Breaking a Fifth-Order Masked Implementation of CRYSTALS-Kyber by Copy-Paste

Elena Dubrova, Kalle Ngo, and Joel Gärtner

KTH Royal Institute of Technology, Stockholm, Sweden
{dubrova,kngo,jgartner}@kth.se

Abstract. CRYSTALS-Kyber has been selected by the NIST as a public-key encryption and key encapsulation mechanism to be standardized. It is also included in the NSA's suite of cryptographic algorithms recommended for national security systems. This makes it important to evaluate the resistance of CRYSTALS-Kyber's implementations to side-channel attacks. The unprotected and first-order masked software implementations have been already analysed. In this paper, we present deep learning-based message recovery attacks on the ω-order masked implementations of CRYSTALS-Kyber in ARM Cortex-M4 CPU for $\omega \leq 5$. The main contribution is a new neural network training method called *recursive learning*. In the attack on an ω-order masked implementation, we start training from an artificially constructed neural network M^{ω} whose weights are partly copied from a model $M^{\omega-1}$ trained on the $(\omega-1)$-order masked implementation, and then extended to one more share. Such a method allows us to train neural networks that can recover a message bit with the probability above 99% from high-order masked implementations.

1 Introduction

Public-key cryptographic schemes used today depend on the intractability of certain mathematical problems such as integer factorization, or the discrete logarithm. However, if large-scale quantum computers become a reality, it will be possible to solve these problems in polynomial time using Shor's algorithm [33]. Therefore current public-key cryptographic schemes will no longer be secure.

To address this issue, the National Institute of Standards and Technology (NIST) launched a process for standardization of post-quantum cryptographic primitives, NIST PQC, in 2016. Candidate primitives rely on problems that are believed to be difficult for quantum computers, such as lattices and decoding in a linear error correcting code. The PQC process is currently in its fourth round. One of the finalists, IND-CCA2 secure public key encryption (PKE) and key encapsulation mechanism (KEM) CRYSTALS-Kyber [3], is already selected to be standardized [23]. It is also included in the National Security Agency (NSA) suite of cryptographic algorithms recommended for national security systems [1].

This makes it important to assess the resistance of CRYSTALS-Kyber implementations to side-channel attacks. Side-channel attacks exploit information

obtained from physically measurable, non-primary channels such as timing or power consumption of a device running the implementation. The first breakthrough in the area was differential side-channel analysis pioneered by Kocher et al. [20]. The second breakthrough was the introduction of deep learning-based side-channel analysis. Apart from improving the differential attacks' effectiveness (e.g. four instead of 400 power traces are required to extract the secret key from a commercial USIM card [10]), the latter enabled attacks of true random number generators [26], physical unclonable functions [42], and non-differential message and secret key recovery attacks on NIST PQC candidates. Deep learning-based side-channel attacks can overcome conventional countermeasures such as masking [27], shuffling [28], random delays insertion [13], constant-weight encoding [22], code polymorphism [7], and randomized clock [11]. The third important contribution is the error injection method of Wang et al. [38] which converts a non-differential attack into a differential. It allows for breaking difficult targets such as a hardware implementation of CRYSTALS-Kyber [19].

Our contributions: The unprotected and first-order masked software implementations of CRYSTALS-Kyber have been already analysed [31,31,34,8,24,40] [36,37,38,41]. That contributed to strengthening the resistance of its subsequently released versions [9,18] and promoted stronger mitigation techniques against side-channel attacks, e.g. [4,35,17]. Side-channel attacks on the second- and third-order masked software implementations of Saber, which is similar to CRYSTALS-Kyber in many aspects, have also been presented [29,30]. In this paper, we demonstrate side-channel attacks on up to the fifth-order masked implementations of CRYSTALS-Kyber in ARM Cortex-M4 CPU.

The first contribution of the paper is a new neural network training method which we call *recursive learning*. In the attack on an ω-order masked implementation, we start training from an artificially constructed neural network M^{ω} whose weights are partly copied from a neural network $M^{\omega-1}$ trained on the $(\omega-1)$-order masked implementation, and then extended to include one more share. Such a method allows us to effortlessly train neural networks that can recover a message bit with the probability above 99%. For higher masking orders, the likelihood of finding such a model without recursive learning is very small.

Another novel contribution is a message recovery method using *cyclic rotations*. In the procedure that is our attack point, the first bit of each message byte leak considerably stronger than the last one. We negacyclically rotate the message to shift its bits from "less leaky" positions to "more leaky" ones. This allow us to increase the success rate of message recovery. The messages are rotated by manipulating the corresponding ciphertexts.

In CRYSTALS-Kyber, a successful message recovery implies the shared session key recovery, as the session key is derived from the message using hash functions. Furthermore, by recovering messages contained in a set of chosen ciphertexts constructed used known methods, e.g. [5,32], one can extract the long-term secret key of CRYSTALS-Kyber.

The rest of the paper is organized as follows. Section 2 describes previous work. Section 3 gives a background on CRYSTALS-Kyber and masking. Sec-

tion 4 presents the equipment and target implementations. Sections 5 and 6 describe the profiling and attack stages, respectively. Section 7 summarizes the experimental results. Section 8 concludes the paper and discusses open problems.

2 Previous work

This section describes previous side-channel attacks on protected implementations of CRYSTALS-Kyber and related schemes based on module lattices.

In [31], a two-step message recovery attack on masked software implementations of LWE/LWR PKE/KEMs is described. At the first step, each share is recovered using templates created on traces with known masks. At the second step, the message is computed from the shares.

In [27], a one-step message recovery method is introduced in which the message is recovered directly, without recovering the shares explicitly. This is done using a neural network which is trained at the profiling stage on traces containing both shares labelled by the value of the corresponding message bit. A great advantage of the one-step method is that it allows for the profiling on traces captured from the device under attack. This helps maximize the neural network prediction accuracy, since neither intra-device variability, nor device aging, degrade the models. In [27], the one-step method is applied to the first-order masked software implementation of Saber from [6] using poly_A2A() and POL2MSG() procedures as the attack points. In [29], it is extended to the second- and third-order masked software implementations of Saber from [21] using A2B_bitsliced_msg() procedure as the attack point. In [30] it is applied to the second-order masked software implementation of Saber from [21] using POL2MSG() as the attack point and the error-injection method [38] as the attack strategy. In both attacks, [29] and [30], neural networks trained at the profiling stage have no trouble recovering the value of a specific message bit from traces that contain all shares corresponding to this bit. However, our experiments show that difficulties begin to appear at higher masking orders.

The one-step message recovery method [27] is utilized in the side-channel attacks on a first-order masked and shuffled software implementation of Saber presented in [28] and [5]. These attacks require 61,680 and 4,608 traces to extract the secret key by deep learning-based power analysis, respectively. The former attack is based on message Hamming weight extraction and bit flipping, while the later on shuffling index recovery and cyclic rotations. In [5], a successful secret key recovery attack on a first-order masked and shuffled software implementation of CRYSTALS-Kyber is also demonstrated. The implementation is built on the top of the first-order masked implementation from [15]. The attack uses masked_poly_tomsg() procedure as the attack point.

In [37], the one-step method [27] is applied to the first-order masked implementation of CRYSTALS-Kyber, targeting the message encoding vulnerability found in [34]. In [8], side-channel attacks on two implementations of masked polynomial comparison are shown on the example of CRYSTALS-Kyber.

CPAPKE.KeyGen()
1: $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $\mathbf{A} \leftarrow \mathcal{U}(R_q^{k\times k}; seed_{\mathbf{A}})$
3: $\mathbf{s} \leftarrow \mathcal{B}_{\eta_1}(R_q^{k\times 1})$
4: $\mathbf{e} \leftarrow \mathcal{B}_{\eta_1}(R_q^{k\times 1})$
5: $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}_{p_1}$
6: $pk = (seed_{\mathbf{A}}, \mathbf{b}), sk = \mathbf{s}$
7: **return** (pk, sk)

CPAPKE.Dec($\mathbf{s}, c = (\mathbf{u}, v)$)
1: $y = \lfloor v \cdot q/2^{d_v} \rceil - \mathbf{s} \lfloor \mathbf{u} \cdot q/2^{d_u} \rceil$
2: $m' = \mathsf{decode}(y)$
3: **return** m'

CPAPKE.Enc($pk = (seed_{\mathbf{A}}, \mathbf{b}), m, r$)
1: $\mathbf{A} \leftarrow \mathcal{U}(R_q^{k\times k}; seed_{\mathbf{A}})$
2: $\mathbf{s}' \leftarrow \mathcal{B}_{\eta_1}(R_q^{k\times 1}; r)$
3: $\mathbf{e}' \leftarrow \mathcal{B}_{\eta_2}(R_q^{k\times 1}; r)$
4: $e'' \leftarrow \mathcal{B}_{\eta_2}(R_q^{1\times 1}; r)$
5: $\mathbf{u} = \lfloor (\mathbf{A}\mathbf{s}' + \mathbf{e}') \cdot 2^{d_u}/q \rceil$
6: $\mathbf{v} = \lfloor (\mathbf{b} \cdot \mathbf{s}' + e'' + \mathsf{encode}(m)) \cdot 2^{d_v}/q \rceil$
7: **return** $c = (\mathbf{u}, v)$

Fig. 1: Pseudocode of CCAPKE algorithms [5].

Kyber.KeyGen()
1: $z \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $(pk, \mathbf{s}) = \mathsf{CPAPKE.KeyGen}()$
3: $sk = (\mathbf{s}, pk, \mathcal{H}(pk), z)$
4: **return** (pk, sk)

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

Kyber.Decaps($sk = (\mathbf{s}, pk, \mathcal{H}(pk), z), c$)
1: $m' = \mathsf{CPAPKE.Dec}(\mathbf{s}, c)$
2: $(\hat{K}', r') = \mathcal{G}(m', \mathcal{H}(pk))$
3: $c' = \mathsf{CPAPKE.Enc}(pk, m', r')$
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: Pseudocode of CCAKEM algorithms [5].

Apart from the one-step message recovery method, an error-correcting code-based chosen ciphertext construction approach is introduced in [27]. The secret key of Saber is recovered from messages extracted from these ciphertexts. In [27], an extended Hamming code with the code distance four is used. It can correct single-bit errors and detect one additional error in each secret key coefficient. More powerful codes with the code distance up to six for Saber and up to eight for CRYSTALS-Kyber are utilized in [5]. The secret key is recovered using $k \cdot l$ chosen ciphertexts, where k is the rank of the module and l is the length of the code. Another code-based chosen ciphertext construction method for CRYSTALS-Kyber is described in [32]. While the non-code-based secret key recovery method presented earlier in [31] uses less ciphertexts than the code-based methods [27,5,32], its drawback is the perfect message recovery requirement.

3 Background

This section provides background information on CRYSTALS-Kyber [3] and masking countermeasure against side-channel attacks.

3.1 CRYSTALS-Kyber

Fig. 1 and 2 show pseudocodes of CPAPKE and CCAKEM algorithms, respectively. We follow the notation of [5]. CPAPKE consists of key generation, CPAPKE.KeyGen(); encryption, CPAPKE.Enc(); and decryption, CPAPKE.Dec(), parts. CPAKEM contains key generation, CCAKEM.KeyGen(); encapsulation, CCAKEM.Encaps(); and decapsulation, CCAKEM.Decaps(), parts.

The ring R_q in CPAPKE is the quotient ring $\mathbb{Z}_q[X]/(X^{256}+1)$, where $\mathbb{Z}_q$ is the ring of integers modulo a positive integer $q = 3329$. Sampling v from a distribution χ_i over a set S is denoted by $v \leftarrow \chi_i(S)$ while $v \leftarrow \chi_i(S;r)$ denotes deterministic sampling from χ_i using seed r. The centered binomial distributions is denoted by $\mathcal{B}_{\eta_i}$ and the uniform distribution is denoted by $\mathcal{U}$.

The security level of CRYSTALS-Kyber is defined by the rank k of the module. In this paper, we focus on $k = 3$, Kyber768. Other cases can be handled similarly.

3.2 Masking

Masking is a well-known countermeasure against power/EM side-channel analysis [12]. A ω-order masking partitions any sensitive variable x into $\omega+1$ shares, $x_1, x_2, \ldots, x_{\omega+1}$, such that $x = x_1 \circ x_2 \circ \ldots \circ x_{\omega+1}$, and performs es all operations separately on the shares. The operator "$\circ$" depends on the type of masking, e.g. in arithmetic masking "$\circ$" is equal to "$+$" and in Boolean masking "$\circ$" is "$\oplus$" .

Since computations do not involve x directly, carrying out operations separately on the shares $x_1, x_2, \ldots, x_{\omega+1}$ should in theory prevent the leakage of side-channel information related to x. Instead, the shares $x_1, x_2, \ldots, x_{\omega+1}$ are linked to the leakage. Since the shares are randomized at each execution, they are not expected to contain any exploitable information about x. Typically, the randomization is performed by assigning random masks $\phi_1, \phi_2, \ldots, \phi_\omega$ to ω shares and deriving the last share as $x - (\phi_1 + \phi_2 + \ldots + \phi_\omega)$ for arithmetic masking or $x \oplus \phi_1 \oplus \phi_2 \oplus \ldots \oplus \phi_\omega$ for Boolean masking.

4 Experimental setup

In this section, we describe the equipment and target implementation.

4.1 Target implementation

To the best of our knowledge, there are no publicly available higher-order masked implementations of CRYSTALS-Kyber[1]. All experiments presented in this paper are performed on the C implementation of the first-order masked CRYSTALS-Kyber from [16] which we modified to extend masked_poly_frommsg() procedure to a higher-order masking. We extended the part of the code required for proving

[1] The higher-order masked software implementation of CRYSTALS-Kyber presented in [9] is not publicly available.

```
void masked_poly_frommsg(uint16 poly[2][256], uint8
msg[2][32])
uint16 c[2];
 1: for (i = 0; i < 32; i++) do
 2:   for (j = 0; j < 8; j++) do
 3:     mask = -((msg[0][i] » j) & 1);
 4:     poly[0][8*i+j] += (mask&((KYBER_Q+1)/2));
 5:   end for
 6: end for
 7: for (i = 0; i < 32; i++) do
 8:   for (j = 0; j < 8; j++) do
 9:     mask = -((msg[1][i] » j) & 1);
10:     poly[1][8*i+j] += (mask&((KYBER_Q+1)/2));
11:   end for
12: end for
13: ...
```

Fig. 3: C code of masked_poly_frommsg() procedure of CRYSTALS-Kyber [16].

the attack's concept only, not the complete CRYSTALS-Kyber package. We verified that power traces captured from the extended implementation for $\omega = 1$ are similar to the traces of the original implementation [16]. We also verified that both implementations have a similar type of side-channel leakage by performing message recovery attacks on both. The resulting empirical mean message bit recovery probabilities were similar in both cases.

Our attack point is the procedure masked_poly_frommsg() shown in Fig. 3. This procedure is called during the re-encryption step of decapsulation (line 3 of CCAKEM.Decaps() in Fig. 2). It performs the encoding of message shares from the Boolean domain into the polynomial domain (line 6 of CPAPKE.Enc() in Fig. 1). The red lines in Fig. 3 show the location of the vulnerability exploited in the attack. Such types of vulnerabilities are known as the *determiner-leakage* in previous works [2,34,31].

4.2 Equipment

For trace acquisition, we use a Chipwhisperer-lite board [25], a CW308 UFO board, and a CW308T-STM32F4 target board. CW308T-STM32F4 contains an ARM Cortex-M4 CPU with STM32F415-RGT6 device. The STM32F415-RGT6 is programmed to a C implementation of CRYSTALS-Kyber compiled with `arm-none-eabi-gcc` with the optimization level -O3 (recommended default). The target board is run and sampled at 24 MHz.

5 Profiling stage

In this section, we describe our profiling strategy. The main difference from previous work is the new neural network training method which we call *recursive*

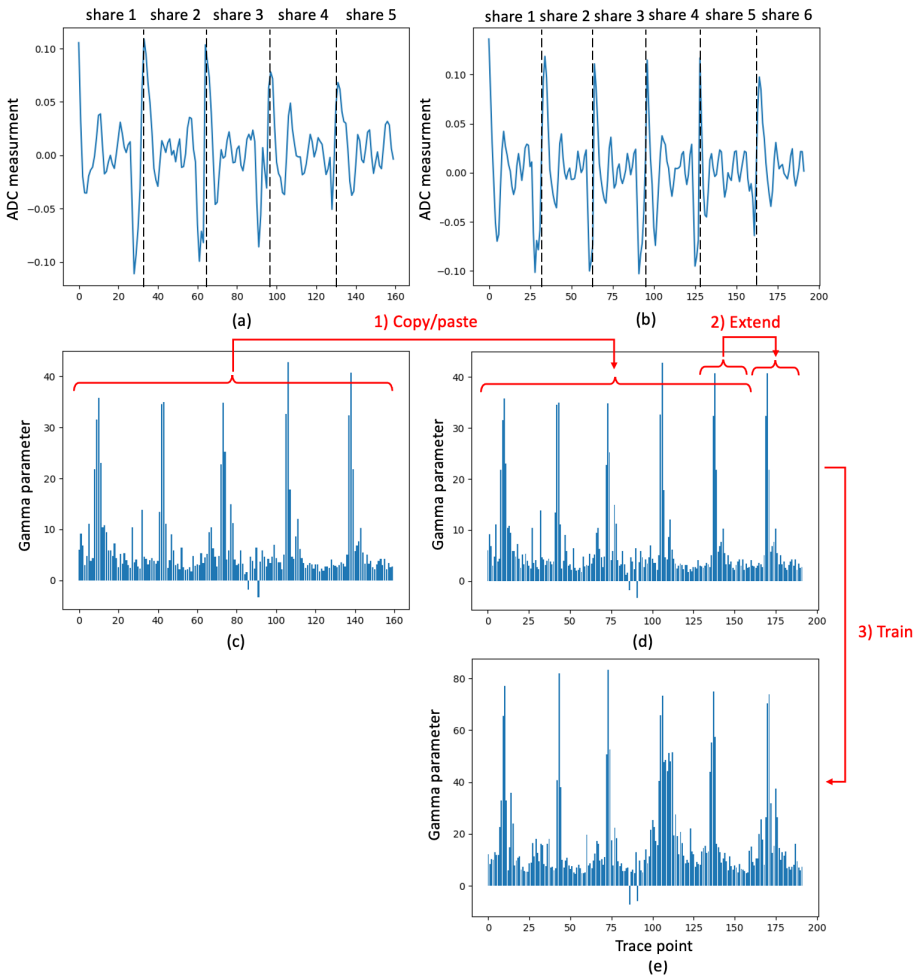

Fig. 4: (a,b) Power traces given as input to neural networks for attacks on 4th- and 5th-order masked implementations, respectively; (c) Weights of input Batch Normalization layer after training for 4th-order; (d) Batch Normalization extended to 5th-order; (e) Batch Normalization after training for 5th-order.

learning. The key idea is to use a neural network model $M^{\omega-1}$, capable of breaking a $(\omega-1)$-order masked implementation, as a stepping stone in the attack on the ω-order masked implementation.

5.1 Recursive learning method

The attack presented in [27] has shown that neural networks are capable to:

1. Identify trace segments corresponding to two message shares in a first-order masked software implementation of an LWE/LWR PKE/KEM scheme, and
2. XOR values of the shares to obtain the ground truth label.

The fact that neural networks are capable of learning the two-argument XOR operation was previously known [14, p. 166]. The results of [27] confirmed that the neural networks can combine the steps (1) and (2). In [29] it was further hypothesized that the complexity of learning the n-argument XOR grows linearly in the number of arguments n. Indeed, neural networks for message recovery from the second- and third-order masked software implementations of LWE/LWR PKE/KEMs can be trained directly [29,30], as in the attack of [27].

However, we found that, for higher masking orders, neural networks need help in order to learn. To help, we start training from an artificially constructed neural network, rather than a network with a standard random weight distribution. For an attack on an ω-order masked implementation, the starting network M^{ω} is constructed by copying the weights of the input Batch Normalization layer of a model $M^{\omega-1}$ trained on the $(\omega-1)$-order masked implementation, and then extending the input Batch Normalization layer of M^{ω} to include one more share, as shown in Fig. 4. In this way, we assist M^{ω} in locating the traces' segments corresponding to the processing of $\omega+1$ shares. The weights of all other layers of M^{ω} are assigned at random.

Recall that a Batch Normalization layer first standardizes the input values x of the layer using their respective mean, μ, and standard deviation, σ, $x_{norm} = (x-\mu)/\sigma$, and then applies the scaling, γ (gamma) and offset, β (beta), parameters to the result, $x' = (\gamma * x_{norm}) + \beta$. The parameters γ and β are learned by the model during the training process by adjusting the back-propagation algorithm to operate on the transformed inputs, and using the error to update the new scaling and offset parameters learned by the model. Thus, a higher value of γ indicates the higher importance of the corresponding input feature in the decision taken by the model.

In our case, a neural network $M^{\omega-1}$ takes input values $x = x_1||x_2||\ldots||x_{\omega}$ of ω concatenated share segments. Let γ_i^j, β_i^j, μ_i^j, and σ_i^j denote scaling parameters, offset parameters, mean and standard deviation of x_i of the input Batch Normalization layer of M^j, for $i \in \{1,2,\ldots,j+1\}$ and $j>0$.

The input Batch Normalization layer of M^{ω}, which takes input values $x = x_1||x_2||\ldots||x_{\omega+1}$, is constructed based on the weights of the input Batch Normalization layer of $M^{\omega-1}$ as follows:

$$\begin{aligned}
&\gamma_i^{\omega} = \gamma_i^{\omega-1} \text{ for all } i \in \{1,2,\ldots,\omega\}, \gamma_{\omega+1}^{\omega} = \gamma_{\omega}^{\omega-1},\\
&\beta_i^{\omega} = \beta_i^{\omega-1} \text{ for all } i \in \{1,2,\ldots,\omega\}, \beta_{\omega+1}^{\omega} = \beta_{\omega}^{\omega-1},\\
&\mu_i^{\omega} = \mu_i^{\omega-1} \text{ for all } i \in \{1,2,\ldots,\omega\}, \mu_{\omega+1}^{\omega} = \mu_{\omega}^{\omega-1},\\
&\sigma_i^{\omega} = \sigma_i^{\omega-1} \text{ for all } i \in \{1,2,\ldots,\omega\}, \sigma_{\omega+1}^{\omega} = \sigma_{\omega}^{\omega-1}.
\end{aligned}$$

In other words, we copy all parameters corresponding to ω shares and then extend by repeating the parameters of the last share one more time. Note that any other share of $M^{\omega-1}$ can be repeated instead of the last one.

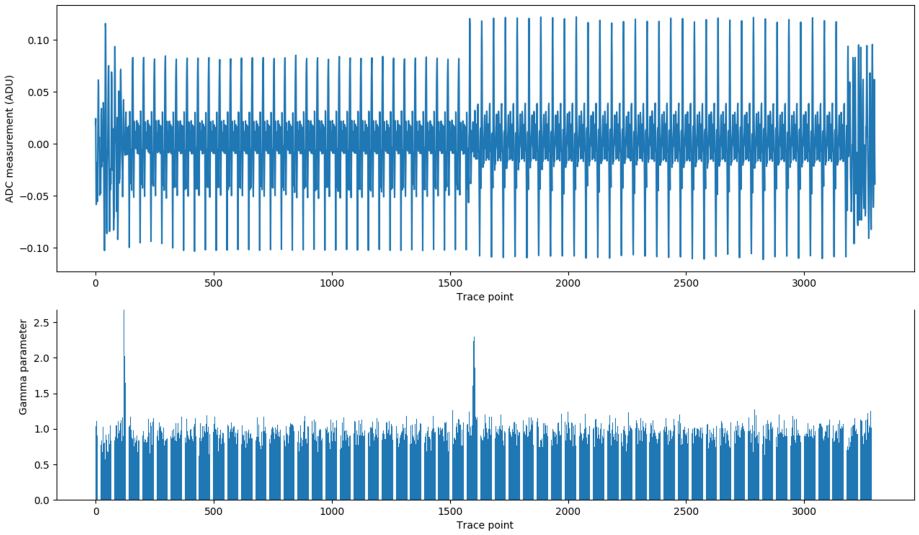

Fig. 5: (top) Full power trace representing the execution of masked_poly_frommsg() in the first-order masked implementation; (bottom) Weights of input Batch Normalization layer of a neural network trained on full traces with the 1st message bit values as labels. The peaks reveal the positions of two shares.

The recursive learning method allows us to easily train good neural network models for message recovery attacks on high-order masked implementations. For masking orders $\omega > 4$, the probability of finding such a model without recursive learning is very small.

The recursive learning method might be viewed as related to transfer learning [39]. Different types of iterative re-training methods were used in side-channel attacks in the past, e.g. in [26,29]. Note, however, that we do not only transfer a model, but also extend it. We construct the initial weights of a larger neural network based on the weights of a smaller neural network. We could not find an analog of such a technique among known transfer learning methods.

5.2 Neural network type

As in the previous deep learning-based message recovery attacks on masked software implementations of LWE/LWR PKE/KEMs [27,28], we train neural networks to recover messages directly, without explicitly extracting the random masks at each execution. Message bits values '0' and '1' are used as labels.

Let $\mathbb{R}$ be the set of real numbers and let $\mathbb{I} := \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$. Let $m = (m[0], m[1], \ldots, m[255])$ be a message of CRYSTALS-Kyber to be recovered, where $m[i]$ is the ith message bit, and $c = (\mathbf{u}, v)$ be a ciphertext generated by CPAPKE.Enc() for m.

To train a neural network $M_i : \mathbb{R}^{|\mathbf{T}|} \rightarrow \mathbb{I}$ which predicts $m[i]$, each trace T in the training set $\mathbf{T}$ is labeled by $m[i]$, where m is the message encrypted in c which is given as input to the device when T is captured. M_i maps each T into a score $s = M_i(T) \in \mathbb{I}$ representing the probability that $m[i] = 1$ in T.

5.3 Input data selection

For the first-order masked implementation, we locate the segments of traces corresponding to the processing of shares by masked_poly_frommsg() by training neural networks on full traces and examining the weights of the input Batch Normalization layer after training, as shown in Fig. 5. The p-point intervals containing the peaks of γ parameters are extracted and concatenated to obtain the input data for to neural networks. In our experiments, we use $p = 32$.

First we determine to location of shares for each message bit $i \in \{0, 1, \ldots, 255\}$ and estimate the distance between two adjacent bits. This is done by training neural networks M_i^1 and M_{i+1}^1 for several bits i and $i + 1$ on full traces from the first-order masked implementation and measuring the distance between the peaks of γ parameters of their input Batch Normalization layers after training.

In the same way we determine the distance between of shares x_1 and x_2, d, by measuring the distance between the peaks of γ parameters of the input Batch Normalization layers of M_i^1 after training.

Once d for x_1 and x_2 is determined for the first-order masked implementation, an approximate location of each share x_j in an ω-order masked implementation can be found by shifting the first share's interval by $(j - 1) * d$ points, for $j \in \{2, \ldots, \omega + 1\}$. For example, if the p-point interval of trace corresponding to x_1 share is $[s : s{+}p]$, then the interval corresponding to $x_{\omega+1}$ share is $[s{+}\omega * d : s{+}\omega * d{+}p]$.

To summarize, for an ω-order masked implementation, the input data given to M_i^ω is a concatenation of $\omega + 1$ p-point trace intervals $[s + (j - 1) * d : s{+}(j - 1) * d{+}p]$, for $j \in \{1, \ldots, \omega + 1\}$ corresponding to the ith bit of shares $x_1, \ldots, x_{\omega+1}$. Fig. 4 (a) and (b) show examples for the fourth- and fifth-order masked implementations, respectively. The black dashed lines show the borders where the shares are concatenated.

5.4 Leakage analysis

We found that the leakage of message bits in masked_poly_frommsg() procedure is non-uniform. The first bit of each message byte leaks considerably stronger than the last one. This is apparent from the difference in the success rate of neural network models which recover these bits. For example, for the first-order masked implementation, the difference between the mean empirical probabilities to recover the bit 0 and the bit 7 is 9%.

We address this issue by applying cyclic rotations to shift the bits which leak less to the bit positions which leak more. The method is described in Section 6.1.

Table 1: MLP architecture for message bits recovery from an ω-order masked implementation of CRYSTALS-Kyber. The neural network input size is $32(\omega+1)$.

Layer type	Output shape
Batch Normalization 1	$32(\omega+1)$
Dense 1	$32(\omega+1)$
Batch Normalization 2	$32(\omega+1)$
ReLU	$32(\omega+1)$
Dense 2	$2^{\omega+4}$
Batch Normalization 3	$2^{\omega+4}$
ReLU	$2^{\omega+4}$
Dense 3	$2^{\omega+3}$
Batch Normalization 4	$2^{\omega+3}$
ReLU	$2^{\omega+3}$
Dense 4	1
Softmax	1

5.5 Architecture and training parameters

To recover messages from an ω-order masked implementation, we use multilayer perceptron (MLP) neural networks with the architecture shown in Table 1. For the attacks on implementations with masking orders $\omega \leq 3$, we start training from a network with a standard random weight distribution. For $\omega > 3$, the recursive learning method is used.

We cut-and-join training traces byte-wise using the technique of [27]. The resulting extended by a factor of 32 set is used to train two universal models, M_0^ω and M_1^ω, which recover the first and the second bit of each message byte (bits with the strongest leakage). For the first-order masked implementation, we also train the models M_i^1 for bits $i \in \{2,3,4,5,6,7\}$, in order to quantify the difference between approaches with and without cyclic rotations.

The neural networks are trained with a batch size of 1024 for a maximum of 100 epochs using early stopping with patience 20. We use Nadam optimizer with a learning rate of 0.01 and a numerical stability constant *epsilon* = 1e-08. Binary 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.

6 Attack stage

This section describes our attack strategy. The main difference from previous attacks of masked implementations of LWE/LWR PEK/KEMs [27,28,30] is the cyclic rotation method which we use to compensate for the non-uniformity of leakage in masked_poly_frommsg() procedure.

6.1 Cyclic rotation method

It is known that a message of a ring-based LWE/LWR can be cyclically rotated by manipulating the corresponding ciphertext. Such a possibility was first described in [31] where it was hypothesised that rotations may be useful in side-channel attacks. Since then two attacks utilizing cyclic rotations were presented.

The first is the template side-channel attack on an unprotected software implementation of CRYSTAL-Kyber presented in [41]. It uses cyclic message rotations to construct ciphertexts which are used during profiling to create templates for the intermediate states of the message decoding operation based on the Hamming weight model. At the attack stage, the templates are applied to recover messages during the encapsulation.

The second is the deep learning-based attack on the first-order masked and shuffled software implementations of CRYSTAL-Kyber and Saber presented in [5]. It recovers a message m contained in a given ciphertext c by modifying c to negacyclically rotate m 128 times by two bits. At each rotation, two message bits with the shuffling indexes 0 and 255 are extracted using a neural network for bits recovery. Another neural network is used for recovering shuffling indexes of the bits.

In the attacks presented in this paper, we negacyclically rotate a message three times by two bits to subsequently shift the last six bits of each byte to the positions of the two first bits. In this way, we make use of "more leaky" bit positions to extract the bit values with a higher probability. This allows us to increase the attack success rate.

The rotation of a message is performed by manipulating the corresponding ciphertext. In CRYSTALS-Kyber, a ciphertext $c = (\mathbf{u}, v)$ consists of polynomials in the ring $\mathbb{Z}_q[X]/(X^{256}+1)$. If c is properly generated, a negacyclic rotation of the message can be performed by multiplying both $\mathbf{u}$ and v by indeterminate X [31]. However, as pointed out in [5], for chosen ciphertexts which are used in secret key recovery attacks, such a method may cause errors because $\mathsf{decode}(-y)$ and $\mathsf{decode}(y)$ can evaluate to different values. A method customized to specific chosen ciphertexts is presented in [5] as a solution to this problem.

7 Experimental results

At the profiling stage, for each ω-order masked implementation of CRYSTAL-Kyber, $\omega \in \{1, 2, \ldots, 5\}$, we capture from the device under attack 30K traces for the training of neural networks. The traces are captured during the execution of CCAKEM.Decaps() with input ciphertexts encrypting messages selected at random. Since CRYSTAL-Kyber is a public-key algorithm, the encryption is performed using the public key of the device under attack. Hence, the attacker can apply CPAPKE.Enc() to generate a proper ciphertext c for any message, and then use the device under attack to decrypt c. The training set is expanded to 960K using the byte-wise cut-and-join technique of [27].

Table 2: Empirical probability (mean over 10K tests) to recover a message bit from a single trace of a first-order masked implementation w/o cyclic rotations.

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	0.9973	0.9977	0.9854	0.9621	0.9697	0.9546	0.9260	0.8958	0.9611
1	0.9995	0.9989	0.9930	0.9732	0.9841	0.9766	0.9501	0.9070	0.9728
2	0.9997	0.9993	0.9939	0.9809	0.9803	0.9794	0.9466	0.9157	0.9745
3	0.9996	0.9992	0.9945	0.9860	0.9868	0.9828	0.9481	0.9204	0.9772
4	0.9996	0.9995	0.9945	0.9881	0.9881	0.9838	0.9479	0.9166	0.9773
5	0.9996	0.9990	0.9962	0.9863	0.9904	0.9834	0.9448	0.9121	0.9765
6	0.9994	0.9990	0.9967	0.9851	0.9898	0.9859	0.9452	0.9044	0.9757
7	0.9990	0.9993	0.9954	0.9860	0.9885	0.9866	0.9442	0.9106	0.9762
8	0.9997	0.9993	0.9972	0.9834	0.9877	0.9835	0.9454	0.9006	0.9746
9	0.9992	0.9990	0.9948	0.9873	0.9902	0.9876	0.9488	0.9113	0.9773
10	0.9995	0.9984	0.9955	0.9824	0.9879	0.9838	0.9395	0.9128	0.9750
11	0.9994	0.9987	0.9961	0.9847	0.9894	0.9854	0.9453	0.9080	0.9759
12	0.9989	0.9993	0.9960	0.9842	0.9883	0.9848	0.9437	0.9066	0.9752
13	0.9992	0.9987	0.9972	0.9845	0.9890	0.9846	0.9358	0.9016	0.9738
14	0.9995	0.9990	0.9953	0.9849	0.9888	0.9847	0.9412	0.9041	0.9747
15	0.9988	0.9988	0.9955	0.9838	0.9897	0.9855	0.9454	0.9090	0.9758
16	0.9993	0.9985	0.9958	0.9866	0.9896	0.9848	0.9349	0.9045	0.9742
17	0.9993	0.9994	0.9957	0.9866	0.9894	0.9825	0.9370	0.9054	0.9744
18	0.9996	0.9988	0.9957	0.9869	0.9884	0.9865	0.9377	0.9118	0.9757
19	0.9989	0.9988	0.9954	0.9849	0.9894	0.9831	0.9410	0.9034	0.9744
20	0.9992	0.9994	0.9952	0.9862	0.9871	0.9877	0.9392	0.9120	0.9758
21	0.9992	0.9988	0.9946	0.9855	0.9868	0.9836	0.9369	0.9036	0.9736
22	0.9990	0.9989	0.9959	0.9836	0.9880	0.9856	0.9303	0.9049	0.9733
23	0.9993	0.9993	0.9967	0.9863	0.9888	0.9849	0.9375	0.9057	0.9748
24	0.9990	0.9990	0.9953	0.9840	0.9885	0.9842	0.9291	0.8905	0.9712
25	0.9991	0.9989	0.9966	0.9864	0.9880	0.9855	0.9368	0.9064	0.9747
26	0.9992	0.9994	0.9962	0.9834	0.9901	0.9866	0.9351	0.9101	0.9750
27	0.9990	0.9991	0.9953	0.9822	0.9869	0.9844	0.9374	0.9111	0.9744
28	0.9986	0.9987	0.9963	0.9855	0.9877	0.9852	0.9342	0.9046	0.9738
29	0.9992	0.9988	0.9955	0.9857	0.9882	0.9863	0.9360	0.9026	0.9740
30	0.9992	0.9987	0.9964	0.9871	0.9891	0.9857	0.9296	0.8987	0.9731
31	0.9993	0.9988	0.9958	0.9859	0.9890	0.9835	0.9282	0.9021	0.9728
avg	0.9992	0.9989	0.9953	0.9841	0.9876	0.9835	0.9393	0.9067	**0.974338**

At the attack stage, for each ω-order masked implementation, we capture 10K traces during the execution of CCAKEM.Decaps(). The attack traces are captured with three two-bit negacyclic rotations for each message. Thus, 2.5K messages, selected at random, are tested in total. For the fifth-order masked implementation, we capture the attack set five times to evaluate the effect of repetitions on the success rate of message recovery.

For the first-order masked implementations, we train eight models, M_i^1, one for each bit position in a byte, for all $i \in \{0, 1, \ldots, 7\}$. Table 2 shows that the resulting empirical probability of successful message bit recovery is too low, $p_{bit} = 0.97434$, on average. This gives us only $p_{message} = p_{bit}^{256} = 0.00127$ average message recovery probability. One can see that the prediction accuracy decreases towards the last bit of a byte.

Then we apply cyclic rotations. For each test message, we capture traces with i two-bit cyclic rotations, for $i \in \{0, 1, 2, 3\}$. The bits $2i$ and $2i + 1$ of each byte are recovered using the models M_0^1 and M_1^1, respectively. As Table 3 shows,

Table 3: Empirical probability (mean over 2.5K tests) to recover a message bit from a single trace of a first-order masked implementation with cyclic rotations.

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	0.9968	0.9984	0.9976	0.9972	0.9980	0.9964	0.9968	0.9988	0.9975
1	0.9996	0.9992	0.9992	0.9988	0.9996	0.9988	0.9996	0.9988	0.9992
2	0.9996	1.0000	1.0000	0.9988	0.9992	0.9988	1.0000	0.9996	0.9995
3	0.9996	0.9992	0.9988	0.9988	1.0000	0.9992	1.0000	0.9996	0.9994
4	0.9996	0.9996	1.0000	0.9996	0.9992	0.9992	0.9996	0.9996	0.9996
5	1.0000	0.9984	0.9992	0.9992	0.9992	0.9992	1.0000	0.9992	0.9993
6	0.9992	0.9992	0.9996	0.9984	0.9996	0.9992	0.9992	0.9992	0.9992
7	0.9992	0.9996	0.9992	0.9992	0.9988	0.9996	0.9988	0.9988	0.9992
8	1.0000	0.9996	0.9996	0.9984	0.9992	1.0000	1.0000	0.9992	0.9995
9	0.9992	0.9988	0.9992	0.9988	0.9988	0.9984	0.9996	1.0000	0.9991
10	0.9992	0.9992	0.9996	0.9992	0.9996	0.9976	0.9996	0.9976	0.9990
11	0.9996	1.0000	0.9992	0.9988	0.9988	0.9976	1.0000	0.9984	0.9991
12	0.9988	0.9996	0.9984	0.9996	0.9988	0.9984	0.9996	0.9996	0.9991
13	0.9996	0.9996	0.9992	0.9988	0.9984	0.9976	0.9996	0.9988	0.9990
14	0.9996	0.9988	0.9992	0.9996	0.9992	0.9984	1.0000	0.9992	0.9993
15	0.9996	0.9984	0.9988	0.9992	0.9980	0.9988	0.9988	0.9988	0.9988
16	0.9996	0.9988	0.9996	0.9980	0.9984	0.9976	0.9996	0.9996	0.9989
17	0.9992	0.9988	0.9996	1.0000	0.9988	0.9992	0.9996	0.9996	0.9994
18	1.0000	0.9988	0.9996	0.9984	0.9992	0.9988	0.9996	0.9992	0.9992
19	0.9992	0.9992	0.9996	1.0000	0.9976	0.9972	0.9992	0.9988	0.9989
20	0.9992	0.9996	0.9996	0.9996	0.9988	0.9988	0.9992	0.9996	0.9993
21	0.9992	0.9984	0.9992	0.9996	0.9984	0.9980	1.0000	0.9992	0.9990
22	0.9996	0.9996	0.9996	1.0000	0.9984	0.9968	0.9984	0.9992	0.9990
23	1.0000	0.9988	0.9992	0.9996	0.9984	0.9988	0.9996	1.0000	0.9993
24	0.9988	0.9988	0.9992	0.9996	0.9988	0.9988	0.9992	0.9988	0.9990
25	0.9992	0.9992	1.0000	0.9984	0.9984	0.9988	0.9988	0.9992	0.9990
26	0.9996	0.9996	0.9984	0.9992	0.9992	0.9988	0.9996	1.0000	0.9993
27	0.9996	0.9984	0.9992	0.9992	0.9972	0.9988	1.0000	1.0000	0.9990
28	0.9992	0.9988	0.9992	0.9980	0.9984	0.9992	0.9976	0.9988	0.9987
29	1.0000	0.9988	0.9996	0.9988	0.9980	0.9984	0.9992	0.9992	0.9990
30	0.9996	0.9984	0.9992	0.9992	0.9992	0.9984	0.9988	0.9988	0.9990
31	1.0000	0.9984	0.9992	0.9992	0.9988	0.9984	0.9992	0.9992	0.9990
avg	0.9994	0.9991	0.9993	0.9990	0.9988	0.9985	0.9993	0.9992	**0.99907**

Table 4: Empirical probability (mean over 2.5K tests) to recover a message from a single trace of an ω-order masked implementation with cyclic rotations.

ω	1	2	3	4	5
p_{bit}	0.99907	0.99829	0.99639	0.98582	0.97995
$p_{message}$	0.78866	0.68567	0.39641	0.02585	0.00560

empirical probability of successful message bit recovery improves considerably, to $p_{bit} = 0.99907$ on average.

We repeat the attack with cyclic rotations on ω-order masked implementations for $\omega \in \{2, 3, 4, 5\}$. The resulting bit and message recovery probabilities as summarized in Table 4. The full tables are shown in the Appendix.

Finally, we check if repeating the same measurement multiple times and applying majority voting to the predictions can further improve the success rate.

Table 5: Empirical probability (mean over 2.5K tests) to recover a message bit from five traces of a fifth-order masked implementation with cyclic rotations .

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	1.0000	0.9764	1.0000	0.9788	1.0000	0.9868	1.0000	0.9760	0.9897
1	1.0000	0.9920	1.0000	0.9924	1.0000	0.9940	1.0000	0.9892	0.9960
2	1.0000	0.9980	1.0000	0.9996	1.0000	0.9972	1.0000	1.0000	0.9994
3	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999
4	1.0000	1.0000	1.0000	0.9996	1.0000	1.0000	1.0000	1.0000	1.0000
5	1.0000	0.9996	1.0000	1.0000	1.0000	0.9988	1.0000	1.0000	0.9998
6	1.0000	0.9996	1.0000	0.9988	1.0000	1.0000	1.0000	1.0000	0.9998
7	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000	0.9992	0.9999
8	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000
9	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000
10	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
11	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000
12	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
13	1.0000	1.0000	1.0000	0.9996	1.0000	1.0000	1.0000	0.9996	0.9999
14	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000
15	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000	1.0000	1.0000
16	1.0000	0.9996	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
17	1.0000	1.0000	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	0.9999
18	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	1.0000	1.0000	1.0000
19	1.0000	1.0000	1.0000	0.9996	1.0000	1.0000	1.0000	1.0000	1.0000
20	1.0000	1.0000	1.0000	0.9992	1.0000	1.0000	1.0000	0.9988	0.9998
21	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	1.0000	0.9996	0.9999
22	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999
23	1.0000	1.0000	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	0.9999
24	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
25	1.0000	0.9996	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
26	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999
27	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
28	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
29	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999
30	1.0000	1.0000	1.0000	1.0000	1.0000	0.9992	1.0000	1.0000	0.9999
31	1.0000	0.9992	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999
avg	1.0000	0.9987	1.0000	0.9989	1.0000	0.9992	1.0000	0.9988	**0.99946**

Table 6: Empirical probability (mean over 2.5K tests) to recover a message from N traces of a fifth-order masked implementation with cyclic rotations.

N	1	3	5
p_{bit}	0.97995	0.99763	0.99946
$p_{message}$	0.00560	0.54530	0.87085

Using the fifth-order masked implementation, we capture test traces with five repetitions. As Tables 5 and 6 show, the repetitions boost the message recovery probability from 0.56% to 87%. We believe that such a significant increase is due to the fact that random masks are updated at each execution. This contributes to the independence of errors in the repeated measurements. Thus, ironically, something that is intended to make the attack harder actually makes it easier.

8 Conclusion

We demonstrated side-channel attacks on up to the fifth-order masked software implementations of CRYSTALS-Kyber exploiting a vulnerability in the procedure masked_poly_frommsg() which is called during the re-encryption phase of decapsulation.

The success of the attacks is due to the recursive learning neural network training method and cyclic rotation-based message recovery method introduced in this paper. To the best of our knowledge, no side-channel attack on a higher than the third order masked implementation of any LWE/LWR PKE/KEM scheme has been demonstrated until now. The presented approach is not specific for CRYSTALS-Kyber and can potentially be applied to other LWE/LWR PKE/KEM schemes. The recursive learning technique might have significance beyond side-channel attacks context.

We are currently working on developing countermeasures against side-channel attacks on LWE/LWR PKE/KEM schemes.

9 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: Int. Conference on Post-Quantum Cryptography. pp. 189–205. Springer (2020). https://doi.org/10.1109/ICDSP.2018.8631824
3. Avanzi, R., Bos, J., Léo Ducas, E.K., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber algorithm specifications and supporting documentation (2020), https://csrc.nist.gov/projects/postquantum-cryptography/round-3-submissions
4. Azouaoui, M., Kuzovkova, Y., Schneider, T., van Vredendaal, C.: Post-quantum authenticated encryption against chosen-ciphertext side-channel attacks. Cryptology ePrint Archive, Paper 2022/916 (2022), https://eprint.iacr.org/2022/916
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. Beirendonck, M.V., D'anvers, J.P., Karmakar, A., Balasch, J., Verbauwhede, I.: A side-channel-resistant implementation of SABER. ACM Journal on Emerging Technologies in Computing Systems (JETC) **17**(2), 1–26 (2021)
7. Belleville, N., Courousse, D., Heydemann, K., Charles, H.P.: Automated software protection for the masses against side-channel attacks. ACM Trans. Archit. Code Optim. **16**(4) (2018)

8. 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
9. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: First- and higher-order implementations. IACR Trans. on Cryptographic Hardware and Embedded Systems **2021**(4), 173–214 (Aug 2021). https://doi.org/10.46586/tches.v2021.i4.173-214
10. Brisfors, M., Forsmark, S., Dubrova, E.: How deep learning helps compromising USIM. In: Proc. of the 19th Smart Card Research and Advanced Application Conference (CARDIS'2020) (Nov 2020)
11. Brisfors, M., Moraitis, M., Dubrova, E.: Side-channel attack countermeasures based on clock randomization have a fundamental flaw. Cryptology ePrint Archive, Paper 2022/1416 (2022), https://eprint.iacr.org/2022/1416, https://eprint.iacr.org/2022/1416
12. 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). https://doi.org/10.1007/3-540-48405-1_26
13. Coron, J.S., Kizhvatov, I.: An efficient method for random delay generation in embedded software. In: Crypt. Hardware and Embedded Systems. pp. 156–170. Springer Berlin Heidelberg (2009)
14. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
15. 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
16. 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
17. 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
18. Jan-Pieter D'Anvers et al.: 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
19. 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, https://eprint.iacr.org/2022/1452
20. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual international cryptology conference. pp. 388–397. Springer (1999)
21. Kundu, S., D'Anvers, J.P., Beirendonck, M.V., Karmakar, A., Verbauwhede, I.: Higher-order masked Saber. Cryptology ePrint Archive, Paper 2022/389 (2022), https://eprint.iacr.org/2022/389
22. Maghrebi, H., Servant, V., Bringer, J.: There is wisdom in harnessing the strengths of your enemy: Customized encoding to thwart side-channel attacks. In: Fast Software Encryption. pp. 223–243 (2016)
23. 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

24. Mujdei, C., Beckers, A., Mera, J.M.B., Karmakar, A., Wouters, L., Verbauwhede, I.: Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. Cryptology ePrint Archive, Paper 2022/474 (2022), https://eprint.iacr.org/2022/474
25. NewAE Technology: Chipwhisperer, https://newae.com/tools/chipwhisperer
26. Ngo, K., Dubrova, E.: Side-channel analysis of the random number generator in STM32 MCUs. In: Proc. of the Great Lakes Symposium on VLSI (GLSVLSI '22) (2022), https://doi.org/10.1145/3526241.3530324
27. 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)
28. Ngo, K., Dubrova, E., Johansson, T.: Breaking masked and shuffled CCA secure Saber KEM by power analysis. In: Proc. of the 5th Workshop on Attacks and Solutions in Hardware Security. pp. 51–61 (2021)
29. Ngo, K., Wang, R., Dubrova, E., Paulsrud, N.: Side-channel attacks on lattice-based KEMs are not prevented by higher-order masking. Cryptology ePrint Archive, Paper 2022/919 (2022), https://eprint.iacr.org/2022/919, https://eprint.iacr.org/2022/919
30. Paulsrud, N.: A Side Channel Attack on a Higher-Order Masked Software Implementation of Saber. Master's thesis, School of Electrical Engineering and Computer Science, KTH (2022)
31. Ravi, P., Bhasin, S., Roy, S.S., Chattopadhyay, A.: On exploiting message leakage in (few) NIST PQC candidates for practical message recovery and key recovery attacks. Cryptology ePrint Archive, Paper 2020/1559 (2020), https://eprint.iacr.org/2020/1559
32. 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. Cryptology ePrint Archive, Paper 2022/563 (2022), https://eprint.iacr.org/2022/563, https://eprint.iacr.org/2022/563
33. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
34. Sim, B.Y., Kwon, J., Lee, J., Kim, I.J., Lee, T., Han, J., Yoon, H., Cho, J., Han, D.G.: Single-trace attacks on the message encoding of lattice-based KEMs. Cryptology ePrint Archive, Paper 2020/992 (2020), https://eprint.iacr.org/2020/992
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). https://doi.org/10.1109/OJCS.2022.3198073
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 Tran. on Cryptographic Hardware and Embedded Systems **2022**(1), 296–322 (Nov 2021). https://doi.org/10.46586/tches.v2022.i1.296-322
37. Wang, J., Cao, W., Chen, H., Li, H.: Practical side-channel attack on masked message encoding in latticed-based KEM. Cryptology ePrint Archive, Paper 2022/859 (2022), https://eprint.iacr.org/2022/859
38. Wang, R., Ngo, K., Dubrova, E.: A message recovery attack on LWE/LWR-based PKE/KEMs using amplitude-modulated EM emanations. In: Proc. of 25th Annual Int. Conf. on Information Security and Cryptology (2022), https://eprint.iacr.org/2022/852
39. Weiss, K., Khoshgoftaar, T.M., Wang, D.D.: A survey of transfer learning. Journal of Big Data **2021**(3) (May 2016). https://doi.org/10.1186/s40537-016-0043-6

40. Xu, Z., Pemberton, O.M., Sinha Roy, 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 pp. 1–1 (2021). https://doi.org/10.1109/TC.2021.3122997
41. Yajing, C., Yan, Y., Zhu, C., Guo, P.: Template attack of LWE/LWR-based schemes with cyclic message rotation. Entropy **24**(10) (2022)
42. Yu, Y., Moraitis, M., Dubrova, E.: Why deep learning makes it difficult to keep secrets in FPGAs. In: Proc. Workshop on DYnamic and Novel Advances in Machine Learning and Intelligent Cyber Security (DYNAMICS '20) (2020), https://doi.org/10.1145/3477997.3478001

10 Appendix

This appendix presents full tables of empirical probabilities (mean over 2.5K tests) to recover a message bit from a single trace of a second-, third-, fourth, and fifth-order masked implementation of CRYSTALS-Kyber using the presented attack method with cyclic rotations.

Second-order masked implementation with cyclic rotations

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	0.9988	0.9972	0.9988	0.9944	0.9992	0.9948	0.9988	0.9940	0.9970
1	0.9972	0.9976	0.9976	0.9980	0.9996	0.9984	0.9976	0.9972	0.9979
2	0.9984	0.9972	0.9992	0.9972	0.9988	0.9976	0.9992	0.9960	0.9980
3	0.9984	0.9976	0.9984	0.9976	0.9996	0.9980	0.9996	0.9984	0.9985
4	0.9980	0.9984	0.9984	0.9980	1.0000	0.9988	0.9984	0.9976	0.9985
5	0.9984	0.9992	0.9984	0.9992	0.9980	0.9984	0.9980	0.9988	0.9986
6	0.9984	0.9972	0.9980	0.9988	0.9996	0.9980	0.9984	0.9972	0.9982
7	0.9988	0.9972	0.9984	0.9992	0.9988	0.9984	0.9984	0.9980	0.9984
8	0.9992	0.9984	0.9988	0.9964	0.9988	0.9984	0.9992	0.9980	0.9984
9	0.9984	0.9976	0.9992	0.9988	1.0000	0.9988	0.9980	0.9984	0.9987
10	0.9992	0.9976	0.9988	0.9984	0.9996	0.9980	0.9992	0.9960	0.9984
11	0.9972	0.9980	0.9988	0.9992	0.9992	0.9980	0.9988	0.9976	0.9984
12	0.9980	0.9968	0.9988	0.9992	0.9992	0.9992	0.9988	0.9980	0.9985
13	0.9988	0.9980	0.9992	0.9988	1.0000	0.9984	0.9984	0.9972	0.9986
14	0.9984	0.9992	0.9992	0.9972	1.0000	1.0000	0.9980	0.9968	0.9986
15	0.9976	0.9976	0.9988	0.9968	0.9988	0.9984	0.9972	0.9980	0.9979
16	0.9980	0.9976	0.9980	0.9964	0.9988	0.9980	0.9980	0.9984	0.9979
17	0.9984	0.9984	0.9988	0.9984	0.9996	0.9984	0.9980	0.9968	0.9984
18	0.9984	0.9968	0.9992	0.9988	0.9984	0.9984	0.9984	0.9968	0.9982
19	1.0000	0.9996	0.9980	0.9968	0.9996	0.9976	0.9968	0.9972	0.9982
20	0.9988	0.9988	0.9980	0.9972	0.9988	0.9988	0.9988	0.9988	0.9985
21	0.9988	0.9972	0.9988	0.9992	0.9996	0.9988	0.9980	0.9972	0.9985
22	0.9976	0.9992	0.9984	0.9984	0.9992	0.9976	0.9980	0.9980	0.9983
23	0.9984	0.9976	0.9988	0.9992	0.9996	0.9984	0.9988	0.9980	0.9986
24	0.9968	0.9976	0.9984	0.9984	0.9992	0.9964	0.9996	0.9980	0.9981
25	0.9984	0.9964	0.9980	0.9980	0.9980	0.9976	0.9996	0.9996	0.9982
26	0.9988	0.9988	0.9980	0.9980	0.9992	0.9980	0.9992	0.9996	0.9987
27	0.9984	0.9976	0.9988	0.9972	0.9988	0.9996	0.9984	0.9976	0.9983
28	0.9984	0.9972	0.9988	0.9972	0.9992	0.9992	0.9988	0.9980	0.9984
29	0.9976	0.9956	0.9984	0.9988	0.9984	0.9980	0.9988	0.9980	0.9980
30	0.9972	0.9988	0.9984	0.9992	0.9988	0.9988	0.9988	0.9968	0.9984
31	0.9984	0.9988	0.9992	0.9984	0.9992	0.9984	0.9988	0.9980	0.9987
avg	0.9983	0.9978	0.9986	0.9980	0.9992	0.9982	0.9985	0.9976	**0.99829**

Third-order masked implementation with cyclic rotations

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	0.9968	0.9820	0.9964	0.9816	0.9976	0.9800	0.9984	0.9808	0.9892
1	0.9940	0.9924	0.9904	0.9892	0.9960	0.9908	0.9924	0.9912	0.9920
2	0.9952	0.9952	0.9960	0.9940	0.9960	0.9928	0.9984	0.9928	0.9950
3	0.9980	0.9944	0.9976	0.9924	0.9956	0.9936	0.9980	0.9960	0.9957
4	0.9984	0.9980	0.9968	0.9940	0.9968	0.9960	0.9980	0.9940	0.9965
5	0.9992	0.9952	0.9976	0.9976	0.9984	0.9952	0.9992	0.9964	0.9973
6	0.9972	0.9976	0.9976	0.9964	0.9980	0.9968	0.9984	0.9956	0.9972
7	0.9984	0.9948	0.9960	0.9956	0.9968	0.9956	0.9984	0.9976	0.9967
8	0.9988	0.9948	0.9976	0.9964	0.9996	0.9948	0.9988	0.9976	0.9973
9	0.9984	0.9932	0.9972	0.9968	0.9968	0.9960	0.9984	0.9952	0.9965
10	0.9984	0.9948	0.9980	0.9960	0.9972	0.9964	0.9988	0.9972	0.9971
11	0.9980	0.9972	0.9972	0.9980	0.9976	0.9964	0.9996	0.9972	0.9977
12	0.9976	0.9960	0.9968	0.9980	0.9968	0.9964	0.9992	0.9944	0.9969
13	0.9968	0.9956	0.9992	0.9960	0.9984	0.9936	0.9984	0.9972	0.9969
14	0.9980	0.9968	0.9984	0.9960	0.9972	0.9960	0.9980	0.9976	0.9973
15	0.9988	0.9964	0.9976	0.9964	0.9976	0.9944	0.9984	0.9980	0.9972
16	0.9984	0.9964	0.9980	0.9960	0.9960	0.9944	0.9980	0.9960	0.9967
17	0.9980	0.9964	0.9984	0.9940	0.9968	0.9980	0.9964	0.9972	0.9969
18	0.9972	0.9952	0.9964	0.9964	0.9976	0.9944	0.9984	0.9964	0.9965
19	0.9984	0.9984	0.9980	0.9956	0.9968	0.9960	0.9984	0.9972	0.9974
20	0.9988	0.9972	0.9968	0.9968	0.9968	0.9908	0.9984	0.9976	0.9967
21	0.9980	0.9980	0.9968	0.9940	0.9964	0.9976	0.9976	0.9972	0.9970
22	0.9988	0.9972	0.9960	0.9948	0.9944	0.9960	0.9976	0.9948	0.9962
23	0.9976	0.9960	0.9960	0.9944	0.9972	0.9948	0.9976	0.9964	0.9963
24	0.9976	0.9984	0.9984	0.9964	0.9980	0.9948	0.9964	0.9968	0.9971
25	0.9992	0.9968	0.9980	0.9964	0.9980	0.9948	0.9984	0.9972	0.9973
26	0.9972	0.9976	0.9968	0.9972	0.9972	0.9920	0.9980	0.9968	0.9966
27	0.9984	0.9944	0.9980	0.9952	0.9964	0.9948	0.9984	0.9972	0.9966
28	0.9988	0.9984	0.9976	0.9952	0.9980	0.9960	0.9976	0.9952	0.9971
29	0.9968	0.9976	0.9972	0.9956	0.9968	0.9948	0.9960	0.9984	0.9967
30	0.9980	0.9976	0.9972	0.9948	0.9968	0.9944	0.9976	0.9980	0.9968
31	0.9976	0.9952	0.9968	0.9968	0.9968	0.9944	0.9988	0.9948	0.9964
avg	0.9978	0.9958	0.9971	0.9951	0.9971	0.9945	0.9979	0.9958	**0.99639**

Fourth-order masked implementation with cyclic rotations

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	0.9844	0.9164	0.9856	0.9060	0.9804	0.9152	0.9812	0.9196	0.9486
1	0.9936	0.9516	0.9920	0.9516	0.9956	0.9416	0.9960	0.9496	0.9714
2	0.9976	0.9776	0.9956	0.9784	0.9948	0.9792	0.9956	0.9744	0.9866
3	0.9968	0.9708	0.9952	0.9744	0.9944	0.9692	0.9936	0.9696	0.9830
4	0.9944	0.9776	0.9932	0.9788	0.9952	0.9808	0.9932	0.9748	0.9860
5	0.9940	0.9808	0.9960	0.9808	0.9940	0.9840	0.9956	0.9772	0.9878
6	0.9948	0.9736	0.9952	0.9808	0.9936	0.9804	0.9936	0.9808	0.9866
7	0.9944	0.9796	0.9960	0.9788	0.9964	0.9824	0.9960	0.9828	0.9883
8	0.9936	0.9864	0.9964	0.9792	0.9948	0.9780	0.9976	0.9816	0.9885
9	0.9960	0.9796	0.9956	0.9812	0.9964	0.9776	0.9952	0.9820	0.9879
10	0.9956	0.9796	0.9960	0.9792	0.9960	0.9788	0.9928	0.9820	0.9875
11	0.9944	0.9804	0.9972	0.9808	0.9948	0.9848	0.9952	0.9752	0.9878
12	0.9952	0.9816	0.9948	0.9732	0.9968	0.9784	0.9928	0.9796	0.9866
13	0.9948	0.9804	0.9968	0.9816	0.9948	0.9792	0.9964	0.9796	0.9879
14	0.9964	0.9812	0.9964	0.9816	0.9944	0.9756	0.9944	0.9844	0.9880
15	0.9964	0.9812	0.9944	0.9756	0.9932	0.9776	0.9952	0.9784	0.9865
16	0.9960	0.9800	0.9972	0.9832	0.9960	0.9752	0.9924	0.9800	0.9875
17	0.9944	0.9844	0.9948	0.9868	0.9940	0.9772	0.9940	0.9836	0.9887
18	0.9948	0.9800	0.9944	0.9812	0.9968	0.9756	0.9952	0.9796	0.9872
19	0.9972	0.9828	0.9964	0.9764	0.9968	0.9800	0.9956	0.9756	0.9876
20	0.9948	0.9840	0.9956	0.9844	0.9944	0.9792	0.9956	0.9816	0.9887
21	0.9944	0.9824	0.9948	0.9764	0.9956	0.9792	0.9968	0.9812	0.9876
22	0.9960	0.9784	0.9932	0.9848	0.9928	0.9788	0.9972	0.9788	0.9875
23	0.9956	0.9788	0.9944	0.9796	0.9956	0.9816	0.9980	0.9792	0.9879
24	0.9964	0.9824	0.9956	0.9828	0.9960	0.9748	0.9952	0.9812	0.9881
25	0.9944	0.9824	0.9952	0.9788	0.9980	0.9836	0.9948	0.9824	0.9887
26	0.9928	0.9856	0.9956	0.9760	0.9936	0.9828	0.9968	0.9776	0.9876
27	0.9928	0.9800	0.9968	0.9812	0.9956	0.9816	0.9972	0.9804	0.9882
28	0.9940	0.9804	0.9940	0.9772	0.9944	0.9856	0.9956	0.9840	0.9881
29	0.9956	0.9792	0.9960	0.9760	0.9956	0.9860	0.9952	0.9832	0.9883
30	0.9948	0.9792	0.9960	0.9784	0.9936	0.9796	0.9924	0.9868	0.9876
31	0.9952	0.9820	0.9968	0.9780	0.9964	0.9780	0.9952	0.9812	0.9878
avg	0.9947	0.9775	0.9951	0.9764	0.9947	0.9763	0.9947	0.9771	**0.98582**

Fifth-order masked implementation with cyclic rotations

Byte	Bit position in byte								**avg**
	0	1	2	3	4	5	6	7	
0	0.9864	0.8524	0.9836	0.8488	0.9840	0.8672	0.9856	0.8520	0.9200
1	0.9816	0.9160	0.9832	0.9152	0.9824	0.9252	0.9828	0.9216	0.9510
2	0.9900	0.9520	0.9916	0.9608	0.9892	0.9572	0.9912	0.9524	0.9731
3	0.9956	0.9684	0.9952	0.9640	0.9916	0.9700	0.9952	0.9696	0.9812
4	0.9912	0.9740	0.9904	0.9708	0.9932	0.9676	0.9940	0.9756	0.9821
5	0.9924	0.9708	0.9924	0.9720	0.9928	0.9684	0.9904	0.9728	0.9815
6	0.9912	0.9752	0.9916	0.9692	0.9892	0.9740	0.9908	0.9732	0.9818
7	0.9920	0.9740	0.9888	0.9720	0.9916	0.9744	0.9932	0.9772	0.9829
8	0.9960	0.9828	0.9916	0.9684	0.9920	0.9720	0.9944	0.9732	0.9838
9	0.9924	0.9688	0.9892	0.9732	0.9932	0.9692	0.9932	0.9724	0.9814
10	0.9936	0.9736	0.9932	0.9732	0.9940	0.9744	0.9960	0.9708	0.9836
11	0.9924	0.9792	0.9956	0.9692	0.9936	0.9740	0.9952	0.9720	0.9839
12	0.9932	0.9760	0.9900	0.9716	0.9964	0.9752	0.9956	0.9724	0.9838
13	0.9924	0.9720	0.9932	0.9756	0.9944	0.9764	0.9932	0.9692	0.9833
14	0.9944	0.9760	0.9940	0.9720	0.9908	0.9720	0.9944	0.9672	0.9826
15	0.9912	0.9716	0.9876	0.9744	0.9928	0.9792	0.9964	0.9720	0.9831
16	0.9924	0.9772	0.9916	0.9740	0.9924	0.9740	0.9976	0.9736	0.9841
17	0.9904	0.9732	0.9936	0.9676	0.9948	0.9736	0.9944	0.9772	0.9831
18	0.9912	0.9724	0.9908	0.9760	0.9916	0.9724	0.9956	0.9752	0.9832
19	0.9936	0.9736	0.9920	0.9672	0.9908	0.9736	0.9936	0.9788	0.9829
20	0.9932	0.9764	0.9948	0.9756	0.9952	0.9676	0.9952	0.9728	0.9839
21	0.9932	0.9736	0.9944	0.9772	0.9952	0.9692	0.9960	0.9756	0.9843
22	0.9940	0.9780	0.9936	0.9704	0.9928	0.9720	0.9944	0.9740	0.9837
23	0.9952	0.9740	0.9912	0.9724	0.9944	0.9748	0.9940	0.9728	0.9836
24	0.9932	0.9744	0.9928	0.9716	0.9920	0.9744	0.9960	0.9752	0.9837
25	0.9920	0.9736	0.9928	0.9720	0.9940	0.9768	0.9960	0.9736	0.9839
26	0.9932	0.9764	0.9944	0.9696	0.9920	0.9712	0.9940	0.9780	0.9836
27	0.9928	0.9772	0.9940	0.9760	0.9940	0.9716	0.9956	0.9728	0.9842
28	0.9960	0.9768	0.9920	0.9756	0.9920	0.9736	0.9944	0.9716	0.9840
29	0.9944	0.9744	0.9924	0.9768	0.9932	0.9800	0.9916	0.9748	0.9847
30	0.9952	0.9748	0.9920	0.9720	0.9944	0.9736	0.9948	0.9768	0.9842
31	0.9920	0.9728	0.9952	0.9720	0.9932	0.9704	0.9948	0.9680	0.9823
avg	0.9924	0.9682	0.9918	0.9661	0.9923	0.9677	0.9937	0.9673	**0.97995**