Adapting Belief Propagation to Counter Shuffling of NTTs

Julius Hermelink[1,2], Silvan Streit[3], Emanuele Strieder[4] and Katharina Thieme[5]

[1] Universität der Bundeswehr München, Munich, Germany julius.hermelink@unibw.de
[2] Infineon Technologies AG, Munich, Germany hermelink.external@infineon.com
[3] Fraunhofer Institute AISEC, Munich, Germany silvan.streit@aisec.fraunhofer.de
[4] Fraunhofer Institute AISEC, Munich, Germany emanuele.strieder@aisec.fraunhofer.de
[5] Fraunhofer Institute AISEC, Munich, Germany k.thieme@stud.uni-goettingen.de

Abstract.
The Number Theoretic Transform (NTT) is a major building block in recently introduced lattice based post-quantum (PQ) cryptography. The NTT was target of a number of recently proposed Belief Propagation (BP)-based Side Channel Attacks (SCAs). Ravi et al. have recently proposed a number of countermeasures mitigating these attacks.
In 2021, Hamburg et al. presented a chosen-ciphertext enabled SCA improving noise-resistance, which we use as a starting point to state our findings.
We introduce a pre-processing step as well as a new factor node which we call *shuffle node*. Shuffle nodes allow for a modified version of BP when included into a factor graph. The node iteratively learns the shuffling permutation of fine shuffling within a BP run.
We further expand our attacker model and describe several matching algorithms to find inter-layer connections based on shuffled measurements. Our matching algorithm allows for either mixing prior distributions according to a doubly stochastic mix matrix or to extract permutations and perform an exact un-matching of layers. We additionally discuss the usage of sub-graph inference to reduce uncertainty and improve un-shuffling of butterflies.
Based on our results, we conclude that the proposed countermeasures of Ravi et al. are powerful and counter Hamburg et al., yet could lead to a false security perception – a powerful adversary could still launch successful attacks. We discuss on the capabilities needed to defeat shuffling in the setting of Hamburg et al. using our expanded attacker model.
Our methods are not limited to the presented case but provide a toolkit to analyze and evaluate shuffling countermeasures in BP-based attack scenarios.

Keywords: Number Theoretic Transform · Shuffling · Kyber · CCA · Belief Propagation · SASCA · Machine Learning · Countermeasures

1 Introduction

All asymmetric cryptographic algorithms rely on computationally hard problems. At the brink of quantum computers, currently used problems like the factorization of integers or discrete logarithm problems might not be appropriately long-term secure anymore. The National Institute of Standards and Technology (NIST) competition for new post-quantum safe public-key algorithms [Nata] selected among others Dilithium and Kyber [BDK+18] for standardization [Natb]. These lattice-based schemes offer high speeds and comparably small key- and ciphertext sizes. Therefore, such schemes are especially well-suited for embedded devices.

On embedded devices, side-channel and fault security are a major concern and countermeasures to defend against such attacks are indispensable. Several Side Channel Attacks (SCAs), e.g. [PH16, PPM17, PP19, ACLZ20, RRCB20, RBRC20, GJN20, BDH$^+$21], have emerged, and especially the combination of side-channel and fault attacks with chosen-ciphertext attacks [RRCB20, HHP$^+$21, HPP21], has proven to be highly effective. Countermeasures such as masking or shuffling have been proposed and first implementations using such countermeasures were published [RRVV15, RRdC$^+$16, OSPG18, RPBC20, BDH$^+$21, HP21]. While these are first steps to achieving practical implementation security, the understanding of SCAs is not comparable to the years of experience with classical schemes and intensified research is crucial for practical usage of post-quantum (PQ) schemes.

Kyber, a Module-Learning with Error (M-LWE) based key exchange mechanism (KEM), uses the Number Theoretic Transform (NTT) [CT65] for fast polynomial multiplication. While using an NTT can provide significant speed-ups [CHK$^+$21] compared to schoolbook multiplication, it turned out to be another possible target for SCAs [PPM17, PP19, HHP$^+$21]. As a major building block in several lattice-based schemes, such attacks on the NTT could be a major threat to a large group of PQ algorithms. Even for schemes previously not using NTTs, adaptations allowing NTT multiplication have been proposed [CHK$^+$21].

In [HHP$^+$21], Hamburg et al. presented an attack using a chosen-ciphertext that is decompressed to a vector containing a large amount of zeros. During Kyber's decapsulation, this vector is multiplied with the secret key and fed into an Inverse Number Theoretic Transform (INTT). When using a template attack on the INTT, the sparseness of the input then allows for greatly improved recovery of the secret. Thereby, the key may be fully recovered using a very low number of traces while maintaining a high noise tolerance. Their attack is unaffected by standard masking and thus evades first countermeasures.

In [RPBC20], Ravi et al. propose several countermeasures aimed at protecting an NTT. Their work consists of different masking and shuffling countermeasures of different complexity and level of protection for an NTT. The proposed shuffling countermeasures mitigate the attack of [HHP$^+$21] when no adaptations are made. The first variant, *fine shuffling*, permutes the inputs to a butterfly. The other two variants, *coarse (block) shuffling* permute nodes in (a block of) a layer. It has yet been unclear to what extend shuffling can be counteracted by specific attack methods.

Prior work. There are several attacks showing successful SCAs on NTTs using Belief Propagation (BP): [PPM17, PP19] have demonstrated that it is possible to use load and store leakage of a profiled SCA on an NTT software implementation using BP. They target the NTT of Kyber [BDK$^+$18] and are able to recover the secret using a single attack trace. The attack of [PPM17] is limited to low measurement noise and depends on being able to create a very high number of templates. The attack of Pessl and Primas [PP19] targets only ephemeral secrets.

Hamburg et al. [HHP$^+$21] improve upon their attack by using a chosen-ciphertext to create a sparse input to the NTT. Thereby, the convergence is considerably improved and the noise tolerance is increased. This allows to extend the attack given in [PP19] to a non-ephemeral setting by attacking the decryption routine. The creation of the sparse ciphertext is done using the lattice-reduction algorithm Blockwise Korkine-Zolotarev (BKZ) [SE94]. Further, Hamburg et al. discuss the influence of the number and the distribution of zeros on the convergence. In a single trace attack scenario, the attack successively recovers all coefficients up to a noise level σ of 0.5 and 0.7 depending on the Kyber parameter set. Using several traces the attack is successful up to $\sigma = 2.7$, depending on the parameter set and number of traces.

Prior works on shuffling attacks include the sliding window DPA and Hamming integration introduced in [CCD00]. For example, Rivain et al. uses this windowing/integration to attack shuffling against AES [RPD09] and Tillich et al. study the windowing approach of Clavier in the context of a masked and randomized AES implementation [THM07]. Additionally, Tillich and Herbst [TH08] provide practical evaluations on a smartcard running AES. Udvarhelyi et al. analyze the combination of masking, re-keying and shuffling countermeasures in the context of lightweight leakage-resilient cryptography on low-end platforms [UBS21]. In order to counter shuffling, they employ an enumeration technique, which reduces the complexity to recover the leaking permutation in order to attack shuffling. Their analysis is further backed by the theoretical work on shuffling countermeasures in combination with leakage-resilient Pseudo-Random Functions (PRFs) by Grosso et al. [GPSG14]. An attack scheme invariant to local shuffling, and thus countering local shuffling, has been proposed by Bruneau et al. in [BGNT15].

In 2012, Veyrat-Charvillon et al. presented several attacks on shuffling in the context of AES which assume either no leakage, hidden leakage, or leakage of permutation indices [VMKS12]. Using a Bayesian strategy, similar to BP, they are able to attack shuffled S-boxes and note that shuffling may be significantly less secure when simplified shuffling variants are used. This has recently been further improved by Azouaoui et al., especially in combination with masking, in [ABG⁺22]. In 2020, Bronchain and Standaert presented a combination of methods to dissect countermeasures of a concrete AES implementation [BS20]. They provide and discuss several attack strategies, which encompass several types of leakages. By, again, using Bayesian updating, they are able to dissect the countermeasure permutations. Embedding their methods into a factor graph is suggested as future work. While their work is focussed on a specific implementation, the presented methods are applicable more generally. In regards to Soft Analytical Side-Channel Attack (SASCA), Guo et al. provide a coding-theoretic model allowing for an evaluation of SASCA and the impact of shuffling [GGSB20]. They thereby derive bounds on leaked information and improve evaluation of countermeasures, including shuffling.

Our contribution. We analyze shuffling countermeasures proposed by Ravi et al. in [RPBC20] hardening the NTT against SCAs. We introduce a number of tools weakening the proposed countermeasures or, under mild noise conditions, even breaking them.

Firstly, we introduce a new factor node which we call *shuffle node* targeting *fine shuffling*. This node is added to the factor graph of [HHP⁺21] and extends the BP algorithm itself. Shuffle nodes iteratively learn the shuffling permutation of *fine shuffling* within a BP run. This allows for a more noise resistant inference, by separating mixed leakage distributions modeling the uncertainty of shuffled in- or output nodes of a butterfly factor node. Additionally, we introduce the pre-processing technique *mixing priors* as an easy to implement adaptation countering fine-shuffling and compare mixing priors against using a shuffle node.

Secondly, we propose several tools targeting the *coarse shuffling* countermeasures. We expand our attacker model and describe several matching algorithms to find interlayer connections based on shuffled measurements. We additionally discuss the usage of sub-graph inference to reduce the uncertainty during the matching of butterflies. By computing matching probabilities using our extended attacker model and then applying the Sinkhorn-Knopp algorithm [Sin64] to the right-stochastic probability matrix, our *two-point matching* algorithm computes a mix-matrix which is applied to the vector of measurements. While this enables running successful BPs on the measurements, it also creates additional noise. Alternatively, we propose *exact permutation matching*, an algorithm that computes a permutation which might represent the shuffling of nodes. *Exact permutation matching* does not increase the noise compared to attacking an NTT with no countermeasures and can be applied to some layers, while working with *two-point matching* in the other layers. Our methods thereby offer a trade-off between available computational power and required noise-level.

Based on our results, we conclude that the proposed countermeasures of [RPBC20] are powerful and partially counter [HHP$^+$21], yet could lead to a false security perception. A resourceful adversary might still be able to launch successful attacks depending on noise-level and available computational resources.

Our contribution is exemplary based on [HHP$^+$21] and re-uses major parts of the implementation and simulation framework [HSSS]. We extend this by taking into account shuffling countermeasures presented by [RPBC20]. Nevertheless, the introduced techniques do not solely apply to the attack of Hamburg et al. but can be generalized to attacks against shuffling countermeasures. Our work is in line with [VMKS12], [BS20], and [ABG$^+$22], showing that simple shuffling methods can only provide limited protection against an attacker. We propose similar methods to be used in a factor graph attacking an NTT. Thereby, we extend adaptations against shuffling countermeasures to the post-quantum world in a SASCA setting. We focus on information already exploited in the template attack against BP in [HHP$^+$21]. This means we do not assume leakage of permutation indices, but only leakage of load and store operations.

To the best of our knowledge, our techniques are the first systematic approach to attacking shuffling countermeasures in BP-based side-channel analysis of post-quantum schemes using NTTs, extending previous analysis in different settings. Our methods are not limited to the presented use-case but are most likely applicable more generally against shuffling countermeasures. We provide a first practical toolkit to evaluate shuffling countermeasures for BP-based attacks and highlight pitfalls leading to potential vulnerabilities when only an insufficient subset of countermeasures is applied.

Outline. In Section 2 we present the necessary background including BP. Section 3 describes the countermeasures introduced by Ravi et al. [RPBC20]. Section 4 discusses new techniques and approaches which weaken these countermeasures. Section 5 states simulated results and discusses advantages and limitations of these techniques. Section 6 summarizes the contribution and describes possible extensions as future work.

2 Preliminaries

This section introduces the mathematical background of the NTT, provides a wrap up of SASCAs, and summarizes previous work in the context of profiled SCAs in combination with BP. Since there are a number of previous contributions in this context, we keep the sections compact and provide references for the interested reader.

2.1 SASCA and Belief Propagation

Soft Analytical Side-Channel Attacks (SASCAs) were introduced by Veyrat-Charvillon et al. [VGS14]. SASCAs have the goal to reduce the guessing entropy by combining side-channel leakage of multiple points within the algorithm execution. The combination of each leakage point represents the joint probability distribution of the secret. Therefore, the distribution of each leakage observation is a marginal of the targeted secret. Due to measurement imperfections or leakage model constraints each marginal distribution has some guessing entropy left. SASCA models the reduction of this guessing entropy as a noisy decoding problem.

Belief Propagation is a message passing algorithm which allows inference in noisy decoding problems. The noisy joint distribution can be modeled efficiently using a factor graph. Factor graphs are bipartite graphs consisting of variable nodes which contain the marginals and factor nodes which model the pair-wise connections between the marginals. BP allows inference on these factor graphs while staying computational efficient. Although, BP is only exact in decoding the marginals in acyclic graphs, it has been shown in several contributions that it can be used in the context of cryptographic SCA, where cycles in the algorithmic flow graph are unavoidable [VGS14, PPM17, KPP20, PP19, GRO18, HHP+21].

BP is a core algorithm in this contribution. Therefore, we re-iterate the basic concepts here and give a more detailed description in Appendix A based on [Mac03] and [VGS14].

As noted before, BP allows efficient computation of marginals to infer the joint probability distribution. Let $P(\mathbf{x})$ be a function of $\mathbf{x}$ where $\mathbf{x}$ is a set of N random variables $\mathbf{x} \equiv \{x_n\}_{n=1}^{N}$ and $P(\mathbf{x})$ a product of M functions $f_m(\mathbf{x}_m)$ with each function using a subset $\mathbf{x}_m$ of $\mathbf{x}$. The initial prior distributions of each $\mathbf{x}_m$ is gained by e.g. a profiled SCA. Variables and functions are modeled as variable nodes and factor nodes arranged in a factor graph reflecting the attacked algorithm. The goal of BP is to calculate the marginal distributions. BP performs this calculation efficiently by iteratively exchanging messages between the nodes. The exchange of messages is continued until convergence or a different break condition is reached. As already briefly mentioned, in cyclic graphs convergence is not guaranteed. Therefore, in cyclic graphs common break condition are a certain number of iterations or a measure of entropy development of each node. The beliefs are finally calculated as normalized product of messages at each variable node.

Belief Propagation Modifications Several modifications of BP have been proposed including the following: Satorras and Welling introduced a hybrid model which executes BP and a Graph Neural Network in parallel while exchanging messages after each iteration [SW21]. Knobelreiter et al. [KSS+20] proposed an adaption of BP which can be used as a layer in a Neural Network and allows back propagation. Weighted BP is an modification which includes scaled messages using pre-trained weights [NBB16]. In this work, we introduce yet another modification by learning weights during a BP-run. This is achieved by introducing a stateful factor node, which we call *shuffle node*.

2.2 Number Theoretic Transform

Lattice-based PQ algorithms like Kyber are using the NTT to speed up polynomial multiplications. The NTT is an adapted form of the Discrete Fourier Transform (DFT) and operates on a prime field $\mathbb{F}_q$ instead of complex numbers. It is a transformation from a polynomial ring, e.g. in Kyber $\mathcal{R}_q = \mathbb{F}_q[x]/(x^n+1)$, to the *NTT domain* which is in most cases given in $\mathbb{F}_q^n$. The NTT defines a forward as well as backward transformation which we call INTT and is therefore a bijective mapping between the *normal domain* and the *NTT domain*. The product of two polynomials a and b can be calculated as $INTT(NTT(a) \circ NTT(b))$, where $\circ$ denotes the point-wise multiplication.

The implementation of the NTT can be efficiently done by chaining so-called *butterflies*, for a more detailed description see Appendix B. Figure 1 (left) shows a translation of an NTT into a factor graph , based on the butterfly factor node (Figure 1 (right)) introduced in [PP19]. We call the number of variable nodes per layer the height of an NTT. For Kyber, the total number of butterfly layers is 7, each with a height of 256, i.e. in each layer there a 256 variable nodes. The factor node f_{bf} is defined as in [PP19]:

$$f_{bf}(x_a, x_b, x_c, x_d) = \begin{cases} 1, \text{if } x_a + x_b \cdot \zeta = x_c \bmod q \text{ and } x_a - x_b \cdot \zeta = x_d \bmod q \\ 0, \text{otherwise} \end{cases}$$

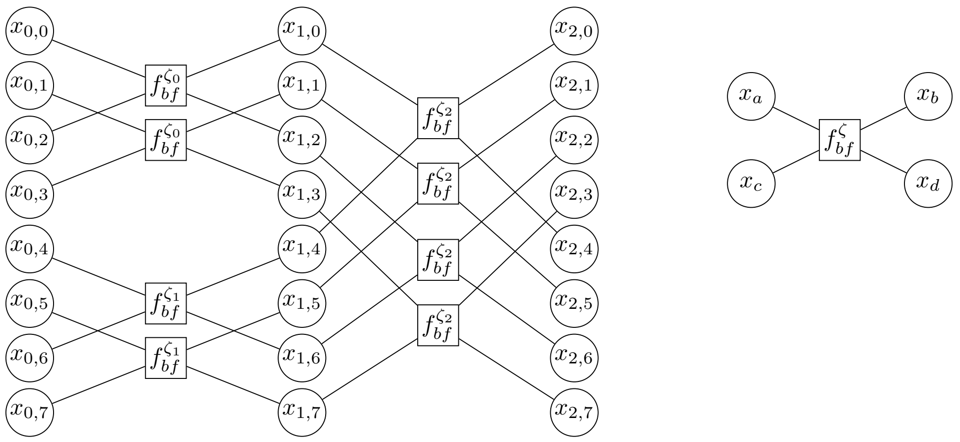


Figure 1: (left) Sub-graph of the INTT with two butterfly layers an a height of eight. The first butterfly layer has two ζ groups and the second one ζ group. The total graph of Kyber has 7 layers and 256 nodes for each layer. The total graph can be separated into two sub-graphs since for Kyber the INTT is incomplete. (right) Generalized naming scheme of a butterfly factor node.

2.3 The attack of Hamburg et al.

The attack of Hamburg et al. improves on the previous BP attacks on the NTT, i.e. [PPM17, PP19], by adding sparseness to the attacked intermediates.

It further targets the INTT of the decryption step, recovering the long-term private key $\mathbf{s}$. As one of the main contributions, it improves on the noise tolerance also in the masked setting by combining the side channel attack with a Chosen Ciphertext Attack (CCA). The factor-graph of Hamburg et al. is similar to the one of [PP19] which uses special butterfly nodes to avoid short cycles in the BP. This improves convergence as it avoids local oscillations and false-positive feedback of the propagated information.

Compressible ciphertext. For an increased noise tolerance in the BP, the intermediates for the input of the attacked INTT are chosen to be sparse. As in Kyber the ciphertext $\mathbf{c}_1$ is transmitted compressed and in standard domain, this is achieved by constructing a termed *compressible* ciphertext, which decompresses into a sparse NTT representation.

The general method of generating sparse inputs in Hamburg et al. is done by solving a short vector problem for a single module component. This way, the distribution of the non-zero coefficients can be adapted to the desired pattern, in order to maximize the noise tolerance of the BP. The lattice problem is constructed by the set of uncompressed ciphertexts u and the required sparse intermediates $\hat{u}$. If the coefficient-wise error introduced by compression $\tilde{u} = u \cdot 2^d - \mathsf{Compress}(u) \cdot q$ is small, the ciphertext u will be compressible. This lattice problem was solved with BKZ-2.0 [CN11] with block sizes 70, in order to generate ciphertexts which result in sparse intermediates with down to 25% sparse coefficients for all Kyber variants. Note, the run-time for the BKZ-solvers are negligible, as an attacker can precompute these compressible ciphertexts, independent of the private key.

Recovering the private key. Once the intermediate $\hat{\mathbf{s}}^T \circ \hat{\mathbf{u}}$ at the non-zero coefficients positions are recovered with BP, the long-term private key $\mathbf{s}$ needs to reconstructed. Depending on the distribution of the non-zero coefficients over the k components of the module, this can be done over a single or multiple traces. Generally per module component, 25% of coefficients in the NTT-domain need to be recovered in order to reconstruct $\mathbf{s}$.

This can again be written as a shortest vector problem for each half-NTT independently and solved analogously with BKZ.

In this work. We take advantage of the improved noise tolerance by reducing the number of non-zero coefficients to 25% in the general case and assume reconstruction of the long-term private key in the k-trace setting. We use the freedom of choice in distributing the non-zero coefficients to minimize the complexity for de-shuffling, i.e. contiguous non-zero coefficients in a single block, as well as pairwise distributed non-zero coefficients.

3 Countermeasures

The information processed within the NTT can contain secret information. For example, in the decryption phase of Kyber, the product of the long-term secret $\hat{s}$ with the ciphertext $\hat{u}$ is the input of the inverse NTT. As shown in Hamburg et al. [HHP+21], leakage recorded during this computation can be used to recover the long-term secret s. Further, as the attack infers a distinct subset of secret coefficients with each single trace, which are recovered from within the single linear NTT operation, splitting the NTT computation into two shares does not alter the attack efficiency significantly. This means that the common countermeasure of masking the secret is not sufficient for efficient protection against this kind of attack.

In Ravi et al. [RPBC20], specific countermeasures against this kind of attack were proposed, consisting of masking and shuffling countermeasures. However, their efficiency against a side-channel attacker was not further studied. In this paper, we are analyzing the effects of these shuffling countermeasures.

3.1 Fine Shuffling

In *fine shuffling* the order of the input loads and output stores for each butterfly are randomized. This way, an attacker can no longer directly assign the recorded leakage value to a specific input/output of a butterfly. As an implementation, Ravi et al. [RPBC20] propose to utilize an arithmetic conditional swap technique after [HS13], which avoids secret branch conditions and secret lookup operations. This is done, with a random bit deciding the order of loading the two coefficients, followed by a conditional swap using bitwise operations. Possible weak points of this shuffling technique discussed in [RPBC20] are on the one side attacking the multiplication step within each butterfly, as done in [PPM17]. On the other side, they also discuss an attack considering the leakage of the mask used in the conditional swap operation as done in [NCOS16, NC17] for ECC. In this paper, we propose another way to circumvent the fine shuffling, without the need of adjusting the leakage model from [HHP+21]. Figure 2 shows four possible factor graph representations of a fine shuffled butterfly.

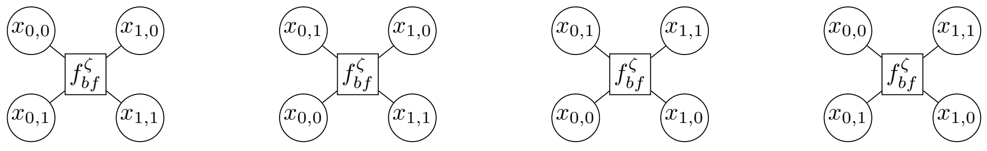


Figure 2: Four possible factor graph representations of a fine shuffled butterfly.

3.2 Coarse Full Shuffling

Each butterfly within a layer of the NTT can be computed independent of the other butterflies. Coarse full shuffling permutes the order of execution of the individual butterflies, and thus loading and storing of the coefficients, within each layer. However, the pair of coefficients belonging to a single butterfly are processed together, and are thus loaded and stored in consecutive order. With n coefficients, this shuffles $(n/2)$ individual butterflies, resulting in $(n/2)!$ permutations per layer.

3.3 Coarse Shuffling in Blocks

In a single layer, several butterflies share the same twiddle factor ζ. Butterflies with the same twiddle factor are referred to as blocks. Coarse shuffling in blocks, in the following called *coarse block shuffling*, randomizes the order of computation of the butterflies within a block. Thereby, in comparison to coarse full shuffling, the permutation applied to each layer is restricted. For a layer with m butterfly groups, this results in $((\frac{n}{2m})!)^m$ permutations. For example, in the INTT of Kyber, with $n = 256$, the first layer consists of $m = 64$ groups, the entropy is $((\frac{n}{2m})!)^m = 2^{64}$. For the last layer, this reduces to a single group, increasing the entropy to 128!. Figure 3 shows a coarse shuffled sub-graph. Shuffling is only performed within ζ blocks and the order of the loads of the input as well as the order of the stores is not shuffled. Note that the layers are shuffled independently. Figure 3 shows an example of a coarse block shuffled sub factor graph.

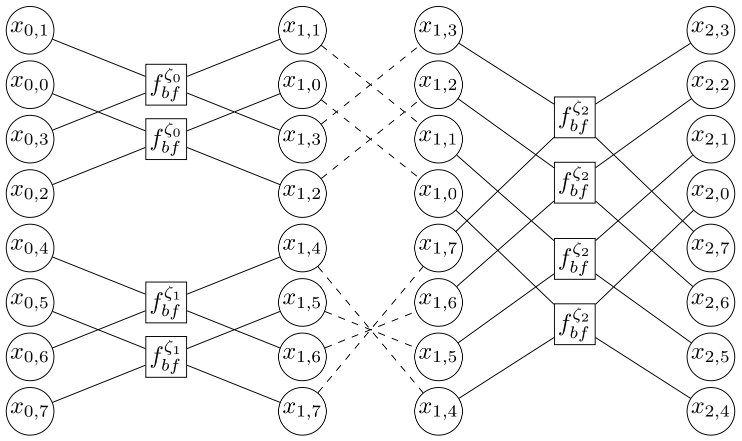


Figure 3: Example of a coarse shuffled sub-graph. Shuffling is only performed within ζ blocks. Note, that the layers can be shuffled independently.

4 Attacking Shuffled Number Theoretic Transforms

In this section, we describe several techniques to attack, and in some cases defeat, the shuffling countermeasures described by Ravi et al. in [RPBC20]. Our general setting is the same as in [HHP$^+$21]: We simulate a template attack on the execution of an INTT running on an embedded device, where the input of the INTT has been sparsified by the techniques described by Hamburg et al. using a CCA as re-iterated in Section 2.3. Our factor graph consists of variable nodes and butterfly nodes modeling an NTT. Our butterfly nodes are of the type introduced by Pessl et al. in [PP19] which causes less short cylces compared to modelling additions and subtractions of a butterfly separately.

4.1 Attacker models

By simulating a profiled SCA on a software implementation, we obtain probability distributions for in- and outputs of butterflies. For a Kyber INTT, we have 8 node layers of height 256, i.e. each layers consists of 256 variables to observe. The height of a block or a sub-graph denotes the number of variable nodes per layer. We assume that 192 positions of the first layer have been set to zero using [HHP+21]. In contrast to [HHP+21], we differentiate between two attacker models. The first model as described in Hamburg et al. assumes leakage of load operations in the first layer and store operations in the following layer. This approach has been verified experimentally by [PP19]. In addition, we assume an attacker may obtain measurements of load operations in the first node layer, and load as well as store operations in each subsequent node layer, expect for the last one, where only store operations are carried out.

Following [PP19] and [HHP+21], at each node with actual value a with Hamming Weight (HW) $h = HW(a)$, we model the leakage by first sampling

$$h' \xleftarrow{\$} \mathcal{N}(h, \sigma)$$

and then obtaining the HW distribution from

$$\mathcal{N}(h', \sigma).$$

In the attacks of [PP19] and [HHP+21], those HW distributions are interpreted as distributions on values and fed into a factor graph. This means the probability of measuring a certain HW is assigned to all indices which evaluate to the respective HW. We describe several pre-processing techniques as well as an additional node, to allow BP to work with measurements obtained from a shuffled NTT. Due to the order of executions being shuffled, measurements are not necessarily of the variable that would have been computed at that position in an un-shuffled NTT. By a measurement $x_{l,h}$ taken at position (l, h), we mean the actual, possibly wrongly assigned measurement, which may belong to a variable $v_{l',h'}$ in an un-shuffled graph.

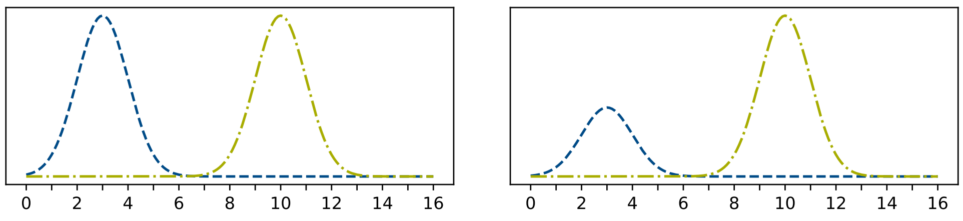
4.2 Defeating Fine-Shuffling

Fine shuffling was proposed by Ravi et al. in [RRCB20] and is re-iterated in Section 3.1. Without additional attack measures, our implementation failed in all cases with all sigmas when using fine-shuffling. Therefore, fine-shuffling provides protection against basic attacks that do not take fine-shuffling into consideration. Nevertheless, it does not fully prevent all considered attacks using the same measurements with different pre-processing techniques. We here describe two methods, *mixing priors* and using a *shuffle node* to defeat fine-shuffling.

Both methods add up prior distributions of nodes which are potentially shuffled. We thereby model the uncertainty introduced by shuffling and avoid contradictions in the BP. *Mixing priors* merely instantiates the BP with adjusted priors and performs standard BP without any modifications afterwards. The shuffled and not-shuffled case are taken to be equaly likely. *Shuffle nodes* on the other hand perform the mixing of priors during BP and adjust their shuffle-factor ω according to information obtained during BP. The shuffle factor ω is initialized to 0.5 which results in the same initial priors as with *mixing priors*. But, in contrast to *mixing priors*, the shuffle factor ω is updated after each BP iteration, tending to 1 in case of a shuffled node or 0 otherwise. This means *shuffle nodes* are not only a different instantiation, but represent a modification to the BP algorithm itself.

4.2.1 Mixing Prior Distributions

We first observe that given a measurement $x_{l,h}$, at position (l, h), where l is the layer and h is the height, $x_{l,h}$ may be a measurement of the variable at (l, h) or $(l, h + d)$ where d is the distance of the layer, for Kyber this is given by $d = 2^{l+1}$. Thus, by replacing $x_{l,h}$ and $x_{l,h+d}$ by $x_{l,h} + x_{l,h+d}$, both measurements certainly do have positive probability at the value of their corresponding variables $v_{l,h}$ and $v_{l,h+d}$. Here, $+$ is point-wise addition and the sum $x_{l,h} + x_{l,h+d}$ is normalized. The point-wise addition of two probability distributions is shown in Figure 4a. Mixing distributions is possible as fine-shuffling only shuffles in



(a) Mixing priors/shuffle node with $\omega = 0.5$ (b) Shuffle node with $\omega = 0.3$

Figure 4: Mixing of two HW distributions according to respective ω's.

a pairwise manner and, therefore, addition of several measurements still gives useful information. While we thereby do not obtain an optimal distribution and significantly decrease our noise tolerance, it is easily integrated into an existing implementation and is successful up to an measurement with noise with a standard deviation of $\sigma = 0.8$, as presented in Figure 8.

4.2.2 Shuffle Node

In the following, we first provide intuition behind the newly introduced *shuffle node*, followed by a thorough definition. Note that our shuffle node is not only a factor node, but a modification to BP itself. The shuffle node allows for self-modification, i.e. learning, in each BP iteration.

Intuition. Mixing two measurements which were fine shuffled removes inconsistent prior distributions. Inconsistent prior distributions are distributions which have zero probability at the correct value. If the two shuffled measurements of two fine shuffled nodes are mixed, inconsistencies are removed since there is a non-zero probability for both possible correct values. As discussed earlier, it is feasible to use these mixed distributions in BP, albeit less noise tolerant in comparison to an un-shuffled distribution setting.

We propose a newly designed factor node which we call shuffle node. The intuition behind the shuffle node is a comparison between the mixed prior distribution and incoming message distributions within a BP run. The shuffle node contains a *shuffle factor* ω which is updated and hence slowly infers how the nodes had been shuffled. Our node thereby learns with each iteration, and the initially mixed distributions tend towards the unmixed priors. The updates of ω are calculated by the Kullback-Leibler (KL)-divergences [KL51] which is a statistical distance given two distributions. The shuffle parameter is updated based on the distance of the message distribution to the prior distributions. KL-divergence expresses how much two distributions differ from one another. Thus, it may be used to compare our incoming message against a measurement in the form of a prior, and subsequently decide on how likely two measurement were shuffled. Thereby, information learned during a BP-run, influences the shuffle factor. We provide a detailed explanation of KL in Appendix C. Figure 4 shows the mixing of distributions with two different ω's.

Definition. Shuffle nodes are factor nodes attached to two variable nodes at positions (l, h) and $(l, h + d)$, as depicted in Figure 2. Each shuffle node holds both priors $x_{l,h}$ and $x_{l,h+d}$. An internal factor ω determines the mixing of $x_{l,h}$ and $x_{l,h+d}$. Initially, ω is set to

0.5 which mixes the distributions exactly as described in *mixing priors*. A shuffle node provides the mixed priors

$$\omega \cdot x_{l,h} + (1-\omega) \cdot x_{l,h+d} \text{ and}$$
$$\omega \cdot x_{l,h+d} + (1-\omega) \cdot x_{l,h}$$

to the attached variable nodes $\tilde{x}_{l,h}$ and $\tilde{x}_{l,h+d}$. This means, $\omega = 0$ corresponds to an un-shuffled measurement and $\omega = 1$ to a switched measurement. If ω is in $]0,1[$ the distributions are being mixed.

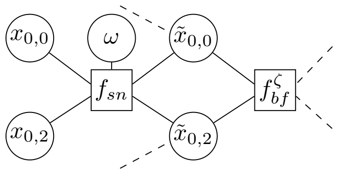


Figure 5: Sketch of a sub-graph with shuffle node f_{sn}, shuffle factor ω, priors $x_{0,0}$, $x_{0,2}$ and respective shuffled variable nodes $\tilde{x}_{0,0}$ and $\tilde{x}_{0,2}$. $\tilde{x}_{0,0}$ and $\tilde{x}_{0,2}$ are connected to a butterfly node f_{bf}. Dashed lines indicate edges connected to the global NTT graph.

In each step, ω is updated by taking both messages $m_{l,h}, m_{l,h+d}$ of the variables nodes into account. This is done by computing the Kullback-Leibler divergences [KL51]

$$\mathcal{D}_{KL}(m_{l,h}, x_{l,h}),$$
$$\mathcal{D}_{KL}(m_{l,h+d}, x_{l,h}),$$
$$\mathcal{D}_{KL}(m_{l,h}, x_{l,h+d}), \text{ and}$$
$$\mathcal{D}_{KL}(m_{l,h+d}, x_{l,h+d}).$$

Then ω is calculated as

$$\omega = \frac{E_{\text{no-shuffle}}}{E_{\text{no-shuffle}} + E_{\text{shuffle}}}$$

where the evidence for a (not-)shuffled version is given by

$$E_{\text{no-shuffle}} = \mathcal{D}_{KL}(m_{l,h}, x_{l,h}) + \mathcal{D}_{KL}(m_{l,h+d}, x_{l,h+d}) \text{ and}$$
$$E_{\text{shuffle}} = \mathcal{D}_{KL}(m_{l,h+d}, x_{l,h}) + \mathcal{D}_{KL}(m_{l,h}, x_{l,h+d}).$$

By updating the shuffle factor ω our modified version of BP self-improves by learning from already obtained information.

4.3 Attacking Coarse Shuffling

Block shuffling, as described by Ravi et al. in [RRCB20], re-iterated in Section 3.2 and Section 3.3, does not shuffle inputs of fixed butterflies, but instead the butterflies of a layer. Coarse block shuffling is limited to shuffling butterflies in a block, i.e. butterflies with the same twiddle factor. A block in layer l is given by the indices

$$\{ (l,h) \mid h \in \{ s, \ldots, s+d \} \}$$

where the start indices s are in $\{ 0, d, 2d, \ldots, 256 \}$ and the distance d is 2^{l+1}. In contrast to fine-shuffling, here the number of shuffling positions does not allow for a simple mixing without increasing the noise to an infeasible level. To counter this additional complexity,

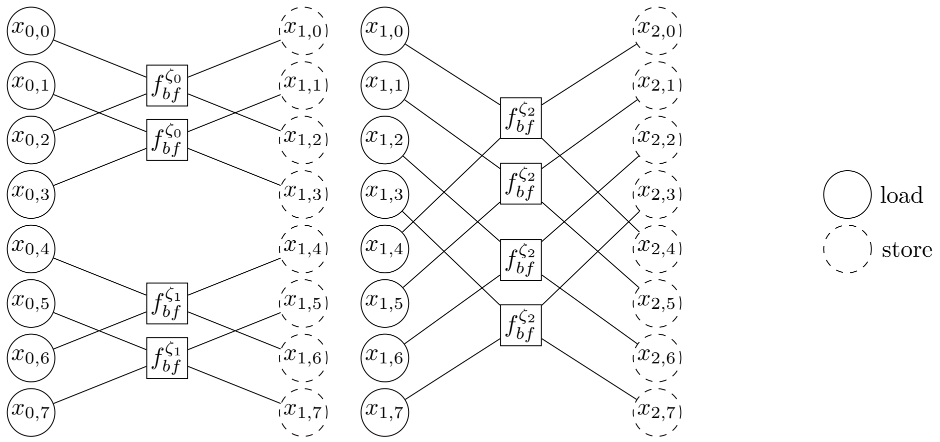


Figure 6: Attacker model showing loads and stores using a subgraph with two butterfly layers and height of eight. Note that butterflies with the same ζ value can be shuffled arbitrary.

we extend the attacker model: We assume that the attacker can not only measure leaking load, but load as well as store operations in each layer. This approach is visualized in Figure 6. Please note, that loads and stores of different layers are independent. However, loads and stores of the same butterflies are considered in-order and are therefore dependent.

Previous work has shown the leakage of load and store operations, but had no necessity to combine it [PP19]. Using this attacker model we can employ a new approach which matches similar probability distributions of loads and stores. Thereby, we obtain probability information which can be used to mix or directly un-shuffle the priors.

4.3.1 Matching Distributions

We now describe the algorithm to match the distributions in each layer given leakage of load and stores of the same value. For each variable in each layer l, except for the first and the last layer, we obtained two distributions, by measuring the stores from butterfly layer l and the loads from butterfly layer $l+1$. Those can be used to find the relative permutation of butterfly layer l. As we assume our error to be Gaussian with fixed standard deviation σ, we identify measurements by their mean.

One-Point Matching. When observing a measurement x of a node v with mean μ, we may compute the probability of v having HW g as follows. For $\epsilon > 0$ and $I =]\mu - \epsilon, \mu + \epsilon[$, using Bayes' rule, we have

$$P(\mathrm{HW}(v) = g \mid x \in I) = \frac{P(x \in I \mid \mathrm{HW}(v) = g) \cdot P(\mathrm{HW}(v) = g)}{P(x \in I)}$$

where the likelihood can easily be computed by using

$$P(x \in I \mid \mathrm{HW}(v) = g) = P_{\mathcal{N}_{g,\sigma}}(I),$$

the evidence as

$$P(x \in I) = \sum_{\gamma} P(x \in I \mid \mathrm{HW}(v) = \gamma) \cdot P(\mathrm{HW}(v) = \gamma)$$

and the prior is derived from the number of occurences of the HW g in the value range. The probability is now given by taking the limit $\epsilon \to 0$ and a pratical approximation may easily be computed by choosing a small enough ϵ. This means for HW g we have

$$P(\mathrm{HW}(v) = g \mid \text{observing } x) = \lim_{\epsilon \to 0} P(\mathrm{HW}(v) = g \mid x \in I).$$

Given two measurements x_{store} and x_{load} of v_{store} and v_{load}, the probability of v_{store} and v_{load} belonging together, i.e. x_{store} and x_{load} being measurements of the same variable, is

$$\sum_{\gamma} P(\mathrm{HW}(v_{store}) = \gamma \mid \text{observing } x_{store}) \cdot P(\mathrm{HW}(v_{load}) = \gamma \mid \text{observing } x_{load}).$$

Computing probabilities for all load and stores gives a right-stochastic matrix $A_{l,s}$ for each layer l and block s. We now apply the Sinkhorn-Knopp algorithm [Sin64], i.e. alternating normalisation of rows and columns until a doubly stochastic matrix is obtained or the change in entropy reached a certain threshold. We now, in each layer, mix the prior distributions according to the obtained doubly stochastic matrix $\tilde{A}_{l,s}$, by applying matrix multiplication to the vector of priors. To account for the mixture in the previous layer, after each layer, we apply the matrix of layer l to the matrix of layer $l+1$. When combining the algorithm with the CCA described in [HHP$^+$21], some nodes in later layers cannot be matched properly, as they are set to zero. Therefore, only a subset of layers may be used as the structure of the NTT is altered. This is not the case for two-point matching.

Algorithm 1 One-point matching without exact permutation extraction.

```
Input: $x_{store,l,h}, x_{load,l,h}$ for $h \in \{0,\dots,\text{height}-1\}, l \in \{0,\dots,\text{layers}-1\}$
Output: Mix matrices $\tilde{A}_l$ for $l \in \{1,\dots,\text{layers}\}$ and mixed measurements
 1: for all $l \in \{0,\dots,\text{layers}-1\}$ do
 2:     $A_l \leftarrow$ Compute priors
 3:     $\tilde{A}_l \leftarrow$ Sinkhorn-Knopp($A_l$)
 4: for all $l \in \{1,\dots,\text{layers}-1\}$ do
 5:     $x_{\text{load},l} = (x_{\text{load},l-1,0},\dots,x_{\text{load},l-1,\text{height}-1})$
 6:     $x_{\text{load},l} \leftarrow \tilde{A}_l \cdot x_{\text{load},l}$
 7:     $x_{\text{store},l} = (x_{\text{store},l,0},\dots,x_{\text{store},l,\text{height}-1})$
 8:     $x_{\text{store},l} \leftarrow \tilde{A}_l \cdot x_{\text{store},l}$
 9:     $\tilde{A}_{l+1} \leftarrow \tilde{A}_l \cdot \tilde{A}_{l+1}$
10: return $\tilde{A}_l$, $x_{\text{store},l,h}$, $x_{\text{load},l,h}$, for all $l, h$
```

Two-Point Matching. As in coarse block shuffling butterflies are shuffled, nodes connected to the same butterfly stay in the same relative position (cf. Section 4.3). Therefore, while computing the priors we may take two observations into account at once. Hence, matching can be improved by considering two blocks at once. The probability of a node with index (l, h) being matched with (l, h') is then given by the product of the previously calculated priors for $(l, (h + d) \mod 2d)$. The modified priors give the basis for Algorithm 1. This means two-point matching is one-point matching with taking into account butterfly instead of node shuffling.

Note that even without initial noise, one-point matching creates significant noise, as values in a block are not necessarily unique. Using two-point matching, we not only decrease errors from noise, but also the likelihood of non-unique matchings resulting in an increased noise regardless of the measurement error.

Reducing Full Shuffling to Block Shuffling. Our matching algorithms perform reasonably well under the assumption of only having to match a small number of nodes in the early layers. When attacking coarse block shuffling, this assumption is true as the number of butterflies is $2^{\text{layer}+1}$ per block and we therefore feed $2^{\text{layer}+2}$ nodes into the two-point matching algorithm. Unfortunately, our techniques alone do not allow for an attack on a coarse full shuffled NTT as, here, no restrictions on the permutation applied to layers are present and, in each layer, all nodes may be permuted. Nevertheless, an attacker might be able to assign twiddle factors to measured butterflies, especially if no additional masking of twiddle factors is applied. In that case, the reduction to coarse block shuffling can clearly be done by applying the previously presented algorithms to nodes attached to butterflies with the same twiddle factor. Here, deep learning as a profiling method, as e.g. described in [PPM+21], may be of high value. We note that practicability of such attacks heavily depends on the concrete setting.

Exact Permutation Matching. Sometimes, it may not be sufficient or desireable to mix distributions as it increases noise. Instead of applying the mix-matrix, one may extract the permutation with the highest probability from it. Unfortunately, this will often not be correct and cause the BP to fail. To account for this, we provide a version of the algorithm computing not one permutation matrix but many likely permutations per layer. An attacker may then run BP with all combinations of all permutations for every layer.

Finding an exact permutation starts with the same algorithm given above. Instead of aborting and returning the mix-matrix after running the Sinkhorn-Knopp algorithm [Sin64], the row with the most entropy is selected. For this row, we fix all values with probability higher than some threshold. We then, for each value, recursively call the algorithm with the row set to 0 except for in the selected position where we set the probability to 1. After having reached the abort depth in each execution path, all obtained permutations are returned.

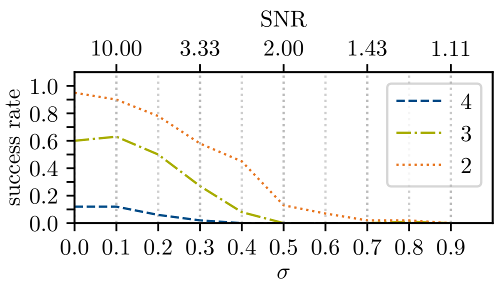
We note that obtaining a set of permutations containing the correct one and exhaustively running all permutations is not realistic in most scenarios. Nevertheless, for a powerful attacker, especially combined with additional information or sub-graph-value matching, this can further decrease noise. Exact permutation matching may also be applied to a subset of the layers while working with two-point matching in the remaining layers. This has the advantage of eliminating mixing noise in early layers which would otherwise also distribute to higher layers due to the mix-matrices being products of matrices obtained in earlier layers.

Algorithm 2 ExactMatch: Finding probable permutations.

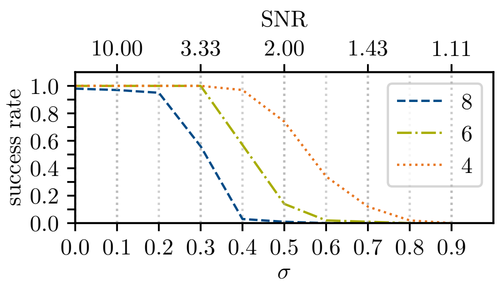
```
Input: Prior matrix $A = A_l$ for some layer $l$, depth level $d$, max depth $d_{\max}$, entropy
    threshold $t_e$, probability threshold $t_p$
Output: A list of permutations.
 1: if $d > d_{\max}$ then
 2:     return []
 3: result_list ← []
 4: $A \leftarrow$ Sinkhorn-Knopp($A$)
 5: $I \leftarrow$ Find indices of rows with entropy higher than $t_e$
 6: for all $i \in I$ do
 7:     $J \leftarrow$ Find indices of columns with probability greater $t_p$
 8:     for all $j \in J$ do
 9:         $A_{ij} \leftarrow$ Set row $i$ to $(\delta_{jk})_k$ where $\delta$ denotes the Kronecker delta.
10:         permutations ← MatchExact($A_{ij}$, $d+1$, $d_{\max}$, $t_e$, $t_p$)
11:         result_list ← Append permutations to result_list
12: result_list ← Sort result_list by likelihood and remove duplicates
13: return result_list
```

4.3.2 Value-Matching using Sub-Graphs

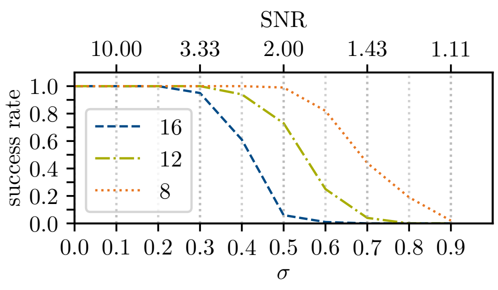
Since matching based on pure HWs is rather imprecise we propose sub-graph evaluation. Sub-graphs of the NTT can contain enough information to converge to the correct values although not connected to the global NTT graph. This clustering allows to use sub-graph convergence as a test routine to enhance certainty into the matching of nodes.



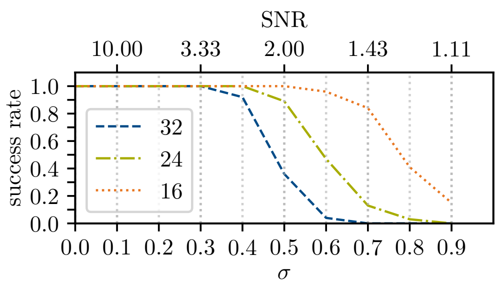
(a) Graphs with 4 input nodes and 2 layers



(b) Graphs with 8 input nodes and 3 layers



(c) Graphs with 16 input nodes and 4 layers



(d) Graphs with 32 input nodes and 5 layers

Figure 7: Attack results for different noise levels σ with distributed non-zero coefficients using sub-graphs of the NTT. From the left top to the right bottom: Graphs with 4, 8, 16, and 32 input nodes reaching over 2, 3, 4, and 5 layers, respectively. For each sub-graph type, three levels of non-zero elements at the input level are depicted. Each sample point is the average of 100 runs.

Figure 7 shows statics for four types of sub graphs: Graphs with 4, 8, 16, and 32 input nodes, which corresponds to graphs with 2, 3, 4, and 5 butterfly layers, respectively. For each sub-graph type we evaluate the influence of zero elements as input on the convergence. As expected and already mentioned in [HHP$^+$21], the number of zero elements heavily improves the noise resistance of the graphs. However, for convergence in the context of evaluating a matching process only the statistics with no zero elements is of importance. Graphs with 3 butterflies and no zero elements reach noise resistance up to a σ equals 0.4. For each added butterfly layer the noise resistance is increased by approximately 0.1.

4.3.3 Unshuffling the First Layer

Our matching algorithms all rely on a correctly un-shuffled first layer. However, it is not possible to un-shuffle the first layer by matching it to a shuffled second layer. In the first layer, only two butterflies have the same twiddle factor ζ. Thus, for height h, that is $\frac{h}{2}$ butterflies, there are $2^{\frac{h}{4}}$ possible permutations. To reduce complexity, we are working with a total of only 64 set values, which is achieved by using the technique described in [HHP$^+$21] using 192 zeros. This results in an effective height of 64. Note that for one-point matching, this reduces the maximal number of layers which may be used, as shuffled nodes in zeros blocks are not corrected but do mix with non-zero values in the last two layers.

Without further information a simple bruteforce in the worst case has to run 2^{16} BP runs. Although 2^{16} BP runs may not be feasible for a small-scale attacker, the resources required to run such an attack are widely available and can easily obtained by a determined adversary. Fortunately, the number of BP runs can be reduced by observing failing nodes in unsuccessful runs on sub-graphs (cf. Section 5.2). Due to an incorrectly permuted pairs of butterflies in the first layer often causing an error in the third layer which propagates back, we are in many cases able to successively rule out incorrect sub-permutations by observing failing sub-graphs (see Figure 10). In addition, observing the matching quality in the first layer may already give an insight on the correctness of the corresponding blocks. Assuming the wrong permutation for the first two butterflies but the correct permutation for the second pair of butterflies (or vice-versa) will in many cases cause higher matching differences when using two-point matching. Thereby, complexity may be further reduced in theory down to 2^8 BP runs (in the worst case), depending on the number of unique HWs per block and the noise level σ. Summarising, our methods require either an increased, but still reasonable amount of computational power or additional manual analysis. We conjecture that this manual analysis may also be automated to some extend.

5 Results

We simulated our attack methods by extending the implementation of [HHP$^+$21]. The implementation of Hamburg et al. provides an Python framework modelling the attack and a BP implemented in Rust. First, we extended their code by adding a shuffling method allowing to shuffle measurements as described by [RPBC20] as well as the different attacker model modelling measurements of load and store operations. Then, we implemented the methods to attack fine-shuffling described in Section 4.2.1 and Section 4.2.2. For the slightly more computationally expensive techniques to attack coarse shuffling, described in Section 5.2, we use a separate Rust implementation which is called from the original implementation. We modeled our measurements following Pessl and Primas as described in [PP19, HHP$^+$21] and state our results depending on the noise standard deviation σ. Note that in unsuccessful BP runs, often enough information is obtained to continue with different methods such as lattice reduction following Dachmann-Soled et al. [DDGR20]. In this section, all results are given for the case of 192 zeros using the chosen-ciphertext attack of [HHP$^+$21]. Figure 8 depicts the success rates against fine shuffling as well as coarse block shuffling. As we evaluate our methods on the example of the attack of [HHP$^+$21], we need the same number of traces as Hamburg et al., i.e. two traces for Kyber512, three traces for Kyber768, and four traces for Kyber1024.

5.1 Fine Shuffling

Fine-shuffling permutes inputs to single butterflies, i.e. each measurement may be swapped with exactly one different measurement. As described in Section 3.1, we provide two different ways to counter fine-shuffling. The first one, mixing priors, does not require any modifications to the BP and is therefore comparably cheap to implement. The second one, using a shuffle node, gives better results but uses a modified BP by adding a special factor node. We implemented both variants.

Mixing Priors. Mixing priors is the point-wise addition (and subsequent normalization) of distributions obtained by measuring a variable. Thereby, we achieve that the resulting distribution at a node certainly is a measurement of the correct variable (cf. Section 4.2.1). This increases the node entropy (see Table 1), but allows for successful recovery up to a σ of 0.8, as depicted in Figure 8.

Table 1: Average node entropy over all layers when applying mixing priors, compared to an attack against an un-shuffled NTT. Note, that mixing priors yields similar entropy levels as by increasing the noise σ by 0.4 to 0.5.

σ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
SNR	10.00	5.00	3.33	2.50	2.00	1.67	1.43	1.25	1.11	1.00
No shuffling	8.60	8.63	8.90	9.30	9.59	9.79	9.96	10.09	10.20	10.30
Mixing priors	9.63	9.63	9.76	10.05	10.29	10.44	10.54	10.63	10.70	10.77

Shuffle Node. Using a shuffle node enables our BP to learn from obtained information and thereby successively choose the correct measurement if the initial noise was small enough. Here, the initial entropy increase is the same as in Table 1, but the updating process of the shuffle node allows for higher noise tolerance. This allows us to recover all coefficients up to $\sigma = 1.0$ with a positive success rate. Note that observing the ω factor (see Section 4.2.2) at a shuffle node gives an indication of whether a measurement at a node was likely shuffled.

Convergence behavior. With a small noise level σ, far enough from a failing case for both variants, we did not see any notable difference in convergence behavior between *shuffle node*-graphs and *mixing priors*. For example, both *shuffle node* graph and the original graph with mixed priors, took an average of 30 iterations to converge at $\sigma = 0.5$. With noise levels where *mixing priors* begins to fail, in the first few iterations, entropy usually decreases similar to when using *shuffle nodes*, but then either stagnates or oscillates. Our assumption is, that this is due to non-updated priors and thus ignoring information present in the graph, hence this is our motivation for using shuffle nodes.

5.2 Coarse Shuffling

In Section 4.3, we discussed several techniques countering coarse block shuffling. We implemented one- and two-point matching as well as exact permutation matching in Rust. In addition, we analyzed the convergence of sub-graphs which allow for better matching and un-shuffling the first layer.

In practice, the attack strategy depends on the concrete situation, i.e. the noise level, the obtained distributions, and possibly additional information from traces. Therefore, we did not combine sub-graph convergence and matching, because we do not believe our model to allow for a realistic analysis of the combination of several techniques. We also do not provide a separate analysis of attacking coarse full shuffle as our reduction to block shuffling depends on additional information and is then trivial (while obtaining such information is not trivial at all).

In one- and two-point matching, we obtain a mix-matrix for each layer as described in Algorithm 1. This matrix is applied to the vector of measurements and expresses how likely a permutation is. The matching quality directly corresponds to the row-entropy and the number of non-zero elements per row. We therefore state row-entropy as well as non-zero elements per row for both one- and two-point matching. Data for the mix-matrix statistics was obtained performing 100 runs per sigma, while success rates were obtained with 10 runs per sigma. We always assume the first layer to be un-shuffled as described in Section 4.3.3.

One-Point Matching. As one-point matching does not incorporate available structural information, it significantly underperformes compared to two-point matching. Nevertheless, we state the result for the rare case where two-point matching may not be feasible, as e.g. in a totally different setting. Even without measurement noise, we did not manage to recover the key when using one-point matching, but for very low noise, in many cases almost all coefficients could be recovered. Nevertheless, we state the entropy increase when using one-point matching in the Appendix D.

Two-Point Matching. Two-point matching uses the same base algorithm as one-point matching but makes use of additional information by matching at two nodes at once. Two-point-matching, allows for a noise tolerance up to $\sigma = 0.4$. Table 2 depicts the average number of distributions being mixed as well as the average entropy per row of the mix matrix for each layer. In Table 3, we show the average entropy over all non-zero nodes per layer compared to an attack on an un-shuffled NTT. Figure 8 show the success rates for the full attack using two-point matching on a coarse block shuffled NTT. For two-point matching, layers where butterflies are connected to zero and non-zero inputs, essentially one-point matching is performed, further adding to the significantly worse performing later layers.

Table 2: Entries in the two-point mix-matrix with probability higher than 0.005 and, in brackets, the entropy per row.

σ	0.1	0.2	0.3	0.4	0.5
SNR	10.00	5.00	3.33	2.50	2.00
Layer 1	1.02 (0.02)	1.02 (0.02)	1.08 (0.05)	1.23 (0.12)	1.36 (0.2)
Layer 2	1.06 (0.06)	1.07 (0.06)	1.23 (0.13)	1.70 (0.33)	2.18 (0.58)
Layer 3	1.15 (0.14)	1.17 (0.15)	1.58 (0.30)	2.96 (0.82)	4.54 (1.43)
Layer 4	1.33 (0.03)	1.37 (0.32)	2.76 (0.75)	6.60 (1.94)	11.58 (2.99)
Layer 5	8.74 (2.85)	8.91 (2.87)	13.07 (3.45)	25.93 (4.80)	37.95 (5.50)
Layer 6	29.13 (4.77)	28.90 (4.79)	30.36 (5.35)	33.52 (6.43)	26.78 (6.80)

Table 3: Average node entropy per layer when applying two-point matching, compared to the total average in an attack against an un-shuffled NTT. The larger increase in entropy with layer 6 is largely influenced by one-sided matching against zero coefficients, and hence effectively only one-point matching.

σ	0.1	0.2	0.3	0.4	0.5	0.6
SNR	10.00	5.00	3.33	2.50	2.00	1.67
No shuffling (avg)	8.69	8.71	8.97	9.35	9.66	9.85
Layer 1	8.70	8.71	8.97	9.38	9.67	9.86
Layer 2	8.68	8.70	8.94	9.43	9.72	9.92
Layer 3	8.79	8.83	9.10	9.61	9.96	10.14
Layer 4	8.92	8.94	9.19	9.75	10.07	10.33
Layer 5	9.01	9.00	9.28	10.06	10.48	10.78
Layer 6	9.82	9.83	9.98	10.51	11.13	11.48
Layer 7	10.21	10.21	10.37	10.89	11.32	11.49

Exact Permutation Matching. Exact permutation matching alone is not practical for most attackers, but does not increase the noise at all. Therefore, using exact permutation matching on all layers, the noise tolerance is as high as in the original attack. We note that applying it to single layers can be practical, especially when combined with observing sub-graph convergences. Figure 9 depicts the average rank, of the correct permutation with

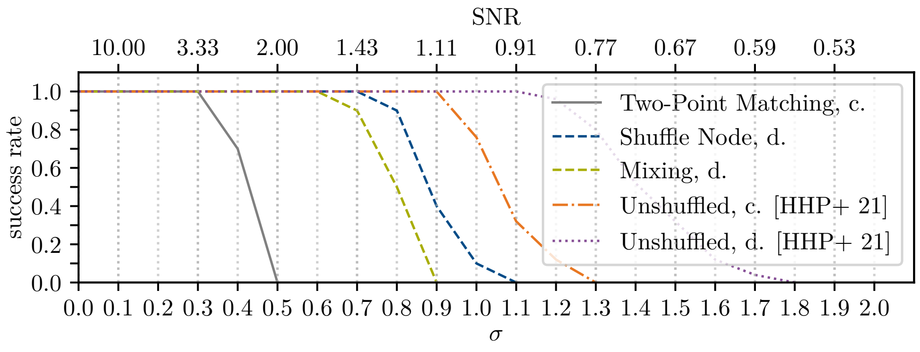


Figure 8: Success rates of mixing distributions and shuffle node against fine-shuffling, as well as two-point matching against coarse block shuffling. As a reference, the results for the unshuffled case from [HHP+21] for both contiguous (c.) and distributed (d.) non-zero coefficients are depicted.

a maximal depth of 2, probability threshold 0.01, entropy threshold 0.1 (cf. Algorithm 2), and we additionally fix no more than 50 coefficients. The rank is the average number of BPs an attacker has to run to correctly un-shuffle one layer. When applying exact permutation matching to multiple layers, the average number of BPs behaves multiplicative with possible improvements by observing sub-graph convergence. For layers 5, 6, and layer 4 with $\sigma > 0.1$, the correct permutation is not found with this parameter set. The choice of parameters heavily influences the runtime and an attacker with high computational power might be able to improve upon the results depicted here.

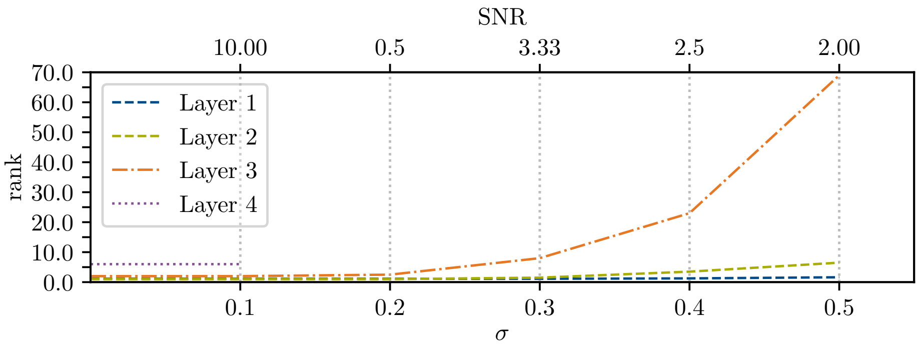
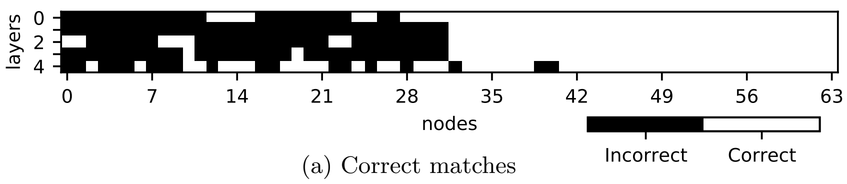
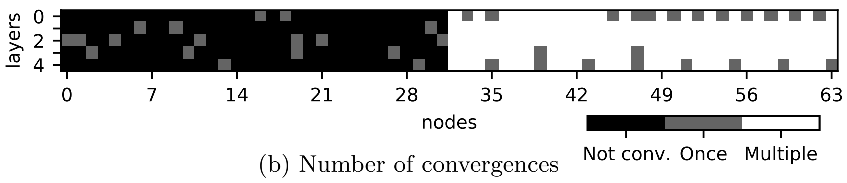


Figure 9: Rank of exact permutation matching per layer and σ.

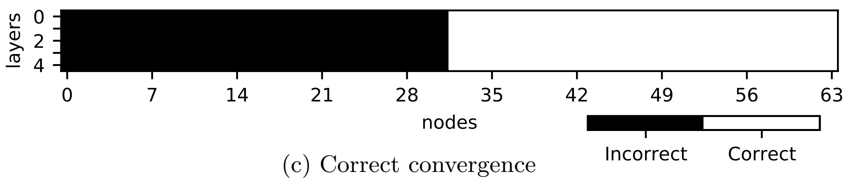
Convergence of Sub-Graphs. We investigated the possibility to use sub-graphs to check for the correct permutation of the first layer. Figure 10 depicts three heat maps. Each heatmap plots layer 0 to 4 and nodes 0 to 63. Further layers as well as the nodes 64 up to 255 did not add any further information. The topmost heatmap indicates how many of the nodes have been matched correctly. We see that the left side has many more faulty matches than the right side. The second heat map depicts the number of convergences of sub-graphs. Since we evaluated multiple sub-graphs with different starting layers and layer numbers, a node could have multiple occurrence in multiple graphs. We notice, that although the left side has bad matching results, some sub-graphs still converge. This was expected because wrongly matched HWs could still have a valid solution and therefore converge. However, it is apparent that correctly matched layers result in convergence of multiple sub-graphs. In the presented case up to 5 graphs converged per node. Hence, in regions with high confidence multiple graphs of different depth converge. This is supported by the final heatmap which shows which nodes converged to the correct value and therefore reveal the secret. This approach is interesting and showed partial success. However, it only was successful up to a σ of 0.2 and we consider the automation and usage in an better than random guess search for the right permutation of the first layer as future work.



(a) Correct matches



(b) Number of convergences



(c) Correct convergence

Figure 10: Evaluation of sub-graph convergence as a measure for correct matching with shuffling of the first layer. Plots where created with a noise level of $\sigma = 0.2$. Depicted are layer 0 to 4 and nodes 0 to 63 since nodes 64 up to 255 are containing zeros and therefore do not add any further findings.

6 Conclusion

In this work, we provide analysis of a subset of NTT countermeasures, *fine shuffling* and *coarse (block) shuffling*, proposed by [RPBC20]. We give further evidence that all variants of shuffling provide basic protection against profiled SCAs that have not been adapted to the countermeasures.

Our attacks are practical in the case of fine-shuffling. We introduce a pre-processing technique, mixing priors, to counter fine-shuffling in a lower noise setting. Mixing priors is easy to implement and does not require the BP to be altered. This makes mixing priors a powerful but inexpensive technique. A more sophisticated approach is our shuffle node, which is a factor node and modifies the BP algorithm to learn whether the inputs of a butterfly had been shuffled. The usage of shuffle nodes does improve on noise tolerance compared to mixing priors. To the best of our knowledge, such as factor node has not been used in SASCAs yet.

We provide several tools to attack coarse block shuffling, making a successful attack practical for a reasonably powerful adversary. We therefor rely on the adversary being able to observe load and store leakage, which was assumed by [HHP$^+$21] and shown to be feasible in [PP19]. The attacker model is extended by taking load and store leakage at each butterfly layer into account. We state an attack on coarse full shuffling assuming an even further extended attacker model leaking twiddle-factor loads or factors of multiplications. Our matching techniques are stated in two variants, allowing either for a less expensive mixing of priors using a mix-matrix applied to the vector of measurements in a layer or the extraction of several exact permutations. The first approach creates additional noise while the latter requires an attacker that has enough resources at their hands to run a large number of BPs. Combining both approaches may be favorable as applying exact permutations in early layers does not cause the need for a large number of BPs, but decreases noise in the layer it is applied to, compared to mixing.

Since several lattice-based PQ schemes rely on an NTT for polynomial multiplication and other schemes may be implemented using an NTT, we contribute to a better overall understanding of the security implications of using PQ cryptography. Furthermore, our tools might also be applicable to counter shuffling in several other attacks.

Other countermeasures Our techniques are adapations to shuffling countermeasures, mainly coarse in-block and fine shuffling. We provide no algorithms to attack other potentially applied countermeasures. While standard masking of the secret does not prevent our attack, as in the attack of [HHP+21], we expect local masking, as also proposed by Ravi et al. in [RPBC20] to counter our attack when no further adaptations are considered. Attacking full block shuffling with our algorithms requires further information and is therefore more difficult to carry out.

Future Work. Our attack leaves room for improvement in un-shuffling the first layer, as e.g. indicated in Section 4.3.3. Using more sophisticated methods, such as algebraic methods to reduce the number of possible permutations, may heavily improve upon the practicability of our attack. This could e.g. include the usage of SAT-solvers or machine learning methods on intermediate results or sub-problems. Such methods could also allow for improving exact permutation matching and thereby increase the overall noise tolerance. Combining and automatization of our techniques to un-shuffle the first layer, perform sub-graph matching, and decide whether to apply exact-permutation matching would be a practical task of high value.

Chunking techniques on sub-graphs, improving our sub-graph methods, might be an option to reduce the noise created by matching in higher layers. For example, iteratively observing overlapping sub-graph convergence and automatically combining results into a meta-graph, might improve overall convergence.

The practical feasibility of reducing from coarse block shuffling to coarse full shuffling is left open. This, as described in Section 4.3.1, requires an attacker to retrieve twiddle-factor information in addition to load and stores already obtained for intermediates. We propose investigating the usage of deep learning on recorded traces to obtain this information which might be hard to observe in a classical template attack.

Acknowledgements

Author list in alphabetical order; see `https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf`. We thank the reviewers for their helpful and constructive feedback which improved this work. This work was supported by the German Ministry of Education, Research and Technology in the context of the project Aquorypt (reference numbers 16KIS1018 and 16KIS1017), and by the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

References

[ABG$^+$22] Melissa Azouaoui, Olivier Bronchain, Vincent Grosso, Kostas Papagiannopoulos, and François-Xavier Standaert. Bitslice masking and improved shuffling: How and when to mix them in software? *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):140–165, 2022.

[ACLZ20] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. Defeating NewHope with a Single Trace. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, volume 12100 of *Lecture Notes in Computer Science*, pages 189–205. Springer, 2020.

[BDH$^+$21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography. *IACR Cryptol. ePrint Arch.*, 2021:104, 2021.

[BDK$^+$18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.

[BGNT15] Nicolas Bruneau, Sylvain Guilley, Zakaria Najm, and Yannick Teglia. Multivariate high-order attacks of shuffled tables recomputation. In *CHES*, volume 9293 of *Lecture Notes in Computer Science*, pages 475–494. Springer, 2015.

[BS20] Olivier Bronchain and François-Xavier Standaert. Side-channel countermeasures' dissection and the limits of closed source security evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):1–25, 2020.

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000.

[CHK$^+$21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[CN11] Yuanmi Chen and Phong Q Nguyen. BKZ 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011.

[CT65] James Cooley and John Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with Side Information: Attacks and Concrete Security Estimation. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 329–358. Springer, 2020.

[GGSB20] Qian Guo, Vincent Grosso, François-Xavier Standaert, and Olivier Bronchain. Modeling Soft Analytical Side-Channel Attacks from a Coding Theory Viewpoint. *IACR TCHES*, 2020(4):209–238, 2020.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. *IACR Cryptol. ePrint Arch.*, 2020:743, 2020.

[GPSG14] Vincent Grosso, Romain Poussier, François-Xavier Standaert, and Lubos Gaspar. Combining leakage-resilient prfs and shuffling - towards bounded security for small embedded devices. In *CARDIS*, volume 8968 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2014.

[GRO18] Joey Green, Arnab Roy, and Elisabeth Oswald. A Systematic Study of the Impact of Graphical Models on Inference-Based Attacks on AES. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*, volume 11389 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2018.

[HHP$^+$21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):88–113, 2021.

[HP21] Daniel Heinz and Thomas Pöppelmann. Combined Fault and DPA Protection for Lattice-Based Cryptography. *IACR Cryptol. ePrint Arch.*, 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, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2021.

[HS13] Michael Hutter and Peter Schwabe. NaCl on 8-Bit AVR Microcontrollers. In *AFRICACRYPT*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013.

[HSSS] Julius Hermelink, Silvan Streit, Emanuele Strieder, and Thomas Schamberger. Source Code for Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber. https://github.com/BayesianSCA/k-trace-CCA.

[KL51] Solomon Kullback and Richard A Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

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

[KSS$^+$20] Patrick Knöbelreiter, Christian Sormann, Alexander Shekhovtsov, Friedrich Fraundorfer, and Thomas Pock. Belief propagation reloaded: Learning bp-layers for labeling problems. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 7897–7906. Computer Vision Foundation / IEEE, 2020.

[Mac03] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

[Nata] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[Natb] National Institute of Standards and Technology. PQ competition, selected algorithms. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[NBB16] Eliya Nachmani, Yair Be'ery, and David Burshtein. Learning to decode linear codes using deep learning. In *54th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2016, Monticello, IL, USA, September 27-30, 2016*, pages 341–346. IEEE, 2016.

[NC17] Erick Nascimento and Lukasz Chmielewski. Applying Horizontal Clustering Side-Channel Attacks on Embedded ECC Implementations. In *CARDIS*, volume 10728 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017.

[NCOS16] Erick Nascimento, Lukasz Chmielewski, David F. Oswald, and Peter Schwabe. Attacking Embedded ECC Implementations Through cmov Side Channels. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 2016.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. *TCHES*, 2018(1):142–174, 2018.

[PH16] Aesun Park and Dong-Guk Han. Chosen ciphertext Simple Power Analysis on software 8-bit implementation of ring-lwe encryption. In *2016 IEEE Asian Hardware-Oriented Security and Trust, AsianHOST 2016, Yilan, Taiwan, December 19-20, 2016*, pages 1–6. IEEE Computer Society, 2016.

[PP19] Peter Pessl and Robert Primas. More Practical Single-Trace Attacks on the Number Theoretic Transform. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2019.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 513–533. Springer, 2017.

[PPM$^+$21] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. SoK: Deep Learning-based Physical Side-channel Analysis. *IACR Cryptol. ePrint Arch.*, page 1092, 2021.

[RBRC20] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. *IACR Cryptol. ePrint Arch.*, page 549, 2020.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT - A Performance Evaluation Study over Kyber and Dilithium on the ARM Cortex-M4. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 10th International Conference, SPACE 2020, Kolkata, India, December 17-21, 2020, Proceedings*, volume 12586 of *Lecture Notes in Computer Science*, pages 123–146. Springer, 2020.

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic Side-channel attacks on CCA-secure lattice-based PKE and KEMs. *TCHES*, 2020(3):307–335, 2020.

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

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A Masked Ring-LWE Implementation. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 683–702. Springer, 2015.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.

[Sin64] Richard Sinkhorn. A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices. *The Annals of Mathematical Statistics*, 35(2):876 – 879, 1964.

[SW21] Victor Garcia Satorras and Max Welling. Neural enhanced belief propagation on factor graphs. In Arindam Banerjee and Kenji Fukumizu, editors, *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event*, volume 130 of *Proceedings of Machine Learning Research*, pages 685–693. PMLR, 2021.

[TH08] Stefan Tillich and Christoph Herbst. Attacking state-of-the-art software countermeasures-a case study for AES. In *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2008.

[THM07] Stefan Tillich, Christoph Herbst, and Stefan Mangard. Protecting AES software implementations on 32-bit processors against power analysis. In *ACNS*, volume 4521 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2007.

[UBS21] Balazs Udvarhelyi, Olivier Bronchain, and François-Xavier Standaert. Security analysis of deterministic re-keying with masking and shuffling: Application to ISAP. In *COSADE*, volume 12910 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2021.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft Analytical Side-Channel Attacks. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2014.

[VMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 740–757. Springer, 2012.

A Belief Propagation

BP allows efficient computation of marginals to infer the joint probability distribution. Let $P(\mathbf{x})$ be a function of $\mathbf{x}$, where $\mathbf{x}$ is a set of N random variables

$$\mathbf{x} \equiv \{x_n\}_{n=1}^N$$

and $P(\mathbf{x})$ as a product of M functions $f_m(\mathbf{x}_m)$

$$P(\mathbf{x}) = f_1(\mathbf{x}_1) \cdot f_2(\mathbf{x}_2) \cdot \ \ldots \ \cdot f_M(\mathbf{x}_M),$$

with each function using a subset $\mathbf{x}_m$ of $\mathbf{x}$. The initial prior distributions of each $\mathbf{x}_m$ is gained by e.g. a profiled SCA. Variables and functions are modeled as variable nodes and factor nodes arranged in a factor graph reflecting the attacked algorithm. The goal of BP is to calculate the marginals

$$Z_n(x_n) = \sum_{\{x_{n'}\}, n' \neq n} P(\mathbf{x})$$

BP performs this calculation efficiently by iteratively exchanging messages between the nodes. BP defines two types of messages, namely *variable-to-factor* messages

$$u_{n \to m}(x_n) \ = \prod_{m' \in \mathcal{M}(n) \setminus \{m\}} v_{m' \to n}(x_n),$$

and *factor-to-variable* messages

$$v_{m \to n}(x_n) \ = \ \sum_{x_m \setminus n} \left(f_m(\mathbf{x}_m) \prod_{n\prime \in \mathcal{N}(m) \setminus m} u_{n' \to m}(x_{n\prime}) \right).$$

The exchange of messages is continued until convergence or a different break condition is reached. The belief or marginal distributions Z_n are finally calculated by

$$Z_n(x_n) = \prod_{m \in \mathcal{M}(n)} v_{m \to n}(x_n)$$

at each variable node.

B Number Theoretic Transform

Lattice-based PQ algorithms like Kyber are using the NTT to speed up polynomial multiplications. The NTT is an adapted form of the DFT and operates on a prime field $\mathbb{F}_q$ instead of complex numbers. It is a transformation from a polynomial ring, e.g. in Kyber $\mathcal{R}_q = \mathbb{F}_q[x]/(x^n+1)$, to the *NTT domain* which is in most cases given in $\mathbb{F}_q^n$. Mathematical it is given by evaluating the original polynomial at the n-th roots of unity. The NTT defines a forward as well as backward transformation which we call INTT and is therefore a bijective mapping between the *normal domain* and the *NTT domain.* For $\mathcal{R}_q$ with a $2n$-th primitive root of unity ζ, the NTT transformation of an n-degree polynomial $f = \sum_{i=0}^{n-1} f_i x^i$ is defined as:

$$\hat{f} = \mathsf{NTT}(f) = \sum_{i=0}^{n-1} \hat{f}_i x^i, \text{ where } \hat{f}_i = \sum_{j=0}^{n-1} f_j \zeta^{(2i+1) \cdot j}.$$

and similarly the INTT

$$f = \mathsf{INTT}(\hat{f}) = \sum_{i=0}^{n-1} f_i x^i, \text{ where } f_i = n^{-1} \sum_{j=0}^{n-1} \hat{f}_j \zeta^{-i\cdot(2j+1)}.$$

The product of two polynomials a and b can be calculated as $INTT(NTT(a) \circ NTT(b))$, where $\circ$ denotes the point-wise multiplication. In the case of Kyber, the NTT is an isomorphism from $\mathcal{R}_q = \mathbb{F}_q[x]/(x^n+1)$ to $\mathbb{F}_{q^2}^{n/2}$, with $\mathbb{F}_{q^2} = \mathbb{F}_q[x]/(x^2-\zeta)$ where ζ is a n-th primitive root of unity. This can be seen as skipping the last layer of the NTT.

The implementation of the NTT can be efficiently done by chaining so-called *butterflies*. The usage of these butterflies allows the iterative reformulation of the NTT into smaller NTTs. This splitting is called decimation and different butterfly types can be used. Figure 11 shows a 8-coefficient NTT using the Cooley-Tukey [CT65] butterfly with decimation in time.

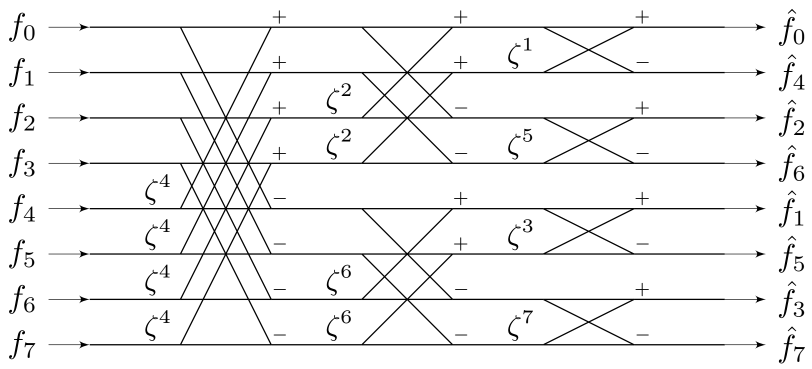


Figure 11: 8-coefficient Cooley-Tukey decimation in time NTT [HHP+21]

C Kullback-Leibler Divergence

The Kullback-Leibler Divergence [KL51], also called relative entropy, measures how much information is expected from p with respect to q.

$$\mathcal{D}_{KL}(p,q) = \sum_x p(x)\ln(p(x)) - p(x)\ln(q(x))$$

Note, that it is not a symmetric measure, as it measures the difference with respect to the second probability distribution. In the context used in this paper, it measures the distance of the message distribution to the prior distributions behind the shuffle nodes. As a measure of distance between probability distributions, it allows for an update of the shuffle parameter.

D One-Point Matching

In Table 4, we state the average number of distributions being mixed with factor higher than 0.005 as well as the average entropy per row of the mix matrix for each layer. Table 5 depicts the average entropy at all non-zero nodes per layer compared to an attack on an un-shuffled NTT. Note that due to zero-values being attached to butterflies with a second non-zero value as input, only the first 4 layers may be used, resulting in 3 layers that have to be matched. Higher layers perform significantly worse as the mix-matrix in layer l is a product of the matrices for layers $1, \dots, l-1$ and the number of nodes to match increases per layer (the constraints for the shuffling permutation decreases with every layer).

Table 4: Entries in the one-point mix-matrix with probability higher than 0.005 and, in brackets, the entropy per row.

σ	0.1	0.2	0.3	0.4	0.5
SNR	10.00	5.00	3.33	2.50	2.00
Layer 1	1.27 (0.25)	1.28 (0.26)	1.53 (0.37)	1.91 (0.57)	2.14 (0.74)
Layer 2	2.04 (0.82)	2.1 (0.83)	2.89 (1.10)	4.31 (1.57)	5.02 (1.86)
Layer 3	4.43 (1.80)	4.57 (2.28)	6.64 (2.28)	10.53 (2.97)	12.42 (3.32)

Table 5: Average node entropy per layer when applying one-point matching, compared to the total average in an attack against an un-shuffled NTT.

σ	0.1	0.2	0.3	0.4	0.5	0.6
SNR	10.00	5.00	3.33	2.50	2.00	1.67
No shuffling (avg)	8.69	8.72	8.97	9.36	9.65	9.85
Layer 1	8.70	8.72	9.00	9.47	9.76	9.93
Layer 2	9.08	9.09	9.40	9.96	10.21	10.33
Layer 3	9.92	9.89	9.89	10.33	10.55	10.68
Layer 4	10.72	10.67	10.59	10.90	11.18	11.36