_1bit()` in the first-order masked implementation of [9] for the direct-copy (top) and additive (bottom) leakage.

first and the last. The trace shape for the first/last bits typically differ from the rest because their previous/next instructions differ [25].

The complete execution of `masked_poly_frommsg()` procedure in the implementation of [9] does not fit into the buffer of ChipWhisperer-Pro which we use for trace acquisition. Therefore, for the first- and second-order masked implementation we capture traces containing the execution of the first 33 bits only and use a union of intervals corresponding to the bits 1-32 for training. For the third-order masked implementation, we capture traces containing the execution of the first 17 bits only and use a union of intervals corresponding to the bits 1-16 for training. In all cases, the interval is a concatenation of three segments covering the three leakage points described in Section 7.3.

Since the implementation in [9] uses a true random number generator (TRNG) with a range check for generating masks, the raw traces are misaligned. We synchronize the traces by cross-correlating with templates, one for each leakage point.

To minimize the total number of traces required from the DUA, we perform profiling on different devices. We use five profiling devices, $D_1 - D_5$, in order to reduce the negative effect of inter-device variation on neural network's classification accuracy. The benefits of such a multi-source profiling are well-known [37].

We also apply standardization to traces. Given a set of traces $\boldsymbol{T}$ with elements $T = (t_1, \ldots, t_{|T|})$, each $T \in \boldsymbol{T}$ is standardized to $T' = (t'_1, \ldots, t'_{|T|})$ such as:

$$t'_i = \frac{t_i - \mu_i}{\sigma_i},$$

Table 2. MLP architecture used for message recovery. The input size is $size = 590, 1000$ and 1610 for the first-, second- and third-order masked implementations, respectively.

Layer type	Output shape
Batch Normalization 1	*size*
Dense 1	512
Batch Normalization 2	512
ReLU	512
Dense 2	256
Batch Normalization 3	256
ReLU	256
Dense 3	128
Batch Normalization 4	128
ReLU	128
Dense 4	2
Softmax	2

where and μ_i and σ_i are the mean and the standard deviation of the elements of $\boldsymbol{T}$ at the ith data point, $i \in \{1, \ldots, |T|\}$.

8.2 Network architecture and training parameters

The neural networks with the architecture listed in Table 2 are trained with a batch size of 1024 for a maximum of 100 epochs using early stopping with patience 10. We use Nadam optimizer with a learning rate of 0.01 and a numerical stability constant *epsilon* = 1e-08. Categorical cross-entropy is used as a loss function to evaluate the network classification error. 70% of the training set is used for training and 30% is left for validation. Only the model with the highest validation accuracy is saved.

9 New chosen ciphertext construction method

It is known that the secret key of an LWE/LWR KEM algorithm can be derived from messages recovered from chosen ciphertexts. Many different methods for constructing the chosen ciphertexts have been presented in the past, including [5, 25, 29, 30, 42]. These methods uniquely map each secret key coefficient into a b-bit binary vector composed from the message bits recovered from b chosen ciphertexts. Some methods, e.g. [5, 25], impose an additional requirement that b-bit binary vectors are codewords of some linear code with the code distance C_d. In the latter case, for each secret key coefficient, c errors in the recovered message bits can be corrected and d additional errors can be detected, where $2c + d + 1 \leq C_d$. The method [5] also minimizes the Hamming weight of the chosen ciphertexts.

However, none of the previous chosen ciphertext construction methods map the secret key coefficients into codewords so that the full secret key recovery probability is maximized for a given message bit recovery probability. We introduce such a method in this section.

First we explain the method of composing chosen ciphertexts and then derive a formula relating the probability of the secret key recovery to the probability of a message bit recovery. Using this formula, we select a best codeword for each key coefficient which maximizes the secret key recovery probability. Finally, we show that there is a considerable difference between secret key recovery probabilities of the best and the worst mappings.

9.1 Composing chosen ciphertexts

In Kyber-768, the secret key $\boldsymbol{s}$ consists of three polynomials $\boldsymbol{s} = (s_0, s_1, s_2)$, and the ciphertext $(\boldsymbol{u}, v)$ consists of three polynomials $\boldsymbol{u} = (u_0, u_1, u_2)$ and one polynomial v. To recover n coefficients of s_i, one of the polynomials of $\boldsymbol{u}$ is set to a non-zero constant k_1 and the other two polynomials of $\boldsymbol{u}$ are set to zero:

$$\boldsymbol{u} = \begin{cases} (k_1, 0, 0) \in R_q^{3\times 1} & \text{for } i = 0, \\ (0, k_1, 0) \in R_q^{3\times 1} & \text{for } i = 1, \\ (0, 0, k_1) \in R_q^{3\times 1} & \text{for } i = 2. \end{cases}$$

All n coefficients of v are set to the same constant k_0:

$$v = k_0 \sum_{j=0}^{255} x^j \in R_q^{1\times 1}.$$

The constants (k_1, k_0) inducing a given mapping between the secret key coefficients and message bits can be found by a brute-force search through all legal pairs (k_1, k_0) (if the solution exists). In the next section, we derive a formula which helps select the best mapping that maximizes the secret key recovery probability.

9.2 Selecting optimal mapping

Let p be the probability of recovering a single message bit, and $d(x, x')$ be the Hamming distance between the codewords representing the secret key coefficients x and x', for $x, x' \in \{-2, -1, 0, 1, 2\}$.

Given a codeword composed from b recovered message bits, there are three possible outcomes:

1. The codeword is recovered correctly (*no errors*). We denote this probability by p_c.
2. The recovered codeword contains errors and matches the codeword representing another secret key coefficient (*undetected error*). We denote this probability by p_u.
3. The recovered codeword contains errors but does not match any codewords of another secret key coefficients (*detected error*). We denote this probability by p_d.

Table 3. Mappings which give the maximum (left) and the minimum p_{sk} (right).

$\mathcal{C}_{best}$: (k_1, k_0)	-2	-1	0	1	2
(1977,208)	1	1	0	1	0
(627,208)	1	1	0	0	1
(731,1040)	0	1	1	0	0

$\mathcal{C}_{worst}$: (k_1, k_0)	-2	-1	0	1	2
(104,1040)	1	1	1	1	0
(419,416)	1	1	0	0	0
(940,1040)	0	1	1	0	1

The secret key coefficients are generated using the centered binomial distribution, see line 3 of KYBER.CPAPKE.KeyGen() in Fig. 1. For Kyber-768, the probability of occurrence of $x \in \{-2, -1, 0, 1, 2\}$, p_x, is given by:

$$p_x = \begin{cases} 1/16, & \text{for } x = -2 \\ 4/16, & \text{for } x = -1 \\ 6/16, & \text{for } x = 0 \\ 4/16, & \text{for } x = 1 \\ 1/16, & \text{for } x = 2 \end{cases} \tag{2}$$

Thus, the expected probabilities of three outcomes listed above are given by:

$$p_c = \sum_x p_x \cdot p^b = p^b \tag{3}$$

$$p_u = \sum_x p_x \cdot \sum_{x', x' \neq x} (1-p)^{d(x,x')} p^{b-d(x,x')} \tag{4}$$

$$p_d = 1 - p_c - p_u \tag{5}$$

Let e be the maximum tolerable number of detected errors. Then, for Kyber-768, the probability of full secret key recovery is:

$$p_{sk} = \sum_{i=0}^{e} \binom{768}{i} p_d^i \cdot p_c^{768-i} \tag{6}$$

By a brute-force search through all possible codewords of the length $b = 3$ for each secret key coefficient, one can find a mapping $\mathcal{C}_{best}$ which results in the highest p_{sk} and a mapping $\mathcal{C}_{worst}$ which gives the lowest p_{sk}. Examples of such mapping are listed in Table 3. They are not unique. One can see that, for the optimal mapping $\mathcal{C}_{best}$, $d(0, x) \geq 2$ for all for $x \in \{-2, -1, 1, 2\}$, while for $\mathcal{C}_{worst}$, $d(0, x') = 1$, for $x' \in \{-1, 1, 2\}$. Thus, single-bit errors are more likely to be undetected in $\mathcal{C}_{worst}$ case.

Fig. 14 plots p_{sk} as a function p for the mappings $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ and a fixed $e = 16$. For $0 < p < 1$, $\mathcal{C}_{best}$ always results in a higher p_{sk}. The difference between the two mappings first grows larger as p increases, reaching the maximium of 39% at $p = 0.9991$. Then, the difference starts decreasing and the plots converge at $p = 1$.

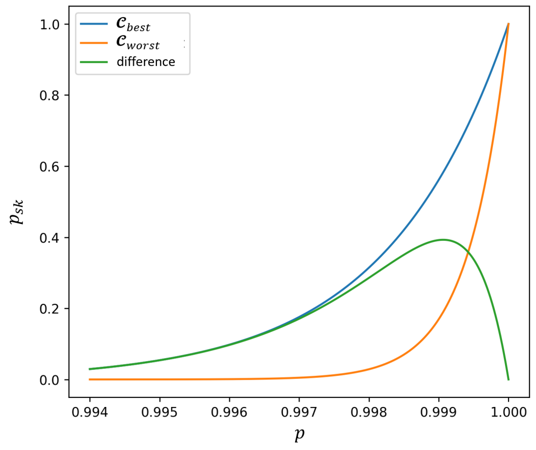

Fig. 14. Secret key recovery probability as a function message bit recovery probability for the best and worst mappings.

10 Experimental results

In this section, we present the results of message and secret key recovery attacks on the higher-order masked implementation of Kyber-768 [9] in ARM Cortex-M4.

10.1 Message recovery attack

Using the strategy described in Section 8, we trained neural network models for message bit recovery. For each ω-order masked implementation in the experiments, a single universal model was trained for recovering all bits. The models were trained using power traces captured from five profiling devices, D_i, $i \in \{1, 2, \ldots, 5\}$. For each implementation, one fifth of training traces were captured from each profiling device. As we mentioned in Section 8.1, the complete execution of `masked_poly_frommsg()` procedure in the implementation of [9] does not fit into the buffer of ChipWhisperer-Pro. For $\omega = 1$ and 2, we captured 2K traces from each D_i and trained the models based on the message bits 1-32. For $\omega = 3$, 4K traces were captured from each D_i and the models were trained based on the message bits 1-16. In all three cases, after the cut-and-join, the total number of training traces is 320K.

The models were tested on power traces captured from the DUA D_6 for 1000 ciphertexts encrypting messages selected at random. For each ciphertext, we repeated the decapsulation twenty times and recorded the corresponding power traces. We use N to denote the number of repetitions of the same decapsulation.

Table 4 summarizes the results of message recovery attacks on the first-order masked implementation of [9]. It lists the empirical average message bit recovery

Table 4. Empirical results of message recovery attack on the first-order masked implementation.

# Repetitions	$N=1$	$N=2$	$N=3$
Avg. message bit recovery prob., p_{bit}	0.99928	0.99997	1
Est. full message recovery prob., p_m	0.83	0.99	1

Table 5. Empirical results of message recovery attack on the second-order masked implementation.

# Repetitions	$N=1$	$N=2$	$N=4$	$N=8$	$N=16$
Avg. message bit recovery prob., p_{bit}	0.97203	0.99519	0.99953	0.99959	0.99994
Est. full message recovery prob., p_m	0	0.29	0.89	0.90	0.98

Table 6. Empirical results of message recovery attack on the third-order masked implementation.

# Repetitions	$N=1$	$N=2$	$N=4$	$N=8$	$N=16$
Avg. message bit recovery prob., p_{bit}	0.95356	0.98219	0.99356	0.99819	0.99994
Est. full message recovery prob., p_m	0	0.01	0.19	0.63	0.98

probability, p_{bit}, and the full message recovery probability, p_m, estimated as $p_m = (p_{bit})^{256}$, for different number of repetitions N.

We can see that the estimated full message recovery probability is 0.83 for a single-trace attack. When the number of repetitions increases to 3, p_m reaches 1. As observed in [13], side-channel attacks of masked implementations benefit from the independence of errors in repeated measurements due to random mask update at each execution.

Table 5 and 6 shows the results of message recovery attacks on the second- and third-order masked implementations of [9]. We increase the number of repetitions exponentially since higher-order masking is more difficult than first-order to break. One can see that, for both $\omega = 2$ and $\omega = 3$, the empirical full message recovery probability p_m reaches 0.98 for sixteen repetitions.

Note that the ChipWhisperer target board used in our experiments has a low noise. For the attacks in noisier conditions, convolutions neural networks [23,27], or transformers [8,17] may be more suitable neural network architectures. Noise reduction e.g. by using autoencoders [23,41] may also be helpful.

Table 7. Empirical results of secret key recovery attack on the first-order masked implementation.

# Repetition	$\mathcal{C}_{best}$ mapping			$\mathcal{C}_{worst}$ mapping		
	p_{bit}	p_{sk}	Enum.	p_{bit}	p_{sk}	Enum.
$N = 1$	0.99761	0.26	5^{16}	0.99746	0	$>5^{16}$
$N = 2$	0.99987	0.94	5^{4}	0.99989	0.84	5^{1}
$N = 3$	0.99995	0.98	5^{4}	0.99997	0.97	5^{1}

Table 8. Empirical results of secret key recovery attack on the second-order masked implementation.

# Repetition	$\mathcal{C}_{best}$ mapping			$\mathcal{C}_{worst}$ mapping		
	p_{bit}	p_{sk}	Enum.	p_{bit}	p_{sk}	Enum.
$N = 1$	0.97466	0	$>5^{16}$	0.97365	0	$>5^{16}$
$N = 2$	0.99600	0.10	5^{11}	0.99570	0	5^{3}
$N = 4$	0.99970	0.79	5^{2}	0.99960	0.60	5^{1}
$N = 8$	0.99975	0.81	5^{5}	0.99971	0.72	5^{1}
$N = 16$	0.99997	0.97	5^{1}	0.99996	0.92	5^{1}

Table 9. Empirical results of secret key recovery attack on the third-order masked implementation.

# Repetition	$\mathcal{C}_{best}$ mapping			$\mathcal{C}_{worst}$ mapping		
	p_{bit}	p_{sk}	Enum.	p_{bit}	p_{sk}	Enum.
$N = 1$	0.94794	0	$>5^{16}$	0.95095	0	$>5^{16}$
$N = 2$	0.98108	0	$>5^{16}$	0.98215	0	$>5^{16}$
$N = 4$	0.99290	0	$>5^{16}$	0.99297	0	$>5^{16}$
$N = 8$	0.99793	0.36	5^{7}	0.99781	0.02	5^{1}
$N = 16$	0.99977	0.82	5^{2}	0.99979	0.74	5^{1}

10.2 Secret key recovery attack

In order to evaluate the effectiveness of the new method of mapping message bits into secret key coefficients, we generated 100 different secret and public key pairs for $\omega = 1, 2$ and 50 different key pairs for $\omega = 3$ using KYBER.CCAKEM.KeyGen(), respectively. According to Table 4, 5 and 6, three and 16 repetitions should give a close to 1 message bit recovery probability for $\omega = 1$ and $\omega = 2, 3$. Then, we captured from the DUA D_6 the coresponding number of traces for each key pair during the decapsulation of chosen ciphertexts constructed for the mappings $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ in Table 3.

Tables 7 summarizes the results for the mappings $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ using the first-order masked implementation of [9]. We can see that the difference in the average message bit recovery probabilities of $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ for a fixed N

is insignificant. However, the difference in the average key coefficient recovery probabilities of $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ is significant for $N = 1$, resulting in a large difference in full key recovery probabilities. For $N = 1$, empirical full secret key recovery probability for the mapping $\mathcal{C}_{best}$ is $p_{pk} = 0.26$ with the maximum enumeration of 5^{16}. In contract, none of the 100 secret keys can be recovered if the mapping $\mathcal{C}_{worst}$ is used. As the number of repetitions N increases, the difference between $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ decreases.

Table 8 shows the results of secret recovery attack on the second-order masked implementation of [9]. We can see that, for $\mathcal{C}_{best}$, 16 repetitions instead of three, are required to reach the full secret key recovery probability $p_{sk} = 0.97$. We can also see that, for $N = 4$, the difference in p_{sk} between $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ is maximum, at 19%.

Similarly, Table 9 shows the results of secret recovery attack on the third-order masked implementation of [9]. The probability of the full secret key recovery is $p_{sk} = 0.82$ using $\mathcal{C}_{best}$ when $N = 16$. The gap in p_{sk} between $\mathcal{C}_{best}$ and $\mathcal{C}_{worst}$ is maximum, at 34%, for $N = 8$.

11 Countermeasures

The presented secret key recovery attack would not be possible if the decapsulating device could refuse decrypting the chosen ciphertexts. This can be realized by authenticating the ciphertexts e.g. using the Encrypt-then-Sign method proposed by [4], or subjecting the ciphertexts to the minimal range check [42].

Another possibility is to update the keys (pk, sk) for each new shared key establishment session rather than keeping them static. In this scenario, the shared key becomes the primary attack target. Note, however, that the dynamic keys (pk, sk) make the likelihood of recovering a shared key less likely, but not impossible. For instance, for the first-order masking, the attacker is expected to recover the shared key from a single trace with the probability of 26% (see Table 7). Since the success probability grows quickly if the decapsulation can be repeated multiple times, designing a mechanism which prevents repeated decapsulations of the same ciphertext could be considered as an option.

The presented message recovery attack would be more difficult if the the procedure `masked_poly_frommsg()` were bitsliced. Using a TRNG with a range check for generating masks is an excellent design choice because the resulting traces' misalignment creates an extra hurdle for the attacker.

12 Conclusion

We demonstrated practical shared and secret key recovery attacks on the higher-order masked implementation of Kyber from Bronchain et al. [9] by profiled deep learning-based power analysis.

We discovered new vulnerabilities in the implementation of arbitrary-order masked Boolean to arithmetic conversion introduced in [9]. Note that such an implementation is applicable not only to Kyber, but also to any algorithm using masked Boolean to arithmetic conversion. Our work shows that, to resist power analysis, the implementation needs to be further strengthened.

Another contribution is a chosen ciphertext construction method that maximizes the likelihood of recovering the secret key for a given message bit recovery probability. The new way of mapping message bits into the secret key coefficients can raise the likelihood of successfully recovering the secret key by up to 39% compared to the worst case.

13 Acknowledgments

This work was supported in part by the Swedish Civil Contingencies Agency (Grant No. 2020-11632) and the Swedish Research Council (Grant No. 2018-04482).

References

1. Announcing the commercial national security algorithm suite 2.0. National Security Agency, U.S Department of Defense (Sep 2022), `https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF`
2. Amiet, D., Curiger, A., Leuenberger, L., Zbinden, P.: Defeating NewHope with a single trace. In: Post-Quantum Cryptography: 11th International Conference, PQCrypto 2020, Paris, France, April 15–17, 2020, Proceedings 11. pp. 189–205. Springer (2020)
3. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber algorithm specifications and supporting documentation (2021), `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf`
4. Azouaoui, M., Kuzovkova, Y., Schneider, T., van Vredendaal, C.: Post-quantum authenticated encryption against chosen-ciphertext side-channel attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 372–396 (2022)
5. Backlund, L., Ngo, K., Gartner, J., Dubrova, E.: Secret key recovery attacks on masked and shuffled implementations of CRYSTALS-Kyber and Saber. Cryptology ePrint Archive, Paper 2022/1692 (2022), `https://eprint.iacr.org/2022/1692`
6. Bhasin, S., D'Anvers, J.P., Heinz, D., Pöppelmann, T., Beirendonck, M.V.: Attacking and defending masked polynomial comparison for lattice-based cryptography. Cryptology ePrint Archive, Paper 2021/104 (2021), `https://eprint.iacr.org/2021/104`
7. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., Van Vredendaal, C.: Masking Kyber: First-and higher-order implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 173–214 (2021)
8. Brisfors, M.: Advanced Side-Channel Analysis of USIMs, Bluetooth SoCs and MCUs. Master's thesis, School of EECS, KTH (2021)
9. Bronchain, O., Cassiers, G.: Bitslicing arithmetic/Boolean masking conversions for fun and profit: with application to lattice-based KEMs. IACR Trans. on Cryptographic Hardware and Embedded Systems pp. 553–588 (2022)

10. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Advances in Cryptology - CRYPTO '99. vol. 1666, pp. 398–412. Springer (1999)
11. D'Anvers, J.P., Beirendonck, M.V., Verbauwhede, I.: Revisiting higher-order masked comparison for lattice-based cryptography: Algorithms and bit-sliced implementations. Cryptology ePrint Archive, Paper 2022/110 (2022), `https://eprint.iacr.org/2022/110`
12. Do, Q., Martini, B., Choo, K.K.R.: The role of the adversary model in applied security research. Computers & Security **81**, 156–181 (2019)
13. Dubrova, E., Ngo, K., Gartner, J.: Breaking a fifth-order masked implementation of CRYSTALS-Kyber by copy-paste. In: Proc. of the 10th ACM Asia Public-Key Cryptography Workshop (APKC 2023) (2023), `https://eprint.iacr.org/2022/`
14. D'Anvers, J.P., Heinz, D., Pessl, P., Van Beirendonck, M., Verbauwhede, I.: Higher-order masked ciphertext comparison for lattice-based cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 115–139 (2022)
15. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Annual international cryptology conference. pp. 537–554. Springer (1999)
16. 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. Cryptology ePrint Archive (2023)
17. Hajra, S., Saha, S., Alam, M., Mukhopadhyay, D.: Transnet: Shift invariant transformer network for side channel analysis. Cryptology ePrint Archive, Paper 2021/827 (2021), `https://eprint.iacr.org/2021/827`
18. Hamburg, M., Hermelink, J., Primas, R., Samardjiska, S., Schamberger, T., Streit, S., Strieder, E., van Vredendaal, C.: Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 88–113 (2021)
19. Heinz, D., Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels, D.: First-order masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2022/058 (2022), `https://eprint.iacr.org/2022/058`
20. Hoffmann, C., Libert, B., Momin, C., Peters, T., Standaert, F.X.: Towards leakage-resistant post-quantum CCA-secure public key encryption. Cryptology ePrint Archive, Paper 2022/873 (2022), `https://eprint.iacr.org/2022/873`
21. Ji, Y., Wang, R., Ngo, K., Dubrova, E., Backlund, L.: A side-channel attack on a hardware implementation of CRYSTALS-Kyber. Cryptology ePrint Archive, Paper 2022/1452 (2022), `https://eprint.iacr.org/2022/1452`
22. Kannwischer, M.J., Petri, R., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, `https://github.com/mupq/pqm4`
23. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking cryptographic implementations using deep learning techniques. In: Carlet, C., Hasan, M.A., Saraswat, V. (eds.) Security, Privacy, and Applied Cryptography Engineering. pp. 3–26. Springer International Publishing, Cham (2016)
24. Moody, D.: Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Nistir 8309 pp. 1–27 (2022), `https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf`
25. Ngo, K., Dubrova, E., Guo, Q., Johansson, T.: A side-channel attack on a masked IND-CCA secure Saber KEM implementation. IACR Trans. on Cryptographic Hardware and Embedded Systems pp. 676–707 (2021)

26. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure and masked ring-LWE implementation. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 142–174 (2018)
27. Picek, S., Samiotis, I.P., Kim, J., Heuser, A., Bhasin, S., Legay, A.: On the performance of convolutional neural networks for side-channel analysis. In: Chattopadhyay, A., Rebeiro, C., Yarom, Y. (eds.) Security, Privacy, and Applied Cryptography Engineering. pp. 157–176. Springer International Publishing, Cham (2018)
28. Rajendran, G., Ravi, P., D'Anvers, J.P., Bhasin, S., Chattopadhyay, A.: Pushing the limits of generic side-channel attacks on LWE-based KEMs-parallel PC oracle attacks on Kyber KEM and beyond. IACR Trans. on Crypto. Hardware and Embedded Systems pp. 418–446 (2023)
29. 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 (2021)
30. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on cca-secure lattice-based PKE and KEMs. IACR Trans. on Cryptographic Hardware and Embedded Systems. pp. 307–335 (2020)
31. Rodriguez, R.C., Bruguier, F., Valea, E., Benoit, P.: Correlation electromagnetic analysis on an FPGA implementation of CRYSTALS-Kyber. Cryptology ePrint Archive, Paper 2022/1361 (2022), `https://eprint.iacr.org/2022/1361`
32. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Public-Key Cryptography–PKC 2019: 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II 22. pp. 534–564. Springer (2019)
33. 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 pp. 89–112 (2023)
34. Sim, B.Y., Kwon, J., Lee, J., Kim, I.J., Lee, T.H., Han, J., Yoon, H., Cho, J., Han, D.G.: Single-trace attacks on message encoding in lattice-based KEMs. IEEE Access **8**, 183175–183191 (2020)
35. Tsai, T.T., Huang, S.S., Tseng, Y.M., Chuang, Y.H., Hung, Y.H.: Leakage-resilient certificate-based authenticated key exchange protocol. IEEE Open Journal of the Computer Society **3**, 137–148 (2022)
36. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. IACR Trans. on Crypt. Hardware and Embedded Systems pp. 296–322 (2022)
37. Wang, H., Forsmark, S., Brisfors, M., Dubrova, E.: Multi-source training deep learning side-channel attacks. In: IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL'2020) (2020)
38. Wang, J., Cao, W., Chen, H., Li, H.: Practical side-channel attack on message encoding in masked Kyber. In: 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 882–889. IEEE (2022)
39. Wang, R., Ngo, K., Dubrova, E.: A message recovery attack on LWE/LWR-based PKE/KEMs using amplitude-modulated EM emanations. In: International Conf. on Information Security and Cryptology. pp. 450–471. Springer (2022)
40. Wang, R., Wang, H., Dubrova, E.: Far field EM side-channel attack on AES using deep learning. In: Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security. pp. 35–44 (2020)

41. Wu, L., Picek, S.: Remove some noise: On pre-processing of side-channel measurements with autoencoders. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 389–415 (2020)
42. Xu, Z., Pemberton, O.M., Roy, S.S., Oswald, D., Yao, W., Zheng, Z.: Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of Kyber. IEEE Transactions on Computers (2021)
43. Yajing, C., Yan, Y., Zhu, C., Guo, P.: Template attack of LWE/LWR-based schemes with cyclic message rotation. Entropy **24**(10) (2022)