Combined blinding Side-Channel and Fault countermeasures for Lattice-Based schemes

Mounir Taha[1], Julien Cam[1,2], Max Duparc[3], Hervé Pelletier[1], and Karine Villegas[1]

[1]Nagra Kudelski Group, Cheseaux-sur-Lausanne, Switzerland, mounir.taha@nagra.com; julien.cam@nagra.com; herve.pelletier@nagra.com; karine.villegas@nagra.com
[2]University of Rennes, Rennes, France, julien.cam@univ-rennes.fr
[3]EPFL, Lausanne, Switzerland, max.duparc@epfl.ch

Abstract. In this paper, we revisit and enhance two blinding countermeasures aimed at protecting implementations of lattice-based cryptographic schemes against side-channel and fault-injection attacks. First, we build upon the Number Theoretic Transform (NTT) blinding technique introduced by Ravi et al., and optimize the use of blinding twiddle factors so as to maximize their number without increasing the total number of modular multiplications. Second, we improve the Chinese Remainder Theorem (CRT)–based blinding scheme proposed by Heinz and Pöppelmann by substantially reducing the randomness-sampling overhead and streamlining the associated fault-detection mechanism. Importantly, the two countermeasures are fully compatible and can be seamlessly combined to provide enhanced resistance to both classical and soft-analytical side-channel attacks, as well as fault-injection attacks, while maintaining a performance overhead comparable to existing approaches. We demonstrate the practicality of our proposals through a proof-of-concept implementation integrated into Kyber/ML-KEM.

Keywords: Blinding · NTT · CRT · Side-Channel · Fault Attacks · ML-KEM.

1 Introduction

1.1 Post-Quantum Cryptography

With the release of the first standards by the NIST [25, 24, 26], post-quantum cryptography has taken a significant step towards integration into everyday applications. Among these standards, ML-KEM [25], previously known as Kyber [34], is currently the only standardized Key Encapsulation Mechanism (KEM), soon to be joined by HQC. Based on the Module-Learning-With-Errors (MLWE) problem, ML-KEM offers relatively small public keys and ciphertexts along with competitive performance.

In the context of embedded devices, its efficiency has been demonstrated in various implementations [16]. However, ML-KEM faces there new challenges, since these devices can typically be the target of physical attacks.

1.2 Physical Attacks

Physical attacks describe a class of attacks against cryptographic implementations, where the attacker not only has access to the input and output, but can also interact with the execution of the algorithms. Side-Channel Attack (SCA) gather the scenarios where a passive attacker observes some leakage from the computing device. Conversely, Fault Injection Attacks denote the ability of the attacker to actively modify the operation of the algorithm.

Power Analysis is a SCA exploiting variations in power consumption caused by intermediate computations and register transitions to infer information about some sensitive data manipulated by the algorithm. Several techniques have been developed, including Simple Power Analysis (SPA), Differential Power Analysis (DPA) and Correlation Power Analysis (CPA) [18, 6].

An electromagnetic attack exploits similar mechanisms, but it can offer better spatial and spectral selectivity, as well as higher sampling rates [9, 23]. This can allow to reduce noise from unrelated components and capture high-frequency effects such as CMOS glitches [22].

A Soft-Analytical Side-Channel Attack (SASCA) is a specific attack operating in two phases. First, the profiling phase collects template traces from a clone device. Then, a single trace from the target device is matched to the templates and modeled as a factor graph, which is then processed by the Belief Propagation (BP) algorithm [21].

Fault injection attacks usually operate in two phases. 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).

1.3 Physical attacks on Post-Quantum schemes

Compared to their classical counterparts (RSA, ECDH), post-quantum schemes exhibit substantially higher algorithmic and implementation complexity, broadening the attack surface, hence the relevance and effectiveness of physical attacks. As a result, achieving robust and leakage-resilient implementations remains particularly challenging in physically exposed or otherwise uncontrolled environments, such as those commonly found in IoT deployments.

Against ML-KEM, Correlation Power Analysis (CPA), has already proven to be particularly effective [20, 37, 19, 17]. SASCAs have also targeted the Number Theoretic Transform (NTT) to recover secret information [30, 28, 10, 14]. Regarding faults, they are partially mitigated by the Fujisaki-Okamoto transform (FO-transform), which ensures that decryption errors (whether natural or deliberately injected), result in a mismatch between the original and re-encrypted ciphertexts, preventing the release of faulty outputs. However, this protection does not totally prevents fault attacks: instruction skips may bypass the re-encryption check, or crafted invalid ciphertexts can induce perturbations that are corrected during decryption while still leaking information [29, 13, 7, 15].

1.4 Countermeasures for Post-Quantum schemes

To prevent these physical attacks, several countermeasures have been proposed.

Masking splits sensitive intermediate values into multiple independent shares, performing computations on each share separately before applying a final recombination step [33, 27, 4]. While masking can provide strong protection against classical side-channel attacks, it does not always prevent more advanced threats such as SASCA [10] or fault-injection attacks [15, 36]. Moreover, it has a significant performance and implementation overhead, as computations must be replicated for each share.

Blinding introduces "noise" to increase the complexity of side-channel analysis. Such techniques may take various forms, like inserting random delays at the instruction level, computing on secret values up to a random scalar (the blinding factor) [32], or leveraging Redundant Number Representation (RNR) based on the Chinese Remainder Theorem (CRT) [12].

Specifically against SASCA, we can increase the number of (small) loops in the factor graph of the NTT [32]. Indeed, the BP algorithm guarantees correct results for acyclic graphs. Although it can also be used for cyclic graphs, its accuracy and convergence degrade as the number of loops in the graph increases, or as they become shorter [30, 28]. Improved methods such as the Generalized Belief Propagation can solve this issue, but their higher computational cost is prohibitive for the large factor graphs used for SASCA on NTT [38].

Some methods, like the CRT-based RNR blinding mentioned above, can provide protection against both side-channel and fault attacks at the same time, which make them very interesting mechanmisms [12].

At the end, designing side-channel resistant implementations requires selecting countermeasures, while carefully evaluating their mutual compatibility, since some may conflict with others, potentially leading to a substantial increase in implementation complexity and resource overhead.

1.5 Contributions

In this paper, we improve the blinding mechanisms introduced in [32] and [12], both in terms of security and efficiency.

First, in section 2, we introduce a new blinded NTT construction, based on mixing blocks in late stages. This design increases the number of cycles in the factor graph underlying SASCA, thereby reducing the efficiency of such attacks while preserving correctness and practical performance.

Second, in section 3, we describe an improved CRT-based RNR blinding mechanism, which requires to sample fewer random bits. The proposed modification accelerates the verification phase and improves the integration of the fault-detection mechanism while providing protection against both side-channel and fault-injection attacks. We also decrease the countermeasure's overhead by designing an ultra-fast NTT which takes advantage of its input's structure.

Since the first countermeasure protects the commonly targeted NTT, while the second protects the linear parts, they are totally compatible with each other,

so they can be combined to protect a larger portion of the algorithms. We illustrate that in section 4, by integrating our two improved countermeasures to an implementation of Kyber/ML-KEM. This also allows to evaluate the computational overheads of our countermeasures, and their resistance to side-channel and fault-injection attacks.

2 Blinded NTT Using Mixed Blocks

In order to protect the Number Theoretic Transform against SASCA, Ravi et al. introduced two methods [32].

Blinding-in-time involves introducing time delays and randomly shuffling the execution of different butterflies within each NTT stage. This increases the complexity of correlating side-channel information with internal computations, making attacks more challenging. However, as shown in [14], while this countermeasure is effective, it is not impervious to attacks by a powerful adversary, who can still achieve success under certain conditions. Moreover, this approach substantially increases implementation complexity, necessitates extensive random shuffling, and disrupts the constant-time properties.

Blinding-in-memory consists in applying multiplicative masking to NTT intermediate values using a blinding twiddle factor. This randomization introduces loops in the associated factor graph, which reduces the effectiveness of SASCA by effectively obstructing the convergence of the Belief Propagation algorithm. Although the approach maintains constant-time properties, a naive application to each butterfly operation requires significant amounts of randomness. To mitigate this overhead, masking blocks can be employed, using a single blinding twiddle factor to mask a group of butterflies, thereby reusing randomness.

In this section, we first detail the blinding-in-memory technique. Then, we describe the logic of masking blocks, which we improve to introduce more loops in the factor graph without any performance overhead.

2.1 The Number Theoretic Transform

The Number Theoretic Transform (NTT) is a method extensively used by schemes based on stuctured lattices to achieve competitive running times. Indeed, since they operate in modular polynomial rings such as $R_q = \mathbb{Z}_q[X]/\langle X^n+1\rangle$, the bottleneck of such schemes often lies in the multiplication of two polynomials, which naively requires $O(n^2)$ multiplications in $\mathbb{Z}_q$. NTT can make these schemes run faster by reducing the cost of one product in R_q to $O(n\log(n))$ multiplications in $\mathbb{Z}_q$. The idea is to leverage an efficiently computable ring isomorphism between $(R_q, +, \times)$ and $(T_q, +, \circ)$, where the product $\circ$ over T_q is easily computable.

More precisely, the `NTT` function is the isomorphism that transforms a polynomial in R_q into its NTT form in T_q, and can be computed with $O(n\log(n))$ multiplications in $\mathbb{Z}_q$. Then, the product $\circ$ of two elements in T_q can be computed with only $O(n)$ multiplications in $\mathbb{Z}_q$. Finally, the `INTT` function is the inverse isomorphism, which recovers a polynomial in R_q from its NTT form in

T_q, with $O(n\log(n))$ multiplications in $\mathbb{Z}_q$. Therefore, for f and g in R_q, directly computing $h = f \times g$ requires $O(n^2)$ multiplications in $\mathbb{Z}_q$, while computing $h = \texttt{INTT}(\texttt{NTT}(f) \circ \texttt{NTT}(g))$ requires only $O(n\log(n))$ such multiplications.

As the NTT is a natural extension of the Fast Fourier Transform (FFT), the same techniques can also be applied for its computation. Most notably, efficient algorithms rely on some predefined primitive $2n$-th root of unity $\omega \in \mathbb{Z}_q$. In this paper, we implement `NTT` using Decimation In Time (DIT) and `INTT` using Decimation In Frequency (DIF). This design choice is guided by Tellegen's principle [5, Proposition 2], which guarantees that any efficient implementation of the DIT-NTT induces a corresponding DIF-INTT with the same computational complexity. The NTT algorithm is illustrated in fig. 1.

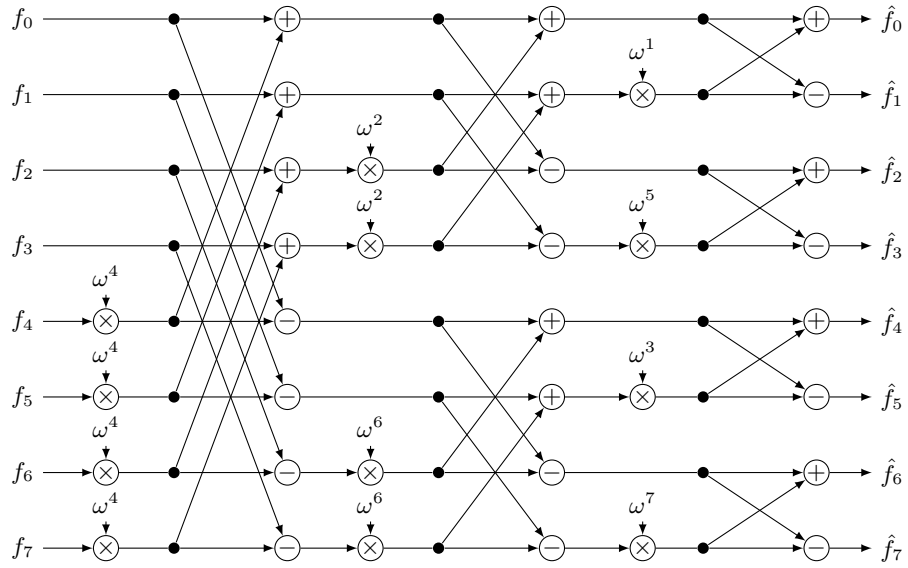


Fig. 1. Toy example of the DIT-NTT with $n = 8$.



(a) The DIF butterfly (b) The DIT butterfly

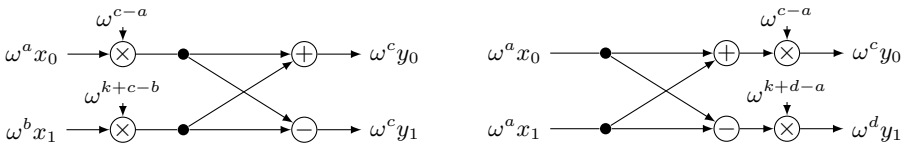
Fig. 2. The two types of butterflies we consider in this paper

At their core, 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 and DIF butterflies are described in fig. 2. Both are dual of each other, which means that the matrix representing a DIT-butterfly is the transpose of the DIF-butterfly matrix. In both `NTT` and `INTT`, each stage consists in groups of interleaved butterflies. In this paper, we refer to such a group as a *swarm*.

2.2 Blinding the Butterflies

Similarly to [32], the fundamental building block of our blinded NTT consists in blinded butterflies, although different ones. A *blinded butterfly* inputs the initial data multiplied by some known blinding twiddle factors, and outputs the result of the butterfly multiplied by some known twiddle factors. As a butterfly has 2 inputs and 2 outputs, we can design 4 distinct types of blinded butterflies, based on whether the blinding twiddle factors on both inputs (and both outputs) are equal. We denote them SISO (Same Input, Same Output), DISO (Different Input, Same Output), SIDO and DIDO. Additionally, all these variants exist for both types of standard butterflies: Decimation In Time (DIT) and Decimation In Frequency (DIF). Finally, this results in a total of 8 possible types of blinded butterflies. Among them, SISO-DIT, SISO-DIF, DISO-DIT and SIDO-DIF require 2 modular multiplications, while the others require 3 of them.

In [32], 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 fig. 3. The operation of the resulting DISO-DIT-NTT is illustrated in fig. 4.



(a) The DISO-DIT butterfly (b) The SIDO-DIF butterfly

Fig. 3. The types of masked butterflies we use

As we can see in fig. 4, each butterfly multiplies each input by some power of ω. The first input should be multiplied by $\omega^{m_o - m_{i_1}}$, where m_{i_1} is the blinding factor used in the previous stage of the NTT, and m_o is the mask we want to use for this butterfly. The second output should be multiplied by $\omega^{k+m_o-m_{i_2}}$, where m_{i_2} is the blinding factor used in the previous stage of the NTT, and k is the twiddle factor from the original unprotected `NTT`. So, each butterfly must be able to compute the blinding twiddle factors used in the previous stage.

In our implementation, we choose to generate an array of masks before the actual computation of the NTT. So, each butterfly can set m_o to the first unused mask in the array, and o can be a counter. Then, each butterfly has to determine i_1 and i_2. Because of the logic of o, this means computing the numbers of the butterflies that processed both inputs in the previous stage. To do that, we note that each swarm at stage i is split into two swarms at stage $i+1$, one little swarm processing the first outputs of all the butterflies in the big swarm, and the other processing their second outputs. We can also note that each stage "consumes" $n/2$ fresh masks, and that $i_2 = i_1 + s$, where s is the size of the butterfly. Everything being numbered from 0, the output mask of the i-th butterfly of `NTT` is $o = i$. Since this butterfly belongs to stage number $t = \lfloor 2i/n \rfloor$, its size is $s = 2^{\log_2(n)-1-t}$, and so it belongs to the w-th swarm of the stage, with $w = \lfloor (i \bmod \lceil n/2 \rceil)/s \rfloor$. Finally, if w is even, then $i_1 = i - n/2$. Otherwise, $i_2 = i - n/2$.

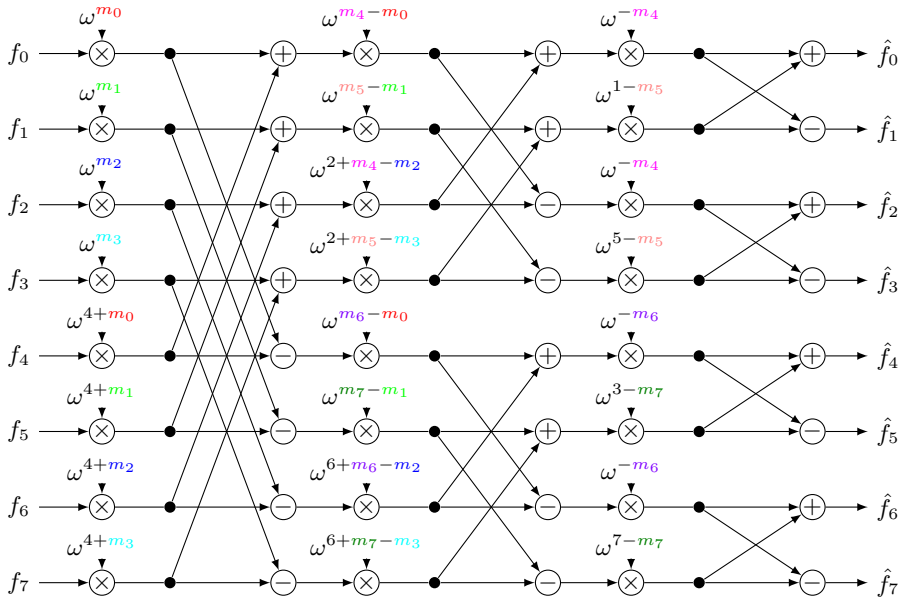


Fig. 4. Toy example of the blinded DISO-DIT-NTT with $n = 8$. Every blinding factor is of a different color for better identification.

2.3 Gathering Butterflies in Blocks

We can see in fig. 4 that every butterfly is protected using a unique mask. This is great for security, but it has a non-negligible impact on performances, because of the amount of random bits that must be sampled. More precisely, one computation of `NTT` has $\log_2(n)$ stages. In order to be easily compatible with the

rest of the algorithms, we chose not to blind the output of NTT. So, the last stage does not use any random mask, while each of the previous stages requires $n/2$ masks. For degree-256 polynomials, we need to sample 896 random 9-bit masks (ω is a primitive 2^9-th root of 1). So, each computation of the NTT algorithm requires to sample 8064 random bits, which is very costly in practice.

One solution, that we chose, is to use *masking blocks* with (fixed) size $B = 2^b$, dividing the amount of random bits needed by a factor B. Indeed, all the butterflies inside a block are blinded with a single twiddle factor:

$$\forall\, i \in \left[\!\left[0, \frac{n\log_2(n)}{2B} - 1\right]\!\right], \quad \forall\, j \in [\![0, B-1]\!], \qquad m_{i\times B+j} = m_{i\times B}$$

2.4 Mixing Blocks in Late Stages

A drawback of this block-based masking lies in the later stages of NTT, when the distance between the inputs of the butterfly is smaller than the block-size B. In that case, both masking twiddle factors are equal, weakening the resistance to SASCA. In implementations with a small randomness budget, many butterflies fall into that category, making this problem really significant.

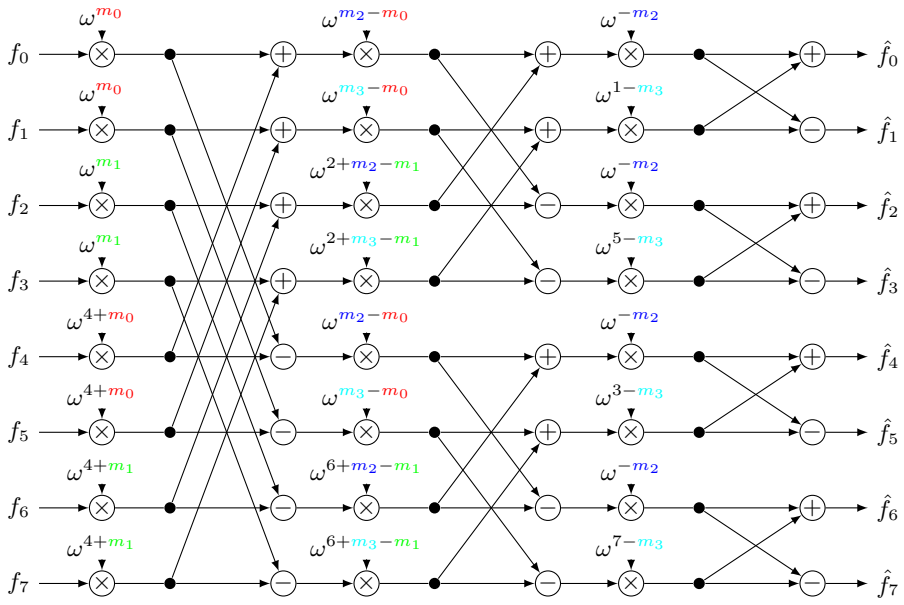


Fig. 5. Toy example of the mixed-block-blinded DISO-DIT-NTT with $n = 8$ and $b = 1$. Same color code as fig. 4.

To address that limitation without additional computational cost, we simply mix the masking blocks in the problematic late stages of the computation. More

precisely, we keep the usual blocks while the butterflies are larger than B, and we use *mixed blocks* when they are too small. In details, we treat two blocks at a time, applying the blinding twiddle factor of the first group to the first half of the swarms that belong to one of the original blocks, and applying the blinding twiddle factor of the second group to the others (illustrated in fig. 5).

In these stages, the blocks are mixed by pairs, and all the inputs of one pair of blocks come from the same pair of blocks from the previous stage. Moreover, the first input always comes from the first previous block, and the second always comes from the second one. Therefore, all masks indexes can be handled as counters. Finally, the last stage must be handled separately, since these swarms (which only contain 1 butterfly each) cannot be split into two mixed blocks.

We provide in section A the final algorithm of our blinded NTT.

3 CRT-based RNR Blinding

In 2021, Heinz and Pöppelmann proposed a side-channel and fault protection model for lattice-based schemes [12]. Their method is faster than standard masking with shares, which works on the algorithm's linear parts.

First, they use Redundant Number Representation (RNR) for side-channel protection, as illustrated in fig. 6. Instead of working in the field $\mathbb{Z}_q$, they perform linear operations in the ring $\mathbb{Z}_{qt}$, where $t \in \mathbb{Z}$ is odd (for fast Montgomery reduction), does not divide the root of unity used for NTT in $\mathbb{Z}_q$ (to avoid de-randomizing the coefficients) and is on the order of $2^{31}/q$ (to keep 32 bits coefficients). The coefficients are randomized by adding cq (with random $c \in \mathbb{Z}_t$), and de-randomized by reducing them modulo q.

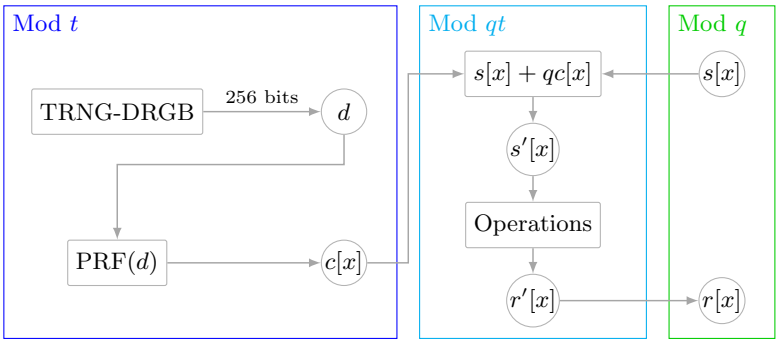


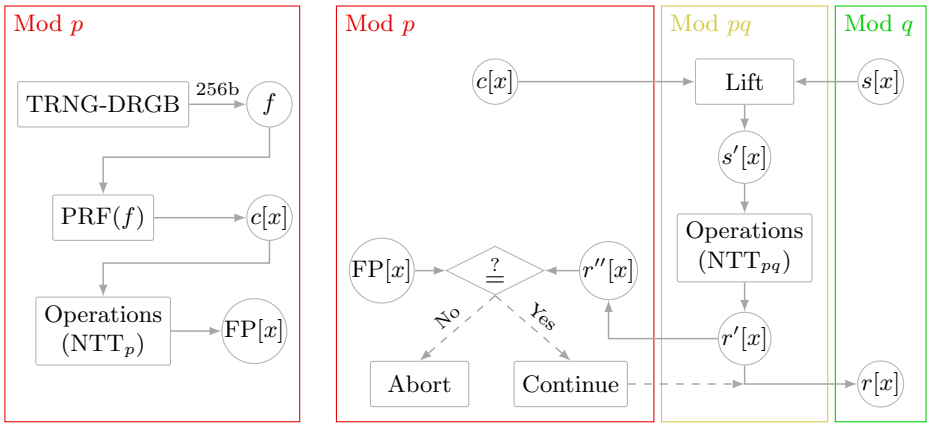
Fig. 6. The side-channel protection proposed in [12].

Second, they use a technique based on the Chinese Remainder Theorem (CRT) for fault protection. Instead of working in the polynomial ring $\mathcal{R}_q$, they work in the ring $\mathcal{R}_{pq} = \mathbb{Z}_{pq}[X]/(X^n + 1)$, with p an odd number co-prime to q. Thanks to the CRT, we have that $\mathcal{R}_{pq} \cong \mathcal{R}_p \times \mathcal{R}_q$ where the ring $\mathcal{R}_q$ contains

the input coefficients to protect, and the ring $\mathcal{R}_p$ contains constants. We will refer to this isomorphism as *lift*. It can be easily computed using the formula:

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

As shown in fig. 7, their method for fault protection is defined by several steps. During precomputations, they sample a random polynomial $\mathbf{c} \in \mathcal{R}_p$ and perform all the operations that should be protected on $\mathbf{c}$, storing the result. They apply the lift operation on the polynomial at runtime, meaning that when given the input polynomial to protect $\mathbf{s} \in \mathcal{R}_q$, they use the isomorphism to pass from $(\mathbf{c}, \mathbf{s})$ to a polynomial in $\mathcal{R}_{pq}$. They, then perform the desired operations in $\mathcal{R}_{pq}$ and finally check that the final value, reduced to $\mathcal{R}_p$, matches the precomputed value. If it does, it returns the value in $\mathcal{R}_q$.



(a) Precomputations (b) Run time

Fig. 7. The fault protection proposed in [12].

Finally, they proposed a technique to combine their side channel and fault protections. To do so they modify the fault protection method. Instead of precomputing the operations on a fixed pre-sampled polynomial, they do everything at run time. They also proposed to use different roots of unity for the NTT in $\mathcal{R}_{pq}$. Using the roots of unity, their proposal makes the NTT in $\mathcal{R}_{pq}$ equivalent to an NTT in $\mathcal{R}_p$ and an NTT in $\mathcal{R}_q$. 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 deployed in a limited resources environment:

- Sampling the random polynomials takes time.
- Storing the random polynomials adds a significant memory overhead.
- The lift operation on polynomials introduces a new leakage point.
- Having to perform a second decryption in $\mathcal{R}_p$ is time-consuming.

3.1 Our Improvements

We instead work in $\mathcal{R}_{pqt}$ where $\mathcal{R}_q$ contains the computations to protect, $\mathcal{R}_t$ is used for side-channel protection, and $\mathcal{R}_p$ 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 large performance drops 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 perform *lift* operations on the coefficients, instead of sampling the polynomials at random in $\mathcal{R}_p$, 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 aspect is to hide the coefficients prior to the lift operation. Indeed, this operation involves multiplying the coefficients by a known constant, which would otherwise create a straightforward and easily exploitable target for side-channel attacks. Via hiding, we ensure that this multiplication produces different leakage traces across executions, thereby preventing direct exploitation.

For the NTT, we perform all operations in $\mathbb{Z}_{pqt}$ and use different roots of unity. Instead of using the 256^{th} roots of unity $\omega_q \in \mathbb{Z}_q$ or $\omega_{pqt} \in \mathbb{Z}_{pqt}$, we define a specific 256^{th} root of unity in $\mathbb{Z}_{pq}$, as: $\omega = \mathsf{Lift}(\omega_p, \omega_q)$. 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 combined fault and side-channel protection is illustrated in fig. 8.

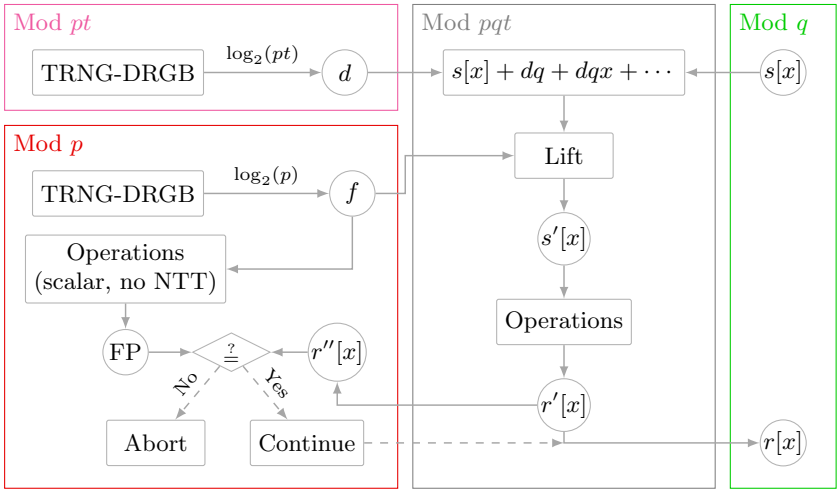


Fig. 8. Diagram illustrating our improved combined protection.

Using a single number to lift all the coefficients allows to reduce the sampling cost and drastically accelerates the verification step. Rather than recomputing

full polynomial operations, it suffices to perform a subset of scalar operations to obtain the expected results in $\mathbb{Z}_p$. Concretely, suppose that the polynomials $\mathbf{a}$ and $\mathbf{b}$ are protected using the scalars f_a and f_b. To verify the correctness of the computation $\mathbf{a} + \mathbf{b}$, it suffices to check that each coefficient of the resulting polynomial equals $f_a + f_b$ in $\mathbb{Z}_p$. This replaces a polynomial addition with a single scalar addition. An analogous simplification applies to subtraction.

For multiplications, the verification step is slightly more complicated, but still much faster than performing a polynomial multiplication. Instead of performing a `NTT`, a point-wise product, and a `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 $(2(i+1) - n)f_{ab}$. The proof of this relation can be found in appendix B.

Our method solves all the problems listed previously but still has one main issue: the coefficient $\frac{n}{2} - 1$ is expected to be 0 in $\mathbb{Z}_p$ right after 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 0 or a multiple of p. Moreover, error propagation through the INTT further complicates the exploitation of this issue.

Another limitation of our method lies in the amount of randomness that can be introduced. Since we operate with a single coefficient, the maximum entropy we can inject is bounded by $\mathsf{WordSize} - \log_2(q) - 1$ bits. Based on the evaluation presented in section 4, we show that 18 bits of randomness allow to secure ML-KEM against the side-channel and fault-injection attacks considered in this work. However, if additional entropy is desired, it is possible to extend the blinding polynomial f by splitting it as $f_a = f_{a_1} + x^{128} f_{a_2}$ where each f_{a_i} is a degree-$(\frac{n}{2} - 1)$ polynomial with repeated coefficients. In this case, the validity of product $f_a \times f_b$ can be verified by checking that, for each i, the result satisfies:

$$f_a \cdot f_b[i] = (i+1) \cdot (f_{a_1} f_{b_1} - f_{a_2} f_{b_2}) + (i + 1 - n/2) \cdot (f_{a_1} f_{b_2} + f_{a_2} f_{b_1}).$$

Further subdivisions of f are also possible, allowing one to reach an arbitrary level of randomness, at the cost of increasingly complex verification logic. [1].

3.2 Ultra-fast NTT

In the context of our fault protection method we use polynomials that have all their coefficients equal in R_p. This structure is not trivial to obtain in T_p.

Some polynomials (for example the secret key) might be stored in T_q, requiring us to perform an NTT in $\mathbb{Z}_p$ on the equal coefficients polynomial before protecting the polynomials in T_q. This NTT in $\mathbb{Z}_p$ is quite costly and we propose an alternative that performs it much faster.

Our method relies on the structure of the polynomial. For every i, we have: $f[i] = c$. Injecting that in the formula of the NTT, we obtain:

[1] A subdivision in 8 parts enables using up to $8 \times (\mathsf{WordSize} - \log_2(q) - 1)$ bits of randomness, while requiring only 64 modular multiplications for verification. This is significantly cheaper than performing a full NTT, as in [12]

$$F[j] = \texttt{NTT}(f)[j] = \sum_{i=0}^{n-1} f[i]\omega^{ji} = \sum_{i=0}^{n-1} c\omega^{ji} = c\sum_{i=0}^{n-1} \omega^{ji} = cg_j$$

By pre-computing the g_j we are able to replace the NTT with a point-wise product, significantly speeding it up. We further elaborate on this ultra-fast NTT across different structures and present additional use cases in section 4.

4 Application to Kyber/ML-KEM

To evaluate the proposed countermeasures, we implemented a proof-of-concept version of Kyber/ML-KEM in C that integrates both techniques described in this work. We use it to quantify the performance overhead induced by our countermeasures and to experimentally assess their resistance to side-channel and fault-injection attacks.

To enable direct comparison with prior countermeasures, our implementation targets Kyber. However, both the analysis and the implementation results apply unchanged to ML-KEM, since the relevant components are identical.

4.1 Background on Kyber/ML-KEM

ML-KEM is a Key Encapsulation Mechanism (KEM) that achieves Indistinguishability under Chosen Ciphertext Attacks (IND-CCA), under the assumption that the MLWE problem is hard. It is derived via a tweaked Fujisaki-Okamoto transform (FO-transform) from K-PKE, which is a Public Key Encryption (PKE) scheme achieving Indistinguishability under Chosen Plaintext Attacks (IND-CPA) [25, 8].

For clarity and brevity, we illustrate our countermeasures only on the decryption algorithm of K-PKE (see algorithm 1). The extension to the remaining algorithms of the K-PKE scheme is straightforward, and the K-PKE scheme is the only component of ML-KEM to which our countermeasures apply. This scheme leverages the polynomial ring $R_q = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$, which is used with $q = 3329$ and $n = 256$ [25].

Algorithm 1 K-PKE.Decrypt(sk, c)

Input: A secret key $sk \in T_q^k$.
Input: A ciphertext $c \in R_q^k \times R_q$.
Output: A message $m' \in \{0,1\}^{256}$.

1: Parse $(\mathbf{u}, v) = c$, and set $\hat{\mathbf{s}} = sk$.
2: Compute $w = v - \texttt{INTT}(\hat{\mathbf{s}}^{\mathrm{T}} \circ \texttt{NTT}(\mathbf{u}))$.
3: **return** $m' = \lfloor w/\lfloor q/2 \rceil \rceil$.

We note here a specificity of ML-KEM, which consists in using a modulus q for which no primitive $2n$-th root of unity exist. Therefore, the scheme leverages

a truncated version of the usual NTT. Instead of splitting a degree-$(n-1)$ polynomial into n degree-0 polynomials, this version splits it into $n/2$ degree-1 polynomials. More precisely, $\omega \in \mathbb{Z}_q$ denoting a primitive n-th root of unity, the NTT domain T_q used in ML-KEM is defined by:

$$T_q = \prod_{i=0}^{n/2-1} \mathbb{Z}_q[X]/\langle X^2 - \omega^{2i+1} \rangle$$

The consequence is that the product in T_q cannot be computed coordinate-wise, but basically consists in multiplying degree-1 polynomials [25]. This specificity, introduced during the second round of NIST standardization, allowed to reduce data sizes with equivalent performance [39, 1].

4.2 Blinded Mixed-Block NTT Countermeasure

The blinded NTT with mixed blocks described in section 2 is directly applicable to Kyber/ML-KEM, apart from the final stage, which must be removed to align with the truncated NTT. Accordingly, the implementation of algorithm 2 can be reused without the "Final stage", thereby reducing the number of required random masks to $r = \frac{n[\log_2(n)-1]}{2B}$.

4.3 Applying CRT-Based RNR Blinding

Protecting both the private key s and the public key (A, t) with the CRT-based RNR blinding countermeasure requires to perform a NTT in $\mathbb{Z}_p$. Since the structure of the NTT changes, we also have to adapt our ultra-fast NTT.

As a reminder, for ML-KEM, the formula of the NTT is:

$$\forall\, j \in [\![0, n-1]\!], \quad \texttt{NTT}(f)[j] = \begin{cases} \displaystyle\sum_{i=0}^{n/2-1} f[2i]\omega^{(2j+1)i} & \text{if } j = 0 \mod 2 \\ \displaystyle\sum_{i=0}^{n/2-1} f[2i+1]\omega^{(2j+1)i} & \text{if } j = 1 \mod 2 \end{cases}$$

Since all coefficients $f[i]$ are set to a randomly sampled value $c \in \mathbb{Z}_p$, we have:

$$\forall\, j \in [\![0, n-1]\!], \quad F[j] = \texttt{NTT}(f)[j] = \sum_{i=0}^{\frac{n}{2}-1} c\omega^{(2j+1)i} = c \sum_{i=0}^{\frac{n}{2}-1} \omega^{(2j+1)i} = cg_j$$

So, precomputing $(g_j)_{j \in [0, \frac{n}{2}-1]}$ allows to replace NTT by a point-wise product.

During key generation, the proposed blinded NTT is used to derive both the private and public keys. Since the keys are stored in T_q, this requires either an additional INTT in $\mathbb{Z}_p$ or fault verification directly in $\mathbb{Z}_p$. The latter option is chosen due to its slightly lower computational cost. For the private key, precomputed g_j values allow an efficient NTT in $\mathbb{Z}_p$. In contrast, for the public key,

executing the ultra-fast NTT twice together with all multiplications in $\mathbb{Z}_p$ is more costly, motivating the alternative approach described below.

In ML-KEM, polynomial multiplication in the NTT domain is performed as follows. For $i \in [\![0, n/2-1]\!]$, we denote $F_1 = F_1[2i] + F_1[2i+1]X$ and $F_2 = F_2[2i] + F_2[2i+1]X$. The product $F_1 \times F_2 = H[2i] + H[2i+1]X$ is given by:

$$F_1[2i]F_2[2i] + (F_1[2i]F_2[2i+1] + F_1[2i+1]F_2[2i])X + F_1[2i+1]F_2[2i+1]X^2$$

In other words, for every $j \in [\![0, n-1]\!]$, we have:

$$H[j] = \begin{cases} F_1[j]F_2[j] + F_1[j+1]F_2[j+1]\omega^{j+1} & \text{if } j = 0 \mod 2 \\ F_1[j-1]F_2[j] + F_1[j]F_2[j-1] & \text{if } j = 1 \mod 2 \end{cases}$$

Since $F_1[j] = c_1 g_j$ and $F_2[j] = c_2 g_j$, we can write:

$$H[j] = \begin{cases} c_1 c_2 g_j^2 (1 + \omega^{j+1}) & \text{if } j = 0 \mod 2 \\ 2 g_j^2 c_1 c_2 & \text{if } j = 1 \mod 2 \end{cases}$$

With $h_j = \begin{cases} g_j^2(1+\omega^{j+1}) & \text{if } j \text{ is even} \\ 2g_j^2 & \text{if } j \text{ is odd} \end{cases}$, we get $H[j] = F_1F_2[j] = c_1c_2h_j$. So, two NTTs and a multiplication can be replaced by a single point-wise product.

4.4 Performance

We implemented the proposed countermeasures in ML-KEM for performance comparison with the state of the art. All experiments use a block size of 8 and parameters $p = 7681$ and $t = 19$.

Table 1. Performance impact of our countermeasures. We give the number of cycles, and the ratio compared to the reference unprotected version in parentheses.

		Reference	Blinded NTT	CRT-RNR	Both
512	KeyGen	136 449	226 852 *(×1.66)*	269 767 *(×1.98)*	324 119 *(×2.37)*
	Encrypt	188 957	279 443 *(×1.48)*	338 183 *(×1.79)*	411 917 *(×2.19)*
	Decrypt	68 841	126 381 *(×1.83)*	143 616 *(×2.14)*	186 344 *(×2.71)*
768	KeyGen	234 887	353 695 *(×1.51)*	441 984 *(×1.88)*	542 439 *(×2.31)*
	Encrypt	301 237	422 959 *(×1.40)*	523 354 *(×1.74)*	621 265 *(×2.06)*
	Decrypt	90 517	168 226 *(×1.85)*	197 802 *(×2.19)*	244 452 *(×2.70)*
1024	KeyGen	374 225	533 417 *(×1.43)*	696 920 *(×1.86)*	796 297 *(×2.13)*
	Encrypt	465 052	624 518 *(×1.34)*	769 873 *(×1.66)*	889 931 *(×1.91)*
	Decrypt	114 078	216 735 *(×1.90)*	239 157 *(×2.10)*	309 953 *(×2.72)*

As shown in table 1, our countermeasures slow the execution down by a factor between 1.91 to 2.72, compared to the unprotected implementation. This overhead stems from several factors: the combination of multiple countermeasures,

the cost of the lift operation and of the polynomial multiplication verification, and the lack of architecture-specific optimizations in our purely software-based implementation. Furthermore, our approach protects all variables rather than only secret ones, improving robustness at the cost of performance.

The use of a relatively small block size compared to prior work further increases the overhead. This cost could be reduced by precomputing blocks, limiting the set of protected variables, and employing larger block sizes. Moreover, the overhead of the countermeasures generally decreases with higher security parameters, as some operations (e.g., randomness sampling and computation of expected values in $\mathbb{Z}_p \dots$) are independent of the polynomial vector dimension.

Concerning memory consumption, we observe an overhead of about 1.42. This arises from the increase in coefficient size from 16 to 32 bits and from the storage of additional constants used in the ultra-fast NTT and reduction steps.

Table 2. Impact of our countermeasures in terms of performances, on the computation of $v - \texttt{INTT}(\hat{s} \circ \texttt{NTT}(u))$ during decryption (Kyber-768, without fault check).

Architecture	Implementation	Source	Overhead
Cortex-M4	Masking & Redundancy	[16, 2]	2.89
Cortex-M4	CRT-RNR	[11]	2.20
x86	CRT-RNR	**This work**	1.95
x86	CRT-RNR & blinded NTT	**This work**	2.58

Table 2 presents a performance comparison with previously published countermeasures. As the implementations target different architectures, overheads are reported relative to the corresponding reference implementation on each platform. The main improvement over [12] lies in fault verification. However, since fault-verification timings are not included in their paper, they are omitted from table 2 and reported separately in table 1. A full end-to-end performance comparison is not possible, as [12] does not provide complete execution times for key generation, encryption, and decryption.

Table 3. Performance impact of our blinded NTT

Block size	Architecture	Source	Function	Overhead
8	x86	**This work**	NTT	2.69
			INTT	1.73
32	x86	**This work**	NTT	2.53
			INTT	1.60
32	Cortex-M4	[32]	NTT	2.32
			INTT	1.72

Table 3 summarizes the performance overhead of our blinded NTT, compared to the unprotected reference. No substantial difference has been observed

between the executions in $\mathbb{Z}_q$ and $\mathbb{Z}_{pqt}$. Unlike [32], which targets the Cortex-M4 platform, the present implementation is architecture-agnostic, so it does not include platform-specific optimizations. Consequently, overheads are computed from the corresponding reference implementations, on the target architecture.

Table 4. Performances of the NTT in $\mathbb{Z}_p$.

Architecture	Implementation	Source	Overhead
Cortex-M4	Classical	[11]	1
x86	Ultra-fast	**This work**	0.43

As shown in table 4, the ultra-fast `NTT` provides a significant speedup (around 2.33) over the unprotected `NTT` in $\mathbb{Z}_p$, in addition to reducing the attack surface.

4.5 Resistance to Side-Channel Attacks

To evaluate the effectiveness of our countermeasures against side-channel attacks, we performed power and electromagnetic radiations measurements using two different experimental set-ups, which are detailed in table 5.

Table 5. The two set-ups used: setup (1) targets an Arduino Due connected to a ChipWhisperer, while setup (2) targets an FPGA AMD Zynq with an EM probe.

Target	Measure	Sample rate	#Samples	Bandwidth	#Traces
(1) @84 MHz	Power	25 Ms/s	5000	100 MHz	up to 10 000
(2) @100 MHz	EM	1-5 Gs/s	2.5×10^6	3 GHz	up to 100 000

The first set-up was used to evaluate both the reference implementation [35] and the protected implementation. The second set-up was used to perform more extensive measurements with electromagnetic radiations on the protected implementation in order to confirm the absence of exploitable leakage.

The analysis focuses on two specific parts of the decryption, namely the multiplication of the ciphertext component u with the secret key s and the INTT that follows this multiplication. Since the ciphertext is public, any consistent variations in power consumption during this operation depend solely on the value of s, making this multiplication a prime target for side-channel analysis. Indeed, several prior works [20, 37, 19, 17] have successfully exploited these steps to carry out CPA attacks.

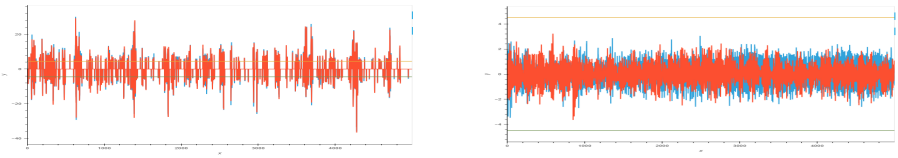
A non-specific fixed-versus-random Test Vector Leakage Assessment (TVLA), following the methodology of Becker and al.[3], is used to evaluate the effectiveness of the proposed countermeasures. The TVLA analysis is applied to both implementations, the reference and the protected one. We used Welch's t-test for the statistical analysis. Further details are provided in appendix C.

Dataset 1 consists of a fixed secret key sk and a set of random ciphertexts [u_0, ..., u_n]. Dataset 2 consists of random secret keys [sk_0, ..., sk_n] and the same set of ciphertexts. In Dataset 1, trace i corresponds to the multiplication of u_i with sk or the following INTT; in Dataset 2, trace i corresponds to the multiplication of u_i using sk_i or the following INTT. Both datasets consist of t traces of s samples, where t and s are specified in table 5.

For the first experiment we performed two independent t-tests with the following sets of traces:

- Test 1: t-test comparing the **first** 5000 traces of both datasets.
- Test 2: t-test comparing the **last** 5000 traces of both datasets.

We consider that the device leaked information about the secret if both Test 1 and Test 2 show a t-score lower than $-4.5 \times \sigma$ or higher than $4.5 \times \sigma$ **at the same point in time**, where σ is the standard deviation of the traces collected. To improve readability, when showing the results of the unprotected multiplication, we zeroed out both t-scores if one of the two tests had a value between $-4.5 \times \sigma$ and $4.5 \times \sigma$. This allows us to only show points where leakage occurs. fig. 9(a) shows the TVLA of the unprotected version, while fig. 9(b) shows the TVLA of the multiplication protected by our countermeasures.



(a) Unprotected (b) Protected

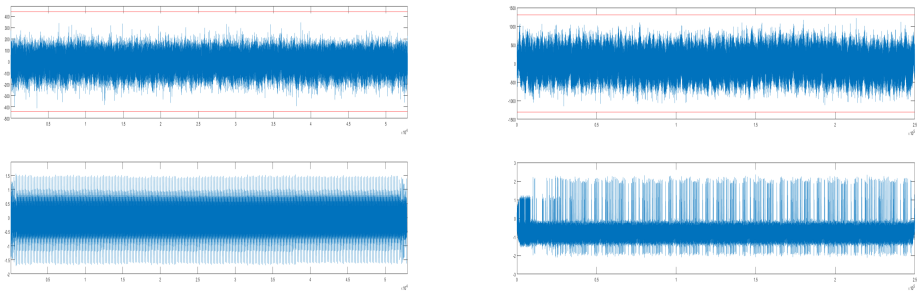
Fig. 9. TVLA for multiplication $sk \times u$. The x-axis represents time, and the y-axis represents the normalized t-score. Test 1 is in blue and Test 2 in red.

As expected, the test shows clear leakage in the unprotected multiplication, while the protected multiplication shows no sign of leakage.

In the second experiment, non-specific fixed-versus-random key TVLA tests were performed using EM traces. Longer traces were collected, allowing a single test to be performed per targeted function, namely the multiplication and the INTT. As shown in fig. 10, no statistically significant leakage was observed for the protected functions. In contrast, EM measurements performed on unprotected portions of the decryption revealed clear leakage, thereby validating the effectiveness of the measurement setup.

4.6 Resistance to Fault Injections

For faults, we will focus on the functions executed during decryption: the NTT of **u**, the point-wise product of **u** and **s**, the INTT of the result and the subtraction of the result to v. These steps cover most of the linear functions in Kyber.



(a) $\texttt{NTT}(u) \odot \texttt{NTT}(sk)$ (b) $\texttt{INTT}(\texttt{NTT}(u) \odot \texttt{NTT}(sk))$

Fig. 10. TVLA results (top) and average EM traces (bottom) of **protected** multiplication of sk with u and the following INTT. The x-axis represents time, and the y-axis represents the non-normalized t-score. The red lines are drawn at $\pm 4.5 \times \sigma$.

We decided to use voltage glitching. We removed some of the capacitors from the Arduino Due target to allow 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 normal execution without faults. Because of this, it was hard to spot effective glitches that led 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 present the results in table 6. We injected faults at different moments of each function to cover a wide variety of instruction skips and/or data corruption.

Table 6. Outcomes of effective fault injections on decapsulation.

		Reference				CRT-RNR			
Target function		NTT	Mul	INTT	Sub	NTT	Mul	INTT	Sub
Total number of faults		2082	2656	2549	792	3905	5095	3455	505
Detection	Crash	1253	1142	2017	684	2371	2195	2108	362
	Countermeasure	0	0	0	0	1479	2900	1347	143
	FO-Transform	829	1514	532	108	55	0	0	0

The unprotected version was subject to many effective faults that were only detected at the FO-transform step. Thanks to our defense, the protected version detected almost all of them, avoiding the reencryption of faulted message.

Some undetected effective faults were still present in the protected decapsulation, all occurring during the NTT of u. Effective faults occurred multiple times at select moments of the NTT and produced the same results. Attacks that rely

on modifying twiddle factors such as that of Ravi et al. [31] would most likely be detected. It would be interesting to go into details to check exactly what faults were produced and whether it is exploitable or not.

5 Conclusion

In this paper, we revisited and improved two complementary blinding-based countermeasures to protect lattice-based cryptographic implementations against side-channel and fault-injection attacks. The first contribution proposes a refined NTT blinding strategy based on mixed masking blocks, designed to increase the number and density of short cycles in the factor graphs exploited by SASCA, while preserving practical performance. The second contribution enhances CRT-based RNR blinding by substantially reducing randomness requirements and verification costs, enabling efficient fault detection tightly integrated with high-performance NTT implementations. Combined, these countermeasures protect both variables and roots of unity, providing strong resistance against a wide range of physical attacks. We validated the practicality of these techniques through a proof-of-concept C implementation integrated into Kyber/ML-KEM. Experimental results show that the proposed protections eliminate detectable side-channel leakage in realistic measurement settings and detect the vast majority of injected faults with a moderate and configurable performance overhead. Although our evaluation focuses on Kyber/ML-KEM, the proposed countermeasures are not scheme-specific. They target algorithmic structures common to many lattice-based primitives——such as NTT/INTT computations, polynomial arithmetic, and modular linear operations——and can therefore be adapted to other NTT-based post-quantum schemes, including ML-DSA and Falcon. This highlights their role as reusable implementation-hardening building blocks rather than specialized defenses for a single algorithm.

References

[1] Amin Abdulrahman et al. "Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4". In: *IACR TCHES* 2022.1 (2022), pp. 127–151. DOI: 10.46586/tches.v2022.i1.127-151.

[2] Erdem Alkim et al. "Cortex-M4 Optimizations for {R,M}LWE Schemes". In: *IACR TCHES* 2020.3 (2020), pp. 336–357. DOI: 10.13154/tches.v2020.i3.336-357.

[3] Georg T. Becker et al. *Test Vector Leakage Assessment (TVLA) methodology in practice*. 2013. URL: https://api.semanticscholar.org/CorpusID:28168779.

[4] Joppe W. Bos et al. "Masking Kyber: First- and Higher-Order Implementations". In: *IACR TCHES* 2021.4 (2021), pp. 173–214. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.173-214.

[5] Alin Bostan, Grégoire Lecerf, and Éric Schost. "Tellegen's principle into practice". In: *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*. ISSAC '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 37–44. ISBN: 1581136412. DOI: 10.1145/860854.860870.

[6] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. LNCS. Springer, Berlin, Heidelberg, Aug. 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2.

[7] Jeroen Delvaux. "Roulette: A Diverse Family of Feasible Fault Attacks on Masked Kyber". In: *IACR TCHES* 2022.4 (2022), pp. 637–660. DOI: 10.46586/tches.v2022.i4.637-660.

[8] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Berlin, Heidelberg, Aug. 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34.

[9] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *CHES 2001*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. LNCS. Springer, Berlin, Heidelberg, May 2001, pp. 251–261. DOI: 10.1007/3-540-44709-1_21.

[10] Mike Hamburg et al. "Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber". In: *IACR TCHES* 2021.4 (2021), pp. 88–113. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.88-113.

[11] Daniel Heinz and Thomas Pöppelmann. *Combined Fault and DPA Protection for Lattice-Based Cryptography*. Cryptology ePrint Archive, Report 2021/101. 2021. URL: https://eprint.iacr.org/2021/101.

[12] Daniel Heinz and Thomas Pöppelmann. "Combined Fault and DPA Protection for Lattice-Based Cryptography". In: *IEEE Transactions on Computers* 72.4 (2023), pp. 1055–1066. DOI: 10.1109/TC.2022.3197073.

[13] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. "Fault-Enabled Chosen-Ciphertext Attacks on Kyber". In: *INDOCRYPT 2021*. Ed. by Avishek Adhikari, Ralf Küsters, and Bart Preneel. Vol. 13143. LNCS.

Springer, Cham, Dec. 2021, pp. 311–334. DOI: 10.1007/978-3-030-92518-5_15.

[14] Julius Hermelink et al. "Adapting Belief Propagation to Counter Shuffling of NTTs". In: *IACR TCHES* 2023.1 (2023), pp. 60–88. DOI: 10.46586/tches.v2023.i1.60-88.

[15] Julius Hermelink et al. "Belief Propagation Meets Lattice Reduction: Security Estimates for Error-Tolerant Key Recovery from Decryption Errors". In: *IACR TCHES* 2023.4 (2023), pp. 287–317. DOI: 10.46586/tches.v2023.i4.287-317.

[16] Matthias J. Kannwischer et al. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. Cryptology ePrint Archive, Report 2019/844. 2019. URL: https://eprint.iacr.org/2019/844.

[17] Alexandre Karlov and Natacha Linard de Guertechin. *Power analysis attack on Kyber*. Cryptology ePrint Archive, Report 2021/1311. 2021. URL: https://eprint.iacr.org/2021/1311.

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Berlin, Heidelberg, Aug. 1999, pp. 388–397.

[19] Yen-Ting Kuo and Atsushi Takayasu. "A Lattice Attack on CRYSTALS-Kyber with Correlation Power Analysis". In: *ICISC 23, Part I*. Ed. by Hwajeong Seo and Suhri Kim. Vol. 14561. LNCS. Springer, Singapore, Nov. 2023, pp. 202–220. DOI: 10.1007/978-981-97-1235-9_11.

[20] Haocheng Ma et al. "Vulnerable PQC against Side Channel Analysis - A Case Study on Kyber". In: *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 2022, pp. 1–6. DOI: 10.1109/AsianHOST56390.2022.10022165.

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

[22] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. "Side-Channel Leakage of Masked CMOS Gates". In: *CT-RSA 2005*. Ed. by Alfred Menezes. Vol. 3376. LNCS. Springer, Berlin, Heidelberg, Feb. 2005, pp. 351–365. DOI: 10.1007/978-3-540-30574-3_24.

[23] Amir Moradi. "Side-Channel Leakage through Static Power - Should We Care about in Practice?" In: *CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. LNCS. Springer, Berlin, Heidelberg, Sept. 2014, pp. 562–579. DOI: 10.1007/978-3-662-44709-3_31.

[24] NIST. *Module-Lattice-Based Digital Signature Scheme Standard*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 204. Washington, D.C.: U.S. Department of Commerce, 2024. DOI: 10.6028/nist.fips.204.

[25] NIST. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 203. Washington, D.C.: U.S. Department of Commerce, 2024. DOI: 10.6028/nist.fips.203.

[26] NIST. *Stateless Hash-Based Digital Signature Standard*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 205. Washington, D.C.: U.S. Department of Commerce, 2024. DOI: 10.6028/nist.fips.205.

[27] 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)*. 2023, pp. 1–6. DOI: 10.1109/ISCTrkiye61151.2023.10336068.

[28] Peter Pessl and Robert Primas. "More Practical Single-Trace Attacks on the Number Theoretic Transform". In: *LATINCRYPT 2019*. Ed. by Peter Schwabe and Nicolas Thériault. Vol. 11774. LNCS. Springer, Cham, Oct. 2019, pp. 130–149. DOI: 10.1007/978-3-030-30530-7_7.

[29] Peter Pessl and Lukas Prokop. "Fault Attacks on CCA-secure Lattice KEMs". In: *IACR TCHES* 2021.2 (2021), pp. 37–60. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i2.37-60.

[30] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption". In: *CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. Springer, Cham, Sept. 2017, pp. 513–533. DOI: 10.1007/978-3-319-66787-4_25.

[31] Prasanna Ravi et al. "Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform". In: *IACR TCHES* 2023.2 (2023), pp. 447–481. DOI: 10.46586/tches.v2023.i2.447-481.

[32] Prasanna Ravi et al. "On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT". In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Lejla Batina, Stjepan Picek, and Mainack Mondal. Cham: Springer International Publishing, 2020, pp. 123–146. DOI: 10.1007/978-3-030-66626-2_7.

[33] Oscar Reparaz et al. "A Masked Ring-LWE Implementation". In: *CHES 2015*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. LNCS. Springer, Berlin, Heidelberg, Sept. 2015, pp. 683–702. DOI: 10.1007/978-3-662-48324-4_34.

[34] Peter Schwabe et al. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022.

[35] Peter Schwabe et al. *CRYSTALS-Kyber reference implementation*. 2018. URL: https://github.com/pq-crystals/kyber.

[36] Jian Wang et al. "Secret in OnePiece: Single-Bit Fault Attack on Kyber". In: *SAC 2025*. Ed. by Christina Boura, Atefeh Mashatan, and Ali Miri. Vol. 16207. LNCS. Springer, Cham, Aug. 2026, pp. 781–804. DOI: 10.1007/978-3-032-10536-3_29.

[37] Yipei Yang et al. "Chosen ciphertext correlation power analysis on Kyber". In: *Integration* 91 (2023), pp. 10–22. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2023.02.012.

[38] Jonathan S Yedidia, William Freeman, and Yair Weiss. "Generalized Belief Propagation". In: *Advances in Neural Information Processing Systems*. Ed.

by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press, 2000. URL: https://proceedings.neurips.cc/paper_files/paper/2000/file/61b1fb3f59e28c67f3925f3c79be81a1-Paper.pdf.

[39] Shuai Zhou et al. "Preprocess-then-NTT Technique and Its Applications to Kyber and NewHope". In: *Inscrypt 2018* (2019). Ed. by Fuchun Guo, Xinyi Huang, and Moti Yung, pp. 117–137. DOI: 10.1007/978-3-030-14234-6_7.

Appendix

A Implementation of our blinded NTT

Algorithm 2 Our blinded Number Theoretic Transform.

Input: The coefficients $(f_0, \cdots, f_{n-1}) \in \mathbb{Z}_q^n$ of a polynomial $f \in R_q$.
Output: The NTT form $(\hat{f}_0, \cdots, \hat{f}_{n-1}) \in \mathbb{Z}_q^n$ of f, in bit-reverse order.

Sample $r = \frac{n \log_2(n)}{2B}$ random masks $(m_0, \cdots, m_r)$ in $[\![0, 2n-1]\!]$.
Set $\texttt{bfSize} = n/2$, $\texttt{blockNum} = 0$ and $\texttt{swarmNum} = 1$.
while $\texttt{bfSize} > B$ **do** ▷ *Normal blocks*
    $\texttt{parity} = 0$
    for $\texttt{swarmStart} = 0$ to $n-1$ with step $2 * \texttt{bfSize}$ **do**
        Increment $\texttt{swarmNum}$, and then compute its bit-reverse $\texttt{swarmNumRev}$.
        for $\texttt{blockStart} = \texttt{swarmStart}$ to $\texttt{swarmStart} + \texttt{bfSize}$ with step B **do**
            Compute $i = \texttt{blockNum} - n/(2B) - \texttt{parity} * \texttt{bfSize}/B$.
            for $\texttt{bfStart} = \texttt{blockStart}$ to $\texttt{blockStart} + B$ **do**
                Set $\alpha = m_{\texttt{blockNum}} - m_i$ and $\beta = \texttt{swarmNumRev} + m_{\texttt{blockNum}} - m_{i+\texttt{bfSize}/B}$.
                Compute $t_0 = \omega^\alpha \times f_{\texttt{bfStart}}$ and $t_1 = \omega^\beta \times f_{\texttt{bfStart}+\texttt{bfSize}}$.
                Update $f_{\texttt{bfStart}} = t_0 + t_1$ and $f_{\texttt{bfStart}+\texttt{bfSize}} = t_0 - t_1$.
            $\texttt{blockNum} = \texttt{blockNum} + 1$
        $\texttt{parity} = 1 - \texttt{parity}$
    $\texttt{bfSize} = \texttt{bfSize}/2$
while $\texttt{bfSize} > 1$ **do** ▷ *Mixed blocks*
    for $\texttt{blockStart} = 0$ to $n-1$ with step $4B$ **do**
        Compute $i = \texttt{blockNum} - n/(2B)$.
        for $\texttt{swarmStart} = \texttt{blockStart}$ to $\texttt{blockStart} + 4B$ with step $2 * \texttt{bfSize}$ **do**
            Increment $\texttt{swarmNum}$, and then compute its bit-reverse $\texttt{swarmNumRev}$.
            for $\texttt{bfStart} = \texttt{swarmStart}$ to $\texttt{swarmStart} + \texttt{bfSize}/2$ **do**
                Set $\alpha = m_{\texttt{blockNum}} - m_i$ and $\beta = \texttt{swarmNumRev} + m_{\texttt{blockNum}} - m_{i+1}$.
                Compute $t_0 = \omega^\alpha \times f_{\texttt{bfStart}}$ and $t_1 = \omega^\beta \times f_{\texttt{bfStart}+\texttt{bfSize}}$.
                Update $f_{\texttt{bfStart}} = t_0 + t_1$ and $f_{\texttt{bfStart}+\texttt{bfSize}} = t_0 - t_1$.
                Set $\alpha = m_{\texttt{blockNum}+1} - m_i$ and $\beta = \texttt{swarmNumRev} + m_{\texttt{blockNum}+1} - m_{i+1}$.
                Compute $t_0 = \omega^\alpha \times f_{\texttt{bfStart}+\texttt{bfSize}/2}$ and $t_1 = \omega^\beta \times f_{\texttt{bfStart}+3*\texttt{bfSize}/2}$.
                Update $f_{\texttt{bfStart}+\texttt{bfSize}/2} = t_0 + t_1$ and $f_{\texttt{bfStart}+3*\texttt{bfSize}/2} = t_0 - t_1$.
            $\texttt{blockNum} = \texttt{blockNum} + 2$
    $\texttt{bfSize} = \texttt{bfSize}/2$
for $\texttt{blockStart} = 0$ to $n-1$ with step $4B$ **do** ▷ *Final stage*
    Compute $i = \texttt{blockNum} - n/(2B)$.
    for $\texttt{bfStart} = \texttt{blockStart}$ to $\texttt{blockStart} + 4B$ with step 2 **do**
        Increment $\texttt{swarmNum}$, and then compute its bit-reverse $\texttt{swarmNumRev}$.
        Set $\alpha = -m_i$ and $\beta = \texttt{swarmNumRev} - m_{i+1}$.
        Compute $t_0 = \omega^\alpha \times f_{\texttt{bfStart}}$ and $t_1 = \omega^\beta \times f_{\texttt{bfStart}+1}$.
        Update $f_{\texttt{bfStart}} = t_0 + t_1$ and $f_{\texttt{bfStart}+1} = t_0 - t_1$.
    $\texttt{blockNum} = \texttt{blockNum} + 2$
return $(f_0, \cdots, f_{n-1})$

Although we did not describe the blinded INTT, it can be directly derived from the blinded NTT by leveraging Tellegen's principle [5]. This ensures that `INTT` achieves the same efficiency as `NTT`, in addition to maintaining an identical structure and blinding mechanism. The key distinction lies in the transformation of DISO-DIT butterflies into SIDO-DIF butterflies.

B Development of polynomial multiplication

If we have two polynomials $f_a[x]$ and $f_b[x]$ with all coefficients equal to c_a and c_b. The result of their multiplication is the following:

$$\begin{aligned} f_a[x] \times f_b[x] =& c_a c_b \Big(\sum_{i=0}^{n-1} (i+1)x^i + \sum_{i=0}^{n-1} (n-(i+1))x^{n+i} \Big) \\ =& c_a c_b \Big(\sum_{i=0}^{n-1} (i+1)x^i - \sum_{i=0}^{n-1} (n-(i+1))x^i \Big) = c_a c_b \sum_{i=0}^{n-1} (2(i+1)-n)x^i \end{aligned}$$

C TVLA details

The TVLA consists of collecting two sets of power traces whose means are expected to differ in the presence of side-channel leakage. A statistical test is then applied to assess whether the observed difference between the means can be explained solely by noise-induced variance in the measurements. If the resulting probability is sufficiently low, the presence of leakage is inferred.

Leakage detection is performed using 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 the dataset i, N_i is the total amount of power traces in the dataset i, and s_i^2 is the variance of the dataset i. Note that each trace contains multiple points corresponding to measurements performed at different time periods. Here, all statistics are performed on a single point of the trace on all traces of the dataset. In practice, multiple point-wise t-tests are computed, each corresponding to the leakage at a given sampling time.