Breaking DPA-protected Kyber via the pair-pointwise multiplication

Estuardo Alpirez Bock[1], Gustavo Banegas[2], Chris Brzuska[3], Łukasz Chmielewski[4], Kirthivaasan Puniamurthy[3] and Milan Šorf[4]

[1] Xiphera LTD, Finland
estuardo.alpirezbock@xiphera.com
[2] Qualcomm France SARL, France
gustavo@cryptme.in
[3] Aalto University, Finland
chris.brzuska@aalto.fi
kirthivaasan.puniamurthy@aalto.fi
[4] Masaryk University, Czech Republic
chmiel@fi.muni.cz
xsorf@fi.muni.cz

Abstract. We present a new template attack that allows us to recover the secret key in Kyber directly from the polynomial multiplication in the decapsulation process. This multiplication corresponds to *pair-pointwise multiplications* between the NTT representations of the secret key and an input ciphertext. For each pair-point multiplication, a pair of secret coefficients are multiplied in isolation with a pair of ciphertext coefficients, leading to side-channel information which depends solely on these two pairs of values. Hence, we propose to exploit leakage coming from each pair-point multiplication and use it for identifying the values of all secret coefficients. Interestingly, the same leakage is present in DPA-protected implementations. Namely, masked implementations of Kyber simply compute the pair-pointwise multiplication process sequentially on secret shares, allowing us to apply the same strategy for recovering the secret coefficients of each share of the key. Moreover, as we show, our attack can be easily extended to target designs implementing shuffling of the polynomial multiplication. We also show that our attacks can be generalised to work with a *known* ciphertext rather than a *chosen* one.

To evaluate the effectiveness of our attack, we target the open source implementation of masked Kyber from the mkm4 repository. We conduct extensive simulations which confirm high success rates in the Hamming weight model, even when running the simplest versions of our attack with a minimal number of templates. We show that the success probabilities of our attacks can be increased exponentially only by a *linear* (in the modulus q) increase in the number of templates. Additionally, we provide partial experimental evidence of our attack's success. In fact, we show via power traces that, if we build templates for *pairs* of coefficients used within a pair-point multiplication, we can perform a key extraction by simply calculating the difference between the target trace and the templates. Our attack is simple, straightforward and should not require any deep learning or heavy machinery means for template building or matching. Our work shows that countermeasures such as masking and shuffling may not be enough for protecting the polynomial multiplication in lattice-based schemes against very basic side-channel attacks.

Keywords: Post-quantum Cryptography · Template attack · Kyber · Side-channel Attack · Single Trace

*Author list in alphabetical order; see https://www.ams.org/profession/leaders/CultureStatement04.pdf. Date of this document: 2023-04-19.

1 Introduction

In the the end of a rigorous competition process, NIST selected Kyber [BDK+18, ABD+20] as one of the post-quantum secure key encapsulation mechanisms (KEM) to be standardized. The main security requirement of the NIST competition is for the KEM to achieve message confidentiality under chosen-plaintext (CPA) and chosen-ciphertext attacks (CCA) based on problems which are plausibly post-quantum hard. A second important consideration in the NIST competition is the resistance of implementations against side-channel attacks. Side-channel vulnerabilities and resistance of Kyber implementations has emerged as a lively field of research over the past few years, and the present article falls into this line of research as well. In this paper we present a novel template attack (cf. [BCP+14, HKP+12a, HMA+08]) on masked implementations of Kyber. Our attack is performed on the decapsulation phase of Kyber and allows us to extract the long term secret key by creating and matching templates corresponding to the polynomial multiplication process performed. In the following, we give an overview on Kyber and recall details relevant to our attack.

Kyber's key encapsulation (encryption) performs a matrix-vector multiplication in the ring of polynomials $\mathcal{R}_q = \mathsf{Z}_q[x]/(x^{256}+1)$ and then adds a small noise vector to the result. In turn, for decapsulation (decryption), Kyber performs a vector multiplication between a ciphertext b and a secret a, each of which corresponds to a polynomial. Polynomials in Kyber are of degree 255 and their coefficients are integers between 0 and $q-1$, with $q = 3329$.

In Kyber, the above core IND-CPA-secure scheme is transformed into a CCA-secure encryption scheme using the Fujisaki-Okamoto (FO) transform [FO13]. FO performs re-encryption in the decryption process and prevents the decryption from returning a message if re-encryption fails, thereby mitigating ciphertext malleability. We note however that the FO transform does not prevent side-channel adversaries from performing CCA attacks on Kyber, since the input ciphertext is always multiplied with the secret key right at the beginning of the decapsulation process, independently of the validity of the ciphertext (see [DTVV19, RRCB20, HHP+21, BNGD22] for more examples of side-channel assisted CCA attacks on post-quantum CCA-secure schemes).

Number theoretic transform. Standard multiplication of two polynomials is *quadratic*. Thus, in Kyber (and other lattice-based systems), polynomials are first translated via the *number theoretic transform* (NTT) into a representation where multiplication only takes *linear* time. Namely, in the NTT domain, polynomial multiplications can be computed *point-wise*. Given two polynomials $\hat{\mathsf{a}}$ and $\hat{\mathsf{b}}$ with coefficients $(a_0, a_1, \ldots, a_{n-1})$ and $(b_0, b_1, \ldots, b_{n-1})$ in the NTT domain, their point-wise multiplication is equal to $\hat{\mathsf{a}} \circ \hat{\mathsf{b}} = (a_0 \cdot b_0, a_1 \cdot b_1, \ldots, a_{n-1} \cdot b_{n-1})$. In Kyber the NTT is not performed on its entirety due to its modulus polynomial, and the multiplication performed on NTT domain is actually *pair-pointwise* (see Subsection 2.2). This property will play a role on the effectiveness of our template attack, since per each pair-point multiplication, the coefficients are multiplied more than once, providing thus more points of comparison between our templates and our target trace.

Pair-pointwise multiplication. The size of the modulus q in Kyber underlies several constraints: (1) The ciphertext size grows *linearly* with the modulus, (2) the modulus needs to be *large enough* to enable a rounding operation required for correctness, and (3) the modulus needs to allow to perform the NTT, which requires decomposing the ring of polynomials $\mathbb{Z}_q[X]/(X^{255}+1)$, which is a 256-dimensional vector space over $\mathbb{Z}_q$. In its original proposal [ABD+17], the selected modulus q for Kyber was 7681. The modulus $q = 7681$ allows to perform a full NTT on the polynomials, as A full NTT requires to find a 512-th root of unity, i.e., a number ζ such that $\zeta^{512} \equiv 1 \mod q$. Therefore in the

original proposal of Kyber, the multiplications between the NTT representations of the secret key and ciphertexts were indeed performed in a point-wise fashion.

Nevertheless, a series of works later showed that Kyber could also be implemented using a smaller modulus $q = 3329$, in exchange of performing *incomplete* NTTs and calculating the multiplication in a *pair-pointwise* fashion [ZXZ+18, LS19]. Using this smaller modulus allows for a more compact implementation of Kyber, since we need one bit less for encoding each coefficient of a ciphertext and secret key. Moreover, this new modulus still allows to perform fast multiplications between the NTT representations of the polynomials. Thus, the specification of Kyber was updated and as of today the chosen modulus is $q = 3329$.

With respect to $q = 3329$, only the 256-th root of unity ζ exists, but not the 512-th root. For this reason Kyber only performs *incomplete* NTTs, where polynomials of degree 256 are transformed into 128 polynomials of degree 1. This is the reason why the multiplication between two polynomials on NTT domain in Kyber corresponds to a *pair-pointwise multiplication*. That is, for two transformed polynomials $\hat{\mathsf{a}}$ and $\hat{\mathsf{b}}$ we multiply $(a_0 + a_1x)(b_0 + b_1x), \ldots, (a_{254} + a_{255}x)(b_{254} + b_{255}x)$.

Unfortunately as we will show in this work, pair-pointwise multiplications also lead to more leakage in the presence of side-channel adversaries. Consider the case that we are multiplying a secret key with some known value. Within each pair-point multiplication, two coefficients of the secret key will be multiplied with with two different known values, and then they will be added to each other. This means that each pair-point multiplication corresponds to a series of operations where the only unknown values are two coefficients with values between 0 and 3328. This simple observation motivates us to study whether we can exploit the leakage coming from a pair-point multiplication and use it for identifying the value of the secret coefficients.

1.1 Our contribution

We propose an attack on the pair-pointwise multiplication of Kyber-like implementations. We present a template attack strategy with different variations for extracting each coefficient used within each pair-point multiplication. The variations of our attack are crafted depending on the type of implementation we are attacking. For instance, our simplest attack should work on implementations where the pair-point multiplication is not optimised, but we find simple adaptations of the attack for targeting optimised implementations. Moreover, we also provide extensions for attacking shuffled implementations. All the attack strategies we present are based on the same idea, which we illustrate next by means of a simple example.

For ease of explanation, we assume in the following that (a) the pair-pointwise multiplication is implemented in a straightforward way without any optimisations (see Equation 2) and (b) each multiplication in Equation 2 will provide enough leakage for successful template matching. In practice, these two assumptions will not always hold, but they will help clarifying the idea behind our attack.

Attack Idea. Let us assume that we want to find out whether the secret $\hat{\mathsf{a}}$ (a on NTT domain) of some implementation of Kyber has *some* coefficient with value, e.g. 328. To determine whether $\hat{\mathsf{a}}$ has some coefficient with value 328, we will construct a template for the case that 328 is a coefficient in $\hat{\mathsf{a}}$ and it is used as an operand in the polynomial multiplication. We construct such a template as follows. In our own device, we fix the key a such that (in NTT domain) all its coefficients have the value 328, i.e.

$$\hat{\mathsf{a}} = (328_0, 328_1, 328_2, \ldots, 328_{254}, 328_{255}).$$

As input ciphertext, we will provide a value b which on NTT domain has the half of its coefficients equal to (e.g.) 2649, and the other half equal to 317,[1] i.e.

$$\hat{\mathsf{b}} = (2649_0,\ 317_1,\ 2649_2,\ 317_3, \ldots, 2649_{254}, 317_{255}).$$

We then run the implementation on said inputs and record a power trace, which we will use as our template and which we denote T_{328}.

We now turn to the target device and run it on an input equal to the ciphertext used for constructing our template, i.e. b s.t. $\hat{\mathsf{b}} = (2649_0, 317_1, 2649_2, 317_3, \ldots, 2649_{254}, 317_{255})$. We record a power trace, which we will denote as our target trace T_t.

We will now perform a template matching which will help us find out whether the secret $\hat{\mathsf{a}}$ running on the target device has some coefficient equal to 328. The template matching will also help us find out exactly *which* coefficient of $\hat{\mathsf{a}}$ has the value 328. Recall first that in the decapsulation process of Kyber, we multiply the ciphertext $\hat{\mathsf{b}}$ with the secret $\hat{\mathsf{a}}$ in a *pair-pointwise* fashion. That means that our template trace T_{328} corresponds to multiplications

$$(328_0 + 328_1)(2649_0 + 317_1), (328_2 + 328_3)(2649_2 + 317_3), \ldots,$$
$$(328_{254} + 328_{255})(2649_{254} + 317_{255}).$$

Note that for each multiplication, we perform the operations described in Equation 2. Moreover, each multiplication is performed sequentially and the result of one multiplication does not affect any other multiplication. If the secret $\hat{\mathsf{a}}$ in the target device has some coefficient a_i equal to 328, we should be able to find good correlations at the points where that coefficient was multiplied by 2649 ($a_i \cdot b_0$ in Equation 2), or at the points where it was multiplied by 317 ($a_i \cdot b_1$ in Equation 2) during some pair-pointwise multiplication. The location of regions where we find good correlations lets us know *which* coefficient in $\hat{\mathsf{a}}$ has the value 328. For instance, assume the coefficients of $\hat{\mathsf{a}}$ are $(7_0, 82_1, 104_2, \ldots, 328_{128}, \ldots, 2013_{255})$. When we perform template matching, we should find good correlations halfway through the trace, at the position corresponding to the pair-pointwise multiplications using the 128th coefficient as operand. Namely, in both the template trace T_{328} and the target trace T_t, the given region corresponds to the power consumption of the same operations using the same operands: $328 \cdot 2649$ (or $328 \cdot 317$).

Moreover, we expect to see bad correlations at all other regions in the trace, (unless a coefficient with value 328 appears elsewhere). Namely in all other regions of the traces, the power consumption corresponds to multiplications between different operands, and thus the template should not match the target trace.

We can perform a complete extraction of all coefficients in $\hat{\mathsf{a}}$ by repeating the same process described above, checking whether $\hat{\mathsf{a}}$ has some coefficient equal to 0, equal to 1, equal to 2, ..., or equal to 3328. Thus, all we need to do is generate a total of 3329 templates and try matching each template with the target trace. Once we have recovered all coefficients in $\hat{\mathsf{a}}$, we just need to transform them back to their standard domain and this will let us recover a. Moreover, this same attack strategy will also allow us to attack masked implementations, since we just need to recover the coefficients of each share, and then combine them to reconstruct the key.

Attacking Kyber in practice and our results. The attack strategy described above corresponds to the basic idea behind our attack and it requires only a total of q templates for a complete key extraction from a masked or unmasked implementation. Later in Section 3 we provide concrete steps for performing our attack. We explain that sometimes,

[1] Note that these are just example values. In principle, we can choose any two values between 0 and $q - 1$. What is important is that the values are located in the ciphertext on NTT domain as shown in the example.

q traces may only allow us to extract one coefficient within each pair-point multiplication. However with knowledge of that one coefficient, we can easily build an additional set of q templates which will allow us to extract the remaining coefficient of each pair-point multiplication. Moreover, we also present an attack strategy where we build templates for each possible *pair of coefficients* within a pair-point multiplication. This attack strategy is much more expensive since it requires a total of q^2 templates. On the positive side however, it has a very high success probability since when performing template matching, we compare regions corresponding to the complete pair-point multiplication process.

We also asses the effectiveness of our attack in practice. For this, we focus on the masked implementation of [cod22] as a case study. We carefully analyse the pair-pointwise multiplication process of this implementation, and perform simulations to determine the success rate of our attack in the Hamming weight model, given only a *single trace* and a *known* ciphertext. After an extensive analysis, we notice that our attack has a very high success probability given an expected number of $\approx 3q$ template traces. The following points explain why the pair-pointwise multiplication in this implementation generates enough leakage for our attack to succeed.

1. Instead of *one* multiplication (as in full NTT), in pair-point multiplications, *three* multiplications (cf. Equation (4)) depend on the same coefficient pair.
2. Since multiplications are performed mod q, the code requires 3 additional operations to execute a *modulus* reduction after each multiplication.
3. While numbers in $[0, \ldots, q-1]$ are 12-bit integers, the code uses 24-bit and 28-bit values due to the modulus reduction. Thus, in the Hamming weight (H) model, the expected information per instruction is $H(24) \approx 3.34$ and $H(28) \approx 3.45$ bits of information rather than only $H(12) \approx 2.84$.

Simulations for the Hamming weight model. On a high level, we want to know whether each possible secret coefficient value (values between 0 and 3328) leads to *unique* hamming weight values during the pair-point multiplication process. If this is the case, then we should see enough leakage allowing us to uniquely determine the value of a secret coefficient processed during a pair-point multiplication. For each possible secret coefficient, we calculate the hamming weight of the result of all instructions executed during the pair-point multiplication. We obtain thus *hamming weight tuples* for each possible secret coefficient. As we show for all odd coefficients in a secret polynomial, about 90% of the coefficient values actually have a unique hamming weight tuple. We interpret this as evidence that we can extract the value of an odd secret coefficient with 90% chances of correctness, using only q templates. Later in the paper we explain how we can use the extracted odd coefficients for extracting all even coefficients. Moreover, our simulation results help us outlining attack extensions which allow us to increase our overall success rates, given additional $2q$, $3q$ or $4q$ templates.

Experiments. We perform partial experiments using a ChipWhisperer Lite platform. In the experiments, we aim for a simple attack where we create templates for possible pairs of secret coefficient used within a pair-point multiplication. We acquire 10000 templates for randomly generated pairs of values and try matching them with a target trace. Our experiments show that it is indeed very easy to extract pairs of coefficients with this simple technique.

State of the art. Attacks on the polynomial multiplication of Kyber have been successfully performed via correlation power analysis [MBB+22]. However already in early stages, it was proposed to mask the polynomial multiplication in lattice based schemes as a means

to mitigate side-channel analysis [OSPG18, RdCR+16, RRdC+16]. For this reason, many works focus on attacking other parts of the decapsulation process of Kyber. Primas, Pessl and Mangard [PPM17], present a template attack on the inverse NTT during decryption. This attack allows them to recover a decrypted message and use it for recovering the session key. The attack is assisted by belief propagation for template matching and was extended and improved in subsequent works [PP19, HHP+21]. In an alternative approach, Dubrova, Ngo and Gärtner propose to use deep learning techniques for recovering the message and then recovering the long term secret key [DNG22] and a line of research has demonstrated the success of deep-learning techniques for attacking lattice based schemes [BNGD22, JWN+22, NWDP22, MKK+23]. We also refer the reader to [RCDB22] for a recent survey on side-channel and fault attacks on Kyber and Dilithium.

The attacks mentioned above also succeed on masked implementations. Our attack however differs significantly from any of these attacks given that (1) we directly extract the long term secret key and (2) we should not require any deep learning, belief propagation or heavy machinery means for constructing templates or matching the templates to our target trace. In other words, our attack is much simpler and easy to replicate. From our perspective, the easier it is to replicate an attack, the bigger the threat it imposes in practice. Moreover, it is also very easy to explain and understand why our attack is successful, and a clear understanding of the success of our attack may lead a clear path towards effective countermeasures. We note that none of the countermeasures mitigating the attacks mentioned in this section should affect the success of our attack, since our attack targets calculations in Kyber which are not covered by those countermeasures.

2 Notation and preliminaries

We represent matrices by bold capital letters $\mathbf{A}$, and vectors by bold small letters $\mathbf{b}$, $\mathbf{b}$. Given a polynomial $a = \sum_{i=0}^{n-1} a_i X^i$ of degree $n-1$, we usually write a as a vector $a = (a_0, a_1, a_2, ..., a_{n-1})$. Also, the operation $\cdot$ represents standard multiplication between two integers, while $\circ$ represents point wise multiplication between two polynomials in NTT domain (cf. Subsection 2.2). When writing polynomial a in NTT domain, we will often write $\hat{a}$ for clarity, and also use the hat notation for matrices, e.g., $\hat{\mathbf{A}}$.

We next provide descriptions of Kyber. Our descriptions of the algorithms will be simplified and we will elaborate mostly on the parts of the KEM which are relevant for our attack. These include the decapsulation algorithm, the NTT transformation and the multiplication between the secret key and the input ciphertext in the decapsulation. We refer the reader to the supporting documentation from Kyber for more details on the KEM [ABD+20].

2.1 Kyber

As previously mentioned, Kyber is a lattice-based key encapsulation mechanism (KEM). it relies in the hardness of the Module-LWE problem. The latest parameters for Kyber are: $n = 256, q = 3329, \eta = 2$ and module dimension $k = 2$, 3, or 4. The security level of Kyber increases with its module dimension (in the case k).

Algorithm 1 gives the overview of the key generation. The private key of Kyber consists of a vector of polynomials of degree $n = 256$, and with coefficients in R_q with $q = 3329$. The k determines the dimension of the vector. The functions Sample_U and Sample_B are functions which uniformly sample values in the ring R_q given a seed. The Sample_U provides a uniform random matrix, and Sample_B gives uniform random vectors. The function H is a secure hash function (SHA3 in Kyber).

Algorithms 2 and 3 describe the encryption and encapsulation functions in Kyber. Particularly relevant for this work are the functions Compress and Decompress, which

Algorithm 1: Kyber-CCA2-KEM Key Generation (simplified)

Output: Public key pk, secret key sk

Choose uniform seeds ρ, σ, z
$\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow \text{SAMPLE}_U(\rho)$
$\mathbf{a}, \mathbf{e} \in R_q^k \leftarrow \text{SAMPLE}_B(\sigma)$
$\hat{\mathbf{a}} \leftarrow \mathsf{NTT}(a)$
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{a}} + \mathsf{NTT}(e)$
$pk \leftarrow (\hat{\mathbf{t}}, \rho)$
$sk \leftarrow (\hat{\mathbf{a}}, pk, \mathsf{H}(pk), z)$
return pk, sk

are defined as $\text{COMPRESS}(u) := \lfloor u \cdot 2^d/q \rceil \bmod (2)^d$ and $\text{DECOMPRESS} := \lfloor q/2^d \cdot u \rceil$, with $d = 10$ if $k = 2$ or 3 and $d = 11$ if $k = 4$. Note that the output of the encryption corresponds to a ciphertext c, which consists of two *compressed* ciphertexts. This ciphertext c will be the input to the decapsulation algorithm.

Algorithm 2: Kyber-PKE Encryption (simplified)

Input: Public key $pk = (\hat{\mathbf{t}}, \rho)$, message m, seed τ
Output: Ciphertext c

$\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow \text{SAMPLE}_U(\rho)$
$\mathbf{r}, \mathbf{e}_1 \in R_q^k, e_2 \in R_q \leftarrow \text{SAMPLE}_B(\tau)$
$\mathbf{b} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \mathsf{NTT}(\mathbf{r})) + \mathbf{e}_1$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \mathsf{NTT}(\mathbf{r})) + e_2 + \text{ENCODE}(m)$
$\mathbf{c}_1, c_2 \leftarrow \text{COMPRESS}(\mathbf{b}, v)$
$c = (\mathbf{c}_1, c_2)$
return c

Algorithm 4 shows the decapsulation algorithm. Note that the ciphertext is first decompressed into its standard form b, and then in line 2 the ciphertext is transformed to its NTT domain. After this transformation, a pair-pointwise multiplication between $\hat{\mathsf{a}}$ and $\hat{\mathsf{b}}$. This operation will be the target of *our attack*.

Algorithm 3: Kyber-CCA2-KEM Encryption (simplified)

Input: Public key $pk = (\hat{\mathbf{t}}, \rho)$
Output: Ciphertext c, shared key K

Choose uniform m
$(\bar{K}, \tau) \leftarrow \mathsf{H}(m || \mathsf{H}(pk))$
$c \leftarrow \text{PKE.ENC}(pk, m, \tau)$
$K \leftarrow \mathsf{KDF}(\bar{K} || \mathsf{H}(c))$
return c, K

2.2 Number Theoretic Transform (NTT)

Kyber (as many other lattice-based cryptosystems) performs polynomial multiplications which naively takes n^2 operations for polynomials of degree $n-1$. However, multiplication is sped up to *linear* time by transforming the polynomials into the NTT domain, allowing

Algorithm 4: Kyber-CCA2-KEM Decryption (simplified)

Input: secret key $sk = (\hat{\mathbf{a}}, pk, \mathsf{H}(pk), z)$, ciphertext $c = (\mathbf{c_1}, c_2)$
Output: Shared key K

$\mathbf{b}, v \leftarrow \textsc{Decompress}(\mathbf{c_1}, c_2)$
$m \leftarrow \textsc{Decode}(v - \mathsf{NTT}^{-1}(\hat{\mathbf{a}})^T \circ \mathsf{NTT}(\mathbf{b})))$
$(\bar{K}, \tau) \leftarrow \mathsf{H}(m||\mathsf{H}(pk))$
$c' \leftarrow \text{PKE.}\textsc{Enc}(pk, m, \tau)$
if $c = c'$ **then**
  $K \leftarrow \mathsf{KDF}(\bar{K}||\mathsf{H}(c))$
else
  $K \leftarrow \mathsf{KDF}(z||\mathsf{H}(c))$
return K

for a so-called *pointwise* multiplication between the polynomials. The NTT is a version of Fast Fourier Transform (FFT) but in a finite ring. To perform the transformation, one can evaluate the polynomial at powers of a primitive root of unity which are usually represented by the symbol ζ.

Kyber has dimension k, each dimension presents their own roots, and we represent it by ζ_k the set $\zeta_k^0, \zeta_k^1, \ldots, \zeta_k^{n-1}$. In the following, we will explain a little bit more the usage of NTT in Kyber since it does not perform a complete NTT but an incomplete NTT. We refer the reader to [Kan22] for more details on how the NTT can be implemented (to Kyber and Dilithium).

The NTT on Kyber implementations. In Kyber since it exist only n-th roots of unity, the modulus polynomial $X^n + 1$ can only be factored into polynomials of degree 2. Hence, an *incomplete* NTT is performed, where we skip the last layer of NTT. Therefore after the (incomplete) NTT transformation, a polynomial a corresponds to 128 polynomials of degree 1 each. Polynomial a is thus transformed to

$$\mathsf{NTT}(a) = a_0 + a_1 x, \quad a_2 + a_3 x, \quad a_4 + a_5 x, \ldots, a_{254} + a_{255} x. \tag{1}$$

The incomplete transformation of the polynomials to their NTT domains has an impact on the way multiplications are performed in Kyber. Namely when computing the multiplication between two transformed polynomials, we are not computing a point-wise multiplication between the coefficients of the polynomials (i.e. $a \cdot b = (a_0 b_0 = c_0, a_1 b_1 = c_1, \ldots, a_n b_n = c_n)$). Instead, we multiply the coefficients pairwise and, for instance, the first two coefficients of the resulting polynomial are obtained as follows:

$$\begin{aligned} c_1 &= a_0 b_1 + a_1 b_0 \\ c_0 &= a_0 b_0 + a_1 b_1 \zeta \end{aligned} \tag{2}$$

We will denote the multiplication in Equation 2 as *pair-pointwise*. We remark that it is different from the point-wise multiplication computed when a complete NTT is applied to the polynomials (such as in Dilithium).

Multiplication optimisations. In Equation 2 we see a very straightforward way of calculating a pair-pointwise multiplication, and obtaining the resulting two adjacent coefficients of a polynomial. We see that a total of 5 multiplications are performed. This multiplication process can be optimised via the Karatsuba algorithm in such way that we only need to perform 3 multiplications per each pair-pointwise multiplication:

$$\begin{aligned}&(a_0 + a_1x)(b_0 + b_1x) \bmod (x^2 - \zeta)\\ &= a_0b_0 + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)x + a_1b_1x^2\\ &= a_0b_0 + a_1b_1\zeta + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)x\end{aligned} \tag{3}$$

Thus we can obtain the resulting polynomial $c_0 + c_1x$ via

$$\begin{aligned}c_0 &= a_0b_0 + a_1b_1\zeta\\ c_1 &= (a_0 + a_1)(b_0 + b_1) - (a_0b_0 + a_1b_1)\end{aligned} \tag{4}$$

Observe that Karatsuba multiplication is the most popular approach for implementing the pair-pointwise multiplication in Kyber. It allows us to reduce the number of multiplications from five to four. This approach has been adapted by the software implementation we analyse in this paper, and also in public hardware implementations of Kyber such as [XL21].

Masking Kyber. There are several proposals to mask lattice-based schemes (NTRU [OSPG18], Saber [BDK+21]), whereby the following works present concrete masking schemes for Kyber [BGR+21, HKL+22]. The schemes consider masking of any secret dependent operation such as the computation of inverse NTT, of the key derivation function on the decapsulation process or, more commonly, masking of the polynomial multiplication with the long-term secret. The idea for masking the polynomial multiplication on Kyber is the same as usually applied in other cryptographic schemes: split the secret into shares and compute secret dependent operations on each share. Then add the results. For Kyber it means that the secret polynomials will be split into shares and then, the input ciphertext will be multiplied with each share separately.

3 Our attack

In this section we explain our template attack on the decapsulation process of Kyber. We recall that our attack allows us to extract the coefficients of the secret a during the polynomial multiplication at the beginning of the decapsulation process. Note that the coefficients that we extract will be in NTT domain, and after correctly recovering, we need to transform them back to their standard domain.

In what follows we first describe our main attack and its steps. Subsequently, we show how variants of our attack with a smaller or larger number of templates affect the success probability of key recovery. Moreover, we explain how our attack can be directly applied for targeting masked implementations and explain how we can extend our attack in order to target implementations which apply shuffling to the polynomial multiplication.

Attacking Kyber in practice. In Subsection 1.1 we provided an example describing the idea behind our attack. However when targeting real life implemenations of Kyber, we should consider several aspects which may affect the success probability of our attack. First as noted in Subsection 2.2, many implementations of Kyber optimise the pair-pointwise multiplication via Karatsuba and thus the multiplication is not performed exactly as described in Equation 2, and we may have less points of comparison for each multiplication with a coefficient value we are trying to extract. Second, it is not clear whether we will always get enough leakage from a multiplication operation, such that it would allow us to distinguish the values of the operands being used. In practice this will depend on the environment running the implementation of Kyber and on the way the multiplication operations are implemented. For instance, the more clock cycles needed

for calculating one multiplication, the more points of comparison we will have when performing template matching. However, some implementations and environments allow to perform multiplications between operands within just one clock cycle. On the other hand, we note that a single multiplication usually involves more operations than just the multiplication itself, such as load operations and modular reductions. In any case, it is worth analysing the number of operations within a pair-pointwise multiplication which depend solely on one of the two coefficients a_0 or a_1 and try to exploit such operations for trying to distinguish. In the following we first analyse possible leakage points for attacking implementations of Kyber which make use of Karatsuba for implementing the pair-pointwise multiplication. This analysis will help us crafting a template attack which will succeed with high probability, and which will not require a very large number of templates.

3.1 Attack steps - extracting the key via $q + q$ templates

As we point out in Subsection 2.2, many implementations of Kyber implement the pair-pointwise multiplication via Karatsuba, reducing thus the number of single multiplications during the process. As we can see in Equation 4, for each pair of coefficients a_0 and a_1, coefficient a_0 is multiplied only once times b_0, while coefficient a_1 is multiplied once with b_1 and their product is multiplied with ζ. If one multiplication is enough for extracting a secret coefficient, then our attack would still work using only q templates. Nevertheless, there exist better chances of extracting each coefficient a_1 alone since there exist more operations within the pair-pointwise multiplication which depend solely on a_1 without any influence of a_0. In the following, we will explain how we can use q templates for extracting all such a_1 values within each pair-point multiplication. These coefficients correspond to all coefficients $a_1, a_3, a_5, a_7, \ldots, a_{253}, a_{255}$ in $\hat{\mathbf{a}}$. Then, with knowledge of all extracted values, we will build new templates and will use them for extracting all remaining values a_0.

Generating the inputs. Note that when building templates and when obtaining the target trace, we will be using chosen ciphertexts (and chosen keys when building templates), which on NTT domain have a *specific* structure. Therefore we need to find polynomials in standard domain which have the desired structure on NTT domain. It turns out we can do this very easily since the NTT (and its incomplete version applied in Kyber) is a bijection. Thus all we need to do is set a polynomial with the desired coefficients and run the inverse NTT on it. More precisely for Kyber, we set 128 polynomials of degree 1, each with the desired coefficients (see Subsection 2.2) and run the inverse NTT on them. In addition, we also need to consider the compression and decompression properties of the ciphertext in standard domain, since the input ciphertexts are provided to the decapsulation algorithm in compressed form (see Algorithm 4). We recall that the compression and decompression algorithms may introduce some errors in the least significant bits of some coefficients of the polynomials. Thus, when setting a value $\hat{\mathbf{b}}$ with a desired structure, and then transforming it into its standard domain $\mathbf{b}$, we should check whether $\mathbf{b}$ can be compressed and decompressed, such that

$$\textsc{Decompress}(\textsc{Compress}(\mathbf{b})) = \mathbf{b}.$$

If the equation above holds, we ensure that on line 2 of Algorithm 4, $\mathsf{NTT}(\mathbf{b})$ is indeed transformed into a vector with the structure we initially desired. In [HHP+21], the authors dealt with the same issue for their chosen ciphertext attack on the decapsulation process of Kyber. The authors needed a ciphertext $\mathbf{b}$ which on NTT domain would be sparse, and they presented two methods for generating such ciphertexts and ensuring that they would preserve the desired properties after compression and decompression. It turns out that for

us it is much easier to deal with this issue, since the structure we desire for the NTT-d value is much more flexible as we explain below (and as will be seen in the attack steps described in the rest of this section).

In essence for our attack, we simply require a ciphertext vector which on NTT domain has either of the two following properties:

- For each pair of coefficient values b_0, b_1, it holds that $b_0 \neq b_1$, or
- For any two coefficients b_i, b_j in $\mathbf{b}$ it holds that $b_i \neq b_j$.

The first property is enough for attacking unprotected and even masked implementations. The second property will be relevant for attacking implementations which implement shuffling of the polynomial multiplication (see Subsection 3.3.1). Naturally, vectors with the second property can also be used for attacking masked or unprotected implementations since the second property implies the first property. Our advantage is that there is no restriction with respect to the specific values these coefficients should have. Thus when generating the inputs, we could simply set the desired vector $\hat{\mathsf{b}}$, run the inverse NTT on it and then check whether the resulting vector preserves its form after compression and decompression. Moreover, it is not even necessary that the vector in standard domain preserves its original form. It is only important that the resulting vector can be transformed via the NTT into a vector with any of the properties listed above. Therefore, it should be very easy to just try out some values. Another simple strategy could be to set a vector in standard domain $\mathbf{b}$ with small coefficient values. The small values ensure that the coefficients will preserve their original values after compression and decompression. Then we can simply apply the NTT to $\mathbf{b}$ and check whether the resulting vector $\hat{\mathsf{b}}$ has the desired properties listed above. Finally we point out that finding input ciphertexts which achieve the second property can be done very easily and we may not even need to choose those ciphertexts ourselves.

We now proceed to explaining the attack steps for extracting the secret a using a total of $2q$ templates.

Step 1: Template building. We start by building templates in the exact same way as described in our earlier example, starting by building the template T_0. That is, we first build a template for the case that the secret $\hat{\mathsf{a}}$ consists completely of coefficients equal to 0

$$\hat{\mathsf{a}} = (0_0, 0_1, 0_2, \ldots, 0_{255}).$$

For the input ciphertext, we can choose a ciphertext equal to the one used in our example. What's important is that the polynomial has a structure where coefficients corresponding to b_0 and b_1 are always different, i.e. $b_0 \neq b_1$. As an example, we consider the ciphertext below.

$$\hat{\mathsf{b}} = (2649_0, 317_1, 2649_2, 317_3, \ldots, 2649_{254}, 317_{255}).$$

We record thus a power trace and obtain the template T_0. We repeat this process for all possible values between 0 and $q-1$. For each new template, we change the value of $\hat{\mathsf{a}}$ accordingly (i.e. setting $\hat{\mathsf{a}} = (1_0, 1_1, 1_2, \ldots, 1_{255}), \hat{\mathsf{a}} = (2_0, 2_1, 2_2, \ldots, 2_{255})$, etc) and we always use the same ciphertext $\hat{\mathsf{b}}$.

Step 2: Obtaining the target trace. We now turn to the target device running a key decapsulation of Kyber and query it using our chosen ciphertext b, which on NTT domain maps to the ciphertext $\hat{\mathsf{b}}$ described above. We record a power trace during execution and obtain our *target trace* T_t.

We now have our set of templates and our target trace and can proceed to perform template matching. The idea is that we will obtain enough information to identify good matches for operations involving the operands a_1, since this coefficient is used independently in several operations during each pair-point multiplication. We will assume that we will not be able to identify any matches for coefficients a_0 since this coefficient is only used once independently during each pair-point multiplication.

Step 3: Template matching. We now match the target trace T_t with each template T_j. We expect to see no correlations between any regions of the traces, *unless* both the target trace and the template used the same operands a_1, b_0, b_1 within some pair-point multiplication. First we compare the target trace with the template T_0. There are a total of 128 pair-pointwise multiplications and thus, a total of 128 regions corresponding to this operation in the power traces. We can numerate each region sequentially from 0 to 127. If we observe some correlations between the target T_t and our template T_0 on region i, then we will know that the operand a_{2i+1} has the value 0. We then repeat the process with all remaining templates, or until we have extracted all a_1 operands of the polynomial $\hat{\mathsf{a}}$.

Step 4: Template building with extracted coefficients. In the previous step, we extracted all operands corresponding to a_1 during a pair-point multiplication. We will now use the knowledge of the extracted coefficients for building a new set of templates which will help us extract all operands corresponding to a_0 in each pair-point multiplication.

Let us denote by ψ an operand a_1 whose value was extracted in the previous step. In essence, we can now build templates in the same way as we did in **Step 1**, but the keys $\hat{\mathsf{a}}$ will now have the following structure. For each value $j \in [0, 1, \ldots, 3328]$ we construct a template for, i.e. each value we set for the key during each template generation, we set the key as follows:

$$\hat{\mathsf{a}} = (j_0, \psi_1, j_2, \psi_3, \ldots, j_{254}, \psi_{255}).$$

We will denote the templates generated during this step as $T_{j,\psi}$, and we will generate all of them the same way as described in **Step 1**, using the same input ciphertext $\hat{\mathsf{b}}$. We obtain a total of q new templates $T_{j,\psi}$.

Step 5: Template matching We now perform template matching in the exact same way as we did in **Step 3**, but using the templates $T_{j,\psi}$ we obtained in **Step 4**. We now expect to see correlations which will let us extract all a_0 values. As opposed to the template matching we performed on **Step 3**, we now will have more points of comparison for finding correlations between some template $T_{j,\psi}$ and the target trace T_t. Namely for a template corresponding to a correct value j for some a_0, we now expect to find correlations not only on the single multiplication $a_0 \cdot b_0$, but also on all remaining operations dependent on a_0 and a_1, i.e. all operations within the pair-point mutliplication. Since the value for a_1 has already been rightly taken into consideration, a correct guess for a_0 will lead to a good match for the *complete* region corresponding to the whole pair-point multiplication.

3.2 Attack alternatives varying the number of templates

We now discuss how the attack above may be implemented using a larger or a smaller number of templates. The attack strategy remains the same, but having a larger number of templates may increase our probabilities of success.

Attack using q templates. As explained in the beginning of this section, ideally our attack would work using only q templates. Here, the templates would allow us to extract

each coefficient in $\hat{\mathsf{a}}$ one by one. This would allow us to attack Kyber with only 3329 templates, which is a fairly small number for such an attack. Moreover, such an attack could potentially generalise to implementations of Dilithium [LDK+20] when collecting q traces for the (larger) Dilithium modulus. Namely, Dilithium actually performs complete NTTs on its polynomials and thus, multiplications are actually point-wise, and not pair-pointwise. Thus each secret coefficient is multiplied once, and then a modulus reduction is performed. In the Hamming weight model (see Section 4), this might not provide sufficient leakage (since Hamming leakage of k bits scales with $\sqrt{k}$), but the real-life leakage might nevertheless suffice to attack also Dilithium.

Attack using q^2 templates. As we have noted throughout this section, each pair-poitnt multiplication provides the result of two coefficients of the product $\hat{\mathsf{a}} \circ \hat{\mathsf{b}}$. Each pair-point multiplication involves two adjacent coefficients of $\hat{\mathsf{a}}$, which we have referred so far as a_0 and a_1 (see Equation 2). Therefore, we could actually build templates for each possible *pair* of coefficients a_0, a_1. When performing template matching, we will have many points of comparison between the templates and the target traces, since we will be comparing regions corresponding to the *complete* pair-point multiplication (similar to how we did in **Step 5** in Subsection 3.1). This increases thus our chances of successfully performing a key extraction.

Making templates for each possible pair of coefficients implies that we need a total of q^2 tamplates, which in Kyber translates to $3329^2 \approx 11M$ templates. While this number is much larger than what we considered initially, this attack strategy is very likely to work. Acquiring 11M traces may need several days. However such an attack complexity is still considered a real threat.

3.3 Attack on DPA-protected Kyber

We now explain how we can apply or extend our attack for targeting DPA-protected implementations of Kyber. We start by discussing our attack on masked implementations.

We can apply the exact same attack as described above on masked implementations of Kyber. Our templates and the corresponding template matching can help us recover each share of the secret key (exactly as described above). Once we have recovered all shares, we just need to add them to obtain the secret key.

Note that one target trace suffices since each share is used independently and sequentially. We are assuming here that in masked Kyber, we first multiply the ciphertext with share one and then multiply the ciphertext with share two (and so on in case of higher order masking). Such an assumption is very likely to hold for software implementations. For hardware implementations, there exists the possibility of performing some multiplications in parallel particularly since each pair of multiplications is performed independently of each other (see Subsection 2.2). However for performing two or more multiplications at once, the hardware design needs to count with two multiplier modules, and not all hardware designs of Kyber will be implemented as such since having extra multipliers may imply large costs in terms of area of the design.

Below we elaborate on how the chosen masking scheme may affect the complexity of our attack. The main takeaways are: if the target implementation uses a masking scheme with a modulus q, then the attack complexity and success probability are barely affected. However if the masked implementation operates on a modulus notably larger than q, the complexity increases linearly, and the success probability is also affected.

Masking schemes with modulus q. As explained in Section 2, masking schemes may vary on the modulus q they operate on. Let us first assume that we are attacking an implementation with a masking scheme which produces shares which all have coefficients

with values between 0 and $q-1 = 3328$. In this case we will be able to perform a key extraction using the same number of templates as for an unmasked implementation. Namely, the templates we need for attacking such a masked implementation correspond to multiplications between known coefficients (for our chosen ciphertext), and unknown coefficients with values between 0 and $q-1$. Thus after obtaining all q templates, we only need to perform the template matching twice with respect to an unmasked implementation (once for each share). The number of template *matchings* we perform increases linearly with the degree of the masking scheme. However, if we perform template matching over a power trace corresponding to the complete multiplication process involving both shares, we only need to perform the matching once for each template. For each $0 \leq j \leq q-1$, each match will reveal which coefficient in any of the two shares has a value equal to j.

Masking schemes with modulus $q' >> q$. Notably, the complexity of our attack increases if the masking scheme generates shares with coefficients with values between 0 and some q' which is notably larger than q. This is simply because we need to generate a corresponding number of q' templates. At the same time, we may have more collisions given the larger number of possible values.

3.3.1 Attack on shuffled implementations - distinguishing via the input ciphertext

Initially, one may think that a straightforward countermeasure against the attack proposed in this section is the randomised shuffling of the pair-point multiplications. Indeed, such pair-pointwise multiplications may be easily shuffled since each pair-point multiplication is independent and it doesn't really matter in which order they are computed, as long as the results are later placed on the correct coefficients of the resulting product. If we target a shuffled implementation of Kyber, our attack as described in Subsection 3.1 would allow us to extract all coefficients correctly, but we would not know the correct order in which they appear on the resulting polynomial. Nevertheless, we observe that our attack can be easily extended such that it is also effective on shuffled implementations, given only one target trace. We explain the attack steps below. The main idea is that we will use a chosen ciphertext whose coefficients all have a different and unique value. We will use such a ciphertext for generating our templates the same way as described before, obtaining thus q templates. Then, we will use the same ciphertext for obtaining our target trace. When performing template matching, for each template, we will try matching it a total of $\frac{n}{2}$ times, where for each try, we will shift the positions of each pair-point multiplication. Whenever we obtain some match, we will know the value of the operands for the chosen ciphertext and since each one of these operands is unique, we will know its original position and this will reveal the position of the extracted secret coefficient. The following description corresponds to an attack where we will first use q templates for extracting all coefficients a_{2i+1} (i.e. the coefficient a_1 within each pair-point multiplication), as we did in Subsection 3.1.

Generating the inputs. We choose an input ciphertext for which (on NTT domain) each of its coefficients has a unique value. That is, given the ciphertext $\hat{\mathsf{b}} = b_0, b_1, b_2, \ldots, b_{255}$, it holds that for each b_i, b_j, with $i \neq j, b_i \neq b_j$. For illustration purposes let us assume we choose $\hat{\mathsf{b}}$ as follows:

$$\hat{\mathsf{b}} = 9_0, 78_1, 1753_2, 7_3, \ldots, 17_{254}, 104_{255}.$$

Template building. We build templates in the exact same way as described in **Step 1** of Subsection 3.1. Thus, we obtain a total of q templates, each template for each possible

coefficient value. For a coefficient j, the templates will be of the form

$$T_j = (j_0 + j_1) \cdot (9_0 + 78_1), (j_2 + j_3) \cdot (1753_2 + 7_3), \ \ldots, \\ (j_{254} + j_{255}) \cdot (17_{254} + 104_{255}).$$

Obtaining the target trace. We obtain the target trace the same way as described in **Step 2** of Subsection 3.1, i.e. by providing our chosen ciphertext $\hat{\mathsf{b}}$ as input. Note however that this time our ciphertext (on NTT domain) consists of coefficients with unique values. Moreover, note that the resulting target trace corresponds to a shuffled evaluation of the pair-pointwise multiplications. For instance, the target trace might correspond to the following shuffled sequence of operations

$$T_t = (a_{22} + a_{23}) \cdot (b_{22} + b_{23}), (a_{104} + a_{105}) \cdot (b_{104} + b_{105}), \ldots, \\ (a_0 + a_1) \cdot (b_0 + b_1), (a_{56} + a_{57}) \cdot (b_{56} + b_{57}).$$

(Secret) coefficient extraction and location identification via template matching. We now proceed to match our templates with the target trace in a similar way as described in **Step 3** of Subsection 3.1 with some additional steps. For each template T_j we will perform a template matching with the target trace as follows.

1. We first test a matching with the template T_j and target T_t the same way as tested for our original attack. Let us assume we find a match at position i, revealing thus that the secret coefficient used at that position has the value j, i.e. $a_{2i+1} = j$. Let us remark that at this point of our analysis, the template T_j corresponds to a non-shuffled sequence of pair-point multiplications. Let us also recall that for generating the template and the target trace we used a ciphertext polynomial whose coefficients (on NTT domain) are all different from each other. Finally, let us recall that for obtaining a match, *all* input operands used within the analysed computations need to be the same. I.e. for a pair-point multiplication, the same b_0, b_1 and a_1 values need to be used in the template *and* in the target.

   Given the observations above, we know that if at this point we obtain a match at position i, then the original, non-shuffled position of the extracted coefficient in the secret key is at position i. The coefficients of our input ciphertext serve as orientation, since they are unique and we know their position in the template traces.

2. We will now try to find out whether a value j appears in some shuffled pair-point multiplication, and we will also find out *where* in the non-shuffled key the value j is located. For this, we start shifting the multiplication regions of our trace T_j. Concretely we will shift the positions of all pair-point multiplications. Thus for each template, there is a total of 128 shifts we can do since each template corresponds to 128 pair-point multiplications. Let w denote the number of shifts we do on a template and let $T_j^{>w}$ denote the template built for the coefficient value j and shifted a total of w times. For instance if we shift the multiplications only once, we obtain the template with the following form:

$$T_j^{>1} = (j_{254} + j_{255}) \cdot (b_{254} + b_{255}), (j_0 + j_1) \cdot (b_0 + b_1), \\ (j_2 + j_3) \cdot (b_2 + b_3), \ldots, (j_{252} + j_{253}) \cdot (b_{252} + b_{253})$$

3. Next we perform template matching with $T_j^{>w}$ and our target trace T_t. Let us assume we find a match at position i. The match tells us that the coefficient a_{2i+1} in the target trace has the value j. However since we know that the template $T_j^{>w}$

shifted the pair-point multiplications a total of w positions, we know that that it is actually the coefficient $a_{2(i-w)+1}$ in the (non-shuffled) secret key which has the value j.

4. We repeat the same matching + shifting process with all templates until we recover all coefficients. Recall that we are recovering all coefficients a_1 for each pair-point multiplication. Once we have recovered them, we can build a new set of q templates, by placing all recovered coefficients in their *shuffled* position and then just repeat the matching process from **Step 5** in Subsection 3.1. This will let us recover all coefficeints a_0 in each (shuffled) pair-point multiplication. Since in the previous step we learnt the original (non-shuffled) position of each pair-point multiplication, we will also know the original position of the extracted a_0 coefficients in the non-shuffled secret key.

4 Simulations and heuristic estimates

This section presents leakage simulation of our attacks (Section 3) on the implementation in [HKL+22, cod22] for Cortex-M4.

4.1 Implementation of pair-point multiplication

The implementation we are analysing implements the pair-pointwise multiplication as shown in Listing 1 and corresponds to the Karatsuba multiplication algorithm [KO63] (see Equation 4 for reference). The procedure first loads a *pair* of secret coefficients $a_0||a_1$ into a 32-bit register poly0 and a pair of public coefficients $b_0||b_1$ into a 32-bit register poly1. The coefficients a_0, a_1, b_0, and b_1 are 12-bit integers in $\{0, .., 3328\}$. In this overview, we skip over the instructions at lines 3 and 4 which are the analogous load operations for the next pair of coefficients in the key and in the ciphertext. Next in line 8, we multiply the top parts of the registers poly0 and poly1, obtaining a product corresponding to $a_1 \cdot b_1$. This product is a 24-bit result and it is stored in **tmp**. The value in **tmp** is then reduced mod 3329 (line 9). Listing 2 gives the code of the Montgomery subroutine and Appendix A explains why the deployed *Montgomery reduction* algorithm for mod 3329 computation induces 3 further operations on 28-bit values. Next, the result is multiplied by ζ (line 10), added to $a_0 \cdot b_0$ (line 11) and reduced mod 3329 via Montgomery reduction (line 12), resulting in the constant term $a_1 \cdot b_1 \cdot \zeta + a_0 \cdot b_1$ (cf. Equation 2). Next, the code computes the sum of the cross term $a_1 \cdot b_0 + a_0 \cdot b_1$ (line 14) and reduces it mod 3329 (line 15).

Listing 1: Instructions for pair-point multiplication.

```
1    ldr poly0, [aptr], #4
2    ldr poly1, [bptr], #4
3    ldr poly2, [aptr], #4
4    ldr poly3, [bptr], #4
5
6    ldrh zeta, [zetaptr], #2
7
8    smultt tmp, poly0, poly1
9    montgomery q, qinv, tmp, tmp2
10   smultb tmp2, tmp2, zeta
11   smlabb tmp2, poly0, poly1, tmp2
12   montgomery q, qinv, tmp2, tmp
13
14   smuadx tmp2, poly0, poly1
15   montgomery q, qinv, tmp2, tmp3
```

Listing 2: Montgomery subroutine.

```
1  .macro montgomery q, qinv, a, tmp
2    smulbt \tmp, \a, \qinv
3    smlabb \tmp, \q, \tmp, \a
4  .endm
```

4.2 Hamming weight model

We analyze leakage in the *Hamming weight* model which counts the number of ones in a state. The assumption in this model is that the power consumption of a device is correlated with the Hamming weight of the states during computations. In our analysis, we will check whether each possible secret coefficient $a_i \in \{0, .., 3328\}$ (or each possible pair of coefficients) leads to unique hamming weight values during the pair-point multiplication. If this is the case, then we expect that the leakage coming from a pair-point multiplication will allow us to identify the value of the secret coefficient(s) used within that pair-point multiplication. We present both heuristic estimates and simulations for Hamming weight.

For a heuristic estimate, we calculate the *expected information* that we obtain from the Hamming weight of a uniformly random k-bit string. Namely, $|\log \Pr[\mathrm{HW} = i]|$ is the number of bits of information which we weigh by the probability of obtaining a state with hamming weight i, leading to the expected information (or *Shannon Entropy*)

$$H(k) := \sum_{i=0}^{k} \Pr[\ \mathrm{HW} = i] \cdot |\log\left(\Pr[\ \mathrm{HW} = i]\right)| = \sum_{i=0}^{k} \frac{\binom{k}{i}}{2^k} \left| \log\left(\frac{\binom{k}{i}}{2^k}\right) \right|$$

for a uniformly random k-bitstring. Asymptotically, the expected information $H(k)$ grows linearly in $\sqrt{k}$. For $k = 24$ and $k = 28$, we have $H(24) = 3.34$ and $H(28) = 3.45$.

Recall that our attack using $q+q$ templates (see Subsection 3.1) first extracts a_1 before extracting a_0. Concretely, the five operations up to and including line 10 in Listing 1 only depend on a_1. They first write a 24-bit value for multiplication of a_1 and b_1, then three 28-bit values in the Montgomery reduction (cf. Appendix A) and then another 24-bit value for multiplication of $a_1 \cdot b_1 \cdot \zeta$. We obtain an overall expected information of

$$\begin{aligned} &H(24) + H(28) + H(28) + H(28) + H(24) \\ =&2 \cdot H(24) + 3 \cdot H(28) \approx 2 \cdot 3.34 + 3 \cdot 3.45 = 6.68 + 10.35 = 13.69 \end{aligned}$$

bits leakage about a_1 only. Since a_1 is a 12-bit value, it is plausible that we extract a_1 correctly with good probability from these five operations, even if not always, since 13.69 bits is only slightly above 12 bits and the random variable is concentrated around its expectation rather than exactly at its expectation.

To extract *both* values a_0 and a_1, we have two further Montgomery reductions (line 12 and line 15), each resulting in 3 more operations, leaking together $6 \cdot H(28) \approx 20.7$ additional bits of information and the computation and addition of cross terms in line 14 which generates another H(24)-bit value, leading to an overall expected leakage of

$$13.69 + 20.7 + 3.34 = 37.73$$

bits to extract a $12 + 12 = 24$-bit value (a_0, a_1), suggesting that an attack trying out all *pairs* should succeed with very high probability. We now turn to our simulation and their results which corroborate our analyses.

4.3 Simulations in the Hamming weight model

We now provide simulations for the operations within the pair-point multiplications. In our simulations we first focus on the first 5 instructions of the pair-point multiplication (see previous subsections). Our simulations calculate which coefficients $a_{2i+1} \in [0, \ldots, q-1]$ have *unique* combinations of hamming weight values (hamming weight tuples) during these instructions. Recall from Equation 4 (and from the listings in this section), that on the pair-point multiplication we have $a_1 b_1 \zeta$. The value of ζ changes for each pair-point multiplication. So for our simulations, we will initially fix ζ_0 and try out all possible values for a_1 and all possible values b_1. We will obtain the average probability that a value for

a_1 leads to a unique hamming weight tuple. Then, we change to ζ_1 and iterate over all possible values for a_3 and all possible values for b_3. We continue this process, obtaining the averages for all a_{2i+1}, given all ζ_i. We thus obtain probabilities for extracting each odd coefficient, given a random ciphertext. Observe that in our simulations we do not consider micro-architectural aspects, like instruction pipelining, of our target. In the next section, we provide partial experimental evidence of the success of our attack.

As we will show, most of the values for an odd coefficient indeed lead to unique hamming weight tuples. Only a small fraction of coefficients have collisions. On average, 3031 of these values have unique hamming weight tuples, i.e. there exist 3031 hamming weight tuples which map to exactly one coefficient value. 259 coefficients lead to 2-way collisions. This means that there exist $259/2 \approx 130$ hamming weight tuples which map to exactly two different coefficient values. Subsequently, there exist 34 coefficients which have 3-way collisions and 4 coefficients which have 4-way collisions each. The exist (on average) only 0.03125 tuples which map to more than 4 different coefficient values. We now go on more detail about the results of our simulations. *Extracting odd coefficients (a_{2i+1}).* Our simulations show that for a uniformly *random* b_{2i+1}, the probability of extracting a_{2i+1} from the first 5 instruction is ≈ 0.90. This means that given a random ciphertext, we have good chances of extracting each odd coefficient. The probability of obtaining *two* possible candidates for each odd coefficient is ≈ 0.085, and the probability of obtaining three possible candidates for each odd coefficient is ≈ 0.011. Thus, taking a union bound, we obtain that the probability that a given a_{2i+1} has either a unique hamming weight tuple, or a 2- or 3-way collision is ≈ 0.996. For this reason in the rest of this analysis we will only consider the case that we are dealing with coefficients with unique hamming weight tuples, or with 2- or 3-way collisions.

Table 1: Simulation results. Under Number of Matches we see the probabilities that given each ζ_i we obtain 1-, 2- or 3-way collisions. The upper part of the table corresponds to probabilities of extracting odd coefficients, given q templates. The lower part corresponds to probabilities of extracting pairs of coefficients, given q^2 templates.

Nr. of templates	**Root**		**Number of Matches**		
			1	**2**	**3**
q-templates	ζ_0	2226	0.8696	0.108	0.018
	ζ_1	-2226	0.9344	0.0603	0.0042
	ζ_2	430	0.8688	0.1087	0.0178
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	ζ_{126}	1628	0.8715	0.1067	0.0173
	ζ_{127}	-1628	0.9329	0.0615	0.0044
q^2-templates	ζ_0	2226	0.9974	0.0025	1.01×10^{-5}
	ζ_1	-2226	0.9973	0.0026	7.1474×10^{-6}
	ζ_2	430	0.9978	0.0021	4.6282×10^{-6}
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	ζ_{126}	1628	0.9973	0.0027	7.4805×10^{-6}
	ζ_{127}	-1628	0.9976	0.0024	5.5263×10^{-6}

In the table under **Number of Matches (1)**, we see the probability that each odd coefficient $a_1, a_3, ..., a_{255}$ has a unique hamming weight tuple. We calculate this probability over all $b_1 \in [1, \ldots, q-1]$, and note that the probability is dependent on the value of ζ. Thus, the probability that a_1 has a unique hamming weight tuple is different from that of a_3, a_5, etc, but the probability is always between 0.801 and 0.937, with an average of 0.90. Under **Number of Matches (2) and (3)** we see the analogous probabilities

that each odd coefficient a_{2i+1} has a hamming weight tuple with a 2- and 3-way collision correspondingly.

We recall that in our attack using $q + q$ templates (cf Subsection 3.1), we use the first set of q templates for extracting the odd coefficients. According to our results, we should have a 90% chance of correctly extracting each odd coefficient - but we should recall that in Kyber, the secret keys consist of polynomials of degree 255. Thus the probability of extracting *all* odd coefficients correctly is notably smaller. In fact, if we consider all probabilities of Table 1 for the chances that each odd coefficient has a unique hamming weight tuple, we get a probability of

$$\Pi_{i=0}^{127} p_i \approx 1.2967 \times 10^{-6}$$

of extracting all odd coefficients from one polynomial, given only q templates. In the end of this section however, we will explain how we can use the results of our simulations for outlining an attack strategy which easily increases our success probabilities, with just a linear increase on the number of templates needed.

Extracting coefficient pairs (a_{2i}, a_{2i+1}). The lower part of Table 1 gives the probabilities that each secret coefficient pair leads to a unique hamming weight tuple. We obtain these probabilities in an analogous way as we did for the odd coefficients. Thus, the probabilities for each pair $(a_0, a_1), (a_2, a_3)(a_4, a_5), \ldots, (a_{254}, a_{255})$ are different as they are dependent on ζ. Note that in this case, the hamming weight tuples consist of more values since we are considering *all instructions* within one pair-point multiplication. Hence the very high probabilities under **Number of Matches (1)**. We can conclude from these results that if we create templates for all possible pairs of secret coefficients, our success probabilities are fairly high (we also provide experimental evidence for this variation of our attack in the next section). However, we should recall that creating templates for all possible coefficient pairs constitutes to creating a total of q^2 templates.

Efficiency Optimizations. While q^2 is a reasonable number of template traces, collecting all of them is still quite consuming. Thus, we may indeed try extracting all odd coefficients first and then extracting all even coefficients with an additional set of templates. From the discussions above we can conclude that our success probabilities of running a $q + q$ attack are not as high as we would originally hope (for the mkm4 implementation in the Hamming weight model). However, the simulation results suggest a natural and very simple way of optimising the success of the attack. In the following we outline an attack adaptation which increases the success probability of our attack and only requires a linear increase in the number of templates.

First, we can perform a template matching using q templates (as originally done in Subsection 3.1). For each coefficient we are trying to extract, we rank the top 3 candidate values for which we get the best matches. Now we build templates for extracting the even coefficients. We will create 3 versions of these templates. In each version we use a different top 3 candidate for each odd coefficient, creating thus an additional set of $3q$ templates. Thus, we first determine the top three candidates for each a_{2i+1} (with high probability), and then try all three of them in combination with all possible a_{2i}, leading to an overall number of $q + 3q$ templates. When trying to extract the even coefficients, we get a very high success rate *iff* we are using the correct odd coefficient a_{2i+1}. Namely as we see in Table 1, each secret coefficient pair has a very high probability of having a unique hamming weight tuple.

We can even optimise our attack further by considering top 4 match candidates for each coefficient, generating thus an additional set of $4q$ templates. Concretely for the optimised attacks using $q + 3q$ and $q + 4q$ templates, we obtain success probabilities of

$$\Pi_{i=0}^{127} p_i \approx 0.6755 \text{ and } \Pi_{i=0}^{127} p_i \approx 0.875$$

respectively. With $6q = 19974$ templates, we have a very high success probability of 0.944, given a single target trace and a random ciphertext.

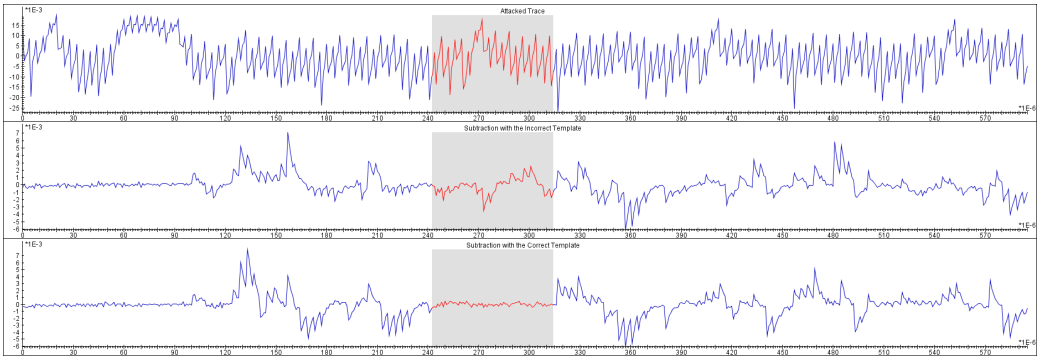

Figure 1: Leakage characterization: target trace with marked pair-point multiplication (top), subtraction of the target trace from an incorrect template (middle), and subtraction of the target trace from the correct template.

5 Experimental evidence

In this section we provide partial experimental results for the simplest, but most expensive, variation of our attack. That is, we assess the success of the attack described in Subsection 3.2 under **Attack using q^2 templates**, but we only generate and test a subset of templates, instead of generating all 11M templates. We want to assess whether we are able to identify the values of *pairs* of secret coefficients from the target trace, and this turns out to be very easy. As we see, we can simply calculate the difference between the target trace and a template, and the resulting *difference trace* will tell us whether the template matches or not. Namely if the pair of secret coefficients used in a template match with the pair of secret coefficients used within *some* pair-point multiplication in the target trace, then we should see a region in the difference trace with values very close to zero. We first describe our experimental setup and then elaborate on our results.

Just as in the previous section, we target the masked implementation of Kyber from the mkm4 repository, described in [HKL+22], and we use the same experimental setup as described in that paper. That is, our experiments are performed on the ChipWhisperer Lite platform with an STM32F303 target [OC14] featuring an Arm Cortex-M4 core. This allows easy verification with a standardised platform, and leads to low noise level and well aligned traces. We will concentrate on the function `poly_basemul` from the implementation, as this is the part of the program executing the pair-pointwise multiplication.

We obtain 10000 templates for randomly generated pairs of coefficients, including some pairs which actually appear in the secret key we are trying to extract. Each template is built using exactly one trace. Thus, for this experiment we generate exactly 10001 traces (10 000 templates plus the target trace).

Figure 1 displays our simple method for visualising leakage. We follow the approach described in [HKP+12b], which as mentioned before, simply consists on calculating the difference between one template and our target trace (see Figures 3 and 4 in [HKP+12b]). The trace at the top of Figure 1 corresponds to our target trace, the area highlighted corresponds to the calculation of one pair-point multiplication. The trace in the middle corresponds to the resulting trace when we subtract the target from a template which *does not* match the secret coefficients used within the pair-point multiplication highlighted. In the bottom we see a trace corresponding to the difference between the target and a template which uses the correct pair of secret coefficients. As we can see, the region highlighted for this last trace has sample values very close to zero.

Whenever we compare a template for a pair of coefficients which appear on the key of the target trace, our difference trace contains some region consisting of sample values very

close to zero (as in the bottom of Figure 1). Whenever we try comparing a template for a pair of coefficients which *do not* appear on the key, our difference trace does not have any region resembling to the highlighted region at the bottom of Figure 1. In other words, this partial experiment has a perfect success rate.

6 Countermeasures

The standard countermeasures of masking or shuffling the polynomial multiplication in Kyber do not seem to be effective for protecting against the type of template attack we present in this paper. In the following, we discuss possible countermeasures which, to the very least, should impose significant obstacles for the success of our attack.

Shuffling of the multiplication steps. One possible countermeasure may be the random shuffling of the operations performed *within* each pair-point multiplication. This would make our template matching steps more difficult since the operation sequence in our templates may not align with the sequential operation of the pair-point multiplication. However, if the pair-point multiplication is optimised and implemented via Karatsuba, there are not many different ways in which the operation sequence can be permuted while maintaining correctness (see the listings in Section 4).

Masking schemes with larger modulus. As discussed in Subsection 3.3, masking schemes which generate shares with coefficients with much larger values would certainly make our attack more difficult. Such schemes would imply an increase on the number of templates needed for our attack, and the chances of getting false positive matches would increase as well. Unfortunately, such masking schemes would clearly imply an increase in the usage of computational resources (e.g. memory and stack usage) and the online complexity of Kyber.

Parallelisation of the pair-point multiplication. Parallelising several of the pair-point multiplications prevents a straightforward application of our attack. Namely, the parallelisation forces us to recover several coefficients simultaneously, so that the *complexity* of our attack is squared when running 2 parallel threads and quadrupled when running 4 parallel threads. The *success probability*, in turn, is expected to decrease, since the expected information increases sub-linearly. Concretely, with 2 threads, the implementation would leak from 56-bit values, whose expected information leakage is ≈ 3.95, which is less than two times the expected information from 28-bit values which is $\approx 2 \cdot 3.45 = 6.9$. With 4 threads, the implementation would leak from 112-bit values, whose expected information leakage is ≈ 4.45 which is less than four times the expected information from 28-bit values which is $\approx 4 \cdot 3.45 = 13.8$. As already discussed however, performing multiplications in parallel seems out of scope for constrained devices as the integration of an additional multiplier entity would imply a big cost in terms of size.

***Complete* NTT and actual point-wise multiplication.** Certainly, some of our attack strategies cannot be applied on schemes which implement a complete NTT to its polynomials and then multiplies them in proper point-wise fashion. For instance, our attack using q^2 templates does not work any more since two adjacent coefficients will be processed in independent multiplications. We could still try applying our simplest attack using only q templates, and the success would be dependent on how much leakage we obtain from one integer multiplication plus one modular reduction. If that sequence of operations leads to enough leakage, we could extend our attack for instance to Dilithium [LDK+20], which was selected as a post-quantum candidate signature scheme. Dilithium performs a *full* NTT and performs a point-wise multiplication.

Acknowledgements. Estuardo Alpirez Bock conducted part of this research while at Aalto University. His work at Aalto and the work from Kirthivaasan Puniamurthy were supported by MATINE, Ministry of Defence of Finland.

References

[ABD+17] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber (version 1.0) – submission to round 1 of the NIST post-quantum project. submission to the NIST post-quantum cryptography standardization project, 2017. https://pq-crystals.org/kyber/data/kyber-specification.pdf.

[ABD+20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber (version 3.0) – submission to round 3 of the NIST post-quantum project. submission to the NIST post-quantum cryptography standardization project, 2020. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[BCP+14] Lejla Batina, Lukasz Chmielewski, Louiza Papachristodoulou, Peter Schwabe, and Michael Tunstall. Online template attacks. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*, volume 8885 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2014.

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *ACM J. Emerg. Technol. Comput. Syst.*, 17(2):10:1–10:26, 2021.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):173–214, 2021.

[BNGD22] Linus Backlund, Kalle Ngo, Joel Gärtner, and Elena Dubrova. 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.

[cod22] Github respository for masked kyber presented in [HKL+22], 2022. https://github.com/masked-kyber-m4/mkm4.

[DNG22] Elena Dubrova, Kalle Ngo, and Joel Gärtner. Breaking a fifth-order masked implementation of crystals-kyber by copy-paste. Cryptology ePrint Archive, Paper 2022/1713, 2022. https://eprint.iacr.org/2022/1713.

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes.

In *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*, TIS'19, page 2–9, New York, NY, USA, 2019. Association for Computing Machinery.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *J. Cryptol.*, 26(1):80–101, 2013.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):88–113, 2021.

[HKL+22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked kyber on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2022/058, 2022. https://eprint.iacr.org/2022/058.

[HKP+12a] Michael Hutter, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Stefan Mangard. Exploiting the difference of side-channel leakages. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[HKP+12b] Michael Hutter, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Stefan Mangard. Exploiting the difference of side-channel leakages. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 1–16, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[HMA+08] Naofumi Homma, Atsushi Miyamoto, Takafumi Aoki, Akashi Satoh, and Adi Shamir. Collision-based power analysis of modular exponentiation using chosen-message pairs. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2008.

[JWN+22] Yanning Ji, Ruize Wang, Kalle Ngo, Elena Dubrova, and Linus Backlund. A side-channel attack on a hardware implementation of crystals-kyber. Cryptology ePrint Archive, Paper 2022/1452, 2022. https://eprint.iacr.org/2022/1452.

[Kan22] M. J. Kannwischer. *Polynomial Multiplication for Post-Quantum Cryptography*. PhD thesis, Nijmegen U., 2022.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595, January 1963.

[LDK+20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-Dilithium, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: truly fast NTRU using NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):180–201, 2019.

[MBB+22] Catinca Mujdei, Arthur Beckers, Jose Bermundo, Angshuman Karmakar, Lennert Wouters, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. Cryptology ePrint Archive, Report 2022/474, 2022. https://eprint.iacr.org/2022/474.

[MKK+23] Soundes Marzougui, Ievgen Kabin, Juliane Krämer, Thomas Aulbach, and Jean-Pierre Seifert. On the feasibility of single-trace attacks on the Gaussian sampler using a CDT. In Elif Bilge Kavun and Michael Pehl, editors, *Constructive Side-Channel Analysis and Secure Design - 14th International Workshop, COSADE 2023, Munich, Germany, April 3-4, 2023, Proceedings*, volume 13979 of *Lecture Notes in Computer Science*, pages 149–169. Springer, 2023.

[NWDP22] Kalle Ngo, Ruize Wang, Elena Dubrova, and Nils Paulsrud. 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.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 243–260. Springer, 2014.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked Ring-LWE implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2019.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 513–533. Springer, 2017.

[RCDB22] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. Cryptology ePrint Archive, Paper 2022/737, 2022. https://eprint.iacr.org/2022/737.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-lwe masking. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography*, pages 233–244, Cham, 2016. Springer International Publishing.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based PKE and kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020.

[RRdC+16] Oscar Reparaz, Sujoy Sinha Roy, Ruan de Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking ring-lwe. *J. Cryptogr. Eng.*, 6(2):139–153, 2016.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *IACR Cryptol. ePrint Arch.*, page 39, 2018.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of cca-secure key exchange mechanism CRYSTALS-Kyber on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):328–356, Feb. 2021.

[ZXZ+18] Shuai Zhou, Haiyang Xue, Daode Zhang, Kunpeng Wang, Xianhui Lu, Bao Li, and Jingnan He. Preprocess-then-NTT technique and its applications to kyber and newhope. In Fuchun Guo, Xinyi Huang, and Moti Yung, editors, *Information Security and Cryptology - 14th International Conference, Inscrypt 2018, Fuzhou, China, December 14-17, 2018, Revised Selected Papers*, volume 11449 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 2018.

A Montgomery reduction

Kyber represents elements in Montgomery representation in order to avoid expensive division by q and computation mod q and replace it by division by 2^{16} (taking the top half of a register) and computation mod 2^{16} (taking the bottom half of a register). In the following, we present the Montgomery reduction with general R and q, but Kyber indeed uses $R = 2^{16}$.Consider $R = 2^k > q$, and an element $a < qR$. To reduce the memory footprint we can store a/R and this reduces the element a by k bits, and it can be efficiently implemented. In Montgomery domain, the idea is to make sure that the element a is a multiple of R by introducing a correction step. More precisely, imagine that we want to find a value t, such that, $a - tq$ is divisible by R. To bring the element to the Montgomery domain, one computes t as $aq^{-1} \pmod R$ in a way that $a - aq^{-1}q \pmod R = 0$. Following closely Section 2.3.2 in [Kan22], Algorithm 6 shows the case of signed Montgomery reduction from [Sei18].

Algorithm 5: Montgomery reduction

Input: modulus q, $R = 2^n > q$, $q^{-1} \bmod (R)$, $a \in \mathbb{Z}$ such that $a < qR$
Output: $t \equiv aR^{-1} \pmod q, 0 \leq t \leq 2sq$

$t \leftarrow a(-q^{-1}) \bmod (R)$
$t \leftarrow (a + tq)/R$
s **return** t

Algorithm 6: Signed Montgomery reduction from [Sei18]

Input: modulus q, $R = 2^n > q$, $q^{-1} \bmod^{\pm} (R)$, $a \in \mathbb{Z}$ such that $a < qR$
Output: $t \equiv aR^{-1} \pmod q, |t| \leq q$

$t \leftarrow aq^{-1} \bmod^{\pm} (R)$
$t \leftarrow (tq)/R$
$t \leftarrow \lfloor a/R \rfloor - t$
return t

To better illustrate the Montgomery representation and reduction, we can show an Example 1. The example is directly from [Kan22].

Example 1. Let $R = 2^{16}$, and that all our elements are in $\mathbb{Z}/q\mathbb{Z}$ for $q = 3329$. The $q^{-1} \bmod (R) = 3327$, the $\cdot$ is the integer multiplication. For the example, we will multiply two numbers $a = 1234$ and $b = 17$. First we bring the elements to the Montgomery domain by computing $a' = 1234 \cdot R \bmod^{\pm} (q) = 27$ and $b' = 17 \cdot R \bmod^{\pm} (q) = 2226$. Then, we compute the multiplication $a' \cdot b' = 60102$. Now, we apply the Montgomery reduction and $t = (60102 \cdot 3327) \bmod (R) = 9018$, and $(a' \cdot b' + t \cdot q)/R = (60102 + 9018 \cdot 3329) = 459$ which equals $abR \bmod^{\pm} (q)$. To bring the result to the "normal" domain, we need to apply the Montgomery reduction again, $t = 19765$ and $(459 + 19765 \cdot 3329)/R = 1004$ which is the result we were looking for. However, it is enough to transform one multiplication into the Montgomery domain, e.g. $b' = 17 \cdot R \bmod (q) = 226$. After multiplication we have $ab' = 2746884$ and apply the Montgomery reduction, $t = 18940$ and $(ab' + tq) = (2746884 + 18940 \cdot 3329)/R = 1004$.

We now provide more details on how we determined the length of values for the Hamming weight that we use in our numerical estimates in Section 4.2.

(1) $a_1 \cdot b_1$	$12 + 12 = 24$ bits
take bottom of register	16 bits
then multiply by q_{inv}	$\lvert q_{\text{inv}} \rvert = 12$ bits
(2) $(a_1 \cdot b_1)_B \cdot q_{\text{inv}}$	$16 + 12 = 28$ bits
take bottom of register	16 bits
then multiply by q	$\lvert q \rvert = 12$ bits
(3) $((a_1 \cdot b_1)_B \cdot q_{\text{inv}})_B \cdot q$	$16 + 12 = 28$ bits
add $(a_1 \cdot b_1)$	$\lvert a_1 \cdot b_1 \rvert = 24$ bits
(4) $((a_1 \cdot b_1)_B \cdot q_{\text{inv}})_B + (a_1 \cdot b_1)$	$\max\{24, 28\} = 28$ bits
take **top** of register and call it c	$\lvert c \rvert = 12$ bits
(5) $c \cdot \zeta$	$12 + 12 = 28$ bits