Improved NTT and CRT-based RNR Blinding for Side-Channel and Fault Resistant Kyber

Max Duparc[1] and Mounir Taha[2]

[1] EPFL, Lausanne, Switzerland, max.duparc@epfl.ch
[2] Nagra Kudelski Group, Cheseaux-sur-Lausanne, Switzerland, mounir.taha@nagra.com

Abstract. In this paper, we build upon the blinding methods introduced in recent years to enhance the protection of lattice-based cryptographic schemes against side-channel and fault injection attacks. Specifically, we propose a cost-efficient blinded Number Theoretic Transform (NTT) that impedes the convergence of Soft Analytical Side-Channel Attacks (SASCA), even with limited randomness sampling. Additionally, we extend the blinding mechanism based on the Chinese Remainder Theorem (CRT) and Redundant Number Representation (RNR) introduced by Heiz and Pöppelmann by reducing the randomness sampling overhead and accelerating the verification phase. These two blinding mechanisms are nicely compatible with each other's and, when combined, provide enhanced resistance against side-channel attacks, both classical and soft analytical, as well as fault injection attacks, while maintaining good performance and low overhead, making the approach well-suited for practical applications, particularly in resource-constrained IoT environments.

Keywords: Kyber · Blinding · NTT · RNR · Side-Channel Attacks · Fault Attacks

1 Introduction

With the release of the first post-quantum cryptographic standards by NIST [NIS24b, NIS24a, NIS24c], post-quantum cryptography has taken a significant step toward integration into everyday applications. Among these standards, ML-KEM [NIS24b], previously known as CRYSTALS-Kyber [SAB+22], is, as of now, the sole standardized Key Encapsulation Mechanism (KEM). Based on the Module Learning With Errors (M-LWE) problem—a variant of the Ring-LWE problem [LPR10]—Kyber offers a practical balance of small public key and ciphertext sizes along with competitive performance. Moreover, its efficiency on embedded devices has been demonstrated in various implementations [KRSS19].

However, a critical aspect that has received comparatively less attention during its development is its vulnerability to *side-channel attacks*. Due to the inherent complexity of post-quantum cryptographic schemes, especially when compared to traditional cryptographic schemes like RSA or ECDHE, their implementations provide an expanded attack surface, heightening the risk of side-channel and fault injection vulnerabilities, posing a challenge for secure deployments in uncontrolled environments, as it is typically the case in IoT devices. For example, Kyber has been subjected to a wide range of side-channel and fault-injection attacks, exposing vulnerabilities in its critical computational steps. Side-channel attacks, such as Correlation Power Analysis (CPA), have targeted power and electromagnetic leakage of the polynomial multiplication and sampling operations [KdG21, MPG+22, YWY+23, KT23]. Soft Analytical Side-Channel Attacks (SASCA) have leveraged recursive patterns in the Number Theoretic Transform (NTT) and modular reductions, using statistical inference to recover private keys [PPM17, PP19, HHP+21, HSST23].

Additionally, *fault-injection attacks*, including techniques like clock glitches and electromagnetic disturbances, have focused on disrupting NTT operations to induce exploitable errors [Del22, HMS+23].

Thankfully, a breath of countermeasures to these attacks do exists and have been proposed thoughout the years. One family of such countermeasures is called *masking* [RRVV15, OY23, BGR+21], which consists in splitting sensitive data in several independent shares, performing leaky operations on each part individually. Although very strong, it must be noted that masking does not always prevent SASCA [HHP+21], nor injection attacks [HMS+23], and is quite costly since it requires performing the same operation multiple times.

Another family of countermeasures is *blinding*, which consists in combining the data with noise to complexify side channel attacks. This can take the form of random delays in the mechanism or of multiplicative masking, meaning that leaky operations over secret values are performed up to a scalar. Examples of blinding include [RPBC20], which provides efficient countermeasure against SASCA and [HP21] which uses Redundant Number Representation (RNR) based on the Chinese Remainder Theorem (CRT) to provide protection against SCA and fault injections.

Creating side-channel-resistant implementations involves selecting appropriate countermeasures while carefully evaluating their mutual compatibility. Some countermeasures may conflict with others, leading to a marked increase in implementation complexity and resource requirements.

Contribution In this paper, we build upon the foundational works of [RPBC20] and [HP21], introducing improvements to their respective blinding mechanisms. Specifically, we propose a novel blinded NTT structure that maximizes the number of loops in the factor graph underlying SASCA, effectively diminishing such attacks efficiency. Additionally, we extend the CRT-based RNR blinding mechanism to require less randomness, substantially accelerating the verification phase. These enhancements not only improves the countermeasure's overall performance but also its practicality.

Thanks to the fact that the first countermeasure target the roots of unity while the second targets the variables, both are nicely compatible with each other. We therefore present a combined implementation of Kyber that integrates both improved countermeasures and evaluate both the overhead and the practical leakage of our implementation compared to the unsecured standard.

Our implementation is available at: https://github.com/moun18/KYBER-CRT-NTT-SEC.

Organization The rest of this paper is structure as follows. In Section 2, we give a reminder on Kyber, the NTT and side channel attacks. In Section 3, we introduce our improved blinded-NTT design that relies on shifted blocks. In Section 4, we introduce our improved method to protect Kyber against side-channel and fault using a CRT-based RNR blinding. Finally, in Section 5, we detail our implementation and its results against SCA and FA.

2 Background

In this section, we present an overview of the Kyber scheme, highlighting its core components, including the Number Theoretic Transform (NTT) and the modular arithmetic operations upon which it is built. We then discuss prior work addressing fault and side-channel attack protections for implementations of lattice-based cryptography.

2.1 CRYSTALS-Kyber

Kyber [SAB+22] is a Key Encapsulation Mechanism (KEM) based on the M-LWE problem (a variant of the R-LWE problem). The KEM is constructed in two parts. The first one is building an IND-CPA PKE scheme that encrypts a 32 bytes message, and the second one is using a tweaked Fujisaki-Okamoto transform (FO-transform) to build an IND-CCA KEM.

Kyber works in the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, with $q = 3329$ and $n = 256$. The PKE scheme consists of three algorithms: KeyGen, Encrypt, and Decrypt. A simplified version of the PKE is given in Algorithms 1-3. Bold uppercase letters represent matrices of size $k \times k$ with elements in R_q. Bold lowercase letters represent vectors of size k with elements in R_q, k depending on the security level. $\mathsf{Sample_U}$ samples from a uniform distribution and $\mathsf{Sample_{CBD}}$ samples from a centered binomial distribution. Expand expands a small seed into a matrix with coefficients in R_q. $\mathsf{Compress}$ loosely maps elements from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^d}$, $\mathsf{Decompress}$ maps elements from $\mathbb{Z}_{2^d}$ to $\mathbb{Z}_q$. $\circ$ denote a Kyber specific baseline product of the polynomials, specified in Section 2.2.

Algorithm 1 ML-KEM/Kyber.CPAPKE.KeyGen()

output Pair of public/secret keys (pk, sk)

1: $\rho \leftarrow \mathsf{Sample_U}()$
2: $\sigma \leftarrow \mathsf{Sample_U}()$
3: $\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow \mathsf{Expand}(\rho)$
4: $\mathbf{s} \in R_q^k \leftarrow \mathsf{Sample_{CBD}}(\sigma, 0)$
5: $\mathbf{e} \in R_q^k \leftarrow \mathsf{Sample_{CBD}}(\sigma, k)$
6: $\hat{\mathbf{s}} \in \hat{R}_q^k \leftarrow \mathsf{NTT}(\mathbf{s})$
7: $\hat{\mathbf{e}} \in \hat{R}_q^k \leftarrow \mathsf{NTT}(\mathbf{e})$
8: $\hat{\mathbf{t}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
9: **return** $pk = (\rho, \hat{\mathbf{t}})$, $sk = \hat{\mathbf{s}}$

Algorithm 2 ML-KEM/Kyber.CPAPKE.Enc(pk, m, r_s)

input pk, the message $m \in \{0,1\}^{256}$ and a random seed $r_s \in \{0,1\}^{256}$
output The ciphertext pair $ct = (\mathbf{u}, v)$

1: $\hat{\mathbf{A}} \in \hat{R}_q^{k \times k} \leftarrow \mathsf{Expand}(\rho)$
2: $\mathbf{r} \in R_q^k \leftarrow \mathsf{Sample_{CBD}}(r_s, 0)$
3: $\mathbf{e_1} \in R_q^k \leftarrow \mathsf{Sample_{CBD}}(r_s, k)$
4: $e_2 \in R_q \leftarrow \mathsf{Sample_{CBD}}(r_s, 2k)$
5: $\hat{\mathbf{r}} \in \hat{R}_q^k \leftarrow \mathsf{NTT}(\mathbf{r})$
6: $\mathbf{u} \in R_q^k \leftarrow \mathsf{INTT}(\hat{\mathbf{A}}^t \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
7: $v \in R_q \leftarrow \mathsf{INTT}(\hat{\mathbf{t}}^t \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}(m)$
8: **return** $(\mathsf{Compress}(\mathbf{u}), \mathsf{Compress}(v))$

2.2 NTT

As shown in Algorithm 1-3, an important part of Kyber is the NTT and its inverse, the INTT operations. These refer to the *Number Theoretic Transform* and the *Inverse Number Theoretic Transform*, and are integral for efficiently performing polynomial multiplication in $\mathcal{R}_q$. Specifically, they reduce the computational cost of polynomial multiplication from $O(n^2)$, with a naive approach, to $O(n \log(n))$ field multiplications in $\mathbb{Z}_q$. This efficiency

Algorithm 3 ML-KEM/Kyber.CPAPKE.Dec(sk,ct)

input sk, ct
output The message m

1: $\mathbf{u}' \in R_q^k \leftarrow \mathsf{Decompress}(\mathbf{ct_u})$
2: $v' \in R_q \leftarrow \mathsf{Decompress}(ct_v))$
3: $\hat{\mathbf{u}}' \in \hat{R}_q^k \leftarrow \mathsf{NTT}(\mathbf{u}')$
4: $m' \in R_q \leftarrow v' - \mathsf{INTT}(\hat{\mathbf{u}}' \circ \hat{\mathbf{s}}))$
5: $m \leftarrow \mathsf{Compress}(m')$
6: **return** m.

gain stems from their ability—analogous to the Discrete Fourier Transform (DFT), which they generalize—to transform a convolution into a Hadamard product. We refer the reader to [SML24] for further details. In the specific case of Kyber, contrary to other lattice based schemes such as Dilithium [LDK+22], the NTT is incomplete and correspond to the following transformation. Let $\omega \in \mathbb{Z}_q$ be a primitive n-th root of unity, usually called a *twiddle factor* and $f = (f_0, \cdots, f_{n-1})$:

$$\mathbf{NTT}_\omega : \mathbb{Z}_q[X]/\langle X^n + 1\rangle \cong \prod_{k=0}^{\frac{n}{2}-1} \mathbb{Z}_q[X]/\langle X^2 - \omega^{2k+1}\rangle$$

$$\widehat{f}_j = \mathbf{NTT}_\omega(f)_j = \begin{cases} \sum_{i=0}^{\frac{n}{2}-1} f_{2i}\omega^{(2j+1)i} & j = 0 \mod 2 \\ \sum_{i=0}^{\frac{n}{2}-1} f_{2i+1}\omega^{(2j+1)i} & j = 1 \mod 2 \end{cases}$$

With $\mathbf{INTT}_\omega = \frac{2}{n}\mathbf{NTT}_{\omega^{-1}}$. The reason for that change during the 2nd round of the NIST standardisation [NIS] was that it enabled smaller parameters and equivalent performance, as detailed in [ZXZ+18, ACC+22]. Due to this change, the NTT transform convolution into the following baseline product:

$$\left(\widehat{f} \circ \widehat{g}\right)_j = \begin{cases} \widehat{f}_{2j}\widehat{g}_{2j} + \omega^{2j+1}\widehat{f}_{2j+1}\widehat{g}_{2j+1} & j = 0 \mod 2 \\ \widehat{f}_{2j}\widehat{g}_{2j+1} + \widehat{f}_{2j+1}\widehat{g}_{2j} & j = 1 \mod 2 \end{cases}$$

As the NTT is a natural extension of the DFT, the techniques developed for performing the Fast Fourier Transform (FFT) can also be applied to the NTT. In this paper, we focus on two main variants: the Decimation-In-Time (DIT-FFT) [GS66] and its dual, the Decimation-In-Frequency (DIF-FFT) [CT65]. Specifically, we implement the NTT using DIT and the INTT using DIF. This design choice is guided by Tellegen's principle [BLS03, Proposition 2], which guarantees that any efficient implementation of the DIT-NTT induces a corresponding DIF-INTT with the same computational complexity. Note that both implementation are bit-reversing, meaning that DIT-NTT$(f)_j = \widehat{f}_{\mathsf{bitrv}(j)}$, where bitrv denotes the bit-reversal of index j, seen as an 8 bit integer. Figure 1 provides an example of an in-line representation of a DIF-FFT. The corresponding algorithms for Kyber's specific NTT and INTT are detailed in [NIS24b, Algorithms 8 and 9].

At their hearth, both mechanisms are based on linear structures called *butterflies*, consisting of a multiplication with a twiddle factor coupled with an addition and subtraction between two coefficients. DIT butterfly and DIF butterfly are described in Figure 2. Both are dual of one another[1].

[1] More specifically, the matrix representing a DIT-butterfly is the transpose of the DIF-butterfly matrix.

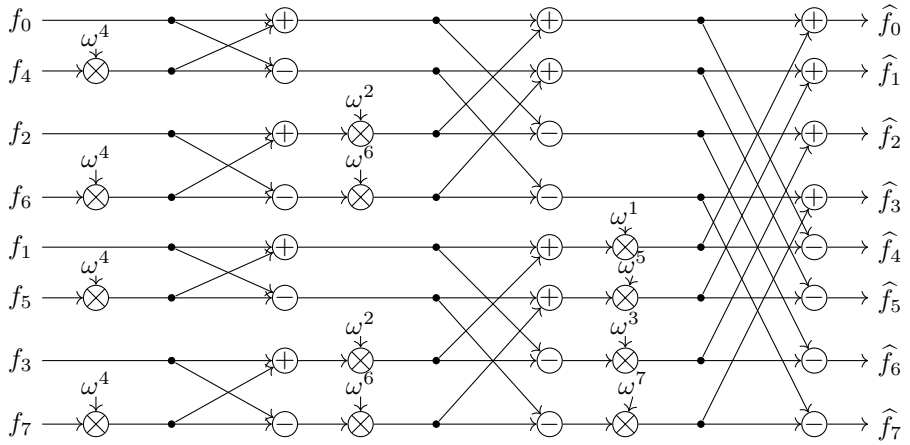


Figure 1: Example of the DIT-FFT for $n = 8$.

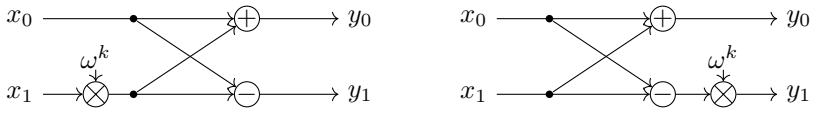


Figure 2: Diagram of DIT-butterfly (left) and DIF-butterfly (right)

2.3 Modular operations

The two primary functions employed for modular arithmetic in Kyber are the Montgomery and Barrett reductions, described in Algorithms 4 and 5, respectively. These functions are constant time and improve efficiency over naïve modular arithmetic approaches by eliminating the need for costly division. Both mechanisms play a crucial role in the implementation of our countermeasures against side channel attacks.

Algorithm 4 Montgomery Reduction

Constant: $2^{t-1} \leq N < 2^t$ an odd number, $R = 2^s$ and $N' = N^{-1} \mod R$
Require: x an positive integer smaller than NR.
Ensure: $xR^{-1} \mod N$
1: $n = (x\&(R-1)) \cdot N'$
2: $m = N \cdot (n\ \&(R-1))$
3: $z = (x - m) \gg s$
4: $z = z + \left(N\& \left(2^{t+1} - (z \gg t)\right)\right)$ ▷ *Ensures conditional addition of N if $z < 0$.*
5: **return** z.

2.4 Side channel and fault attacks.

2.4.1 Power analysis

Power analysis is a widely used category of side-channel attacks against cryptographic implementations. It exploits variations in power consumption caused by intermediate computations and register transitions, as the current drawn by a device often correlates with the Hamming weight of processed data. Consequently, an unprotected implementation of a cryptographic algorithm may inadvertently leak sensitive information. By collecting and analyzing power traces over multiple runs, an attacker can potentially recover the secret key using only public information and the captured traces.

Algorithm 5 Barrett Reduction

Constant: $N < 2^t$ an odd number, $R = 2^s$ and $Q = \left\lfloor \frac{2^{t+s}}{N} \right\rfloor \mod N$
Require: x an positive integer smaller than NR.
Ensure: $x \mod N$
1: $m = (x \cdot Q) \gg s + t - 1$
2: $z = m \cdot N$
3: $z = x - z$
4: **return** $z + \left(N\& \left(2^{t+1} - (z \gg t)\right)\right)$ ▷ *Ensures conditional addition of N if $z < 0$.*

Over the years, several power analysis techniques have been developed, including Simple Power Analysis (SPA), Differential Power Analysis (DPA), and Correlation Power Analysis (CPA). Among these, CPA has proven to be particularly effective against Kyber [KdG21, MPG+22, YWY+23, KT23], as it exploits statistical correlations between power consumption and key-dependent operations.

The classical way to protect against side channel attack is masking by using shares [RRVV15, OY23, BGR+21]. This method offers a strong protection against power analysis, however it also introduces a significant overhead that makes it less attractive. An alternative to shares is to introduce noise inside the computation, as proposed in [HP21].

2.4.2 Soft Analytical Side Channel Attacks (SASCA)

Soft-Analytical Side-Channel Attacks (SASCA) are very specific single-trace attacks. Originally applied to AES and SHA-3 [VGS14, KPP20], SASCA uses a single trace matched against precomputed templates to reconstruct the secret key. In lattice-based cryptosystems, SASCA primarily targets the NTT (Number Theoretic Transform) [PPM17, PP19, HHP+21, HSST23], exploiting sensitive intermediate variables to retrieve secret-dependent inputs. SASCA operates in two phases:

1. Profiling: Templates are built using leakage from a clone device.
2. Execution: A single trace from the target device is segmented, matched to templates, and modeled as a factor graph. The graph is processed using the Belief Propagation (BP) algorithm, combining intermediate leakage to recover secrets.

While BP guarantees correct results for acyclic graphs, "loopy BP" is used for graphs with cycles. However, loopy BP's accuracy and convergence degrade as loops in the graph increase or shorten, often causing oscillations and overconfidence in beliefs. Further details on BP are in [Mac03, Chapter 26] and [PPM17, PP19]. Improved method such as the Generalized Belief Propagation [YFW00] do exist to remedy this problem, but comes at the cost of significantly higher computational complexity, making it unsuitable for applications such as SASCA, where computational efficiency is critical, as factor graph of the NTT are large. We represent in Figure 3 how a DIT-butterfly is represented in the factor graph of the NTT in [PPM17, PP19], in which $f_{leak}(x_i) = \mathbb{P}[x_i|leak]$ are the observed side-channel information resulted from the outcome of the template matching while $f_{ADD}(y) = \mathbf{1}_{x_0+\omega x_1}(y)$, $f_{SUB}(y) = \mathbf{1}_{x_0-\omega x_1}(y)$ and $f_{BTF}(y_0, y_1) = f_{ADD}(y_0) f_{SUB}(y_1)$.

Using these representations, [PPM17] create a SASCA targeting generic Ring-LWE-based scheme. Their proposed attack required over a million templates for a successful key recover of Kyber. Importantly, their representation of a DIT-butterfly in the factor graph enabled [PPM17, Section 5.3] a quasi-linear propagation of parts of the BP algorithm. [PP19] is an improvement on the previous attack requires only hundreds of templates, when the target vector has a small range. This improvement comes at the expense of a slower computation of the BP. [HHP+21] combined SASCA with a Chosen Ciphertext

Figure 3: Representation of a DIT butterfly in the respective factor graph of [PPM17](left) and [PP19, HHP+21](right).

Attack strategy in order to construct sparse polynomials at the input of inverse NTT computations, secret key. A final important point to note is that, in order to be efficient, SASCA requires quite low noise level during sampling. (From $\sigma = 0.5$ to 2.)

2.4.3 Fault attacks

Fault injection attacks are another threats to cryptographic implementation. These attacks typically proceed in two stages: first, an adversary injects faults into a cryptographic computation, often by introducing errors into hardware or software operations. Second, the adversary analyzes the resulting faulty outputs to infer secret information, leveraging techniques such as Differential Fault Analysis (DFA).

In the context of Kyber, fault attacks are partially mitigated by its use of the Fujisaki-Okamoto (FO) transform. This transform ensures that decryption errors—whether caused by natural faults or deliberate injections—result in a mismatch between the ciphertext generated during reencryption and the original ciphertext, preventing the return of faulty outputs. However, the FO transform does not completely eliminate the vulnerability to fault attacks. Indeed, several effective fault attacks have been demonstrated against Kyber. Pessl and Prokop [PP21] showed that skipping a single instruction during decryption can bypass the FO transform's protection. By analyzing whether the error is detected, attackers can gather valuable information about the secret key.

Additionally, other fault injection strategies exploit slightly invalid ciphertexts to induce perturbations that are later corrected during decryption, effectively bypassing protections. Notable examples include the techniques described in [HPP21, Del22], where attackers carefully craft faulty inputs to extract secret key material.

3 Blinded-NTT using mixed blocks

Among the proposed countermeasures against SASCA, Ravi et al.[RPBC20] introduced two methods named blinding in time and blinding in memory. Blinding in time involves introducing time delays and randomly shuffling the execution of different butterflies within each NTT stage. This strategy increases the complexity of correlating side-channel information with internal computations, thereby making attacks more challenging. However, as shown in [HSST23], while this countermeasure is effective, it is not impervious to attacks by a powerful adversary who can still achieve success under certain conditions.

The second approach involves applying multiplicative masking to intermediate values within the NTT using a masking twiddle factor. This technique mitigates SASCA attacks by introducing loops into the factor graph, effectively obstructing the convergence of the BP algorithm. While this method remains constant-time, it requires significant random sampling if applied to every butterfly operations. For Kyber, this amounts to 6144 random bits per NTT computation. To reduce this overhead, blocks of butterflies are often masked using the same masking twiddle factors, thereby reusing randomness.

Our proposed solution enhances the block-based blinding-in-memory technique. Specifically, it maximizes the number of loops in the factor graph while maintaining the same number of multiplications as the original solution. In exchanges, we no longer allow for the number of masks to vary across the different NTT stages.

3.1 Masked-butterflies

The fundamental building block of our blinded NTT consist, similarly to [RPBC20], in masked butterflies, but different ones. A *masked butterfly* takes as input its initial data, multiplied by one or two masking twiddle factors, and return the standard output, also multiplied by one or two twiddle factors. As a butterfly has 2 input and 2 outputs, we have that a masked butterfly can be of 4 distinct types, based on whether the masking twiddle factors for the input and output are the same or different. We denote that SISO (Same Input, Same Output), DISO (Different Input, Same Output), SIDO and DIDO. Additionally, each of the butterflies can be DIT or DIF, resulting in a total of 8 masked butterflies. Among these 8 possibilities, 4 (SISO-DIT, SISO-DIF, DISO-DIT and SIDO-DIF) can be performed using only 2 modular multiplications while the others (SIDO-DIT, DISO-DIF, SISO-DIT and DIDO-DIF) require 3 multiplications.

In [RPBC20], the authors primarily used SISO butterflies (both DIT and DIF), with limited use of DISO-DIT and SIDO-DIF butterflies. In contrast, our approach exclusively employs DISO-DIT butterflies for the NTT and SIDO-DIF butterflies for the INTT, both illustrated in Figure 4. For even further clarity, the pseudocode for the DISO-DIT butterfly is detailed in Algorithm 6.

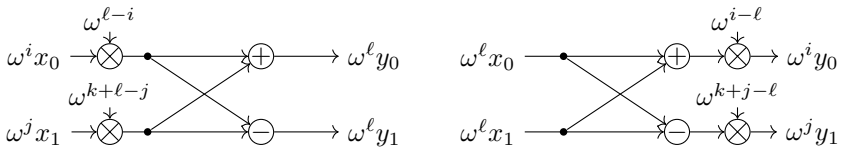


Figure 4: The DISO-DIT (left) and SODI-DIF (right)

Algorithm 6 – `DISO-DIT-BF`$(f_0, f_1, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k})$: (Different Input Same Output Derivation in Time Butterfly).

Require: f_0, f_1 in $\mathbb{Z}_q$ and $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, k \in \mathbb{Z}_{2^n}$, ω a fixed 2^n root of unity.
Ensure: f_0, f_1 in $\mathbb{Z}_q$.
1: $q \leftarrow \omega^{\mathtt{bitrv}(k)+m_{\mathsf{p}_3}-m_{\mathsf{p}_2}} f_1$ ▷ *bitrv*(k) *is the bit-reverse of* k *in* $\mathbb{Z}_{2^n}$.
2: $p \leftarrow \omega^{m_{\mathsf{p}_3}-m_{\mathsf{p}_1}} f_0$
3: $f_0 \leftarrow p + q$
4: $f_1 \leftarrow p - q$
5: **return** f_0, f_1

Although our method requires two modular multiplications instead of one, as in the standard butterfly (Figure 2). Some implementations, such as [ABD+18], perform one Montgomery multiplication and one Barrett reduction on an alternate branch. In these cases, the computational cost of our approach is approximately equivalent.

3.2 Masking Blocks

As touched priorly and in [RPBC20, Section 3.2], if we decided to sample a random mask for every butterfly inside our NTT, we would have to sample up to $6144 = 8 \cdot 6 \cdot 128$

random bits per NTT in Kyber [2], which significantly temper the performance.

This is why we employ the concept of *masking blocks*. Concretely, masking blocks are defined as sets of butterflies, where every butterfly within a block uses the same masking twiddle factor for both input and output. The size of each block, noted B (and thus their total number) is a power of two and remains fixed throughout the NTT. By using blocks of size B, the amount of randomness required is significantly reduced. Specifically, for one NTT computation in Kyber, we only need to sample $\frac{6144}{B}$ bits of randomness, making this approach efficient while maintaining the desired security properties.

Since B is a power of 2, the logic to manage and track the various twiddle factors can be efficiently implemented using, in the case of Kyber, 8-bit additions combined with bit shifts. Notably, the connections between successive masking blocks within the NTT can themselves be represented as an NTT diagram of length 7 and height $\log_2(128/B)$, which simplifies both construction and control. However, the challenge with block-based masking lies in handling the later stages of the NTT, when the distance between the two inputs of the butterfly is smaller than the block size. In this case, both masking input twiddle factors are equal, weakening the resistance to SASCA. This issue is particularly significant since the majority of butterflies fall into this category, especially when using smaller randomness budgets (e.g., sampling only 256 or 512 bits of randomness per NTT).

To address this limitation without incurring additional computational cost, we employ two types of blocks in our NTT design:

- Normal Blocks: Used during stages where the distance between the two inputs of the butterfly is greater than the block size. These blocks are defined based on the butterfly index. Specifically, the i-th butterfly of stage s belongs to the j-th block if and only if:
$$\left\lfloor \frac{i}{B} \right\rfloor = j$$

- Mixed block: Used when the distance of two input of the butterfly is smaller than the block-size. Here, the block also depends on the spacing between the two inputs of the butterfly, i.e. the current stage in the NTT. In this case, we have that the i-th butterfly of stage s is in the j-th block if and only if:
$$2\left\lfloor \frac{i}{2B} \right\rfloor + \left(\left\lfloor \frac{2^s(i \,\&(B-1))}{2^8} \right\rfloor \&1 \right) = j$$
Mixed block induces a shift in the different butterflies which ensures that the masking input twiddle comes from two different blocks, that are in fact concomitant.

Finally, the first and last masking twiddle factor used in our blinded NTT are set to 1. This choice ensures compatibility with Kyber's vector tests. Using this mask structure, we can efficiently determine the input and output twiddle factors for each masking block in the case of a DISO-DIT implementation. To do so, we assume that our different masks are available in an array, at different position. These computations are handled by Algorithms 8 and 9, which together enable the construction of the blinded NTT, as described in Algorithm 7. We implemented our NTT such that it is modular with respect to the block size. If a fixed block size is preferred, the relevant positions could be precomputed for greater efficiency.

For a detailed representation of the graph of the blinded NTT, we refer the reader to Appendix A and Figure 11.

[2] and up to 8064 bits per NTT in Dilithium

Algorithm 7 – `DISO-DIT-NTT`(f): Our Kyber Blinded Number Theoric Transform.

Constant: ω – a primitive 256th root of unity in $\mathbb{Z}_q$, $B = 2^b$ the block size and $N = 2^{7-b}$ the number of blocks.
Require: $f = (f_0, \ldots, f_{255})$ – coefficients of $f(x) \in \mathbb{Z}_q[x]$.
Require: $m = (m_0, \ldots, m_{8N-1})$ – list of mask index with $m_i \in \mathbb{Z}_{2^8}$. First and last N masks set to 0.
Ensure: $\widehat{f} = (\widehat{f_0}, \ldots, \widehat{f_{255}})$ – the NTT of $f(x)$. It is in bit reverse order.

$\mathsf{k} = 0$ ▷ *k denote which roots of unity*
$\mathsf{pad} = 0$
`//Part I, with Normal Blocks`
for $\mathsf{len} = 7$ **to** $\mathsf{len} = b + 1$ **do** ▷ *Stage loop*
  $\mathsf{num} = 0$
  for $\mathsf{st} = 0$ **to** 256 with **step** $2^{\mathsf{len}+1}$ **do** ▷ *NTT standard loop*
    $\mathsf{k} = \mathsf{k} + 1$
    for $j = \mathsf{st}$ **to** $\mathsf{st} + 2^{\mathsf{len}}$ with **step** B **do** ▷ *Block loop*
      $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ `Compute-Normal_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$
      for $i = j$ **to** $j + B$ **do**
        $f_i, f_{i+2^{\mathsf{len}}} \leftarrow$ `DISO-DIT-BF`$(f_i, f_{i+2^{\mathsf{len}}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k})$
        $\mathsf{num} = \mathsf{num} + 1$
  $\mathsf{pad} = \mathsf{pad} + N$
`//Part II, with Mixed Blocks`
for $\mathsf{len} = b$ **to** $\mathsf{len} = 1$ **do** ▷ *Stage loop*
  $\mathsf{num} = 0$
  for $\mathsf{st} = 0$ **to** 256 with **step** $2B$ **do** ▷ *Block loop*
    $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ `Compute-Mixed_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$
    for $j = \mathsf{st}$ **to** $j + 2B$ with **step** $2^{\mathsf{len}+1}$ **do** ▷ *NTT standard loop*
      $\mathsf{k} = \mathsf{k} + 1$
      for $j = i$ **to** $i + 2^{\mathsf{len}-1}$ **do**
        $f_i, f_{i+2^{\mathsf{len}}} \leftarrow$ `DISO-DIT-BF`$(f_i, f_{i+2^{\mathsf{len}}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k})$
        $f_{i+2^{\mathsf{len}-1}}, f_{i+3\cdot 2^{\mathsf{len}-1}} \leftarrow$ `DISO-DIT-BF`$(f_{i+2^{\mathsf{len}-1}}, f_{i+3\cdot 2^{\mathsf{len}-1}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 + 1, \mathsf{k})$
        $\mathsf{num} = \mathsf{num} + 2$
  $\mathsf{pad} = \mathsf{pad} + N$
return f

3.3 A quick cryptanalysis of our Blinded-NTT

We give a quick intuition of why our blinded NTT hampers SASCA. First, the randomness inside the NTT makes it significantly harder from a practical side-channel perspective, though not impossible, to identify the underlying targeted operations within the NTT and compute templates. Nevertheless, Similarly to [RPBC20], our design has the great property to make our butterflies no longer one-to-one. They are in fact surjective (in the case of Kyber, they are exactly 2^{24}-to-one). This hampers the BP algorithm, as we now have to consider the butterfly function

$$f_{BTF}(x_a, x_{a+l}, y_a, y_{a+l}, i, j, m) = 1 \iff \begin{cases} y_i &= \omega^{m-i}x_a + \omega^{k+m-j}x_{a+l} \text{ and} \\ y_{i+l} &= \omega^{m-i}x_a - \omega^{k+m-j}x_{a+l} \end{cases}$$

This therefore heavily slows down the propagation in the BP algorithm.

Secondly, we have that our different masks create numerous additional loops inside the factor graph that represent our NTT. Indeed, as shown in Figure 5, each masks values are linked to $3B$ distinct butterfly functions inside our NTT. Those additional connections ensure the creation of many 4-cycles, the smallest possible. They are of two types:

Algorithm 8 – `Compute_Normal_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$.

Require: $\mathsf{num}, \mathsf{len}, \mathsf{pad}, b$.
Ensure: $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3$

1: $\mathsf{p}_1 \leftarrow \Big(\big((\mathsf{num} \gg b) \gg (\mathsf{len}+1-b)\big) \ll (\mathsf{len}+1-b)\Big) + \big((\mathsf{num} \gg b)\&(2^{\mathsf{len}-b}-1)\big) \,|\mathsf{pad}$
2: $\mathsf{p}_2 \leftarrow \big((\mathsf{p}_1 + 2^{\mathsf{len}-b})\&(N-1)\big) \,|\mathsf{pad}$
3: $\mathsf{p}_3 \leftarrow (\mathsf{num} \gg b)|(\mathsf{pad}+N)$

Algorithm 9 – `Compute_Mixed_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$.

Require: $\mathsf{num}, \mathsf{len}, \mathsf{pad}, b$.
Ensure: $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3$.

1: $\mathsf{p}_1 \leftarrow \big((\mathsf{num} \gg b+1) \ll 1\big)|\mathsf{pad}$
2: $\mathsf{p}_2 \leftarrow \mathsf{p}_1 + 1$
3: $\mathsf{p}_3 \leftarrow \mathsf{p}_1 + \big(\mathsf{num}\&((B-1) \gg (\mathsf{len}-1))\&1\big) + N$

- Variable cycles: These cycles are induced by the masking twiddle factor around each variable inside our NTT. Both nodes are connected by two functions at different stage of the NTT. In our proposed Kyber NTT, there are $5 \cdot 2^8$ such cycles.
- Block cycles: These cycles are induces by the fact that our masking twiddle variables are used for multiple butterflies. Here, both nodes are connected via two functions in the same stage of the NTT. For each pair of butterflies inside a block, there exists 3 such cycles. In our proposed Kyber NTT, there are $16 \cdot \binom{B}{2}$ such cycles.

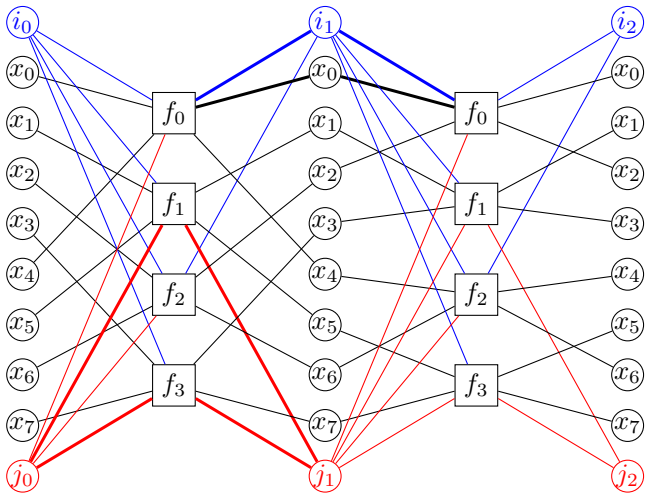


Figure 5: Subset of the factor graph on our blinded NTT with 8 variables and using mixed blocks of size 2. Leak functions are omitted. Examples of variable and block cycles are highlighted.

3.4 Blinded INTT

Our blinded INTT is derived directly from the blinded NTT by leveraging Tellegen's Principle. This ensures that the INTT achieves almost[3] same efficiency as the NTT while maintaining an identical structure and blinding mechanism. The key distinction lies in the

[3]This difference is due to the last scalar multiplication of the INTT.

transformation of DISO-DIT butterflies into SIDO-DIF butterflies. For the sake of brevity, we do not include a detailed description of the INTT in this section. A comprehensive explanation, including all relevant details, is available in Appendix A.

To conclude, it is worth noting that our proposed design can be adapted to incorporate DIDO butterfly operations. Such an adaptation would increase the computational cost by requiring one additional multiplication per butterfly operation compared to the current scheme. However, this approach would further complexify side-channel attacks.

4 CRT-based RNR blinding

In 2021, Heinz and Pöppelmann proposed a side channel and fault protection model for lattice-based schemes [HP21]. Their method is faster than standard masking with shares, which works on the algorithm's linear parts. First, they use RNR for side channel protection. Instead of having coefficients in the field $\mathbb{Z}_q$, they perform linear operations (multiplications, additions, subtractions) in the ring $\mathbb{Z}_{qt}$, where t is an odd number. The coefficients are randomized by adding rq with random $r \in \mathbb{Z}_t$ before operations and derandomized by reducing coefficients modulo q. There are two restrictions on the choice of t. It must be odd to allow fast Montgomery reduction, and it must not be a divider of the 256^{th} root of unity in $\mathbb{Z}_q$; otherwise, going through the NTT will de-randomize the coefficient. Their method is depicted in figure 6. The numbers written on the arrows are the numbers of bits sampled with the TRNG/PRNG.

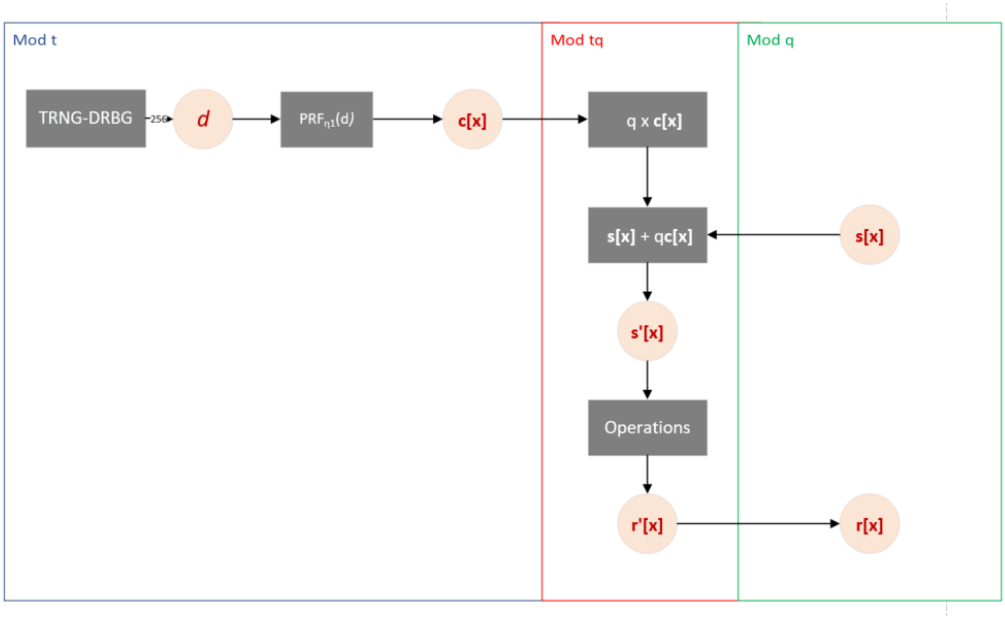
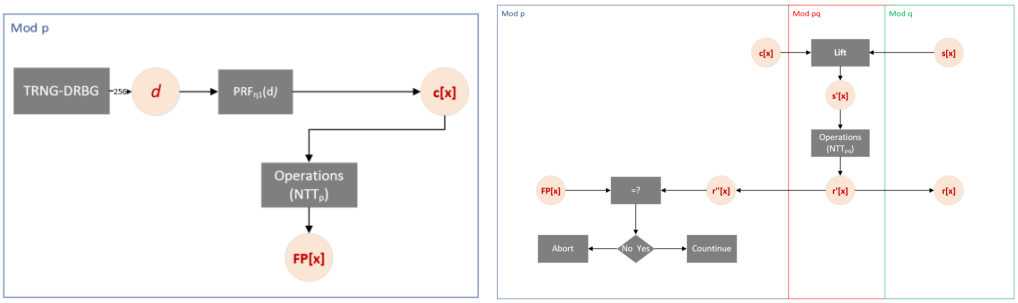


Figure 6: Diagram of the side channel protection of [HP21]

Second, they use a technique based on the CRT for fault protection. Instead of working in the polynomial ring R_q, they work in the ring $R_{pq} = \mathbb{Z}_{pq}[X]/(X^n + 1)$, with p an odd number coprime with q. Thanks to the CRT, we have that $R_{pq} \cong R_p \times R_q$ where the ring R_q contains the input coefficients to protect, and the ring R_p contains constants. We will call this isomorphism *lifting*. It can be easily computed using the formula:

$$\mathsf{Lift}(c_p, c_q) = c_p \cdot q \cdot (q^{-1} \bmod p) + c_q \cdot p \cdot (p^{-1} \bmod q)$$

As shown in Figure 7, their method for fault protection is as follows. During precomputations, they sample a random polynomial $\mathbf{c} \in R_p$ and perform all the operations that have to be protected on $\mathbf{c}$, storing the result. At run time, they lift the polynomial, meaning that when given the input polynomial to protect $\mathbf{s} \in R_q$, use the isomorphism to pass from $(\mathbf{c}, \mathbf{s})$ to a polynomial in R_{pq}. They then perform the desired operations in R_{pq} and finally check that the final value, reduced to R_p, matches the precomputed value. If it does, it returns the value in R_q.



(a) Precomputations. (b) Run time.

.

Figure 7: Diagram of the fault protection of [HP21]

Finally, they proposed a technique to combine both protections. In the fault protection method, instead of precomputing the operations on a single sampled polynomial, they do everything at run time. They also proposed to use different roots of unity for the NTT in R_{pq}. Using the roots of unity they proposed makes the NTT in R_{pq} equivalent to an NTT in R_p and an NTT in R_q. This method requires p to be coprime with q and $\mathbb{Z}_p$ to contain a 256^{th} root of unity.

Although their method costs less than masking with shares, it still has some issues, especially when trying to deploy it in a limited resources environment:

- Sampling the random polynomials takes a non-negligible amount of time.
- Storing the random polynomials adds a significant memory overhead.
- Lifting the polynomials (applying the isomorphism from $R_p \times R_q$ to R_{pq}) introduces a new leakage point.
- Having to perform a second decryption in R_p is time-consuming.

In this section, we tackle these issues and propose an improved countermeasure.

4.1 Our improvements

We instead work in R_{pqt} where R_q contains the computations to protect, R_t is used for side channel protection, and R_p for fault protection, with p an odd prime number greater than q, and t an odd number. p, q and t are pairwise co-prime. The value of t can be changed at each execution, or even between each reduction. The best approach to avoid plummeting the performances is to store a table containing the possible values of pqt and a table for each constant that depends on pqt for the reductions.

To lift the coefficients, instead of sampling the polynomials at random in R_p, we decided to sample a random number in $\mathbb{Z}_p$ and use polynomials with all coefficients equal to this number. This significantly reduces the overhead caused by sampling and verification.

Another critical point is to hide the coefficients before lifting. Lifting the coefficients requires multiplying them with a known constant, which opens a very obvious and easy-to-exploit target for side-channel attacks. By first hiding, we ensure that the multiplication with the constant gives a different trace at each execution.

For the NTT, we perform all the operations in $\mathbb{Z}_{pqt}$ and use different roots of unity. Instead of using the 256^{th} roots of unity ω_q in $\mathbb{Z}_q$ or ω_{pqt} in $\mathbb{Z}_{pqt}$, we use a specific 256^{th} root of unity ω in $\mathbb{Z}_{pq}$:

$$\omega = \mathsf{Lift}(\omega_p, \omega_q)$$

This will lead to some undefined behavior in $\mathbb{Z}_t$ but will result in a normal NTT being performed in $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Our method for combined fault and side channel protection is:

1. Sample a random number $d \in \mathbb{Z}_{pt}$.
2. Hide the polynomial by adding $d \times q$ to each coefficient.
3. Sample a random number $f \in \mathbb{Z}_p$.
4. Lift each coefficient of the hidden polynomial with f.
5. Perform all the operations with the lifted polynomials and obtain $\mathbf{r}'$.
6. Reduce the polynomial modulo p to obtain $\mathbf{r}''$.
7. Compute the expected values based on f.
8. Verify that $\mathbf{r}''$ matches the expected results.
9. Reduce $\mathbf{r}'$ modulo q to obtain the final result.

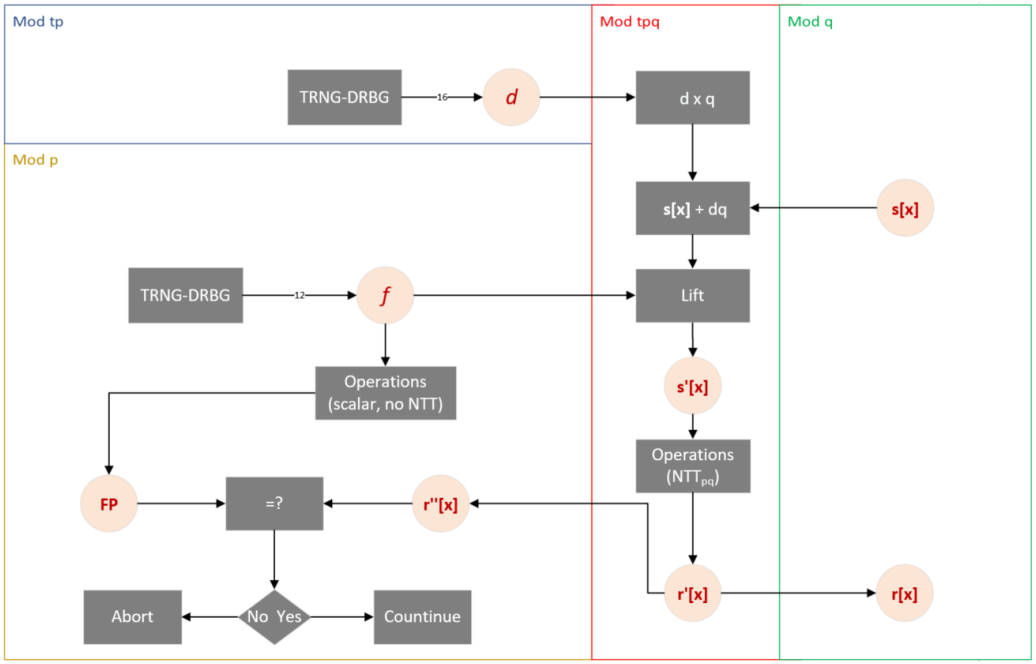


Figure 8: Diagram of our protection.

The main advantages of using a single number to lift all the coefficients are that it reduces the sampling cost and drastically increases the speed of the verification step. Instead of reproducing all the polynomial operations, it suffices to perform a subset of scalar operations to obtain the expected results in $\mathbb{Z}_p$. Let's say we fault protect the polynomials $\mathbf{a}$ and $\mathbf{b}$ with the numbers f_a and f_b. To check that $\mathbf{a} + \mathbf{b}$ ran correctly, it suffices to check that all the coefficients of the result are equal to $f_a + f_b$ in $\mathbb{Z}_p$. We just need a scalar addition instead of a polynomial addition. The same goes for the subtraction. For the multiplication, the verification step is slightly more complicated but still much faster than performing a polynomial multiplication. Instead of performing an NTT, a pointwise product, and an INTT, we first compute $f_{ab} = f_a \times f_b$ and then verify that the coefficient i (for $i \in \mathbb{Z}_n$) of the result is equal to $(2(i+1) - n)f_{ab}$.

Our method solves all the problems listed previously but still has one main issue: the coefficient 127 is expected to be 0 in $\mathbb{Z}_p$ right after a multiplication, regardless of the value of f. This is not trivial to exploit as it involves a single coefficient and requires any fault produced to set this coefficient to either 0 or a multiple of p. The error propagation through the INTT makes it even harder to exploit this problem.

5 Implementation and Evaluation

To evaluate the effectiveness of our countermeasures, we developed a proof-of-concept implementation of Kyber in C, incorporating both countermeasures described in this paper. This implementation is publicly available for review and experimentation at the following repository:

https://github.com/moun18/KYBER-CRT-NTT-SEC

We used this implementation to evaluate the impact of to assess the impact of our countermeasures on performance and security. Specifically, we measured the overhead introduced by our countermeasures and checked that they significatively hindered both side-channel and fault injection attacks.

5.1 Performance

We tested our implementation on an x86 architecture. With blocks of size 8.

The result of our assessment can be found in Tables 1. The overhead represent the slowdown compared to the unprotected version. For example, a 2 means that our protected version is twice as slow as the reference version.

Table 1: Table of our countermeasure's performance (number of cycles) compared to reference.

Algorithm		Reference	CRT-RNR & NTT Blind	Overhead
Kyber-512	indcpa-Key	128 762	521 930	4.05
	indcpa-Enc	179 150	471 779	2.63
	indcpa-Dec	66 404	212 286	3.19
Kyber-768	indcpa-Key	228 819	822 304	3.59
	indcpa-Enc	287 089	705 402	2.45
	indcpa-Dec	88 097	279 121	3.16
Kyber-1024	indcpa-Key	365 707	1 174 802	3.2
	indcpa-Enc	457 149	972 513	2.12
	indcpa-Dec	112 034	332 016	2.96

As can be seen in Table 1, our defense makes the algorithms between 2.12 and 4.05 times slower than an unprotected version, depending on which algorithm is being run. The

slowdown can be explained by the fact that we use multiple defenses, the cost of lifting and verifying polynomial multiplications, and by the non optimization of our code. We also protect every variable, not only the secret ones, making our code more robust but also slower. On top of that we used a relatively small block size.

To reduce this overhead, we could precompute blocks, protect less variables and use bigger blocks.

We can observe a much bigger overhead for the KeyGen function, this is because this function outputs the keys in the NTT domain. So, to be able to perform the known answer tests, we need to first perform an INTT in $\mathbb{Z}_p$ which has a significant cost.

Concerning the memory, we have an overhead of about 1.42. This overhead is explained by the size of the coefficients passing from 16 to 32 bits integers and the extra roots of unity and constants that have to be stored for the reductions.

Table 2: Overhead factor for v - INTT($\hat{s}\circ$NTT(u)) in Kyber768.CPA.Dec decryption.

Architecture	Implementation	Source	Nb. of cycles	Overhead
Cortex-M4	Masking & Redundancy	[KRSS20, ABCG20]	229 922	2.89
Cortex-M4	CRT-RNR	[HP21]	174 858	2.20
x86	CRT-RNR & NTT blinding	**This work**	187 777	2.24

In Table 2, we compare our defense performance with other published defenses. Since we are using a different architecture/reference code and did not optimize our code, the comparison may not be the most relevant. Despite that, we still obtain very good results.

Table 3: Table of our countermeasure's NTT performance (number of cycles) compared to reference.

Block size	Architecture	Implementation	Source	Function	Nb. cycles	Overhead
8	x86	CRT-RNR & NTT Blind	**This work**	NTT INTT	33 895 37 505	2.69 1.93
32	x86	CRT-RNR & NTT Blind	**This work**	NTT INTT	32 888 35 605	2.61 1.84
32	Cortex-M4	NTT Blind	[RPBC20]	NTT INTT	72 100 87 200	2.32 1.72

In Table 3, we show the performance of our NTT compared to the unprotected NTT. We don't observe significant differences between the NTT in $\mathbb{Z}_p$ and the NTT in $\mathbb{Z}_{pqt}$, so we reported both results under the generic NTT/INTT line. This is due to the fact that we use 32 bit int for coefficients in $\mathbb{Z}_p$ even though they should not exceed 12 bits. This explains why we are slightly slower than [RPBC20] who probably still use 16 bit for the NTT. The difference in cycle between [RPBC20] and us is due to the difference in architecture. To compare our works we would recommend looking at the overhead, which was computed based on the reference implementation running on the relevant architecture/reference code.

5.2 Side-Channel resistance

We collected power traces using a ChipWhisperer-Lite. Our analysis was conducted using it for measurement and faults injection and using a separate Arduino Due board running Kyber as the target. Thanks to this setup, we were able to target specific instructions and did not have to spend much time preprocessing the collected data. The experiment's goal was to ensure that our defense fixed, or at least lessened, some of the side channel and fault vulnerabilities. We used this setup to target both the reference implementation [ABD+18] and ours.

We will focus on a specific part of the decryption, where the ciphertext u is multiplied by the secret s. More specifically, we insert an instruction to trigger the capture just after the NTT of u is finished and just before the pointwise product is called. Since the ciphertext is public, the consistent variance in the power consumption will only depend on the value of s, making this product a target of choice for side-channel attacks. Many researchers [KdG21, MPG+22, YWY+23] have successfully targeted this part with CPA attacks.

We choose to go for a Test Vector Leakage Assessment (TVLA), as described by Becker et al. [BCD+13], to evaluate our defense's effectiveness. The idea is to collect two sets of power traces that we expect to have different means due to side channel leakage. We can then assess the likelihood that the difference in their means is only due to variance in the power traces. If the probability is small enough, we consider that there is leakage. We performed the TVLA on both the reference implementation and the protected implementation.

To assess the likelihood of leakage, we used Welch's t-test:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

Where $\overline{X}_i$ is the mean of the power traces of dataset i, N_i is the total amount of power traces in dataset i, and s_i^2 is the variance of the dataset i. Note that each trace contains multiple points corresponding to measurements done at different time periods. Here, all the statistics are performed on a single point of the trace over all the traces of the dataset. So, in practice, we will compute multiple t-tests, each one representing the leakage at a specific time.

We performed a non-specific, fixed vs random key TVLA. Data set 1 consists of a fixed key Dk and a set of random ciphertexts [c_0 to c_n]. Data set 2 consists of a set of random keys [Dk_0 to Dk_n] and the same set of ciphertexts as in dataset 1. For data set 1, the trace i corresponds to the decapsulation of the c_i with the key Dk. For dataset 2, the trace i corresponds to the decapsulation of the c_i with the key Dk_i.

We collected 10 000 traces per dataset, each trace containing 5000 measurements. All the measurements are performed during the pointwise product.

We performed two independent t-tests with the following sets of traces:

1. Test 1: t-test comparing the **first** 5000 traces of dataset 1 to the **first** 5000 traces of dataset 2.

2. Test 2: t-test comparing the **last** 5000 traces of dataset 1 to the **last** 5000 traces of dataset 2.

We consider that the device leaked information about the secret if both Test 1 and Test 2 show a t-score smaller than -4.5 or bigger than 4.5 **at the same point in time**. To improve readability, when showing the results of the unprotected decapsulation, we zeroed out both t-scores if one of the two tests had a value between [-4.5 and 4.5]. This allows us to only show points where leakage occurs. Figure 9 shows the TVLA of the unprotected version, while Figure 10 shows the TVLA of the decapsulation protected by our countermeasures.

As expected, the test shows clear leakage on the unprotected decapsulation meanwhile the protected decapsulation shows no sign of leakage. In theory, if we could take pt times more traces (about 1 billion traces), we should again see clear leakage even with the protected decapsulation. However, collecting so many traces of decapsulation with the same key would require an unrealistic amount of time and storage.

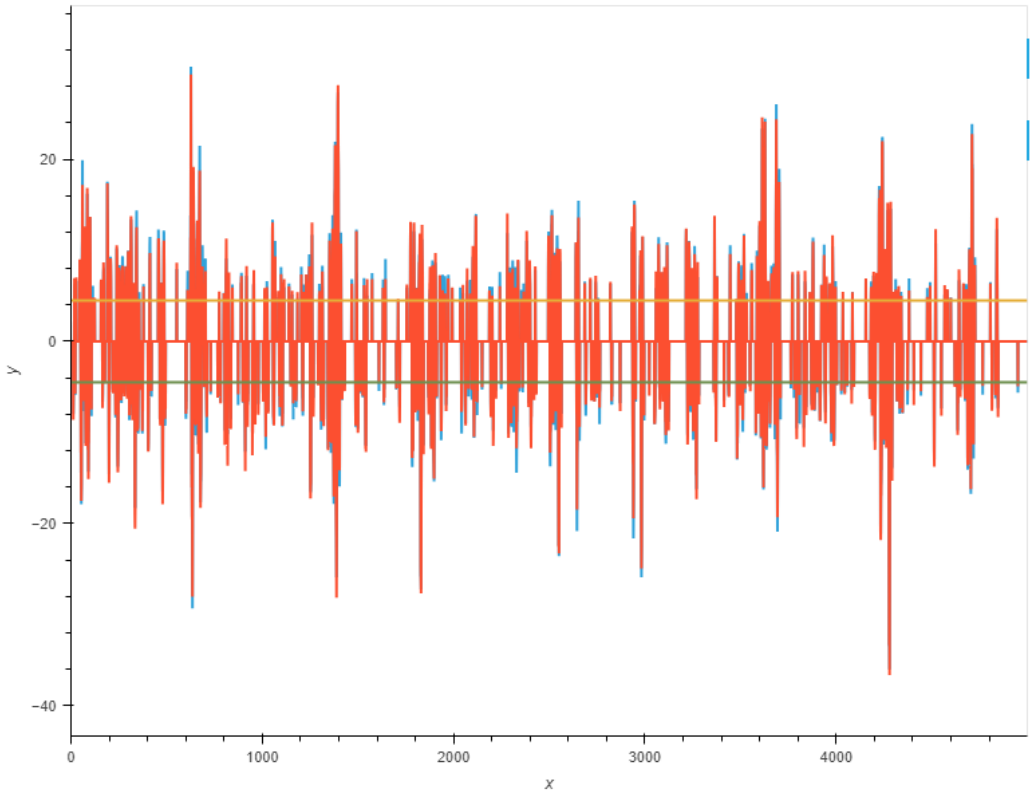


Figure 9: TVLA results of **unprotected** decapsulation. The x-axis represents time, and the y-axis represents the t-score. Test 1 is represented in blue; Test 2 is represented in red.

5.3 Fault attacks

For the faults, we will focus on the different functions executed during the decryption: the NTT of u, the pointwise product of u and s, the INTT of the result, and the subtraction of the result to v. This covers most of the linear functions in Kyber.

We decided to use voltage glitching. We removed some of the capacitors of the Arduino Due target to permit sudden tension changes. Those changes will cause instruction skips or data corruption, leading to either a crash of the decapsulation or a wrong value computed.

In our setup, the glitches were ineffective the majority of the time, leading to a normal execution without faults. Because of this, it was hard to spot effective glitches leading to ineffective faults. To evaluate our defense, we only considered the effective faults to get a better overview. In a more consistent setup, where the fault almost always occurs, it would be interesting to study ineffective faults and see how well our defense works against them.

We applied at different points in time for each function and tried the glitch about 300 times per point in time.

The results are compiled in the Tables 4 and 5. For each function, we tried to fault it at different moments to cover a wide variety of instruction skips and/or data corruption. Most of the time, the plaintext obtained in the faulted decapsulation produced a different ciphertext when re-encrypted. We marked those cases as “Arrived to FO”. Note that in those cases, the faulted plaintext is written in memory, even if it is not returned at the end. If the plaintext was detected by our defense, we marked it as “Detected before FO-transform”. In those cases, the decapsulation is aborted before writing the plaintext in memory.

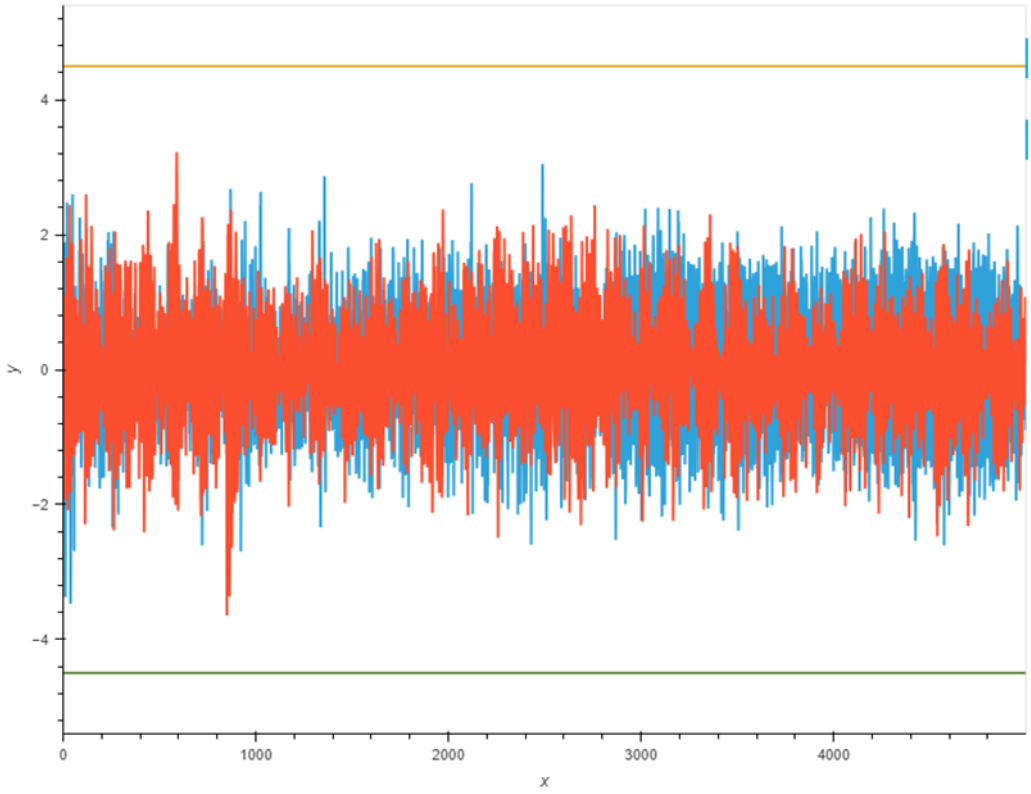


Figure 10: TVLA results of **protected** decapsulation. The x-axis represents time, and the y-axis represents the t-score. Test 1 is represented in blue; Test 2 is represented in red.

Table 4: Table of the effective faults obtained for the **unprotected** decapsulation.

Function	Total faults	Crash	Detected before FO-transform	Arrived to FO
NTT	2082	1253	0	829
Pointwise mul	2656	1142	0	1514
INTT	2549	2017	0	532
Sub	792	684	0	108

Table 5: Table of the effective faults obtained for the **protected** decapsulation.

Function	Total faults	Crash	Detected before FO-transform	Arrived to FO
NTT	3905	2371	1479	55
Pointwise mul	5095	2195	2900	0
INTT	3455	2108	1347	0
Sub	505	362	143	0

As expected, the unprotected version was subject to many effective faults that required the FO-transform to be detected. Thanks to our defense, the protected version detected almost all of them, avoiding the reencryption of faulted message. In both versions, there were no time points during which no effective faults occurred.

It is important to note that there were still some undetected effective faults in the protected decapsulation, all occurring during the NTT of u. There were few points in

time where the effective fault could happen; however, during those points, it was common to have the effective fault occurring multiple times and producing the same result. This means that attacks that rely on modifying the twiddle factors like the one of Ravi et al. [RYB+22] would probably be detected. It would still be interesting to go into details to check exactly what were the faults produced and if it is exploitable or not.

6 Conclusion

In this paper, we introduced two key improvements to blinding countermeasures: a Blinded NTT leveraging distinct masking blocks and an enhanced RNR mechanism based on the CRT. These countermeasures complement each other effectively, as the first focuses on the roots of unity while the second targets the variables. By incorporating both into our proof-of-concept C implementation of Kyber, we demonstrated their practical feasibility. Our evaluation showed promising results in terms of both efficiency and resistance to side-channel attacks, paving the way for further research. In particular, exploring optimized hardware implementations of these mechanisms presents an exciting avenue for future work.

References

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR TCHES*, 2020(3):336–357, 2020.

[ABD+18] Roberto Avanzi, Joppe Bosa, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber. Submission to [NIS], 2018.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR TCHES*, 2022(1):127–151, 2022.

[BCD+13] Georg T. Becker, Jim Cooper, Elizabeth K. DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, T. Kouzminov, Andrew J. Leiserson, Mark E. Marson, Pankaj Rohatgi, and Sami Saab. Test Vector Leakage Assessment (TVLA) methodology in practice, 2013. https://api.semanticscholar.org/CorpusID:28168779.

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

[BLS03] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, ISSAC '03, pages 37 – 44, New York, NY, USA, 2003. Association for Computing Machinery.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[Del22] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked Kyber. *IACR TCHES*, 2022(4):637–660, 2022.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. Association for Computing Machinery.

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

[HMS+23] Julius Hermelink, Erik Mårtensson, Simona Samardjiska, Peter Pessl, and Gabi Dreo Rodosek. Belief propagation meets lattice reduction: Security estimates for error-tolerant key recovery from decryption errors. *IACR TCHES*, 2023(4):287–317, 2023.

[HP21] Daniel Heinz and Thomas Pöppelmann. Combined fault and DPA protection for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/101, 2021.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on kyber. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *INDOCRYPT 2021*, volume 13143 of *LNCS*, pages 311–334. Springer, Cham, December 2021.

[HSST23] Julius Hermelink, Silvan Streit, Emanuele Strieder, and Katharina Thieme. Adapting belief propagation to counter shuffling of NTTs. *IACR TCHES*, 2023(1):60–88, 2023.

[KdG21] Alexandre Karlov and Natacha Linard de Guertechin. Power analysis attack on Kyber. Cryptology ePrint Archive, Report 2021/1311, 2021.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on Keccak. *IACR TCHES*, 2020(3):243–268, 2020.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019.

[KRSS20] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. Pqm4: Post-quantum crypto library for the arm cortex-m4, 2020, accessed 12/16/2020, 2020. https://github.com/mupq/pqm4.

[KT23] Yen-Ting Kuo and Atsushi Takayasu. A lattice attack on CRYSTALS-kyber with correlation power analysis. In Hwajeong Seo and Suhri Kim, editors, *ICISC 23, Part I*, volume 14561 of *LNCS*, pages 202–220. Springer, Singapore, November / December 2023.

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Berlin, Heidelberg, May / June 2010.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[MPG+22] Haocheng Ma, Shijian Pan, Ya Gao, Jiaji He, Yiqiang Zhao, and Yier Jin. Vulnerable PQC against Side Channel Analysis - A Case Study on Kyber. In *2022 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6, 2022. https://doi.org/10.1109/AsianHOST56390.2022.10022165.

[NIS] NIST. Post-Quantum Cryptography Standardization.

[NIS24a] NIST. Module-lattice-based digital signature scheme standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 204, U.S. Department of Commerce, Washington, D.C., August 2024.

[NIS24b] NIST. Module-lattice-based key-encapsulation mechanism standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 203, U.S. Department of Commerce, Washington, D.C., August 2024.

[NIS24c] NIST. Stateless hash-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 205, U.S. Department of Commerce, Washington, D.C., August 2024.

[OY23] Sila Ozeren and Oguz Yayla. Methods for Masking CRYSTALS-Kyber Against Side-Channel Attacks. In *2023 16th International Conference on Information Security and Cryptology (ISCTürkiye)*, pages 1–6, 2023. http://dx.doi.org/10.1109/ISCTrkiye61151.2023.10336068.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *LATINCRYPT 2019*, volume 11774 of *LNCS*, pages 130–149. Springer, Cham, October 2019.

[PP21] Peter Pessl and Lukas Prokop. Fault attacks on CCA-secure lattice KEMs. *IACR TCHES*, 2021(2):37–60, 2021.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 513–533. Springer, Cham, September 2017.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable sca countermeasures against single trace attacks for the ntt. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 123–146, Cham, 2020. Springer International Publishing.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 683–702. Springer, Berlin, Heidelberg, September 2015.

[RYB+22] Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform. Cryptology ePrint Archive, Paper 2022/824, 2022. https://eprint.iacr.org/2022/824.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[SML24] Ardianto Satriawan, Rella Mareta, and Hanho Lee. A complete beginner guide to the number theoretic transform (NTT). Cryptology ePrint Archive, Paper 2024/585, 2024.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 282–296. Springer, Berlin, Heidelberg, December 2014.

[YFW00] Jonathan S Yedidia, William Freeman, and Yair Weiss. Generalized belief propagation. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000.

[YWY+23] Yipei Yang, Zongyue Wang, Jing Ye, Junfeng Fan, Shuai Chen, Huawei Li, Xiaowei Li, and Yuan Cao. Chosen ciphertext correlation power analysis on Kyber. *Integration*, 91:10–22, 2023. https://doi.org/10.1016/j.vlsi.2023.02.012.

[ZXZ+18] Shuai Zhou, Haiyang Xue, Daode Zhang, Kunpeng Wang, Xianhui Lu, Bao Li, and Jingnan He. Preprocess-then-NTT technique and its applications to KYBER and NEWHOPE. Cryptology ePrint Archive, Report 2018/995, 2018.

A Generic blinded NTT

The blinded NTT we described in Section 3 is specialized to Kyber, but the method can be adapted to any NTT-based cryptographic scheme (e.g., Dilithium) and, more broadly, to any FFT-based scheme. In this appendix, we provide the generalized pseudocode for both a complete NTT and its corresponding INTT.

The full pseudocode for the complete blinded NTT is provided in Algorithm 12, while the corresponding INTT is described in Algorithm 13, which employs the SIDO-DIF method (as defined in Algorithm 11).

Additionally, for clarity, Figure 11 presents a fully specified diagram of our NTT, including the computations and masking strategies used across all stages.

Algorithm 10 – `Compute-Final_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$. Compute positions in the final column

Require: $\mathsf{num}, \mathsf{len}, \mathsf{pad}, b$.
Ensure: $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3$.
1: $\mathsf{p}_1 \leftarrow ((\mathsf{num} \gg b + 1) \ll 1) + \mathsf{pad}$
2: $\mathsf{p}_2 \leftarrow \mathsf{p}_1 + 1$
3: $\mathsf{p}_3 \leftarrow (\mathsf{num} \gg b) + \mathsf{pad} + N$

Algorithm 11 – SIDO-DIF-BF$(f_0, f_1, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, k, t)$: (Different Input Same Output Derivation in Time Butterfly).

Require: f_0, f_1 in $\mathbb{F}$ and $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, k \in \mathbb{Z}_{2^n}$.
Ensure: f_0, f_1 in $\mathbb{F}$.
1: $p \leftarrow f_0 + f_1$
2: $q \leftarrow f_0 - f_1$
3: $f_0 \leftarrow \omega^{m_{\mathsf{p}_1} - m_{\mathsf{p}_3}} p$
4: $f_1 \leftarrow \omega^{\mathtt{bitrv}(k) + m_{\mathsf{p}_2} - m_{\mathsf{p}_3}} q$ ▷ *$bitrv(k)$ is the bit-reverse of k in $\mathbb{Z}_{2^n}$.*
5: **return** f_0, f_1

Algorithm 12 – DISO-DIT-NTT(f): Tukey–Cooley Blinded Number Theoric Transform.

Constant: ω - a 2^n root of unity in $\mathbb{F}$, $B = 2^b$ the block size and $2N = 2^n/B$ the number of blocks.
Require: $f = (f_0, \ldots, f_{2^n-1})$ - coefficients of $f(x) \in \mathbb{F}[x]$.
Require: $m = (m_0, \ldots, m_{(n+1)N-1})$ - list of mask index with $m_i \in \mathbb{Z}_{2^n}$. First and last N masks set to 0.
Ensure: $\widehat{f} = (\widehat{f_0}, \ldots, \widehat{f_{2^n-1}})$ – the NTT of $f(x)$. It is in bit reverse order.
1: $\mathsf{k} = 0$
2: $\mathsf{pad} = 0$
3: `//Part I, with Normal Blocks`
4: **for** $\mathsf{len} = n - 1$ **to** $\mathsf{len} = b + 1$ **do**
5: $\quad \mathsf{num} = 0$
6: $\quad$ **for** $\mathsf{st} = 0$ **to** 2^n with **step** $2^{\mathsf{len}+1}$ **do**
7: $\quad\quad \mathsf{k} = \mathsf{k} + 1$
8: $\quad\quad$ **for** $j = \mathsf{st}$ **to** $\mathsf{st} + 2^{\mathsf{len}}$ with **step** B **do**
9: $\quad\quad\quad \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ Compute-Normal_Block_Pos$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$ ▷ *Algorithm 8*
10: $\quad\quad\quad$ **for** $i = j$ **to** $j + B$ **do**
11: $\quad\quad\quad\quad f_i, f_{i+2^{\mathsf{len}}} \leftarrow$ DISO-DIT-BF$(f_i, f_{i+2^{\mathsf{len}}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k})$
12: $\quad\quad\quad\quad \mathsf{num} = \mathsf{num} + 1$
13: $\quad \mathsf{pad} = \mathsf{pad} + N$
14: `//Part II, with Mixed Blocks`
15: **for** $\mathsf{len} = b$ **to** $\mathsf{len} = 1$ **do**
16: $\quad \mathsf{num} = 0$
17: $\quad$ **for** $\mathsf{st} = 0$ **to** 2^n with **step** $2B$ **do**
18: $\quad\quad \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ Compute-Mixed_Block_Pos$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$ ▷ *Algorithm 9*
19: $\quad\quad$ **for** $j = \mathsf{st}$ **to** $j + 2B$ with **step** $2^{\mathsf{len}+1}$ **do**
20: $\quad\quad\quad \mathsf{k} = \mathsf{k} + 1$
21: $\quad\quad\quad$ **for** $j = i$ **to** $i + 2^{\mathsf{len}-1}$ **do**
22: $\quad\quad\quad\quad f_i, f_{i+2^{\mathsf{len}}} \leftarrow$ DISO-DIT-BF$(f_i, f_{i+2^{\mathsf{len}}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k}, 0)$
23: $\quad\quad\quad\quad f_{i+2^{\mathsf{len}-1}}, f_{i+3*2^{\mathsf{len}-1}} \leftarrow$ DISO-DIT-BF$(f_{i+2^{\mathsf{len}-1}}, f_{i+3*2^{\mathsf{len}-1}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 + 1, \mathsf{k})$
24: $\quad\quad\quad\quad \mathsf{num} = \mathsf{num} + 2$
25: $\quad \mathsf{pad} = \mathsf{pad} + N$
26: `//Part III, Last Stage`
27: $\mathsf{num} = 0$
28: **for** $\mathsf{st} = 0$ **to** 2^n with **step** $2B$ **do**
29: $\quad \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ Compute_Final_Block_Pos$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$ ▷ *Algorithm 10*
30: $\quad$ **for** $i = \mathsf{st}$ **to** $\mathsf{st} + 2B$ with **step** 2 **do**
31: $\quad\quad \mathsf{k} = \mathsf{k} + 1$
32: $\quad\quad f_i, f_{i+1} \leftarrow$ DISO-DIT-BF$(f_i, f_{i+1}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k})$
33: $\quad\quad \mathsf{num} = \mathsf{num} + 1$
34: **return** $f = (f_0, \ldots, f_n)$

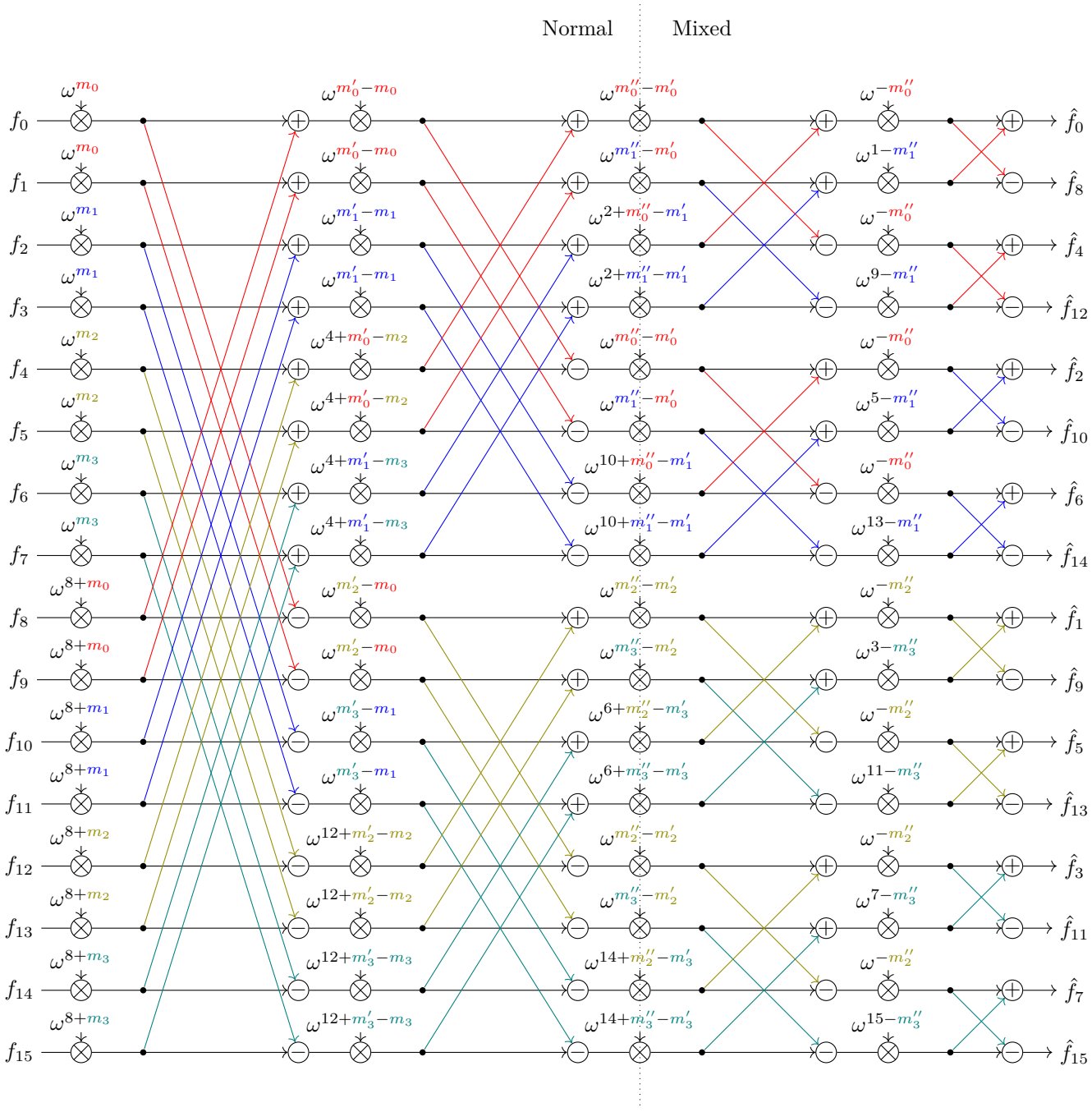


Figure 11: Diagram of a DIT-NTT using 4 blocks (n=16)

Algorithm 13 – `SIDO-DIF-INTT`(f): Gentelman-Sande fast Blinded Inverse Number Theoric Transform.

Constant: ω - a 2^n root of unity in $\mathbb{F}$, $B = 2^b$ the block size and $2N = 2^n/B$ the number of blocks.
Require: $\widehat{f} = (\widehat{f_0}, \ldots, \widehat{f_{2^n-1}})$ - the NTT of $f(x)$.
Require: $m = (m_0, \ldots, m_{(n+1)N-1})$ - list of mask index with $m_i \in \mathbb{Z}_{2^n}$. First and last N masks set to 0.
Ensure: $f = (f_0, \ldots, f_{2^n-1})$ - in bit reverse order.

1: $\mathsf{k} = 2^n$
2: $\mathsf{pad} = nN$
3: $\mathsf{num} = 0$
4: `//Part III, First Row`
5: **for** $\mathsf{st} = 0$ **to** 2^n with **step** $2B$ **do**
6: $\quad \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ `Compute_Final_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$ ▷ *Algorithm 10*
7: $\quad$ **for** $i = \mathsf{st}$ **to** $\mathsf{st} + 2B$ with **step** 2 **do**
8: $\quad\quad \mathsf{k} = \mathsf{k} - 1$
9: $\quad\quad f_i, f_{i+1} \leftarrow$ `SIDO-DIF-BF`$(f_i, f_{i+1}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k}+1)$
10: $\quad\quad \mathsf{num} = \mathsf{num} - 1$
11: $\mathsf{pad} = \mathsf{pad} - N$
12: `//Part II, with Mixed Blocks`
13: **for** $\mathsf{len} = 1$ **to** $\mathsf{len} = b$ **do**
14: $\quad \mathsf{num} = 0$
15: $\quad$ **for** $\mathsf{st} = 0$ **to** 2^n with **step** $2B$ **do**
16: $\quad\quad \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ `Compute_Mixed_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad})$ ▷ *Algorithm 9*
17: $\quad\quad$ **for** $j = \mathsf{st}$ **to** $j + 2B$ with **step** $2^{\mathsf{len}+1}$ **do**
18: $\quad\quad\quad \mathsf{k} = \mathsf{k} - 1$
19: $\quad\quad\quad$ **for** $j = i$ **to** $i + 2^{\mathsf{len}-1}$ **do**
20: $\quad\quad\quad\quad f_i, f_{i+2^{\mathsf{len}}} \leftarrow$ `SIDO-DIF-BF`$(f_i, f_{i+2^{\mathsf{len}}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k}+1)$
21: $\quad\quad\quad\quad f_{i+2^{\mathsf{len}-1}}, f_{i+3*2^{\mathsf{len}-1}} \leftarrow$ `SIDO-DIF-BF`$(f_{i+2^{\mathsf{len}-1}}, f_{i+3*2^{\mathsf{len}-1}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3+1, \mathsf{k}+1)$
22: $\quad\quad\quad\quad \mathsf{num} = \mathsf{num} - 2$
23: $\quad \mathsf{pad} = \mathsf{pad} - N$
24: `//Part I, with Normal Blocks`
25: **for** $\mathsf{len} = b + 1$ **to** $\mathsf{len} = n - 1$ **do**
26: $\quad \mathsf{num} = 0$
27: $\quad$ **for** $\mathsf{st} = 0$ **to** 2^n with **step** $2^{\mathsf{len}+1}$ **do**
28: $\quad\quad \mathsf{k} = \mathsf{k} - 1$
29: $\quad\quad$ **for** $j = \mathsf{st}$ **to** $\mathsf{st} + 2^{\mathsf{len}}$ with **step** B **do**
30: $\quad\quad\quad \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \leftarrow$ `Compute_Normal_Block_Pos`$(\mathsf{num}, \mathsf{len}, \mathsf{pad}, b)$ ▷ *Algorithm 8*
31: $\quad\quad\quad$ **for** $i = j$ **to** $j + B$ **do**
32: $\quad\quad\quad\quad f_i, f_{i+2^{\mathsf{len}}} \leftarrow$ `SIDO-DIF-BF`$(f_i, f_{i+2^{\mathsf{len}}}, \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{k}+1)$
33: $\quad\quad\quad\quad \mathsf{num} = \mathsf{num} - 1$
34: $\quad \mathsf{pad} = \mathsf{pad} - N$
35: **for** $i = 0$ **to** 2^n **do**
36: $\quad f_i \leftarrow f_i * 2^{-n}$ ▷ 2^{-n} *the inverse of* 2^n *in* $\mathbb{F}_q$.
37: **return** $f = (f_0, \ldots, f_n)$