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,kirthivaasan.puniamurthy}@aalto.fi
[4] Masaryk University, Czech Republic {chmiel,xsorf}@fi.muni.cz

Abstract. We introduce a novel template attack for secret key recovery in Kyber, leveraging side-channel information from polynomial multiplication during decapsulation. Conceptually, our attack exploits that Kyber's incomplete number-theoretic transform (NTT) causes each secret coefficient to be used multiple times, unlike when performing a complete NTT.

Our attack is a single trace *known* ciphertext attack that avoids machine-learning techniques and instead relies on correlation-matching only. Additionally, our template generation method is very simple and easy to replicate, and we describe different attack strategies, varying on the number of templates required. Moreover, our attack applies to both masked implementations as well as designs with multiplication shuffling.

We demonstrate its effectiveness by targeting a masked implementation from the *mkm4* repository. We initially perform simulations in the noisy Hamming-Weight model and achieve high success rates with just 13 316 templates while tolerating noise values up to $\sigma = 0.3$. In a practical setup, we measure power consumption and notice that our attack falls short of expectations. However, we introduce an extension inspired by known online template attacks, enabling us to recover 128 coefficient pairs from a single polynomial multiplication. Our results provide evidence that the incomplete NTT, which is used in Kyber-768 and similar schemes, introduces an additional side-channel weakness worth further exploration.

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. E. Alpírez Bock conducted part of this research while at Aalto University. His work at Aalto and the work of K. Puniamurthy were supported by MATINE, Ministry of Defence of Finland. The work of Ł. Chmielewski and M. Šorf was supported by the Ai-SecTools (VJ02010010) project. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. Date of this document: 2024-04-05.

1 Introduction

NIST selected Kyber [9,4] to be standardized as a post-quantum secure key encapsulation mechanism (KEM) after a rigorous competition. The primary security requirement of the NIST competition is achieving message confidentiality against chosen-plaintext (CPA) and chosen-ciphertext attacks (CCA) based on plausibly post-quantum hard problems. Additionally, the competition emphasizes the resistance of implementations to side-channel attacks. This paper builds upon the previous research exploiting differences in side-channel traces based on the chosen inputs [21,20,7] to design a new single-trace template attack against masked Kyber implementations. In particular, we target the decapsulation phase, leveraging templates to extract the long-term secret key from the polynomial multiplication process. Our goal is to show that in this context, masking is not sufficient protection, even considering relatively simple attacks.

Kyber's key encapsulation (encryption) performs a matrix-vector multiplication in the ring of polynomials $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256}+1)$ and then adds a small noise vector to the result. In turn, Kyber's decapsulation (decryption), multiplies 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 $q = 3329$. Kyber turns this core IND-CPA-secure scheme into IND-CCA-secure encryption using the Fujisaki-Okamoto (FO) transform [16]. Black-box security against IND-CCA security, however, does not protect against known/chosen ciphertext *side-channel* attacks, since the input ciphertext is always multiplied with the secret key right at the beginning of the decapsulation process, cf. [14,35,17,5].

Number theoretic transform. The 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 $\hat{\mathsf{a}} \circ \hat{\mathsf{b}} = (a_0 \cdot b_0, a_1 \cdot b_1, \ldots, a_{n-1} \cdot b_{n-1})$. However, at this point 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 in the effectiveness of our template attack since, per each pair-point multiplication, the coefficients are multiplied more than once, thus providing 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 the rounding operation required for the 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 [3], the selected modulus q for Cyber 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 for performing *incomplete* NTTs and calculating the multiplication in a *pair-pointwise* fashion [45,26]. Using this smaller modulus allows for a more compact implementation of Kyber since we need one bit less to encode each coefficient of a ciphertext and secret key. Moreover, this new modulus still allows fast multiplications between the NTT representations of the polynomials to be performed. 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, 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 the 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 $(a_0 + a_1 x)(b_0 + b_1 x), \ldots, (a_{254} + a_{255} x)(b_{254} + b_{255} x)$.

Unfortunately, as we will show in this work, pair-pointwise multiplications also lead to more leakage in the presence of side-channel adversaries. Consider 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 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 to identify the value of the secret coefficients.

1.1 Our contribution

We propose an attack on the pair-pointwise multiplication of Kyber-like implementations and start by observing that Kyber executes more secret-dependent operations than lattice-based schemes, which perform a full NTT:

1. Instead of *one* multiplication (as in full NTT), in pair-point multiplications, *three* multiplications (cf. Equation (3)) 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 $a_i \in [0, \ldots, q-1]$ are 12-bit integers, the registers operate on 24-bit and 28-bit integers before the modulus reduction. Thus, in the Hamming weight model, the expected information per instruction is $H(24) \approx 3.34$ and $H(28) \approx 3.45$ bits of information rather than only $H(12) \approx 2.84$.

Starting from these observations, we devise an attack which extracts each coefficient from a pair-point multiplication individually. We also present variations of our attack, changing the number of templates needed. All our attack strategies 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 1) and (b) each multiplication in Equation 1 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 half of its coefficients equal to (e.g.) 2649, and the other half equal to 317,[5] 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

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

Note that we perform the operations described in Equation 1 for each multiplication. Moreover, each multiplication is performed sequentially, and the result

[5] 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.

of one multiplication does not affect any other . Suppose the secret $\hat{\mathsf{a}}$ in the target device has some coefficient a_i equal to 328. In that case, 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 1), or at the points where it was multiplied by 317 ($a_i \cdot b_1$ in Equation 1) 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 an 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 low correlations at all other regions in the trace (unless a coefficient with a value of 328 appears elsewhere). Namely, in all other regions of the traces, the power consumption corresponds to multiplications between different operands; 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, we all need to 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 need to transform them back to their standard domain, which will let us recover a. Moreover, this same attack strategy will also allow us to attack masked implementations since we 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, 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. Thus, we devise an attack requiring $q+q$ templates. Finally, we explore an extension of our attack that extracts pairs of coefficients from each pair-point multiplication via q^2 templates but has a much higher success probability given that the templates target complete regions of pair-point multiplications and thus have more samples for comparison with the target trace. We refer the reader to Section 3 for the specific steps of our attack and its adaptations. Then, we validate our attacks against the masked implementation of [2], first via simulations and then via experiments with power consumption measurements. We show how our attack strategy requires a single target trace from a known ciphertext and avoids complex attack methods like machine learning since it succeeds by performing simple correlation analysis.

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, we should see enough leakage to uniquely determine the value of a secret coefficient processed during 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 positive evidence of our chances of extracting odd secret coefficients. Later in the paper, we explain how to use the extracted odd coefficients to extract all even coefficients. Finally, our simulations show that a template attack with $100q$ templates succeeds with the probability ≥ 0.999 even in the presence of Gaussian noise with standard deviation $\sigma \leq 0.87$.

Experimental results. We perform a power analysis attack also on the masked implementation of Kyber [2] using the ChipWhisperer Lite platform [31]. We detect leakage for both $q{+}q$ and q^2 attacks, but unfortunately it is not enough to recover a pair of coefficients from a pair-point multiplication. We show that the low success of these experiments is influenced by microarchitectural aspects and the implementation we target: essentially, the power profile of a pair-point multiplication is slightly influenced by the operations done before it started [6].

However, the success rate, especially for the q^2 attack, is quite promising and therefore, to make the attack work we come up with an extension inspired by the Online Template Attack (OTA), originally used to attack elliptic curve cryptography [6,7]. OTA is a powerful technique residing between horizontal and template attacks with the main distinctive characteristics of building the templates after capturing the target trace and not before. The combined attack works as follows: first we reduce the number of candidate templates using the q^2 attack and then we launch iteratively OTA to limit the microarchitectural noise. This way we are able to recover all the coefficients of 128 pair-pointwise multiplications. In particular, we completely recover all coefficients for 3 attacked target traces at the cost of maximum 43M templates. While these numbers are high, they are required to recover all the coefficients from a single trace.

We also estimated how many templates we need to attack masked Kyber768 with the order 2. Here we need more templates since such implementation uses 6 full polynomial multiplications. For such attack we would need 78M to achieve 43% success rate and to increase it to 90% we need approximately 105M traces.

With respect to the experiments it is also an interesting question whether our experiments may provide better results if we use electro-magnetic emanations

[6] For details the attacks and the experiments see Section 5.

as the side-channel information instead of power consumption (see for instance [38,32] for EM attacks on real life implementations). It would be also interesting to see whether we can lower the number of used templates. We leave these investigations as future work.

1.2 State of the art

Attacks on the polynomial multiplication of Kyber were successfully performed using correlation power analysis techniques [28]. However, early proposals recognized the need to apply masking to the polynomial multiplication in lattice-based schemes as a countermeasure against side-channel analysis [30,36,37]. Consequently, many research efforts have focused on attacking other components of the Kyber decapsulation process. Primas, Pessl, and Mangard introduced a template attack on the inverse NTT during decryption, enabling them to recover a decrypted message and subsequently extract the session key [34]. This attack leverages belief propagation for template matching and has since been extended and improved in subsequent works [33,17]. In a different approach, Dubrova, Ngo, and Gärtner propose the use of deep learning techniques to recover the message and subsequently extract the long-term secret key [15] from the re-encryption step of decapsulation. Notably, research in this area has demonstrated the success of deep learning in attacking lattice-based schemes [5,22,29,27]. Further SCA attacks on masked implementations of Kyber were presented on the message encoding [41] and on the arithmetic-to-boolean conversion step [42]. Note that all works cited above attack parts of Kyber other than the pointwise multiplication.

Our attack differs from the previous attacks in two significant ways when applied to masked implementations: we directly extract the long-term secret key from pointwise multiplication and we do not require deep learning or belief propagation for template construction and matching. Although machine learning (ML) techniques were shown to be particularly successful again post-quantum schemes, for example, in [15], we prefer a more classical approach based on Pearson correlation matching due to the following reasons: (1) the attack description is simpler, (2) the attack is easier to replicate since the adversary does not require the knowledge of ML, (3) it is easier to explain where the leakage comes from and thus come up with countermeasures, and (4) crucially we wanted to show that (even) classic side-channel methods effectively extract the key from masked Kyber.

In parallel to this work, the authors of [44] also developed a template attack on the polynomial multiplication in Kyber. Their attack targets key generation and encryption. It thus exploits the fact that each secret polynomial is multiplied with k different values in the matrix A (see Algorithm 1), leading thus to more side-channel leakage for each secret polynomial. Their approach relies on Hamming Weight templates for multiple intermediates and thus utilizes a key enumeration, similar to belief propagation. Unfortunately, the current manuscript does not provide many details about template building and matching, including how many templates their attack requires. Therefore, currently it is difficult to fairly compare our attacks to theirs. We do observe from their results that the complexity of their attack grows drastically as the dimension k of Kyber decreases (see Table 5 in [44]). In fact, the smaller k, the smaller the probability of success for their attack. This is because their attack relies on multiple polynomial multiplications for each secret polynomial, and thus, the larger k (with $k = 2, 3$ or 4), the more multiplications are performed during key generation. Our attack, on the other hand, does not have such dependencies on the size of k. The complexity of our attack increases slightly with the size of k since there are more coefficients to extract, but the attack steps and their success probability remain basically the same.

Following our remarks above, we should point out that our attack can also be applied to target the key generation in Kyber, since we just require one target trace and *knowledge* of the operand coefficients (coefficients of the polynomials of the public value A). On the other hand, it is unlikely that the attack from [44] succeeds on the key decapsulation, given that each secret polynomial is only multiplied once in the key decapsulation.

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 that are relevant to our attack. We refer the reader to the supporting documentation from Kyber for more details on the KEM [4].

2.1 Kyber

As previously mentioned, Kyber is a lattice-based KEM. It relies on 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 $\textsc{Sample}_U$ and $\textsc{Sample}_B$ are functions which uniformly sample values in the ring R_q given a seed. The $\textsc{Sample}_U$ provides a uniform random matrix, and $\textsc{Sample}_B$ 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

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

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

Decompress, which are defined as $\textsc{Compress}(u) := \lfloor u \cdot 2^d/q \rceil \bmod (2)^d$ and $\textsc{Decompress} := \lfloor q/2^d \cdot u \rceil$, 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
$\hat{\mathbf{A}} \in R_q^{k\times k} \leftarrow \textsc{Sample}_U(\rho)$
$\mathbf{r}, \mathbf{e}_1 \in R_q^k, e_2 \in R_q \leftarrow \textsc{Sample}_B(\tau)$
$\mathbf{b} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \mathsf{NTT}(\mathbf{r})) + \mathbf{e}_1$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \mathsf{NTT}(\mathbf{r})) + e_2 + \textsc{Encode}(m)$
$\mathbf{c}_1, c_2 \leftarrow \textsc{Compress}(\mathbf{b}, v)$
$c = (\mathbf{c}_1, c_2)$
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.

2.2 Number Theoretic Transform (NTT)

Kyber performs polynomial multiplications and speeds it up to *linear* time by transforming the polynomials into the NTT domain, allowing for a so-called *pointwise* multiplication between the polynomials. The NTT is a version of Fast Fourier Transform (FFT) over a finite ring. To perform the transformation, one evaluates the polynomial at powers of a primitive root of unity, which are usually represented by the symbol ζ. We refer to [23] for details on how to implement

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

Input: Public key $pk = (\hat{\mathbf{t}}, \rho)$
Output: Ciphertext c, shared key K
Choose uniform m
$(\bar{K}, \tau) \leftarrow \mathsf{H}(m||\mathsf{H}(pk))$
$c \leftarrow \text{PKE.}\textsc{Enc}(pk, m, \tau)$
$K \leftarrow \mathsf{KDF}(\bar{K}||\mathsf{H}(c))$
return c, K

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
$\mathbf{b}, v \leftarrow \textsc{Decompress}(\mathbf{c_1}, c_2)$
$m \leftarrow \textsc{Decode}(v - \mathsf{NTT}^{-1}(\hat{\mathbf{a}} \circ \mathsf{NTT}(\mathbf{b})))$
$(\bar{K}, \tau) \leftarrow \mathsf{H}(m||\mathsf{H}(pk))$
$c' \leftarrow \text{PKE.}\textsc{Enc}(pk, m, \tau)$
if $c = c'$ **then**
  $K \leftarrow \mathsf{KDF}(\bar{K}||\mathsf{H}(c))$
else
  $K \leftarrow \mathsf{KDF}(z||\mathsf{H}(c))$
return K

the NTT (in Kyber and Dilithium) and cover relevant aspects of Kyber below. Kyber has dimension k, and each dimension has its own roots $\zeta_k^0, \zeta_k^1, \ldots, \zeta_k^{n-1}$. In the following, we focus on a single dimension for ease of presentation.

The NTT on Kyber. In Kyber, the n-th root of unity does not exist and therefore, the $2n$-th roots of unity are used so that modulus polynomial $X^n + 1$ is factored into polynomials of degree 2 rather, i.e., Kyber performs an *incomplete* NTT, where the last layer is not executed. Therefore, in Kyber, after the (incomplete) NTT transformation, a polynomial a corresponds to 128 polynomials of degree 1 each. Polynomial a is thus transformed to $\mathsf{NTT}(a) = a_0 + a_1 x, \ldots, a_{254} x + a_{255} x$. 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:

$$c_1 = a_0 b_1 + a_1 b_0, \quad c_0 = a_0 b_0 + a_1 b_1 \zeta. \tag{1}$$

We will denote the multiplication in Equation (1) as *pair-pointwise*.

Multiplication optimizations. In Equation (1), 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 optimized via the Karatsuba algorithm in such a way that we only need to perform 4 multiplications per each pair-pointwise multiplication:

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

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

$$c_0 = a_0b_0 + a_1b_1\zeta, \qquad c_1 = (a_0 + a_1)(b_0 + b_1) - (a_0b_0 + a_1b_1). \tag{3}$$

Observe that Karatsuba multiplication is the most popular approach for implementing pair-pointwise multiplication in Kyber. It allows us to reduce the number of multiplications from five to four. The software implementation has adopted the approach we analyze in this paper; it was also used in public hardware implementations of Kyber such as [43].

Masking Kyber. There are several proposals to mask lattice-based schemes such as NTRU [30] and Saber [8], whereby the following works present concrete masking schemes for Kyber [11,18]. The masking of the schemes addresses various secret-dependent operations, such as computing inverse NTT, the key derivation function in the decapsulation process, or more commonly, masking polynomial multiplication with the long-term secret. The approach for masking polynomial multiplication in Kyber follows a similar pattern to other cryptographic schemes: the secret is divided into shares, and secret-dependent operations are performed on each share. The results are then combined. In the case of Kyber, this involves splitting the secret polynomials into shares and multiplying the input ciphertext separately with each share.

2.3 Online Template Attacks

Online Template Attack (OTA), introduced in [6,7], is a powerful technique residing between horizontal and template attacks. The main distinctive characteristic is building the templates after capturing the target trace and not before like in classical template attacks [13]. In general, creating templates in advance is feasible when the number of possible templates is small, like for example, for a binary exponentiation algorithm, where templates need to distinguish a single branch result, which only requires two templates [13]. However, if the number of leaking features increases, the number of different templates could be infeasible to generate in advance. This scenario is where OTAs enter into play by capturing templates on-demand based on secret guesses [6,7].

In general, OTA works as follows: the attacker creates templates corresponding to partial guesses of the secret and then matches the templates to the target trace; the best matching indicates which guess was correct. The attacker continues by iteratively targeting new parts of the secret until it is fully recovered.

In recent years OTA was applied in many scenarios, most notably, against Frodo post-quantum proposal [10] and several crypto-libraries (`libgcrypt`, `mbedTLS`, and `wolfSSL`) using microarchitectural side-channels [12].

We will use OTA in our experiments to improve the success rate of our attacks to 100%, namely, we will first use attacks to learn the secret coefficients and the remaining entropy we will recover using OTA (for details see Section 5).

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 implementations of Kyber, we should consider several aspects that may affect the probability of success of our attack. First, as noted in Subsection 2.2, many implementations of Kyber optimize the pair-pointwise multiplication via Karatsuba, and thus, the multiplication is not performed exactly as described in Equation 1, and we may have fewer 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 how 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 multiplications between operands to be performed 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 analyzing the number of operations within a pair-pointwise multiplication that depends solely on one of the two coefficients a_0 or a_1 and tries to exploit such operations for trying to distinguish. In the following, we first analyze possible leakage points for attacking implementations of Kyber, which uses Karatsuba to implement pair-pointwise multiplication. This analysis will help us craft a template attack that will succeed with high probability and will not require many 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 3, 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 $\hat{\mathsf{a}}$. 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 $\hat{\mathsf{b}}$ 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 [17], 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 3.4). 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 [25] 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 1). 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.

Improving success rates of the attacks using Online Template Attack. We now consider the case where the success rate of an attack (either q or q^2) is too low to recover all coefficients, e.g., when mounting a single-trace attack or when the attack is affected by noise. Then, in the q^2 attack, correlation analysis might not rank the template with the correct pair (a_0, a_1) first, but rather as the x-th most likely template. To recover (a_0, a_1), enumerating over all possible x pairs is prohibitive for all 128 coefficient pairs since it would require 2^{128} trials.

In this case, it is worth to check whether the *first* pair of coefficients is always determined correctly. Indeed, this is the case in our experiments (Section 5). Our interpretation is that values in registers set by multiplications in previous iterations slightly affect the power consumption when the registers are overwritten. On the other hand, since there is no previous operation for the first multiplication, the initial register state is deterministic, and the attack is successful. Thus, the attack improves if we proceed *adaptively* and only attack the y-th pair after having correctly recovered the $y-1$ coefficient pairs before. Since all registers are now set correctly, the attack on the y-th multiplication should succeed similarly to the attack on the first multiplication. This attack creates template *online*, i.e., after obtaining the target power trace. Similarly to improving the q^2 attack, it can also improve the accuracy of the $q + q$ attack and all intermediate variants. For details about this method in practice, see Section 5.

3.3 Attack on DPA-protected Kyber

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

We can apply the 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 assume 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 high costs in terms of the design area.

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 probability of success 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 that produces shares that 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 concerning 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.

3.4 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 randomized shuffling of the pair-point multiplications. Indeed, such pair-pointwise multiplications may be easily shuffled since each pair-point multiplication is independent, and it does not 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. However, 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 to use a chosen ciphertext whose coefficients all have a different and unique value. We will use such a ciphertext to generate our templates the same way described before, thus obtaining q templates. Then, we will use the same ciphertext to obtain 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. Since each of these is unique, we will know its original position, revealing 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 the 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 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

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

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 our ciphertext (on the NTT domain) consists of coefficients with unique values this time. 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

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

(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 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., $a_{2i+1} = j$. 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 to generate the template and the target trace, we used a ciphertext polynomial whose coefficients (on the NTT domain) are different from each other. Finally, let us recall that for obtaining a match, *all* input operands used within the analyzed computations must 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, we can do a total of 128 shifts for each template 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:

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

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 coefficients a_0 in each (shuffled) pair-point

multiplication. Since we learned the original (non-shuffled) position of each pair-point multiplication in the previous step, we will also know the original position of the extracted a_0 coefficients in the non-shuffled secret key.

4 Simulations

This section presents leakage simulation of our attacks (Section 3) on the implementation in [18,2] for Cortex-M4.

4.1 Implementation of pair-point multiplication

The code which we analyze implements the pair-pointwise multiplication as in Listing 1.1 and corresponds to the Karatsuba multiplication algorithm [24] (see Equation (3) 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 $\{0, \ldots, 3328\}$. 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 1.2 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 term $a_1 \cdot b_1 \cdot \zeta + a_0 \cdot b_0$ (cf. Equation (1)). Next, the code sums of the cross terms as $a_1 \cdot b_0 + a_0 \cdot b_1$ (line 14) and reduces it mod 3329 (line 15).

Listing 1.1: 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 1.2: Montgomery subroutine.

```
.macro montgomery q, qinv, a, tmp
  smulbt \tmp, \a, \qinv
  smlabb \tmp, \q, \tmp, \a
.endm
```

4.2 Hamming weight model

We analyze our attack in the *Hamming weight* model which leaks the number of ones in the processed values. We assume that the power consumption of a device correlates with the Hamming weights of the computed states. In our analysis, we will check whether each possible secret coefficient $a_i \in \{0, .., 3328\}$ (or each possible pair of coefficients) leads to a unique sequence of 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 coefficients used within that pair-point multiplication.

For the first heuristic estimate, let us compute an *upper bound* on the leaked information by assuming that all computations correspond to *independent* uniformly random k-bit strings. The *expected information* we obtain from the Hamming weight of a uniformly random k-bit string $|\log \Pr[\mathrm{HW} = i]|$ 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*)

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

for a uniformly random k-bitstring. Asymptotically, the expected information $H(k)$ grows linearly in $\sqrt{k}$. For example, 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 1.1 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 the overall expected information of $H(24) + 3 \cdot H(28) + H(24) \approx 13.69$ 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 Montgomery reductions (line 12 and line 15), each resulting in 3 more operations, leaking together $6 \cdot H(28) \approx 20.7$ additional bits and the computation and addition of cross terms in line 14, which generate another H(24)-bit value, leading to an overall 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 trying out all *pairs* should succeed with a high probability. Appendix B confirms our heuristic calculus with simulations. Additionally, the heuristic calculations and the simulations from the next section suggest that the $q+q$ attack and the q^2 attack are robust even when adding a certain amount of Gaussian noise.

4.3 Simulations of Gaussian Noise

We now simulate the aforementioned operations while adding a small Gaussian noise with standard deviation σ to the simulated target trace. Subsequently, we list the best coefficient candidates according to the L_2-norm.

Using this method (see Appendix B for details), we analyze the probability of a_{2i} being amongst the top 1, 2, 3, 10, 100 candidates (cf. Fig. 6) when analyzing only the operations that depend on a_{2i} alone as well as the probability of (a_{2i}, a_{2i+1}) being amongst the top candidates (cf. Fig. 7) when analyzing all operations depending on (a_{2i}, a_{2i+1}). Since the probability of a_{2i} being the top 1 candidate is only 0.9475 when no noise is added, the probability of obtaining all 128 correct a_{2i} is $(0.9475)^{128} \approx 0.001$ and thus too low to be useful. However, up to $\sigma = 0.87$, the probability of a_{2i} being amongst the top 100 candidates is ≥ 0.999 and thus, up to a noise of $\sigma = 0.7$, with probability $0.99^{128} \approx 0.88$, we can significantly reduce the search space for the coefficient pairs from q^2 to $100q$.

For larger noise, we need to run the q^2 attack. The probability of (a_{2i}, a_{2i+1}) being the top 1 candidate drops below $\frac{15}{16}$ at $\sigma = 0.54$. In turn, the probability of (a_{2i}, a_{2i+1}) being amongst the top 100 candidates stays above 0.99 up to $\sigma = 0.72$. When aiming to brute-force the remaining uncertainty, in expectation, for $\sigma = 0.72$, we have $\frac{15}{16} \cdot 128 \approx 16$ positions where we need to try out 100 candidates yielding a computation cost of $100^{16} \leq 2^{20}$ times $\binom{128}{16} \approx 2^{128}$. The brute-forcing cost is thus dominated by the binomial coefficient $\binom{128}{\ell}$, determined by the number ℓ positions which we need to brute-force. $\binom{128}{\ell}$ remains below 2^{40} for $\ell \leq 5$. For each noise rate, we can now compute the probability of extracting all 128 coefficients if we brute-force only up to 5 positions as follows:

$$p_{100}^{128} \cdot \sum_{\ell=0}^{5} \binom{128}{\ell} \cdot (1 - p_1)^{\ell} \cdot p_1^{128-\ell},$$

where p_{100} is the probability that (a_{2i}, a_{2i+1}) is amongst the top 100 candidates and p_1 is the probability that it is the top candidate. This probability is almost 1 when $\sigma \leq 0.4$ and then drops to almost 0 sharply for $0.4 \leq \sigma \leq 0.55$, also see the dashed line in Fig. 7.

5 Experimental evidence

This section presents experimental results for three attack variations from Section 3: q^2, q, and an improved version using an online template attack (OTA)[7]. Similar to the original OTA [6,7], we calculate the correlation between the target trace and a template, resulting in a *matching trace* that indicates a match. If the secret coefficient pair in the template matches that used in *some* multiplication in the target trace, we observe a region in the matching trace with values close to one. We first describe our experimental setup and then discuss our results.

We target the masked Kyber implementation from the mkm4 repository [18]. Our experiments use the same setup as described in that paper, utilizing the ChipWhisperer Lite platform with an STM32F303 target [31], featuring an Arm Cortex-M4 core. This setup ensures low noise and well-aligned traces. Our focus is the `poly_basemul` function, where we compute pair-pointwise multiplication.

In our experiments, we use the same physical instance of the ChipWhisperer device for profiling and attacking, which is the best scenario for an attacker. However, this might not reflect a real-world scenario and we leave investigating the portability of templates in our attack as future work.

[7] Paper supplementary materials, the attack scripts in particular, are available at: https://github.com/crocs-muni/Attack_Kyber_ACNS2024

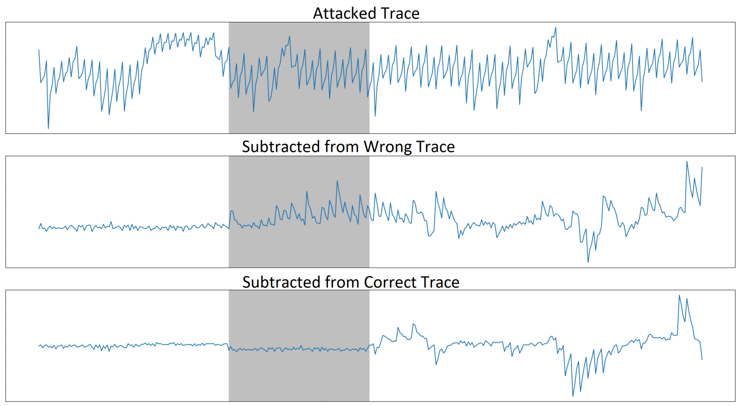

Fig. 1: Characterization: target trace (top), subtraction of the target trace from an incorrect template (middle) and from the correct template (bottom).

Before launching the attack, we need to select relevant regions of the traces. After testing multiple methods and approaches, the Difference-of-Means approach described in [6] proved to be the best. We always select 33 points of interest per pair-pointwise multiplication for all our attacks (since that number yiels the best results).

In the $q + q$ attack, we observe a limited leakage and the results are rather modest. We obtain a more accurate success rate for the first pair-pointwise multiplication than the remaining ones. On average, the correct candidate for the first multiplication is ranked at 282, and for all multiplications, it is at 1623 (out of 3329). This is insufficient for the attack to succeed. Improving the success rate, possibly using deep learning, is left for future work.

5.1 q^2 attack

Next, we attempt q^2 attack. We obtain the q^2 templates for all pairs of coefficients and each template is exactly *one trace*. Therefore, for this experiment, we use exactly 11082241 template traces to attack single target traces separately.

In Figure 1, we illustrate our method for visualizing leakage, following the approach outlined in [21]. This approach involves calculating the difference between a template and our target trace, as depicted in Figures 3 and 4 of [21]. The top trace in Figure 1 represents our target trace, with the highlighted area indicating the calculation of a pair-point multiplication. The middle trace shows the result when we subtract the target from a template that *does not* match the secret coefficients used in the highlighted pair-point multiplication. The bottom trace corresponds to the difference between the target and a template using the correct pair of secret coefficients. Notably, the highlighted region in this trace contains sample values very close to zero.

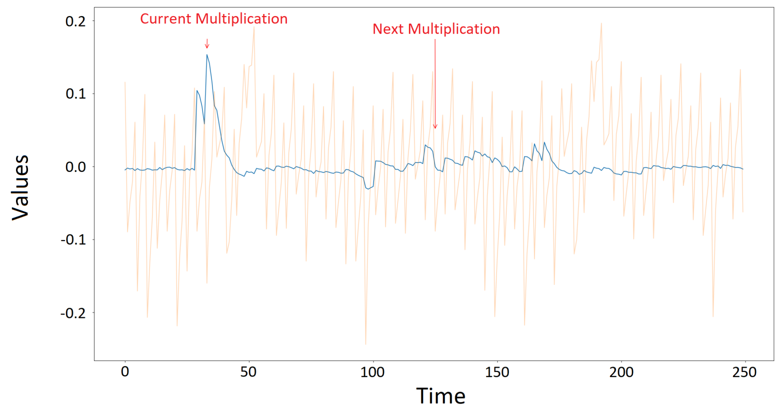

Fig. 2: The effect of previous multiplication on the following one: the correlation between the current multiplication value and the whole trace (in blue).

When comparing a target traces to the template corresponding to the pair of coefficients found in the secret key, our difference trace consistently contains a region with samples close to zero, as shown at the bottom of Figure 1. However, when attempting to compare a template for a pair of coefficients that *do not* appear in the key, the difference trace does not exhibit such a low region.

In the q^2 attack, we compare each pair of coefficients with templates, resulting in an ordered list of candidate values. Notably, there is a significant difference in accuracy between the first pair of coefficients and the rest. As shown in Figure 3, the first pair is correctly recovered in about 86% of cases, while the average success rate across all multiplications is 34%. This discrepancy is due to traces being influenced by previous multiplications, as illustrated in Figure 2, where the coefficient from the first multiplication affects slightly the subsequent multiplication, too. The first multiplication is not affected by any previous multiplication and that is why the corresponding success rate is much better.

Given the high success rates of the q^2 attack in recovering the first multiplication, we can reduce the number of candidate templates and initiate a combined attack using both q^2 and OTA. We begin with the q^2 attack. Assuming successful recovery of the first multiplication, we generate a new set of templates by combining the top two results for the first multiplication with a select number of top candidates for the second multiplication. These new templates cover a larger portion of the trace and are fewer in number, resulting in improved matching rates. We now repeat this process, assuming the first two multiplication coefficients have been recovered correctly, iterating through the whole trace. The main downside of this approach is requiring additional templates.

We *successfully recover all coefficients* for 3 attacked traces with this approach, at the cost of the increased number of templates – 20 600 000, 43 000 000,

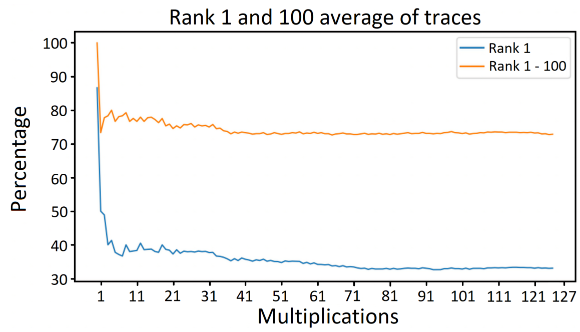

Fig. 3: q^2 attack success rate: blue line corresponds to the first candidate being correct and orange line to the correct candidate being in the top 100 results.

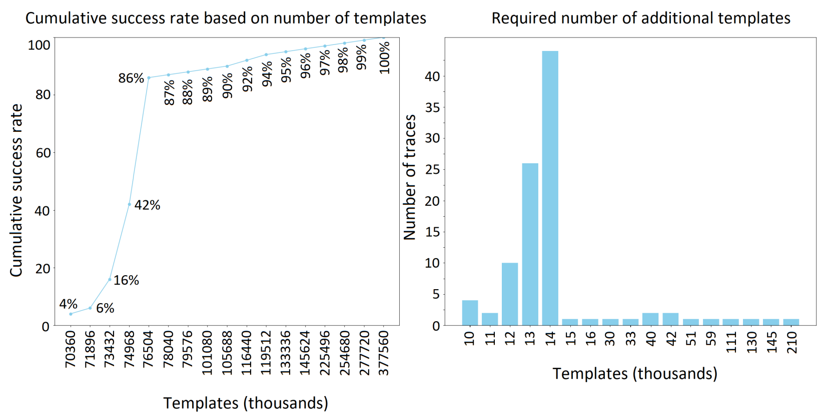

Fig. 4: *Left:* success rates of the full attack on masked Kyber768 wrt. the number of captured templates, estimated from 100 random target traces. *Right:* the extra number of templates required for the OTA attack (only non-zero values).

and 20 600 000, respectively. These numbers can be lowered, as described in the analysis of the required number of traces in the following section. With our setup, gathering additional 15 000 templates per multiplication takes about 9 days [8] and cover 87% of attacked traces. The success rates for different amounts of templates for the full attack on masked Kyber768 are shown in Figure 4.

5.2 Attack analysis

In order to launch the q^2 + OTA attack, it is necessary to collect the 11M templates for the q^2 attack and the additional traces for each multiplication. Based on the analysis of 100 random traces, the additional requirement is, on average 13 000 - 15 000 per candidate for each multiplication, as shown in Figure 4.

[8] Note, however, that we did not optimize our setup for the speed of acquisition.

To successfully attack unmasked Kyber768, we need to repeat the attack 3 times, reducing the experimental success rate to 65%. Kyber768 performs three polynomial multiplications: the initial `poly_basemul` and two subsequent `poly_basemul_acc` operations. The `poly_basemul_acc` function is similar to operation `poly_basemul` but also accumulates its results into the previous multiplication, hence the name "accumulation."

The code of `poly_basemul_acc` mixes accumulation instructions with other multiplication instructions, necessitating separate template collection. These templates rely on results from previous multiplications. However, we already have these coefficients from previous attacks (notably, on `poly_basemul`). While the attack on `poly_basemul_acc` should perform better due to more leaking instructions, new templates must be collected for each execution, depending on the previously recovered coefficients.[9] For a complete attack on unmasked Kyber768, we would need approximately 44.5M templates: 3×11 million (for 3 executions) and $3 \times 15\,000 \times 2 \times 128$. Here, we assume that we need 15 000 additional templates per multiplication and a conservative estimate that we cannot reuse templates for `poly_basemul_acc` if accumulation inputs differ. Based on preliminary characterization, it seems that re-using templates for different inputs is challenging and we leave it to be investigated in future work.

To attack masked Kyber768 with order 2, we need to execute attack 6 times: 2 times for `poly_basemul` and 4 times for `poly_basemul_acc`. For `poly_basemul` we would need to collect templates once, but for `poly_basemul_acc` templates need to be collected each time. Therefore, we would need the following number of templates: $5 * 11\text{M} + 6 * 15000 * 2 * 128 \approx 78\text{M}$ to achieve 43% success rate; to increase it to 90% we need approximately 105M traces as shown in Figure 4. At the time of writing, the current setup was able to capture 1 500 traces per minute. At this rate, gathering the full 78M templates would take about 45 days. In general, we leave improving the efficiency of this attack as future work.

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 to 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

[9] Initial tests hint at a 30% acquisition reduction for the OTA step with a single `poly_basemul_acc` experiment. However, we exclude this result from our estimates, reserving exploration of this optimization for future work.

optimized 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 that generate shares with coefficients with much larger values would certainly make our attack more difficult. Such schemes would imply an increase in the number of templates needed for our attack, and the chances of getting false positive matches would also increase. Unfortunately, such masking schemes imply an increase in the usage of computational resources (e.g. memory and stack usage) and the online complexity of Kyber.

Randomisation of the secret coefficients. The works in [46,19] propose and improve a countermeasure based on randomizing the polynomials of the secret coefficients via the redundant number representation (RNR). Secret coefficients are randomized by adding a randomly chosen value $r \cdot q$, with $r \in [1, 2^k)$. The computations are then performed $\mod (2^k q)$. Given such a countermeasure, we can still apply the same attack described in this paper and extract the randomized secret coefficients. A polynomial with such randomized coefficients is equivalent to the original secret polynomial, and thus, the extracted values are just as helpful in breaking the security of decryption. We note, however, that this countermeasure would make our attack more difficult since now there is a larger set of possible values for the secret coefficients.

Masking of the input ciphertext. Reparaz et al. proposed masking of the input ciphertext as a countermeasure against side-channel attacks [36]. Such an approach would effectively mitigate our attack since we would not know the value of the ciphertext polynomial used when performing the polynomial multiplication during decryption. However, masking of an input ciphertext is an expensive countermeasure against SCA given that (1) one needs to integrate a source of randomness during decapsulation. Such a source is needed because for blinding the ciphertext, one needs first to generate a random message; (2) the newly generated message needs to be encrypted (see Algorithm 2); and (3) the extra noise added to the encryption of the new message may affect the homomorphic property of the complete scheme and the chances of decryption failure increase.

Polynomial blinding. Polynomial blinding may be a straightforward way of mitigating our attack [39]. The idea of this countermeasure is to multiply both polynomials $a, b \in R_q$ by a randomly chosen integer $t \in \mathbb{Z}$, s.t. $ta \cdot t^{-1}b = a \cdot b$. Since the value t is unknown, the adversary does not know the value of any of the two operand polynomials and thus cannot construct useful templates. The adversary could attempt to extract either the value t or its inverse via some other side-channel attack. Recall that the adversary knows the value of one of the two polynomials, which would be blinded.

Parallelisation of the pair-point multiplication. Parallelizing several of the pair-point multiplications prevents a straightforward application of our attack. Namely, the parallelization 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 integrating an additional multiplier entity would imply a high cost in terms of size.

***Complete* NTT and actual point-wise multiplication.** Certainly, some of our attack strategies cannot be applied to schemes that implement a complete NTT to its polynomials and then multiply them in a proper point-wise fashion. For instance, our attack using q^2 templates does not work anymore 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 [25], which was selected as a post-quantum candidate signature scheme. Dilithium performs a *full* NTT and a point-wise multiplication.

References

1. Github repository: Collection of post-quantum cryptographic algorithms for the arm cortex-m4. Last modified: 2023, https://github.com/mupq/pqm4.
2. Github respository for masked Kyber presented in [18], 2022. https://github.com/masked-kyber-m4/mkm4.
3. 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.
4. 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.

5. 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.
6. Lejla Batina, Łukasz 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.
7. Lejla Batina, Łukasz Chmielewski, Louiza Papachristodoulou, Peter Schwabe, and Michael Tunstall. Online template attacks. *Journal of Cryptographic Engineering*, 9:1–16, 04 2019.
8. 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.
9. 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.
10. Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Assessing the feasibility of single trace power analysis of frodo. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 2018.
11. 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.
12. Alejandro Cabrera Aldaya and Billy Bob Brumley. Online template attacks: Revisited. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):28–59, July 2021. Artifact available at https://artifacts.iacr.org/tches/2021/a11.
13. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
14. 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.
15. Elena Dubrova, Kalle Ngo, Joel Gärtner, and Ruize Wang. Breaking a fifth-order masked implementation of crystals-kyber by copy-paste. In *Proceedings of the 10th ACM Asia Public-Key Cryptography Workshop*, APKC '23, page 10–20, New York, NY, USA, 2023. Association for Computing Machinery.
16. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *J. Cryptol.*, 26(1):80–101, 2013.

17. 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.
18. 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.
19. Daniel Heinz and Thomas Pöppelmann. Combined fault and dpa protection for lattice-based cryptography. *IEEE Transactions on Computers*, 72(4):1055–1066, 2023.
20. 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.
21. 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.
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.
23. M. J. Kannwischer. *Polynomial Multiplication for Post-Quantum Cryptography.* PhD thesis, Nijmegen U., 2022.
24. A. Karatsuba and Yu. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595, January 1963.
25. 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.
26. Vadim Lyubashevsky and Gregor Seiler. NTTRU: truly fast NTRU using NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):180–201, 2019.
27. 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.
28. Catinca Mujdei, Lennert Wouters, Angshuman Karmakar, Arthur Beckers, Jose Maria Bermudo Mera, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. *ACM Trans. Embed. Comput. Syst.*, Nov 2022.
29. 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.
30. 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.

31. 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.
32. David F. Oswald and Christof Paar. Breaking mifare desfire MF3ICD40: power analysis and templates in the real world. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2011.
33. 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.
34. 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.
35. 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.
36. 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.
37. Oscar Reparaz, Sujoy Sinha Roy, Ruan de Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking ring-lwe. *J. Cryptogr. Eng.*, 6(2):139–153, 2016.
38. Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to titan. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 231–248. USENIX Association, August 2021.
39. Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures - engineering a side-channel resistant post-quantum signature scheme with compact signatures. *J. Cryptogr. Eng.*, 8(1):71–84, 2018.
40. Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *IACR Cryptol. ePrint Arch.*, page 39, 2018.
41. Jian Wang, Weiqiong Cao, Hua Chen, and Haoyuan Li. Practical side-channel attack on masked message encoding in latticed-based kem. Cryptology ePrint Archive, Paper 2022/859, 2022. https://eprint.iacr.org/2022/859.
42. Ruize Wang, Martin Brisfors, and Elena Dubrova. A side-channel attack on a bitsliced higher-order masked crystals-kyber implementation. *IACR Cryptol. ePrint Arch.*, page 1042, 2023.
43. 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.
44. Bolin Yang, Prasanna Ravi, Fan Zhang, Ao Shen, and Shivam Bhasin. STAMP-single trace attack on M-LWE pointwise multiplication in Kyber. Cryptology ePrint Archive, Paper 2023/1184, 2023. https://eprint.iacr.org/2023/1184.

Alg. 5: Montgomery reduction

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;

Alg. 6: Signed Montgomery reduction from [40]

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;

45. 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.
46. Timo Zijlstra, Karim Bigou, and Arnaud Tisserand. FPGA implementation and comparison of protections against SCAs for RLWE. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019*, pages 535–555, Cham, 2019. Springer International Publishing.

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 the 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 [23], Algorithm 6 shows the case of signed Montgomery reduction from [40].

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} $|q_{\text{inv}}| = 12$ 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	$\|q\| = 12$ bits
3. $((a_1 \cdot b_1)_B \cdot q_{\text{inv}})_B \cdot q$	$16 + 12 = 28$ bits
add $(a_1 \cdot b_1)$	$\|a_1 \cdot b_1\| = 24$ bits
4. $((a_1 \cdot b_1)_B \cdot q_{\text{inv}})_B + (a_1 \cdot b_1)$	$\max\{24, 48\} = 28$ bits
take **top** of register and call it c	$\|c\| = 12$ bits
5. $c \cdot \zeta$	$12 + 12 = 28$ bits

B Details on noiseless and noisy simulations

We now discuss our simulations for noiseless operations within the pair-point multiplications comprehensively and additionally explain how we calculated probabilities in our noisy simulations. We first focus on the first 5 instructions of the pair-point multiplication, cf. Section 4.2. 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 3 that pair-point multiplication also computes the term $a_1 b_1 \zeta$, where the value of ζ changes for each pair-point multiplication. So for our simulations, we initially fix ζ_0 and try out all possible values for a_1 and all possible values b_1. We 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.

As we 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. On the average only a 0.03125 fraction of tuples maps to more than 4 different coefficient values. We now provide further details 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 only consider the case that we are dealing with coefficients with unique hamming weight tuples, or with 2- or 3-way collisions.

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 Figure 5 for the chances that each odd coefficient has a unique hamming weight tuple, we obtain 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. We will explain later in this section how we can use the results of our simulations to outline an attack strategy that easily increases our success probabilities, with just a linear increase in the number of templates needed.

Extracting coefficient pairs (a_{2i}, a_{2i+1}). The lower part of Figure 5 gives the probabilities that each secret coefficient pair leads to a unique hamming weight tuple. We obtain these probabilities in an analogous way as 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

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}

Fig. 5: Number of Matches: given ζ_i, probability of a 1-, 2- or 3-way collision. Upper part: the probability of extracting odd coefficients with q templates. Lower part: probability of extracting pairs of coefficients with q^2 templates.

# templates	σ	Probability of being amongst top .. matches 1	2	3
q-templates	0.3	0.8915	0.9775	0.9936
	0.4	0.7851	0.9205	0.9617
	0.5	0.6530	0.8231	0.8948
	0.6	0.5291	0.7027	0.7911
	0.7	0.4214	0.5860	0.6775
q^2-templates	0.5	0.9336	0.9788	0.9890
	0.6	0.8234	0.9112	0.9415
	0.7	0.6707	0.7906	0.8419
	0.8	0.4998	0.6310	0.7027
	0.9	0.3697	0.4839	0.5517
	1.0	0.2581	0.3559	0.4135

Table 1: Simulation results for noisy traces.

success probabilities are fairly high, while, on the other hand, it also requires 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 optimizing the success of the attack. In the following, we outline an attack adaptation that 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 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 Figure 5, each secret coefficient pair has a very high probability of having a unique hamming weight tuple.

We can even optimize our attack further by considering the top 4 match candidates for each coefficient, generating an additional set of $4q$ templates. Concretely for the optimized attacks using $q+3q$ and $q+4q$ templates, we obtain success probabilities of $\Pi_{i=0}^{127} p_i \approx 0.6755$ 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. Subsequently, we can use our analysis of the coefficients to determine the (expected) ≈ 0.875 fraction of coefficients that are unique, given our list of coefficients that have a unique Hamming weight pattern. For the remaining ≈ 0.125 coefficients, brute-forcing over $4^{0.125 \cdot 128} = 2^{32}$ coefficients is feasible.

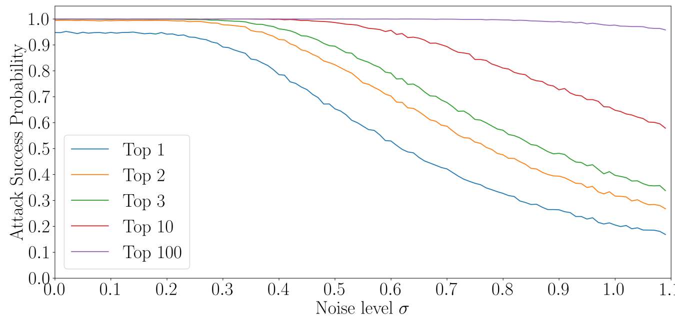

Fig. 6: Noisy $q + q$ attack simulations.

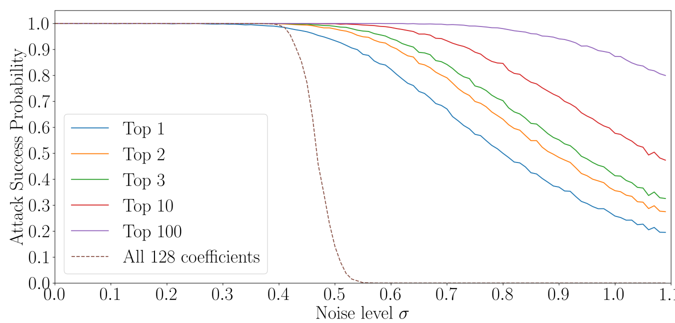

Fig. 7: Noisy q^2 attack simulations.

Noise. We now add Gaussian noise with standard deviation σ to the target trace and see for which σ we can still extract one or both coefficients. Instead of searching for perfect matchings, we minimize the L_2-norm of the *differences* between the simulated target trace and the template. Unfortunately, even for the q^2 attack, the best match under the L_2 norms provides the correct (a_{2i}, a_{2i+1}) value with probability ≤ 0.5 when $\sigma \geq 0.8$. All probabilities are calculated via 10,000 samples and using a random root out of all possible 128 roots.

C Comparison

To the best of our knowledge, there exist two other works in the literature that target polynomial multiplication in Kyber. In [28], the authors present a CPA attack on an unprotected polynomial multiplication implementation of Kyber. This attack led to the extraction of the long-term secret using approximately 200 traces. The main difference in comparison to our work is that the attack [28] requires multiple target traces and thus is not successful in the presence of a masking countermeasure. Our attack, on the other hand, requires a single target trace and, therefore, can successfully target masked implementations. The drawback of our approach is that we consider an adversary who can build template traces using a profiling device on which the secret can be freely changed. A classic CPA attack, as presented in [28], does not require any such profiling.

Another related work [44] presents a single-trace template attack on the polynomial multiplication of an unmasked implementation *pqm4* [1] during key generation [10]. There are several differences between this work and ours. First, note that they did not attack any masked implementation, but only argue about the attack's applicability to masking schemes since it attacks single traces. The attack is performed against a non-optimized implementation, utilizing straightforward polynomial multiplication without Karatsuba, leading to each secret coefficient being loaded twice, while our attack is on the *mkm4* masked implementation, which accesses the secret only once. Second, the attack [44] cannot be replicated on decapsulation since their template requires the leakage from the multiplication of k different polynomial values in the matrix A — which happens in the key generation. On the other hand, our attack can be applied to the key generation by utilizing the public polynomial values in A. Finally, their attack does not recover the full secret, but employs an extra key enumeration to finish the attack; as a result, their attack works for Kyber768 and Kyber1024, but not for Kyber512. Precise performance comparison is challenging due to uncertainties about the number of required templates in [44]. The authors mention using 500 traces to build templates for each intermediate, with approximately 14 attacked intermediates in each multiplication. This means that their attack would require only 7 000 templates if one template can be created for all pairwise multiplications or 896 000 if each multiplication needs to be templated separately. Consequently, it seems that the attack [44] requires fewer template traces for profiling than our approach, albeit with increased complexity and a lower success rate, necessitating final key enumeration.

Comparing our approach with [44] is intricate due to the mentioned differences. Foremost, [44] attacks key generation of the unprotected implementation, which involves a broader range of secret-dependent operations than our target. Therefore, we cannot estimate how well the attack from [44] would work against protected implementation like *mkm4*. In summary, the attack in [44] has advantages as it exploits various leaks and capitalizes on them. However, it is not easy to adapt to other procedures, such as the technique presented in this paper. Thus, this makes our attack more generic than the one presented in [44].

In Table 2, we give a summary of the comparison with [28] and [44]. From our work, we present the two versions, i.e., "Simulation" refers to the numbers of the original introduction of our attack described in Section 3 and concerning the results obtained via simulations in Section 4. The "Experiment" work refers to the real-world attack from Section 5, where 78M traces give a 43% success of extracting the secret key, while 105M traces give over 90% success rate.

[10] They also attack a reference implementation, but we do not concentrate on that since this implementation leaks much more than *pqm4* and the attacked by us *mkm4*. We are only looking at the long-term secret key and we do not consider the attacks on the encryption procedure.

Table 2: Comparison of attacks on the long-term secret key from the polynomial multiplications; the analysis is made for Kyber768 unless stated otherwise.

<table>
<tr><th>Work</th><th>Implementation</th><th>No. of target traces</th><th>No. of templates</th><th>Target algorithm</th><th>Remaining Brute-Force</th></tr>
<tr><td>[28]</td><td>Non-masked pqm4</td><td>200</td><td>0</td><td>Decapsulation</td><td>No</td></tr>
<tr><td>[44]</td><td>Non-masked reference and pqm4 implementations</td><td>1</td><td>Not provided, estimation: 7 000 or 896 000</td><td>Key generation</td><td>For pqm4 Kyber: 512 – infeasible; 768 – 2^{40}; 1024 – 2^5.</td></tr>
<tr><td>This work (Simulation)</td><td rowspan="2">Optimized masked mkm4 imp.</td><td rowspan="2">1</td><td>6 628 (q + q attack), or 11 082 241 (q^2 attack)</td><td rowspan="2">Key generation and Decapsulation</td><td rowspan="2">No</td></tr>
<tr><td>This work (Experiment)</td><td>q^2+OTA attack: 78M (43% SR) or 105M (90% SR)</td></tr>
</table>