INDIAN STATISTICAL INSTITUTE, KOLKATA

R. C. BOSE CENTRE FOR CRYPTOLOGY AND SECURITY

KATHOLIEKE UNIVERSITEIT LEUVEN

COMPUTER SECURITY AND INDUSTRIAL CRYPTOGRAPHY

Master's Thesis Report

by Tamas Kanti Garai (CrS2116)

SCA Resistant Implementation of Post-Quantum Scheme: CRYSTALS - Kyber

Advisors: Prof. Dr. Ingrid Verbauwhede & Prof. Dr. Bimal Kumar Roy
Daily Supervisors: Suparna Kundu, Angshuman Karmakar & John Gaspoz

July, 2023

Acknowledgements

I wish to express my sincerest gratitude to Prof. Ingrid Verbauwhede and Prof. Bimal Kumar Roy to give me an opportunity to work in this thesis and also to my daily supervisors Suparna Kundu, Angshuman Karmakar and John Gaspoz for their continuous guidance and mentorship that they provided me during the project. They showed me the path to achieve my targets by explaining all the tasks to be done and explained to me the importance of this project. They were always ready to help me and clear my doubts regarding any hurdles in this project. Without their constant support and motivation, this project would not have been successful.

Place: KU Leuven, Belgium Tamas Kanti Garai
Date: 04.07.2023

Declaration

I hereby declare that the project entitled "SCA Resistant Implementation of Post-Quantum Scheme: CRYSTALS - Kyber" submitted in partial fulfillment for the award of the degree of Master of Technology in Cryptology and Security completed under the supervision of Prof. Dr. Ingrid Verbauwhede and Prof. Dr. Bimal Kumar Roy, at ISI Kolkata is an authentic work. Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Place: KU Leuven, Belgium Tamas Kanti Garai
Date: 04.07.2023

Abstract

Masking and shuffling are two well-known countermeasures against *Side-Channel Attacks (SCA)* on cryptographic schemes. But masking causes huge performance and resource overhead. Whereas Shuffling is less expensive yet an effective countermeasure. It uses a random permutation to effectively scramble the order of operations inside a loop. In the year 2022, *National Institute of Standards and Technology (NIST)* has selected CRYSTALS-Kyber as a new standardized post-quantum public-key encryption and key-encapsulation scheme. In the meantime some studies have already been done on Kyber's side-channel security. Some papers have been published on first- and higher-order masking [10, 4] on Kyber. But, no proper study on implementing shuffling on Kyber has been done yet. But recently an attack by Backlund et al. in [2] has been done on combined masked-shuffled implementation of Kyber. They have designed the attack on self-implemented shuffling on Kyber, taking the already published first-order masked implementation of Kyber in mkm4 project [10] as a baseline. In this paper, we have implemented shuffling on a specific variant of Kyber (Kyber768) from the pqm4 project [12], for an ARM Cortex-M4 microcontroller. Our shuffled implementation, taking $873,757$ CPU cycles, features a 1.1x overhead factor over the implementation in the pqm4 project. We have checked the leakage evaluation of our shuffled implementation component-wise. Then we combined our shuffled implementation with the first-order masked version of Kyber768 presented in mkm4 project [10]. Our combined masked-shuffled implementation, taking $3,081,293$ CPU cycles, features a 3.86x overhead factor over the implementation of Kyber768 in the pqm4 project.

Table of Contents

Chapter 1

INTRODUCTION

In recent years, seeing the uninterrupted advancement in the construction of quantum computer, scientists and researchers have realized the importance of post-quantum cryptography. Due to Shor's polynomial-time [26] algorithm for solving prime factorization problem and discrete logarithm problem, currently employed public key schemes like RSA [24] and elliptic curve Diffie-Hellman schemes [7] are no more a safe option against large-scale quantum computers. Consequently, in the year 2016, the National Institute of Standard and Technology (NIST) [27] has initiated a process to develop and standardize one or more public-key algorithms that are quantum-resistant.

In the selection process of NIST [21], CRYSTALS - Kyber [3] is selected as a standard post-quantum public-key encryption and key-establishment algorithm in the year 2022. The security of Kyber is based on ring-learning with error (RLWE), which is reduced to the shortest vector problem (SVP) and α-SVP of the lattice. Since no polynomial-time algorithm to solve the SVP has been evolved till date, it is believed that the public key cryptography based on this hard problem will be resistant to quantum computers also.

In the year 1995, Kocher introduced in [14], a new kind of attack on the implementations of Diffie-Hellman, RSA, DSS, and other Systems. This attack was named as timing attack since it is done by carefully measuring the time required for each operation related to the secret keys of those schemes. Nowadays every cryptographic protocol is implemented in constant-time to prevent this type of attacks. Again in 1999. Kocher et al. have introduced in [15] another attack that also does not target the mathematical security directly like timing attack. This attack was named as power analysis attack as this attack recovers secret keys by analyzing the power consumption of

the target device. To prevent this specific attack, only constant-time implementation of cryptographic schemes is not enough.

Due to these kinds of already published successful attacks, nowadays secure implementation of post-quantum schemes is as important as mathematical security of the schemes. So after standardizing post-quantum schemes, the remaining task for the researchers is to make the standardized quantum schemes side-channel attack (SCA) resistant.

One very famous provably-secure countermeasure against these kinds of SCA is masking [6]. But masked implementations of quantum schemes have huge overhead in run-time. There are already some published works on the masked implementation of KYBER. In [10], first-order masked kyber on ARM cortex-M4 has been implemented. But first-order masking cannot resist higher-order side-channel leakages and higher-order masking increases the complexity of a scheme exponentially.

Shuffling is another countermeasure against SCA, which is not as expensive as masking. One idea is to combine masking with shuffling in the implementation of KYBER instead of using higher-order masking in the implementation.

1.1 Motivation

Although there are already some published papers on the masked implementation of KYBER, both first-order [10] and higher-order [4]; there is no published work on the shuffling of KYBER. Some attacks (e.g., [2]) has also been published on combined masked-shuffled implementation of KYBER with self-implemented shuffling. There should be some study on the shuffled implementation of KYBER. In this thesis, we have tried to do that. Combining shuffling with the masked implementation of KYBER can give some improvement in security against SCA with very less overhead in run-time. We tried that aspect also in this work.

1.2 Our Contribution

This work is focusing on implementing shuffling on KYBER and testing the effect of shuffling on the run-time and security of the scheme. During this short period of six months, we have done two things:

- At first we have implemented shuffling on the Kyber768, taking the implementation of the pqm4 [12] project as baseline. We checked the overhead and improvement in the security of our shuffled implementation compared to the plain implementation.
- Then we have combined our shuffling implementation on the mkm4 [10] project and checked the overhead in run-time of this combined implementation.

1.3 Thesis Outline

The topics of each chapter in this thesis are briefly described below:

- **Chapter 2 :** The second chapter consists of the prerequisites of this thesis. Here we have briefly described the scheme Kyber, the idea behind masking and shuffling. Later some published attacks on Kyber are also mentioned in this section.
- **Chapter 3 :** In the third chapter we have explained how we implemented shuffling countermeasures explained in chapter 2 in different components of Kyber.
- **Chapter 4 :** This chapter consist of the measurements of run-time of different components of plain, shuffled and combined masked-shuffled implementation of Kyber and their comparisons. We also present detailed evaluation of the improvement in leakages in the modified implementation in this chapter.

Chapter 2

PRELIMINARIES

2.1 Kyber Key Encapsulation Mechanism

For later reference the public key encryption scheme for KYBER:

KYBER.CPA = (KYBER.CPA.GEN, KYBER.CPA.ENC, KYBER.CPA.DEC)

in Algorithm 1, 2 and 3 are provided respectively, where the function definitions are as follows:

$$\text{COMPRESS}_q(x, d) \equiv \lceil (2^d/q).x \rfloor mod(2^d)$$

$$\text{DECOMPRESS}_q(x, d) \equiv \lceil (q/2^d).x \rfloor$$

$\chi_{n,\eta}$ be the distribution of polynomials of degree n with entries independently sampled from the centered binomial distribution χ_η with support $\{-\eta, ..., \eta\}$.

Let k, d_t, d_u, d_v be positive interger parameters, $n = 256$. Let $\mathcal{M} = \{0,1\}^{256}$ denote the massage space, where every maessage $m \in \mathcal{M}$ can be viewed as a polynomial in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with coefficients in $\{0, 1\}$. Here ciphertext are of the form $(\mathbf{u}, v) \in \{0,1\}^{256.kd_u} \times \{0,1\}^{256.d_v}$. Parameter sets for different variants of KYBER is mentioned in table 2.1.

To obtain a CCA2-secure KEM from the CPA-secure building blocks, KYBER uses a tweaked version of the Fujisaki-Okamoto transform [11]. The algorithms of the KYBER KEM are provided in algorithm 4, 5 and 6.

Algorithm 1 KYBER.CPA.GEN[10]

1: $(\rho, \sigma) \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$;
2: $\mathbf{A} \xleftarrow{\rho} U(q)^{k \times k}$;
3: $(s, e) \xleftarrow{\sigma} \chi^k_{n,\eta_1} \times \chi^k_{n,\eta_1}$;
4: $\hat{\mathbf{s}} \leftarrow NTT(\mathbf{s})$;
5: $\hat{\mathbf{e}} \leftarrow NTT(\mathbf{e})$;
6: $\hat{\mathbf{t}} \leftarrow \mathbf{A} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$;
7: **return** $pk_{CPA} := (\hat{\mathbf{t}}, \rho), sk_{CPA} := \hat{\mathbf{s}}$

Algorithm 2 KYBER.CPA.ENC[10]

Input: $pk_{CPA} = (\hat{\mathbf{t}}, \rho), m \in \mathcal{M}, r \xleftarrow{\$} \{0,1\}^{256}$
1: $\mathbf{A} \xleftarrow{\rho} U(q)^{k \times k}$;
2: $(\mathbf{r}, \mathbf{e_1}, \mathbf{e_2}) \xleftarrow{r} \chi^k_{n,\eta_1} \times \chi^k_{n,\eta_2} \times \chi_{n,\eta_2}$;
3: $\hat{\mathbf{r}} \leftarrow NTT(\mathbf{r})$;
4: $\mathbf{u} \leftarrow INTT(\mathbf{A} \circ \hat{\mathbf{r}}) + \mathbf{e_1}$;
5: $\mathbf{v} \leftarrow INTT(\hat{\mathbf{t}} \circ \hat{\mathbf{r}}) + \mathbf{e_2} + \lceil \frac{q}{2} \rfloor .m$;
6: $c_1 \leftarrow \text{COMPRESS}_q(\mathbf{u}, d_u)$;
7: $c_2 \leftarrow \text{COMPRESS}_q(\mathbf{v}, d_v)$;
8: **return** $c := (c_1, c_2)$

Algorithm 3 KYBER.CPA.DEC[10]

Input: $sk_{CPA} = \hat{\mathbf{s}}, c = (\mathbf{u}, \mathbf{v})$
1: $\mathbf{u} \leftarrow \text{DECOMPRESS}_q(\mathbf{u}, d_u)$;
2: $\mathbf{v} \leftarrow \text{DECOMPRESS}_q(\mathbf{v}, d_v)$;
3: **return** $m = \text{COMPRESS}_q(\mathbf{v} - INTT(\hat{\mathbf{s}} \circ NTT(\mathbf{u})), 1)$

Algorithm 4 KYBER.CCAKEM.GEN[10]

1: $z \xleftarrow{\$} \{0,1\}^{256}$;
2: $(pk, sk') = \text{KYBER.CPA.GEN}()$;
3: $sk := (sk'||pk||H(pk)||z)$;
4: **return** pk, sk

Algorithm 5 KYBER.CCAKEM.ENCAPS[10]

Input: pk

$M \xleftarrow{\$} \{0,1\}^{256}$;
$M \leftarrow H(m)$;
$(\bar{K}, r) := G(m||H(pk))$;
$c := \text{KYBER.CPA.ENC}(pk, m, r)$;
$K := KDF(\bar{K}||H(c))$;
return c, k

Algorithm 6 KYBER.CCAKEM.DECAPS[10]

Input: c, sk

Extract $(sk'||pk||H(pk)||z)$ from sk;
$m' =:= \text{KYBER.CPA.DEC}(sk', c)$;
$(\bar{K}', r') := G(m'||H(pk))$;
$c' := \text{KYBER.CPA.ENC}(pk, m', r')$;
if $c = c'$ **then**
    $K := KDF(\bar{K}'||H(c))$;
else
    $K := KDF(z||H(c))$;
end if
return K

KYBER variant	n	k	q	η_1	η_2	(d_u, d_v)	δ
KYBER512	256	2	3329	3	2	$(10, 4)$	2^{-139}
KYBER768	256	3	3329	2	2	$(10, 4)$	2^{-164}
KYBER1024	256	4	3329	2	2	$(11, 5)$	2^{-174}

Table 2.1: Parameter sets for different variants of KYBER [1]

2.2 Masking

In masking countermeasure we split every sensitive variable M into $d+1$ shares $M_0, M_1, ..., M_d$ randomly so that $M = M_0 * M_1 * ... * M_d$, where $*$ is a group operation. $M_1, M_2, ..., M_d$, called the *masks*, are usually chosen randomly and M_0, called the *masked variable*[16], is processed in such a way that the relation mentioned is satisfied. The parameter d is called the *masking order*. A d order masked implementation can withstand an DPA exploiting upto d leakage signals simultaneously. Although attacks exploiting more than d leakages are always possible theoretically, but practically complexity of those attacks grows exponentially with d.

There are two kinds of masking: Boolean masking, that uses XOR-operation to add up the shares to get the original variable and arithmetic masking, that uses modular arithmetic addition to add up the shares to get the original variable.

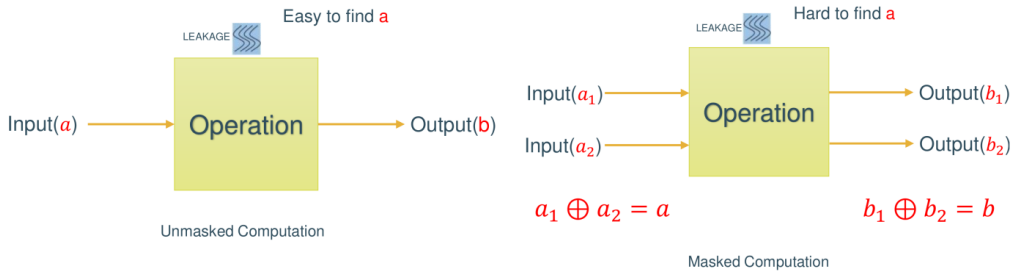


Figure 2.1: Masked vs Unmasked Computation

In masked implementation, as shown in figure 2.1, due to the randomness of the masks and execution all the computations inside an algorithm separately on each of the masks, it is hard for an adversary to find the value of the sensitive variable, from the side-channel leakage. But when the masking order is higher, there is too much overhead in the implementation due to the repetition of same computation in each of the masks. For that reason shuffling together with first order masking is preferable in the implementation.

2.3 Shuffling

In shuffling, at first a random permutation is generated and later this permutation is used to scramble up the order of computation. This can misguide the attacker about the order of the operations, the correlation between time and execution of a certain instruction. Applying shuffling is very straight forward and it is usually significantly less expensive than higher-order masking.

In figure 2.2, suppose $m_0, m_1, ..., m_7$ is an array of output after the execution of some independent operation on each variable. In case of shuffling, the sequence of outputting the array element after the execution of the independent operation is reordered. As a result if the permutation, which is used to reorder the sequence of the output array, is kept secret then from side channel leakage it is impossible to map the power trace peaks to specific array element.

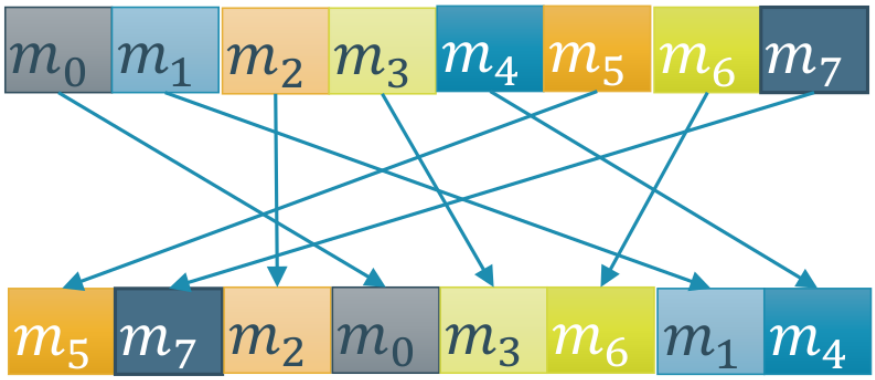


Figure 2.2: Shuffling the order of computation using a random permutation

2.4 Shuffling on NTT

Inside the Kyber algorithms, the NTT used are one of the easy target in the recently published attacks. There are some shuffling countermeasures on the NTT as well mentioed by Ravi et al. in [23].

2.4.1 Fine-Shuffled NTT

Each NTT layers contains a number of butterflies. Some Belief Propagation based attack on the NTT has targeted the SCA leakage from the loading of inputs of every butterfly. To directly counter that, fine shuffling [23] simply randomize the order of the input loads and output stores for each butterfly, as shown in 2.3. As a result an attacker cannot assign the leakage value to a specific input/output of a butterfly.

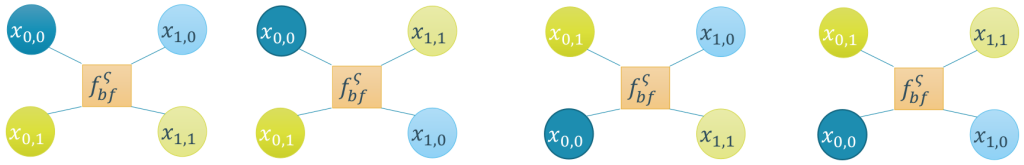


Figure 2.3: Four possible fine shuffled butterflies

2.4.2 Coarse-Full-Shuffled NTT

It is known that butterflies within a layer of the NTT can be computed independent of the one another. Coarse full shuffling [23] uses this advantage to permute the order of execution of the individual butterflies, as shown in figure 2.4, within each layer. However, the pair of coefficient of a single butterfly has to be processed together, so loads and stores of a single butterfly are processed in a consecutive order. With n coefficients, this permutes $(n/2)$ individual butterflies, resulting an entropy of $(n/2)!$, which is beyond realistic brute-force for typical parameter sets.

2.4.3 Coarse-In-Group-Shuffled NTT

In a single layer, butterflies with the same twiddle factor ζ are referred to as group. Instead of performing a full layer shuffling for every stage, coarse-in-group shuffling [23] randomizes the order of computation of the butterflies within a group. For a layer with m butterfly groups, this results in $((\frac{n}{2m})!)^m$ permutations. For example, in the INTT of Kyber, with $n = 256$, the first layer consists of $m = 64$ groups, the entropy is $((\frac{n}{2m})!)^m = 2^{64}$. For the last layer, this reduces to a single group, increasing the entropy to 128!. Figure 2.5 shows a coarse-in-group-shuffled sub-graph.

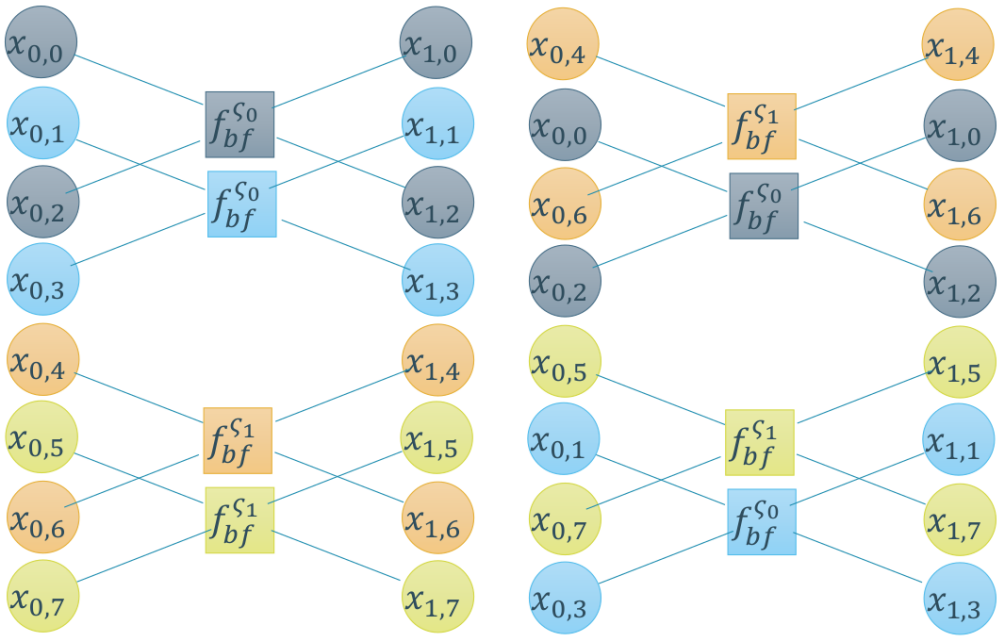


Figure 2.4: Example of coarse full shuffling

2.5 Some Attacks on Kyber

2.5.1 Neural Network Based Attacks

In the paper [18], Ngo et al. has attacked a masked implementation of Saber, which is a post quantum scheme very much similar to Kyber. They have attacked the decapsulation device to recover the long-term secret key as well as the session key. Later in [17] and [2] the attacks has been improved and generalized so that it can be done on masked and shuffled Kyber also.

In [17], they have attacked on the Decoder of the decapsulation device. During decapsulation of both Saber and Kyber, a given ciphertext is decrypted at first and then it is re-encrypted and checked if the newly generated ciphertext matches with the original ciphertext or not. If the two match, then the decrypted message is used to generate the common key otherwise a randomly sampled element is used to generate the key. In the decryption part, just before the message is output, a decoder is used to give the message bit as output in a packed fashion. This decoder processes the message array

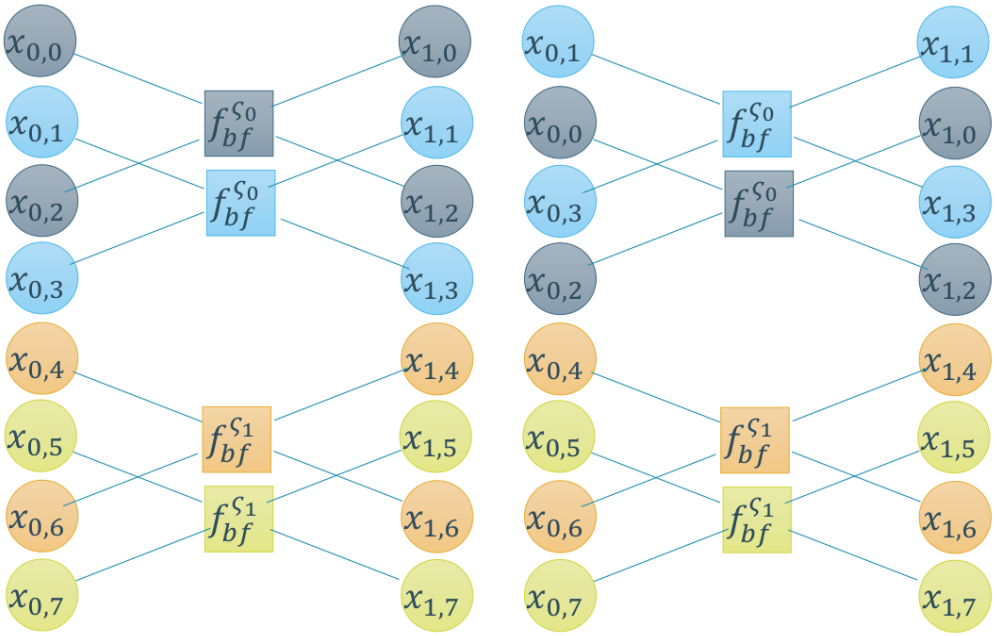


Figure 2.5: Example of coarse-in-group shuffling

in a bit by bit fashion. As a result the power consumption of the decapsulation device during the execution of this decoder leaks a lot of information about the message bits.

In the above mentioned attack, a neural network has been trained with millions of power traces of one bit processing by the decoder. After the training the neural network can distinguish between the message bit 0 and 1 just by looking at the power trace of that bit's processing by the decoder. This way the whole message can be recovered. Masking has no effect in this attack since in case of masking the decoder function processes the masked share of each message bit together, so power trace corresponding to all the shares corresponding to a single message bit is used for the training of neural network.

Once the message corresponding to a certain number of chosen ciphertexts is recovered, from the relation between the message, ciphertext and key, the key is recovered. The same attack works for the shuffled case also. By taking the Hamming weight of messages corresponding to chosen ciphertexts and using bit flip technique, one can recover the messages in the shuffled case

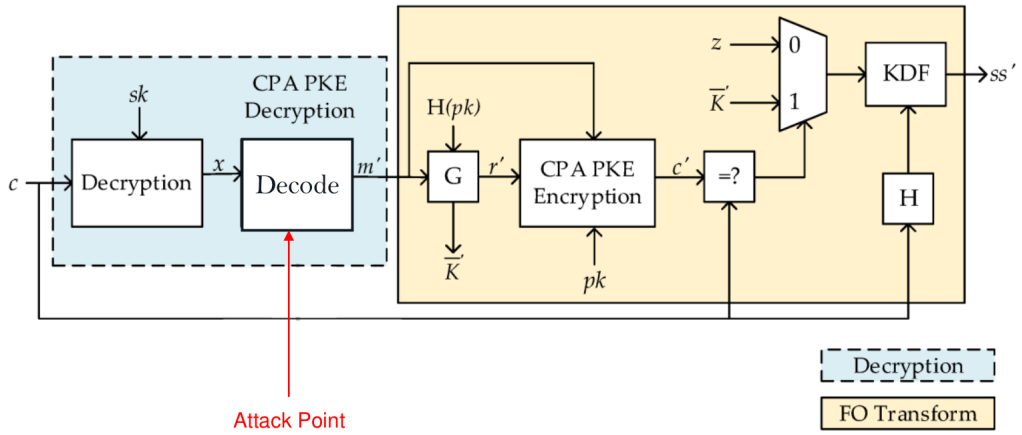


Figure 2.6: Attack point in the Decapsulation of Kyber (diagram taken from [5])

also.

The attack process in the shuffled case is more complicated than the unmasked and masked both the cases. So shuffling is clearly giving more security in this case than plain masking.

2.5.2 Belief Propagation Based Attacks

In the implementation of Kyber KEM, for efficient polynomial multiplication number theoretic transformation (NTT) is used and the input of these NTT and inverse NTT are sometimes related to sensitive variables like the long term secret or the session keys. In [22], [20] and [9] these NTT/INTT has been targeted. In all the attacks, belief propagation has been used to attack the butterflies which are the building block of NTT/INTT.

The butterflies inside NTT/INTT are comprised of many potentially leaking modular operations like modular addition multiplication etc. Power traces during the execution of these operation is used for template matching. At first factor graphs are designed for the belief propagation. The initial distribution of the nodes of these factor graphs are obtained by template matching from millions of templates of those butterflies with different set of inputs. Once the template matching is done, some iteration of belief propagation is executed. After some iteration of the BP, the input to the NTT/INTT

can be obtained if the the coefficients of the input polynomials are chosen from a set of small support or if the input polynomials are coarse.

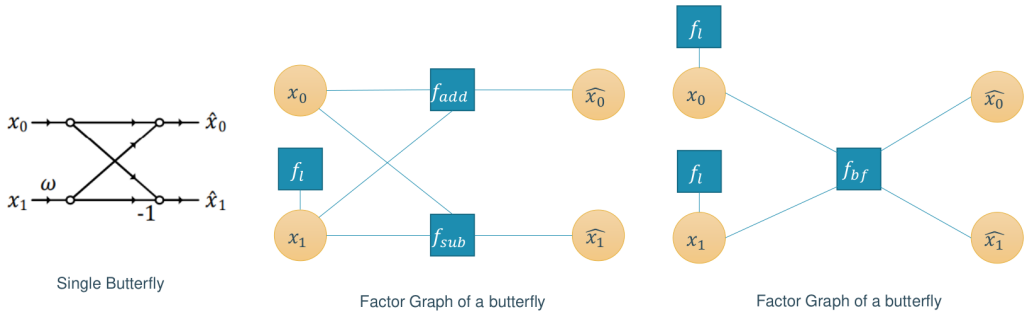


Figure 2.7: Butterflies and Corresponding Factor Graphs used in many attacks

For example, in figure 2.8, the input to the red marked NTT is the polynomial r, whose coefficients are chosen using centered binomial distribution. As a result the coefficients of r are from a set of small support. So this attack can be successfully done on this NTT. In [22], this NTT has been attacked.

In [20], an attack on the NTT inside the key generation of Kyber is mentioned. The key generation contains an NTT whose input is the long-term secret key (figure 2.9) which is polynomial whose coefficient are also sampled from a centered binomial distribution. So similar attack can be done on this also.

In line 2 of the decapsulation algorithm of Kyber shown in algorithm 6, KYBER.CPA.DEC is called. There is an inverse NTT inside KYBER.CPA.DEC whose input involves with the secret key and ciphertext. If the ciphertext can be chosen in such a way that this input to the INTT become coarse then this INTT can also be exploited by the similar BP based attack. In fact in [9], this approach is taken to extract the input to the mentioned INTT, and from the input the secret key is recovered using some look up table and Brute Force search on a small sample.

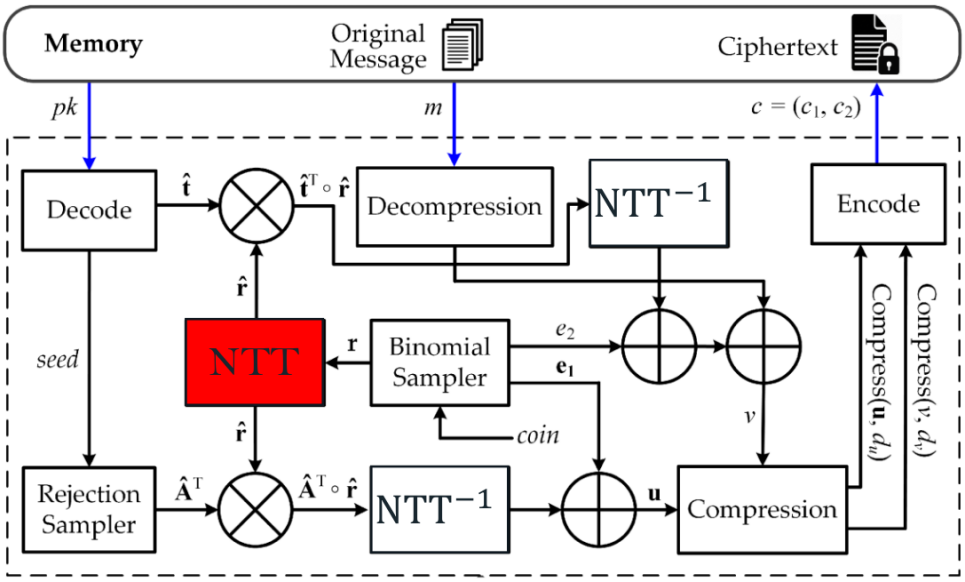


Figure 2.8: Attack point in the Encryption of Kyber (diagram taken from [19])

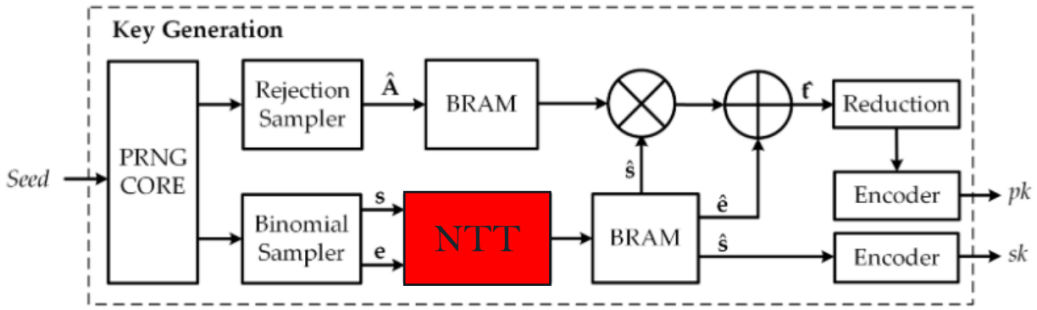


Figure 2.9: Attack point in the Key Generation of Kyber (diagram taken from [19])

Chapter 3

IMPLEMENTATION

In this section, we will discuss the implementation of shuffling in different components of the Kyber decapsulation algorithm. For the implementation, we first need a permutation generator to generate a random permutation. Also, the implementation of the permutation generator has to be constant time so that we can do a t-test on the shuffled components. We will use this permutation generator whenever we need a random permutation for shuffling throughout the code. Inside the Kyber decapsulation algorithm, there are two major parts where we apply shuffling, one is the CPA-secure decryption part and the other is the merged-up CPA-secure re-encryption and comparison of ciphertexts part.

3.1 Permutation Generation

The main building block of implementing shuffling is the permutation generator. Using the Fisher-Yates algorithm [13] (shown in algorithm 7) we can generate a random permutation of finite sequences in linear time. Given a finite sequence $S = \{s_0, s_1, ..., s_{n-1}\}$, the original algorithm iterates over every element s_i for i in $\{n-1, n-2, ..., 1\}$ and swaps it with a random element in the set $S_i = \{s_0, s_1, ..., s_i\}$. But sampling from $\{0, 1, ..., i\}$ will need an approach with probabilistic run-time or a modulo operation together with a random number generator. To generate the random number we used the *randombytes*() function that is already defined in the KYBER code present in the pqm4 library [12]. But the modulo operation cannot be avoided if we do not change the Fisher-Yates algorithm a little bit. Instead of sampling

from $\{0, ..., i\}$ if we sample from $\{0, ..., n-1\}$ then the modulo operation can be replaced by bit-wise AND operation whenever n is a power of 2. And in shuffling the decapsulation of KYBER, we only need random permutation of power of 2 sizes.

Algorithm 7 FISHERYATES[13]

Input: *size*
Output: *permutation*[*size*], an array of random permutation of size *size*

1: **for** i in 0 to $size - 1$ **do**
2: $permutation[i] = i$;
3: **end for**
4: **for** i from $size - 1$ down to 1 **do**
5: $k \leftarrow$ random integer, such that $0 \leq k \leq i$;
6: $temp = permutation[i]$;
7: $permutation[i] = permutation[k]$;
8: $permutation[k] = temp$;
9: **end for**
10: **return** *permutation*

Although this tweaked version of the Fisher-Yates shuffle (shown in algorithm 8) will generate all possible permutations but with a slight bias. In [28], it is shown that the bias caused by the tweaked version of the Fisher-Yates shuffle should not lead to a significant security reduction of the shuffling countermeasure.

3.2 Kyber Decryption

3.2.1 Decoder

Inside the decryption algorithm of Kyber, just before the plaintext is generated, there is a decoder (shown in figure 3.1) that packs the 256 message bits (each of them are coefficient of a polynomial) in a 32-byte array. This decoder is a good attack point for many of the published attacks. So we apply shuffling here. Since the decoder packs the message bits in a sequential manner, we shuffle the order of computation of each of the message bits using the permutation generated by the implemented permutation generator.

Algorithm 8 FisherYates_modified

Input: *size*
Output: *permutation*[*size*], an array of random permutation of size *size*

```
for i in 0 to size − 1 do
    permutation[i] = i;
end for
for i from size − 1 down to 1 do
    k ← random integer, such that 0 ≤ k ≤ size − 1;    ▷ modified line
    temp = permutation[i];
    permutation[i] = permutation[k];
    permutation[k] = temp;
end for
return permutation
```

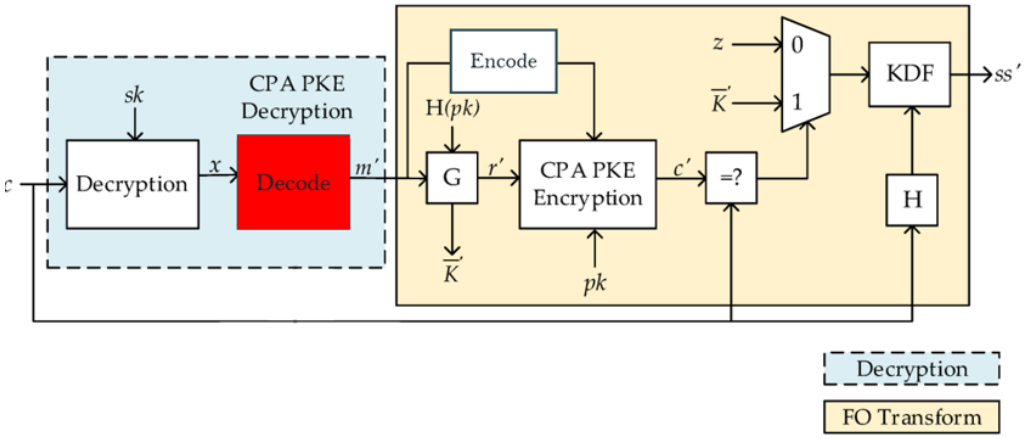


Figure 3.1: The decoder function inside the Kyber.CCAKEM.Decaps

3.2.2 Subtraction of Two Polynomials

In the decryption algorithm of Kyber, there is a subtraction of two polynomials involving secret key and ciphertext. As a result, this subtraction can be a target to secret key recovery attacks. We have shuffled the order of pointwise subtraction of polynomials using a random permutation generated from the tweaked version of Fisher-Yates algorithm.

Algorithm 9 `poly_tomsg`

Input: $poly[256]$
Output: $msg[32]$, 32-byte message
1: **for** i in 0 to 255 **do**
2: $x = i/8$;
3: $y = i\%8$;
4: y-th bit of $msg[x] = \text{Compress}_q(poly[i], 1)$;
5: **end for**
6: **return** msg

Algorithm 10 `poly_tomsg_shuffled`

Input: $poly[256]$
Output: $msg[32]$, 32-byte message
1: $permutation[256] = \text{FisherYates\_modified}(256)$;
2: **for** z in 0 to 255 **do**
3: $i = permutation[z]$;
4: $x = i/8$;
5: $y = i\%8$;
6: y-th bit of $msg[x] = \text{Compress}_q(poly[i], 1)$;
7: **end for**
8: **return** msg

Algorithm 11 `poly_sub_shuffled`

Input: $poly1[256], poly2[256]$
Output: $poly3[256]$, after component-wise adding $poly1[256]$ and $poly2[256]$
1: $permutation[256] = \text{FisherYates\_modified}(256)$;
2: **for** i in 0 to 255 **do**
3: $x = permutation[i]$;
4: $poly3[x] = poly1[x] - poly2[x]$;
5: **end for**
6: **return** $poly3$

3.3 Kyber Re-encryption & Compare

3.3.1 Encoder

Inside the encryption of Kyber, before generating the ciphertext, the 32-byte message has to be encoded into a polynomial with 256 coefficients. This

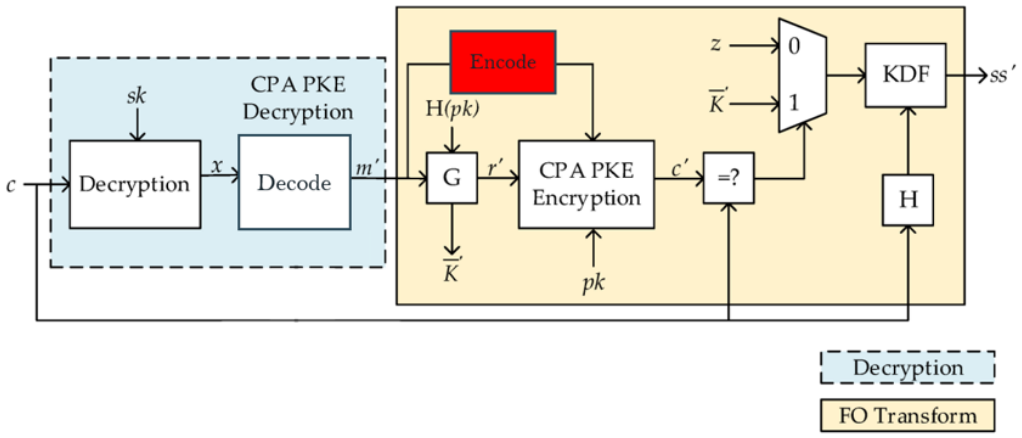


Figure 3.2: The encoder function inside the KYBER.CPA.ENC

encoder function (shown in figure 3.2) can be an easy target to message recovery attacks. So we reordered the generation of polynomial coefficients from the input message using a random permutation. The implementation of this part is exactly similar to the implementation of the decoder function mentioned above.

Algorithm 12 `poly_frommsg`

Input: $msg[32]$, 32-byte message
Output: $poly[256]$

1: **for** i in 0 to 255 **do**
2: $x = i/8$;
3: $y = i\%8$;
4: $poly[i] = \text{DECOMPRESS}_q(y\text{-th bit of } msg[x], 1)$;
5: **end for**
6: **return** $poly$

3.3.2 Addition of Two Polynomials

Inside the encryption algorithm of Kyber (in line 5 of algorithm 2), there is an addition involved with the message. As a result, some information about the message can be leaked during re-encryption. We implemented shuffling

Algorithm 13 `poly_frommsg_shuffled`

Input: $msg[32]$, 32-byte message
Output: $poly[256]$

$permutation[256] = \text{FisherYates\_modified}(256)$;
for z in 0 to 255 **do**
    $i = permutation[z]$;
    $x = i/8$;
    $y = i\%8$;
    $poly[i] = \text{Decompress}_q(y\text{-th bit of } msg[x], 1)$;
end for
return $poly$

to this component as well with a similar technique mentioned in subsection 3.2.2.

Algorithm 14 `poly_add_shuffled`

Input: $poly1[256], poly2[256]$
Output: $poly3[256]$, after component-wise adding $poly1[256]$ and $poly2[256]$

$permutation[256] = \text{FisherYates\_modified}(256)$;
for i in 0 to 255 **do**
    $x = permutation[i]$;
    $poly3[x] = poly1[x] + poly2[x]$;
end for
return $poly3$

3.3.3 Compress Function

In the line 7 of Kyber.CPA.Enc algorithm, there is a compress function (shown in figure 3.3), which is used on **v**. Since this **v** is directly related with the input message, message recovery attacks can exploit this compression. To make it resistant to message recovery attacks, we need to secure this compress function. So we apply shuffling here also.

The compress function takes a 256-coefficient polynomial as input. The function applies the Compress_q operation on each of the coefficients to make every co-efficient a 4-bit integer. Then packs every 2 coefficient into a single 8-bit variable.

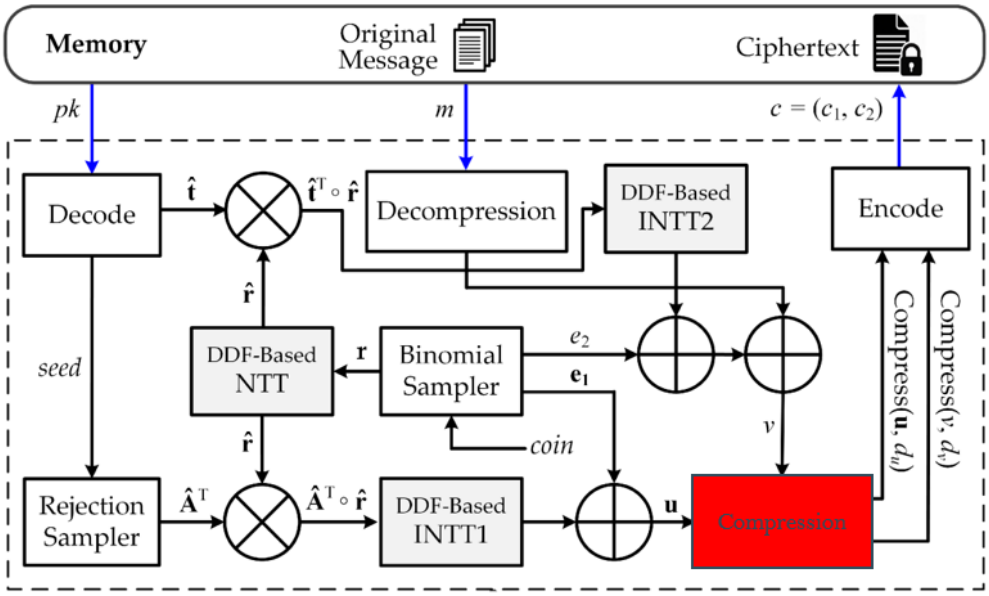


Figure 3.3: The compress function inside the KYBER.CPA.ENC

Algorithm 15 `poly_compress`

Input: *poly*[256]
Output: *compressed_poly*[128]

1: **for** i in 0 to 31 **do**
2: **for** j in 0 to 7 **do**
3: $t[j] = \text{COMPRESS}_q(poly[8i + j], 4)$;
4: **end for**
5: $compressed\_poly[k] = t[0] \mid t[1] << 4$;
6: $compressed\_poly[k + 1] = t[2] \mid t[3] << 4$;
7: $compressed\_poly[k + 2] = t[4] \mid t[5] << 4$;
8: $compressed\_poly[k + 3] = t[6] \mid t[7] << 4$;
9: $k = k + 4$;
10: **end for**
11: **return** *compressed_poly*

3.4 NTT & Inverse NTT

The belief propagation-based attacks mentioned in 2.5.2 are targeting the NTT/INTTs. There are already some published works on SCA countermea-

Algorithm 16 `poly_compress_shuffled`

Input: $poly[256]$
Output: $compressed\_poly[128]$

1: $permutation[32] = \text{FisherYates\_modified}(32)$;
2: **for** z in 0 to 31 **do**
3: $\quad i = permutation[z]$;
4: $\quad$ **for** j in 0 to 7 **do**
5: $\quad\quad t[j] = \text{Compress}_q(poly[8i + j], 4)$;
6: $\quad$ **end for**
7: $\quad compressed\_poly[k] = t[0] \mid t[1] << 4$;
8: $\quad compressed\_poly[k + 1] = t[2] \mid t[3] << 4$;
9: $\quad compressed\_poly[k + 2] = t[4] \mid t[5] << 4$;
10: $\quad compressed\_poly[k + 3] = t[6] \mid t[7] << 4$;
11: $\quad k = k + 4$;
12: **end for**
13: **return** $compressed\_poly$

sure on NTTs. In [23], Ravi et al. has implemented some shuffling countermeasures on NTT/INTTs described in 2.4. We tried to use those implementation in our code. But noticed that their implementation is not constant time, as a result it is not possible to do the t-test on the NTT/INTT parts of our shuffled implementation of Kyber. So we are now trying to make the implementation constant time so that we can include the shuffling on NTT into our code.

Chapter 4

MEASUREMENT

In this section we present the performance and leakage evaluation of our implementation. We have used the Kyber768 (parameter set mentioned in table 2.1) implementation in the PQM4 [12] project as baseline and modified the code to include the shuffling countermeasures mentioned in chapter 3. And for the masked and shuffled implementation we have taken the masked implementation of [10] of Kyber768 as baseline and modified the code to include the same shuffling countermeasures.

4.1 Performance

We measured the performance of our shuffled implementation of Kyber768 in STM32F407VG microcontroller with 32-bit ARM Cortex-M4 with FPU core that is mounted on the STM32F407VG Discovery board. The measurement setup is based on the PQM4 [12] project. For the compilation we used `arm-none-eabi-gcc` version `10.3.1` with compiler flags `-O3 -std=gnu99 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16`.

In table 4.1, the comparison in the CPU cycles of different components of the decapsulation of Kyber768 between plain implementation, shuffled implementation and masked & shuffled implementation has been shown. Our shuffled implementation and masked & shuffled implementation of decapsulation of Kyber768 only has overhead factors 1.1x and 3.86x respectively, over the plain implementation for CPU cycles. In the mkm4 project [10], the CPU cycles for the decapsulation in Kyber768 were $2,978,441$ with the similar set up and same microcontroller. So our shuffling implementation on

Operation	Cycles		
	Plain	Shuffled	Combined Masked-Shuffled
`crypto_kem_dec`	797,645(1.0x)	873,757(1.1x)	3,081,293(3.86x)
`indcpa_dec`	43,902	80,865	170,361
`indcpa_enc`	612,132	650,940	2,580,950
`indcpa_dec`	43,902	80,865	170,361
`poly_tomsg`	3,296	21,826	84,782
`poly_sub`	765	19,199	21,266
`indcpa_enc`	612,132	650,940	2,580,950
`poly_frommsg`	1,716	19,710	147,530
`poly_add`	765	19,199	38,370
`poly_compress`	2,889	5,127	347,430

Table 4.1: CPU cycles of different components of plain implementation, shuffled implementation and combined masked-shuffled implementation of Kyber768

the masked Kyber768 has added almost 100,000 CPU cycles, which is a 1.03x overhead factor only.

4.2 Leakage Evaluation

The leakage evaluation of our Kyber768 implementation was done in STM32-F415RG microcontroller. We have used a Tektronix DPO 70404C digital oscilloscope to collect instantaneous power measurements during executions of `crypto_kem_dec`. We have used the Test Vector Leakage Assessment (TVLA) methodology introduced by Goodwill et al. [8] to check the improvement in security with our shuffling implementation. In our experiments we have used fixed vs. random t-test as it is proposed in [25]. The fix class, $\mathcal{Q}_0$, contains the power traces obtained when the algorithm's input is a fixed secret key sk_{fix}, while random class, $\mathcal{Q}_1$, contains power traces obtained when the input to the algorithm is a random secret key sk_{rand}. In TVLA the Welch's t-test is used to detect the differences in the mean power consumption between the two classes mentioned. The t-test statistic is computed

as:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}}},$$

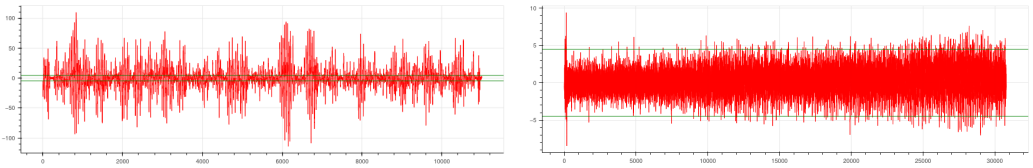
where μ_0 (resp. μ_1) and σ_0^2 (resp. σ_0^2) stand for the sample mean and sample variance of the set $\mathcal{Q}_0$ (resp. $\mathcal{Q}_1$) and n_0 and n_1 are the cardinality of the two sets. A t-test gives a probability to examine the validity of the null hypothesis as the samples in both sets were drawn from the same population, i.e., the two sets are not distinguishable. The null hypothesis is rejected with 99.999% confidence when the t value is exceeded the ± 4.5 range for a large number of measurements.

The TVLA experiment cannot be done on the whole `crypto_kem_decaps` function since the decapsulation function is not constant time. Kyber uses prime modulo $\mathbb{Z}_q$, where $q = 3329$ which is a 12 bit prime number. Inside the CPA-secure encryption algorithm of Kyber, as shown in line 1 of algorithm 2, the coefficientes of entries of the public matrix A is sampled from $\mathbb{Z}_q$. At first 12-bit numbers are sampled from the XOF (Keccak) and then checked if that number is less than q or not. If yes it is accepted otherwise discarded. As a result the implementation become non-constant time. But this does not lead to any security risk since the matrix A inside Kyber algorithm is public anyway.

For reasons mentioned above we have done component-wise TVLA experiment. In our TVLA experiments on components of `crypto_kem_decaps`, we divide between measurements with a fixed secret key, sk_{fix} and measurements with a random secret key, sk_{rand}. The null hypothesis in our experiment is that the implementation does not leak the long term secret key, sk. The input ciphertext to the `crypto_kem_decaps` function is kept as constant valid ciphertext of a fixed message encrypted with the secret key sk_{fix}.

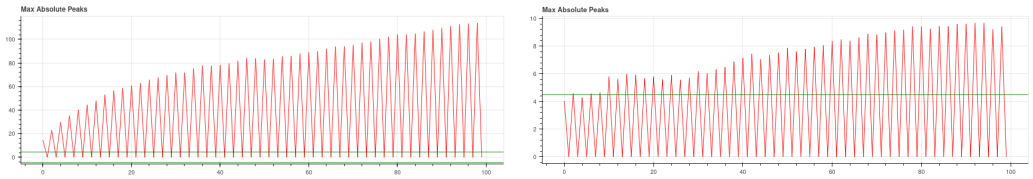
4.2.1 `poly_tomsg`

In figure 4.1a, we can see in the t-test of the plain implementation of the function `poly_tomsg`, after 10000 measurement the highest peak of the t-statistic has crossed the value 100. On the other hand, in figure 4.1b, for the shuffled implementation of the same function the value of the t-statistic has not even crossed the value 10 after 10,000 measurement. It took almost 1000 measurements to cross the value 4.5 (shown in figure 4.2b) in our shuffled implementation.



(a) Plain (b) Shuffled

Figure 4.1: T-statistic as a function of time after applying TVLA with a pool of 10 000 measurements of the component `poly_tomsg`



(a) Plain (b) Shuffled

Figure 4.2: Peak value of t-statistic after every 200 measurement of the component `poly_tomsg`

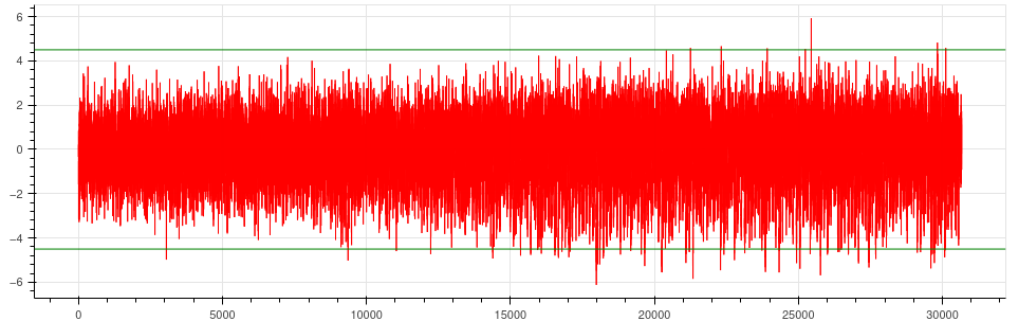


Figure 4.3: t-test of `poly_tomsg` with 10 000 measurements after removing the first message bit processing

But from figure 4.1b it is clear that the starting point of the graph is contributing to the peak t-test value. If we remove the processing of first message bit (figure 4.3) from the trace then it is taking almost 3000 measurements to cross the value 4.5 (shown in figure 4.4). So it is clear that the power trace of the first message bit processing is leaking more information than the other bits. But since in case of shuffling the first bit processing is randomized, the attacker cannot surely guess the first bit's exact position

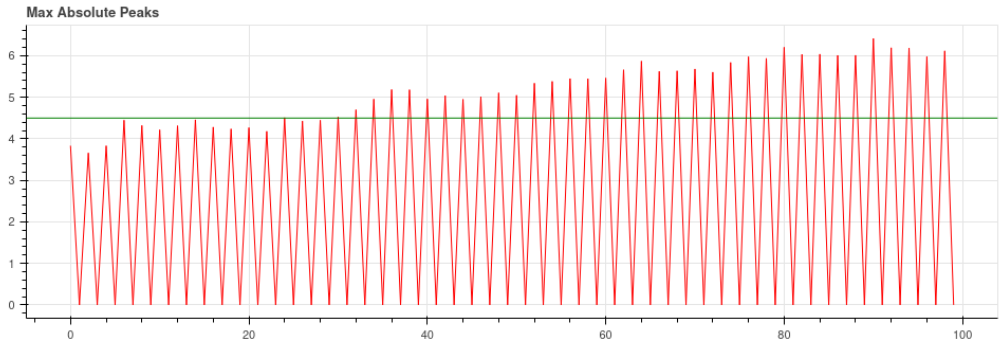
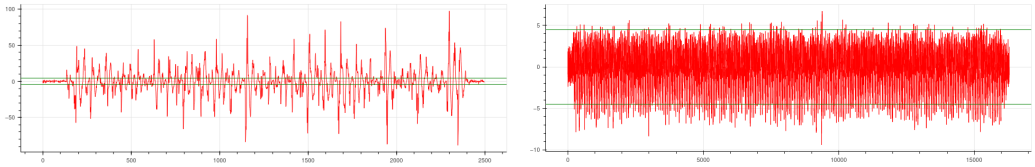


Figure 4.4: Peak value of t-test of `poly_tomsg` after every 200 measurements after removing the first message bit processing

in 32 bit message. Although template-attack like bit-flip technique can be used to recover message, but altogether shuffling is actually increasing the complexity of the attack with very little overhead in performance.

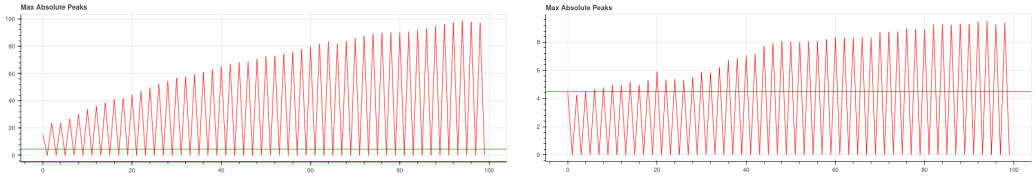
4.2.2 `poly_sub`

In figure 4.5a, after 10,000 measurements the peak t-statistic value for the plain implementation of the function of `poly_sub` has almost touched the value 100. Whereas in our shuffled implementation the peak value of the t-test has not even crossed the value 10. Although the value 4.5 is crossed but from the figure 4.5, it is clear that there is a improvement in the side channel leakage after our shuffled implementation.



(a) Plain (b) Shuffled

Figure 4.5: T-statistic as a function of time after applying TVLA with a pool of 10 000 measurements of the component `poly_sub`

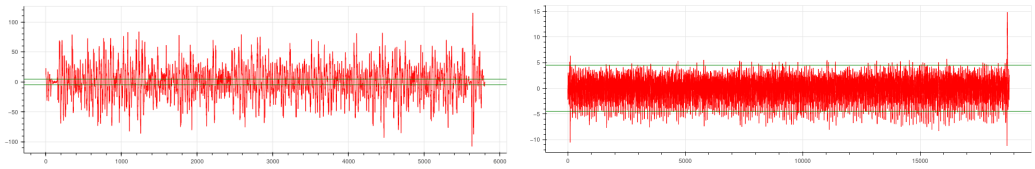


(a) Plain (b) Shuffled

Figure 4.6: Peak value of t-statistic after every 200 measurement of the component `poly_sub`

4.2.3 `poly_frommsg`

In figure 4.7a, we can see that after 10,000 measurements the peak t-statistic value for the plain implementation of the function of `poly_frommsg` has crossed the value 100. Whereas in our shuffled implementation the peak value of t-statistic after 10,000 has barely crossed the value 15. Since from figure 4.7b, it can be seen that the end point of the graph is giving the peak t-statistic values, we thought that the processing of the last message bit is responsible for that. We checked the t-test once again after removing the processing of the last bit of the message (shown in figure 4.9). In this t-test, it took 2000 measurements to cross the range 4.5 (shown in figure 4.10).

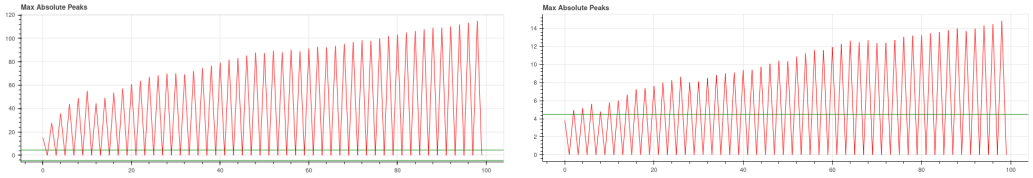


(a) Plain (b) Shuffled

Figure 4.7: T-statistic as a function of time after applying TVLA with a pool of 10 000 measurements of the component `poly_frommsg`

4.2.4 `poly_add`

In figure 4.11a, after 10,000 measurements the peak t-statistic value for the plain implementation of the function of `poly_add` has almost touched the value 100. Whereas in our shuffled implementation the peak value of the t-test has barely crossed the value 10. In 4.11b, it can be seen that the



(a) Plain (b) Shuffled

Figure 4.8: Peak value of t-statistic after every 200 measurement of the component `poly_frommsg`

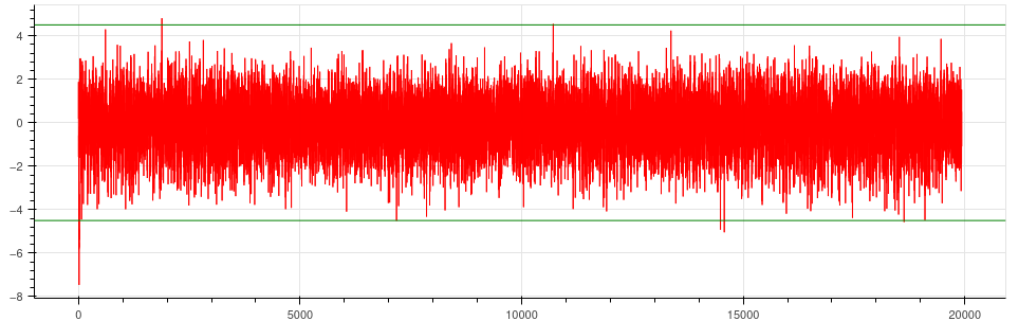


Figure 4.9: t-test of `poly_frommsg` with 10 000 measurements after removing the last message bit processing

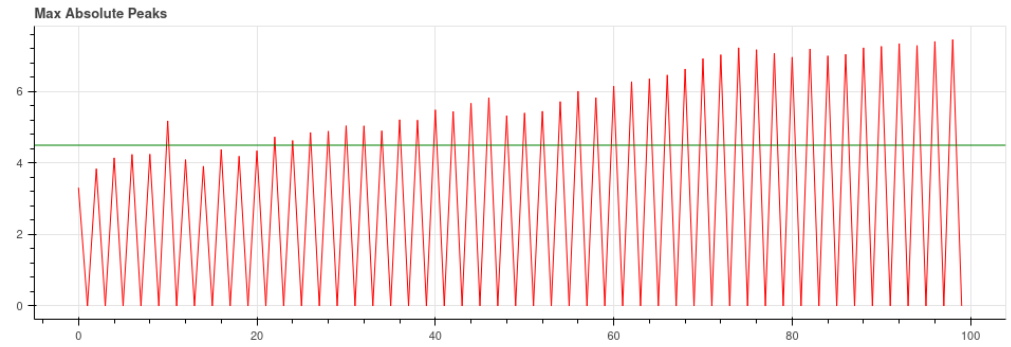
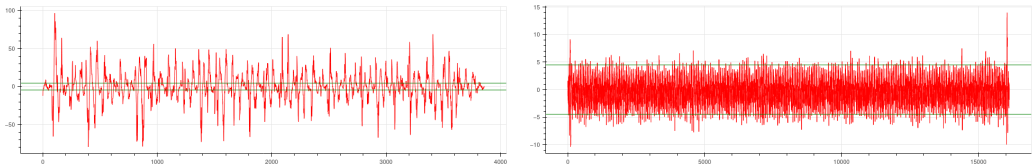


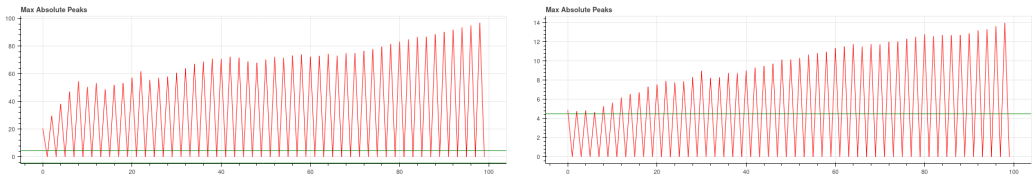
Figure 4.10: Peak value of t-test of `poly_frommsg` after every 200 measurements after removing the last message bit processing

two extreme point in the graph are contributing to the highest peak of the t-statistic value. Those two are corresponding to the addition of the first and the last coefficient of the two polynomials with 256 coefficients. We did



(a) Plain (b) Shuffled

Figure 4.11: T-statistic as a function of time after applying TVLA with a pool of 10 000 measurements of the component `poly_add`



(a) Plain (b) Shuffled

Figure 4.12: Peak value of t-statistic after every 200 measurement of the component `poly_add`

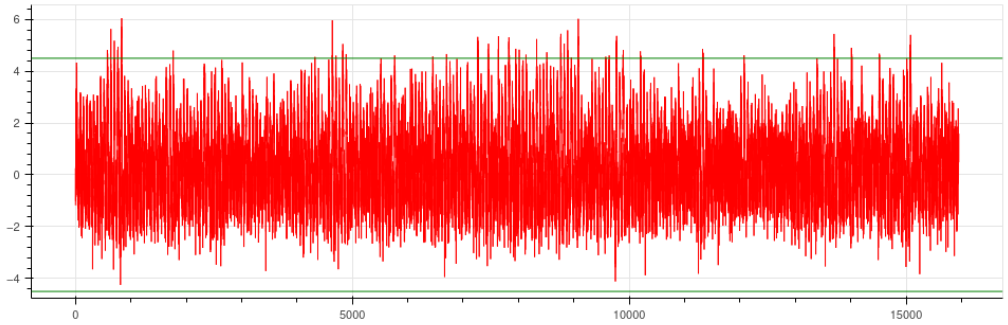


Figure 4.13: t-test of `poly_add` with 10 000 measurements after removing the first and last co-efficient addition

the t-test again by taking the trace of the additions corresponding to 253 coefficients (removing the addition of the two extreme coefficients from the trace) and noticed that (in figure 4.13) the peak value of the t-statistic in this t-test has not even crossed the value 6 with 10,000 measurements.

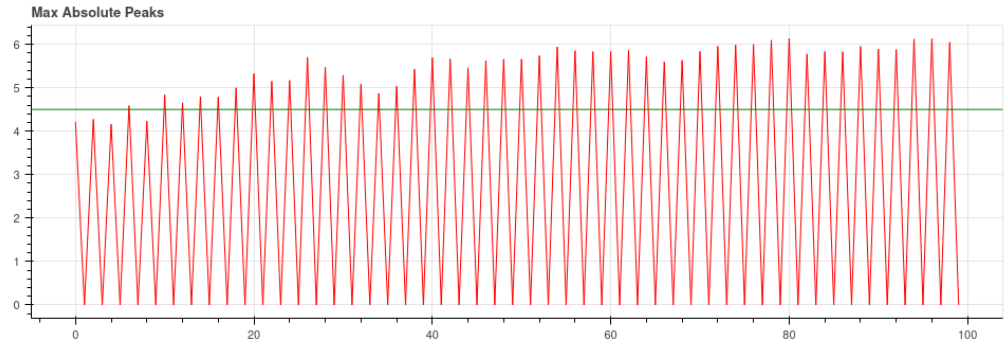
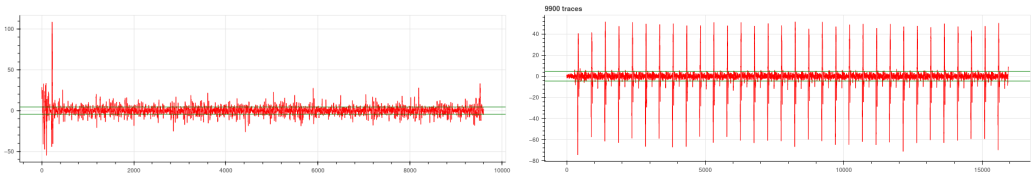


Figure 4.14: Peak value of t-test of `poly_add` after every 200 measurements after removing the first and last co-efficient addition

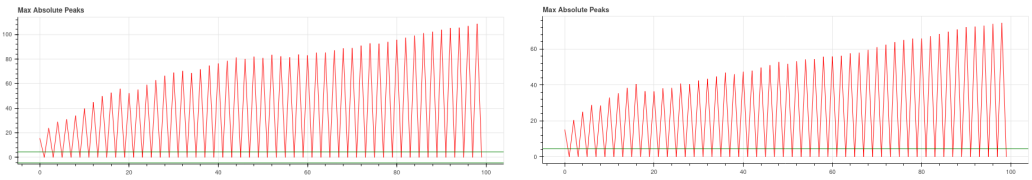
4.2.5 `poly_compress`

The t-test for the plain and shuffled version of the function `poly_compress` is shown in figure 4.15. It is quite clear from the figure that shuffling is not providing any benefit to the security of this function. Since we have implemented shuffling on c language only, this behavior of the t-test can occur due to some optimization during compiling of the c-code in assembly level.



(a) Plain (b) Shuffled

Figure 4.15: T-statistic as a function of time after applying TVLA with a pool of 10 000 measurements of the component `poly_compress`



(a) Plain (b) Shuffled

Figure 4.16: Peak value of t-statistic after every 200 measurement of the component `poly_compress`

Chapter 5

CONCLUSION

In this thesis, we have seen that only shuffling on post-quantum scheme Kyber can give impressive improvement against side channel attack with very little overhead in the run-time. Although in this paper we have implemented shuffling countermeasure on KYBER768, but the same implementation techniques can be easily extended to the other versions (KYBER512 and KYBER768) also. We have already combined shuffling countermeasure together with the mkm4 [10] project, which is the first order masked implementation of Kyber768. We have presented the run-time of the combined implementation in section 4.1. But due to time constraint we are unable to run t-test on the combined implementation.

In some of the components (e.g., `poly_compress`), shuffling has no visible effect in the leakage evaluation. But since we have done all implementation in c and the pqm4 project uses `-O3` optimization flag during compilation. So it is possible that there is no effect of shuffling due to some optimizations during compilation. In a future work these things can be observed thoroughly with implementation in ARM assembly with increased efficiency in the run-time.

References

[1] Roberto Avanzi et al. *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.02)*. 2021. URL: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[2] Linus Backlund et al. *Secret Key Recovery Attacks on Masked and Shuffled Implementations of CRYSTALS-Kyber and Saber*. Cryptology ePrint Archive, Paper 2022/1692. https://eprint.iacr.org/2022/1692. 2022. URL: https://eprint.iacr.org/2022/1692.

[3] Joppe Bos et al. *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM*. Cryptology ePrint Archive, Paper 2017/634. https://eprint.iacr.org/2017/634. 2017. DOI: 10.1109/EuroSP.2018.00032. URL: https://eprint.iacr.org/2017/634.

[4] Joppe W. Bos et al. *Masking Kyber: First- and Higher-Order Implementations*. Cryptology ePrint Archive, Paper 2021/483. https://eprint.iacr.org/2021/483. 2021. URL: https://eprint.iacr.org/2021/483.

[5] Yajing Chang et al. "Template Attack of LWE/LWR-Based Schemes with Cyclic Message Rotation". In: *Entropy* 24.10 (2022). ISSN: 1099-4300. DOI: 10.3390/e24101489. URL: https://www.mdpi.com/1099-4300/24/10/1489.

[6] Suresh Chari et al. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 398–412. ISBN: 978-3-540-48405-9.

[7] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *IEEE Trans. Inf. Theor.* 22.6 (Sept. 2006), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: https://doi.org/10.1109/TIT.1976.1055638.

[8] Gilbert Goodwill et al. *A testing methodology for side channel resistance*. 2011. URL: https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[9] Mike Hamburg et al. *Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber*. Cryptology ePrint Archive, Paper 2021/956. https://eprint.iacr.org/2021/956. 2021. URL: https://eprint.iacr.org/2021/956.

[10] Daniel Heinz et al. *First-Order Masked Kyber on ARM Cortex-M4*. Cryptology ePrint Archive, Paper 2022/058. https://eprint.iacr.org/2022/058. 2022. URL: https://eprint.iacr.org/2022/058.

[11] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. *A Modular Analysis of the Fujisaki-Okamoto Transformation*. Cryptology ePrint Archive, Paper 2017/604. https://eprint.iacr.org/2017/604. 2017. URL: https://eprint.iacr.org/2017/604.

[12] Matthias J. Kannwischer et al. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896842.

[14] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[15] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397. ISBN: 3540663479.

[16] Julien Doget Matthieu Rivain Emmanuel Prouff. *Higher-order Masking and Shuffling for Software Implementations of Block Ciphers*. Cryptology ePrint Archive, Paper 2009/420. https://eprint.iacr.org/2009/420.pdf. 2009. DOI: 10.46586/tosc.v2021.i3.137-169. URL: https://eprint.iacr.org/2009/420.pdf.

[17] Kalle Ngo, Elena Dubrova, and Thomas Johansson. *Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis*. Cryptology ePrint Archive, Paper 2021/902. https://eprint.iacr.org/2021/902. 2021. URL: https://eprint.iacr.org/2021/902.

[18] Kalle Ngo et al. *A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM*. Cryptology ePrint Archive, Paper 2021/079. https://eprint.iacr.org/2021/079. 2021. URL: https://eprint.iacr.org/2021/079.

[19] Tuy Tan Nguyen et al. "Area-Time Efficient Hardware Architecture for CRYSTALS-Kyber". In: *Applied Sciences* 12.11 (2022). ISSN: 2076-3417. DOI: 10.3390/app12115305. URL: https://www.mdpi.com/2076-3417/12/11/5305.

[20] Peter Pessl and Robert Primas. *More Practical Single-Trace Attacks on the Number Theoretic Transform*. Cryptology ePrint Archive, Paper 2019/795. https://eprint.iacr.org/2019/795. 2019. URL: https://eprint.iacr.org/2019/795.

[21] *Post-Quantum Cryptography: Selected Algorithms 2022*. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed: 2010-03-03.

[22] Robert Primas, Peter Pessl, and Stefan Mangard. *Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption*. Cryptology ePrint Archive, Paper 2017/594. https://eprint.iacr.org/2017/594. 2017. URL: https://eprint.iacr.org/2017/594.

[23] Prasanna Ravi et al. *On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT - A Performance Evaluation Study over Kyber and Dilithium on the ARM Cortex-M4*. Cryptology ePrint Archive, Paper 2020/1038. https://eprint.iacr.org/2020/1038. 2020. URL: https://eprint.iacr.org/2020/1038.

[24] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[25] Tobias Schneider and Amir Moradi. *Leakage Assessment Methodology - a clear roadmap for side-channel evaluations*. Cryptology ePrint Archive, Paper 2015/207. https://eprint.iacr.org/2015/207. 2015. URL: https://eprint.iacr.org/2015/207.

[26] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. URL: https://doi.org/10.1137/S0097539795293172.

[27] National Institute of Standard and Technology (NIST). *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Processe*. 2016. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[28] Nicolas Veyrat-Charvillon et al. "Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note". In: *ASIACRYPT*. Vol. 7658. Springer, 2012, pp. 740–757. DOI: 10.1007/978-3-642-34961-4_44. URL: https://www.iacr.org/archive/asiacrypt2012/76580728/76580728.pdf.