Unshuffling Fisher-Yates: Practical Recovery of Length-128 Permutations and Application to ML-DSA

Jonas Schupp[1], Anna Baumeister[1], Antonia Wachter-Zeh[1] and Georg Sigl[1,2]

[1] TUM School of Computation, Information and Technology, Technical University of Munich, Munich, Germany,
{jonas.schupp,anna.baumeister,antonia.wachter-zeh,sigl}@tum.de
[2] Fraunhofer Institute for Applied and Integrated Security, Garching, Germany,
georg.sigl@aisec.fraunhofer.de

Keywords: PQC · ML-DSA · Side-Channel Attack · Shuffling · Fisher-Yates Algorithm

Abstract. Shuffling is a standard hiding countermeasure against (single-trace) side-channel attacks on the number-theoretic transform (NTT) of lattice-based post-quantum schemes, such as ML-DSA. The Fisher-Yates algorithm can be used to generate the underlying random permutation. Recovering this permutation itself from a single trace has so far only been studied for short sequences. We present a practical attack on a software implementation of Fisher-Yates for sequences of length 128, the number of butterfly operations in one layer of ML-DSA's NTT. We identify the exploitable leakage of the algorithm, detect and remove the timing variations caused by rejection sampling and extract Hamming weight and identity leakage from a single trace per shuffle. We encode these hints as a constraint-programming problem with a soft objective function and enumerate the most likely permutations in order of descending likelihood. Applied to a shuffled ML-DSA-44 NTT, the recovered permutations are used to re-order the butterfly leakage, and recover the secret polynomial $\mathbf{s}_1$ either directly or up to a few coefficients that are enumerated directly. Protecting ML-DSA's NTT with Fisher-Yates-based shuffling does, therefore, not, by itself, prevent side-channel-assisted key recovery.

1 Introduction

Given a growing concern about the security of classical asymmetric cryptography in the presence of a large scale quantum computer, the National Institute of Standards and Technology (NIST) standardized the first Post-Quantum algorithms, namely the Key-Encapsulation Mechanism (KEM) ML-KEM (former CRYSTALS-Kyber) as well the signature algorithms Module-Lattice-Based Digital Signature Algorithm (ML-DSA) (former CRYSTALS-Dilithium) and SLH-DSA (former Sphincs+). The now ongoing transition from classic asymmetric cryptography (RSA, ECC) to Post-Quantum Cryptography poses the significant

challenge of achieving a similar level of implementation security for these new algorithms as for the long-established classical ones.

Naturally, numerous side-channel attacks as well as countermeasures on these algorithms have been proposed in recent years. With respect to the lattice-based schemes (ML-DSA, ML-KEM), Ravi et al. [17] provide a now slightly outdated overview of attacks and possible countermeasures. Their work highlights the limitations of masking as a countermeasure against single-trace attacks on software implementations of the Number Theoretic Transform (NTT) in ML-KEM and ML-DSA and suggests shuffling as an effective mitigation. This, however, ignores the practical challenges of securely generating a sufficiently long permutation for a PQC algorithm.

In contrast to most other works such as [14] or [7] which exclude the shuffling algorithm from their analysis, we explicitly analyze the attack surface of the Fisher-Yates Algorithm (FY). This is motivated by the fact that, while there seems to be a general consensus that such an attack is possible, to the best of our knowledge, no practical attack has been described for instance sizes required in the context of PQC.

Within this work, when talking about shuffling, we specifically refer to countermeasures which randomize the execution order of a set of independent operations within an algorithm. In a cryptographic context, operations to which shuffling can be easily applied are e.g. the 16 S-Boxes in one round of AES or the 128 butterfly operations executed for one layer of ML-DSA's NTT. While the specifics of the implementations differ, shuffling generally requires to permute a previously ordered sequence which is subsequently used to define the execution order of some independent operations. The security of the shuffling therefore depends not only on the shuffled operations but also on the algorithm that generates the permutation. Both [19] and [21], for instance, suggest FY for this purpose.

Contribution Within this work we show a practical attack against a software implementation of FY for sequences of length $L = 128$ following the implementation provided with [20] but adapted for rejection sampling to make the results as transferrable as possible for a real-world attack. We apply this attack against a NTT implementation of ML-DSA protected by FY based shuffling and show that we can extract the entire secret polynomial up the point where the correct key can be directly enumerated in most cases. More specifically our contribution can be summarized as follows:

- We analyze leakage points within a software implementation of FY and exploit them via the power side-channel.
- We show how the hints obtained from this leakage can be modeled into a CP-SAT problem with objective and how we can use it to enumerate possible solutions to a given shuffle.
- Based on this approach we can estimate how many measurements are required on average to recover the correct shuffle at least once for a list length k of enumerated solutions per measurement.

– Using these enumerated lists of possible shuffles and additional hints from a shuffled NTT, we can recover a secret polynomial $\mathbf{s}_1$ from ML-DSA-44 within 1000 measurements to the point where it can usually be directly enumerated. The entire attack including measurements takes a few hours on a single machine.

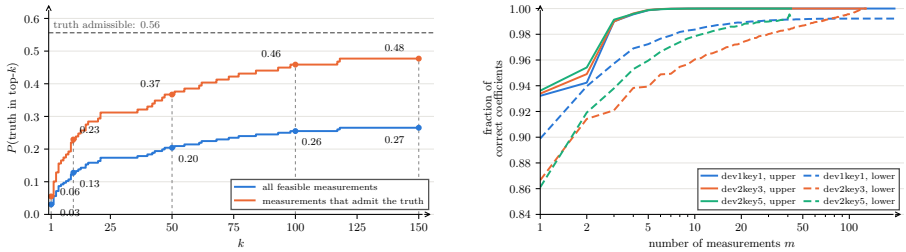

Fig. 1. Main results of the paper. Left: probability of reconstructing the complete FY-sequence from a single trace with increasing reconstruction effort. Right: Fraction of correctly recovered coefficients of the $\mathbf{s}_1$ polynomial as a function of measurements required. Discussed in detail in Section 7.

The implementations of this work are available at `https://github.com/joschupp/unshuffling-fy`.

Organization This paper is split into two parts (measurement and reconstruction) which are mostly self-contained. The first part is concerned with the acquisition of hints from software implementations of FY and ML-DSA's NTT. The second part covers the reconstruction of the FY-sequence described by the hints through constraint programming and subsequently the recovery of the $\mathbf{s}_1$-polynomial of ML-DSA through the leakage information and (partial) knowledge of the shuffle.

We first describe how to obtain the necessary side-channel leakage for this attack from a software implementation and how to derive hints from it, and evaluate both in practice. Section 2 provides the background on shuffling as a countermeasure, existing attacks on shuffling, and ML-DSA, together with existing attacks on it. The first part is concluded by our measurement strategy and hint generation approach in Section 3.

In the second part, we describe how the hints on FY can be cast to a CP-SAT problem with an objective function in Section 4, and how we can use it to enumerate those sequences consistent with the hints in descending order. Next, we map back from a hypothesis about the shuffled sequence to ML-DSA's secret key as used in the NTT in Section 5. We conclude the second part by describing the end-to-end attack chain from measurements to the recovered key in Section 6. Finally, we evaluate and demonstrate the practicality of both the attack on FY

itself and the resulting recovery of the polynomial $\mathbf{s}_1$ in Section 7. We conclude our work in Section 8.

2 Background on Shuffling and ML-DSA

Within this chapter, we provide the necessary background on shuffling as a side-channel countermeasure, FY and ML-DSA. We furthermore discuss existing attacks on shuffling and ML-DSA.

2.1 Side-Channel Attacks

Side-channel attacks aim at extracting information from the execution of an algorithm such as the power consumption of the device during execution to recover knowledge about the internal state of the algorithm and thus enable attacks on an otherwise hard problem with this additional information. Countermeasures aim at preventing such an attack, *shuffling* as discussed in this work, belongs to the family of hiding countermeasures as explained in [11]. Hiding countermeasures aim to make the power consumption of a cryptographic device independent of the processed data by removing or at least reducing the exploitable link between the processed data and the power consumption. Shuffling achieves this by randomizing the execution order of independent operation within an algorithm such that an attacker looses the link between the secret data he tries to recover and the corresponding point in time in his measurements where this data is processed.

2.2 The Fisher-Yates Algorithm

The Fisher-Yates Algorithm (FY) generates a uniformly random permutation of a sequence and is therefore well suited for the use in shuffling. Other methods of permuting sequences of course exist, for these we refer to [26], but FY seems to be the most frequent suggestion in literature.

Algorithm 1 shows FY permuting an array from right to left. Starting with an ordered sequence of length L, we start from the rightmost index at $i = n - 1$. A number r in the interval $[0 : i]$ is sampled uniformly at random, and the elements at position r and i are swapped. The rightmost position is now frozen and will never be touched again. i is decreased by 1 and the process repeats, until i reaches 1, where it is decided whether the last two elements are swapped or not. Evidently, the algorithm requires random numbers from intervals of decreasing size which we generate by rejection sampling as suggested in [24], we again refer to [26] for a discussion of alternatives.

2.3 Attacks on the Generation of Permuted Sequences in Related Work

Attacks on Shuffling as a Side-Channel Countermeasure. There is, to the best of our knowledge, limited work on attacking algorithms generating random permu-

Algorithm 1: The Fisher-Yates algorithm

Input: sequence length n
Output: permuted sequence $\mathtt{a}$
for i **from** 0 **to** $n-1$ **do**
  $\mathtt{a}[i] \leftarrow i$;
end
for i **from** $n-1$ **to** 1 **do**
  $r \xleftarrow{\$} \{0, i\}$;
  $\mathtt{a}[i], \mathtt{a}[r] \leftarrow \mathtt{a}[r], \mathtt{a}[i]$;
end

tations for the purpose of shuffling in general and on FY in particular. Veyrat-Charvillon et al. provide an analysis and successful attack using Beam-Search for $L = 16$ in [26]. Using Beam-Search is intractable for the larger sequences we are interested in, since the beam width required to always retain the true solution is far too large, we thus have to come up with another solution.

Some other works propose attack strategies which require only information about single positions within a permuted sequence, for example [1]. Our aim is, however, to recover the entire sequence.

Attacks on Polynomial Sampling. Other works focus on FY in the context of fixed-weight polynomial sampling which exhibits different properties limiting the applicability of the attacks presented (see e.g. [9] and [8]) to shuffling. More specifically, these works rely on retrieving limited information, often only a single bit, from a full invocation of the used permutation generator. The specific attack strategies employed can deal with this amount of information, as they e.g. rely on only identifying different seeds to the permutation for a Chosen-Ciphertext-Attack on a KEM thus bypassing the FO-transform's rejection protection. It should be noted that these works even mention the theoretical possibility of targeting FY without further assessing it. The authors of [8] state that for an earlier version of Dilithium's sampling process, that the swap operations within a variant of FY are "hard to track for the propagated coefficients and make the attack complicated", while only extracting the number of zero and non-zero coefficients. In addition, [23] proposes an implementation protected against timing side-channels. Following on this, [9] enhances this implementation with a protection against power side-channels and states with respect to a timing attack on BIKE's variant of FY that: "While this attack exploits timing differences, there is no reason to believe that a power side channel cannot be exploited analogously." Still, none of these works provides a practical or theoretical assessment of the actual recoverability of the entire permuted sequence by an execution of FY.

2.4 ML-DSA

To showcase the practical implications of the proposed attack, we focus on NIST's recent PQC signature standard ML-DSA [12].

Brief Introduction to ML-DSA On a high level, ML-DSA is a digital signature algorithm which bases its security on the Module Learning With Errors (MLWE) problem and a variant of the Module Shortest Integer Solution (MSIS) problem. As this work focuses on attacks against FY and uses ML-DSA primarily for demonstration purposes of such an attack, we exclude further details on ML-DSA and refer the reader to the standard [12]. Nevertheless, we outline the components of ML-DSA relevant for our attack in the following.

Attacking ML-DSA For a passive side-channel attack on a signature scheme, usually the only interesting component is the secret key, which in the case of ML-DSA consists of two matrices of polynomials, $\mathbf{S}_1$ and $\mathbf{S}_2$. From an attackers perspective, only $\mathbf{S}_1$ is of real interest, as [2] and [18] both showed that the knowledge of $\mathbf{S}_1$ is sufficient to forge signatures albeit with a larger error probability. We thus limit our analysis in the following on $\mathbf{S}_1$. Further, as the polynomials within $\mathbf{S}_1$ can be recovered individually for our attack, we limit our analysis to one polynomial of $\mathbf{S}_1$ which we refer to as $\mathbf{s}_1$ for simplicity. Still, for a recovery of $\mathbf{S}_1$ we need to recover all the polynomials it consists of. For the smallest parameter set, ML-DSA-44, $\mathbf{S}_1$ consists of four polynomials.

The operations most attacked in the literature for ML-DSA are the NTT transform of $\mathbf{s}_1$, the coefficient-wise multiplication of $\mathbf{s}_1$ and the public challenge c and the subsequent addition of $c\mathbf{s}_1$ and $\mathbf{y}$ to form $\mathbf{z}$. Of these options, we opted to attack the NTT transform of $\mathbf{s}_1$. Our choice for this operation is on the one hand motivated by the fact that, to the best of our knowledge, no published attack exists on a shuffled instance of a NTT where a complete layer is shuffled and ML-DSA uses the largest NTT instance of the currently standardized PQC algorithms, such that an attack applicable here is also applicable to smaller instances such as the one used in ML-KEM. On the other hand, shuffling as a countermeasure for NTTs has been proposed and evaluated in recent years, as outlined below.

ML-DSA uses the NTT for faster polynomial multiplication. Typical algorithms to compute this transform split it into several layers and the operations in each layer into so-called *butterfly units* such as the Cooley-Tukey [4] and Gentleman-Sande [22] algorithms. An example of such a butterfly unit as used

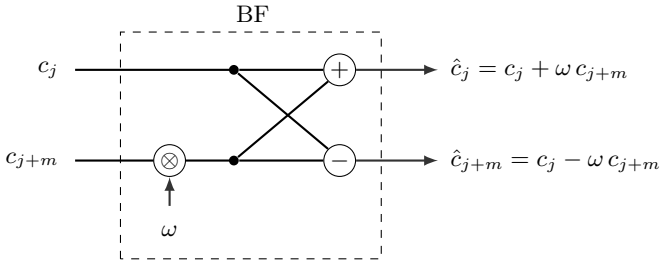

Fig. 2. Single "forward" butterfly of a NTT

in the ML-DSA reference implementation and thus in our attack is shown in Figure 2. Here the second input coefficient is multiplied by a precomputed twiddle-factor ω and subsequently added/subtracted from the first input to obtain the two resulting outputs. The complete NTT used in ML-DSA has $\log_2(256) = 8$ layers for the secret polynomial $\mathbf{s}_1$. A toy example of such a NTT for polynomial length 8 is shown in Figure 3.

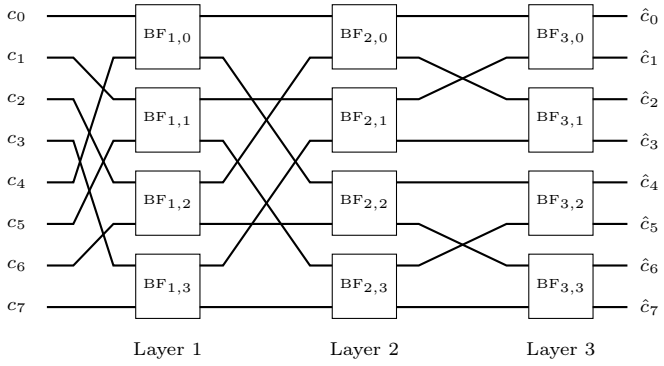

Fig. 3. Toy example NTT for $n = 8$

Existing Attacks on ML-DSA's NTT of $\mathbf{s_1}$ To the best of our knowledge, only [6] shows a passive attack against an NTT in ML-DSA. The authors of [6] focus on the forward transform of $\mathbf{s}_1$ and recover the upper part of this polynomial within a single trace on only the first layer of the NTT utilizing a machine-learning based distinguisher. The lower half of $\mathbf{s}_1$ is harder to attack due to the structure of the NTT, here the authors utilize leakage from several subsequent layers to recover the coefficients. We discuss the impact of this difference on our attack below. Given that FY was used to shuffle each NTT layer, we can use our attack to reorder a captured trace of the NTT such that it is again susceptible to this attack.

Several other works investigated on attacks against NTTs (see e.g. [16] and [15]). The only difference is that most of these attacks target ML-KEM which has a smaller (incomplete) NTT instance but this should only make our attack easier, as a smaller instance generally also means smaller sets to permute/shuffle.

Shuffling the NTT As a countermeasure against the attacks mentioned above Ravi et al. suggest in [20] to shuffle the NTT operations. This is motivated by the ineffectiveness of straightforward masking of the coefficients against single trace attacks while shuffling efficiently prevents such single trace attacks. Ravi et al. furthermore suggest a set of different shuffling options with varying trade-offs of performance overhead and shuffling complexity. More precisely, one can, in

increasing order of shuffling complexity and computational overhead either shuffle only the loading order of the coeffcients used within one NTT butterfly, the execution order of all operations using the same twiddle factor or *all* operations within one NTT-layer.

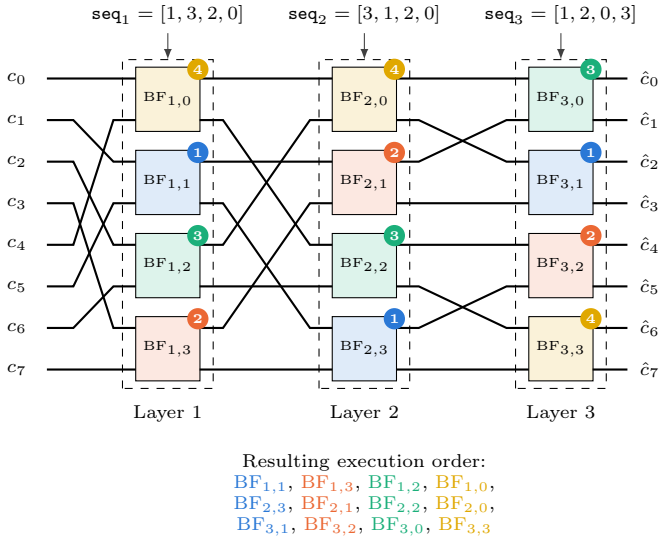

Fig. 4. Toy example NTT for $n = 8$ with shuffled layers and resulting execution order of the butterflies

Figure 4 outlines how shuffling for a complete layer is applied to an NTT instance. One first generates a permuted sequence of the length of one layer and executes the butterfly operations in the layer according to that sequence. Based on [20], Hermelink et al. propose adaptions to their Belief Propagation based attack to counter these shuffling countermeasures in [7]. We want to emphasize here, that their adaption does not work against a complete shuffled layer without prior knowledge as they also acknowledge at the end of their paper. This leaves the practical attackability of a NTT with shuffled layers unanswered for which we aim to provide an assessment for FY as shuffling algorithm.

3 Measurement Strategy and Hint Generation

This section describes the leakages we expect from an execution of FY and explain the hints we generate from this leakage. Furthermore, we outline the leakage we expect from ML-DSA's NTT and provide measurement results on the hints obtainable from these leakages in practice. We assume that we are in general unable to directly observe the resulting permuted sequence by e.g. directly building templates on it and recovering the corresponding values with

sufficient accuracy. We thus need some kind of post-processing that efficiently combines the hints we obtained to recover the correct permutation and resulting sequence.

3.1 Attack Preliminaries

For this work, we assume an attacker with access to a target device as well as a profiling device of the same type, mounting template attacks where a model of the leakage is built on the profiling device and later applied to the target device. We furthermore focus solely on software implementations. Our Fisher-Yates implementation is as generic and straightforward as possible, while the shuffled NTT is adapted from the ML-DSA reference code following the approach suggested in [20]. To this extent, the attacker has full control over the profiling device and knows e.g. the randomness required for the shuffling process. We furthermore assume that the attacker has access to multiple traces from the target device with the same secret material but different randomness for the shuffling countermeasure. This is e.g. a realistic scenario for a device computing multiple signatures with the same signing key. Please note, however, that we only use information from a single measurement to recover a permuted sequence, as the changing input randomness results in a different permutation for each invocation of the shuffling algorithm.

For the practical evaluation, we rely on standard side-channel attack methods such as SNR [10] calculation for POI detection and multivariate gaussian templates with reduced dimensionality by the help of LDA ([3], [25]). For the different attack points outlined below, we use between 500 and 1000 POIs per instance with a dimensionality reduction to between 2 and 15 dimensions. We use the implementations available within the SCALib.[3]

3.2 Possible Leakages for FY

We assume that we can, agnostic of a leakage model, observe leakage of the variables outlined below from each loop iteration of the for-loop in Line 4 of Algorithm 1 independently. Beyond these per-iteration leakages, there is likely also leakage stemming from data overwrites between iterations as an unprotected software implementation will most likely use the same registers for the same variable across multiple iterations. While such leakages are in principle possible, they are harder to model and to incorporate into our attack. We therefore opted to omit them in our analysis and leave this open for future work.

Leakage of the random number r. Within each loop iteration of the algorithm, a fresh random number is sampled from the interval $[0 : i]$. Furthermore, as the domain of this number shrinks with each loop iteration, it seems sensible to build a separate set of templates for each iteration, accounting for the changing interval and thus the decreasing number of classes to be distinguished.

[3] `https://scalib.readthedocs.io`

Leakage of the value being replaced. With each loop iteration, the element at position i, i.e. at the position determined solely by the current iteration number is swapped with another element (or itself). The domain of this number also changes with each iteration, but in contrast to the random number above, its cardinality *increases* by one with each iteration. In the first iteration, we know the value being replaced since we know the original sequence, each subsequent iteration increases the domain by one. We refer to this value as `s1` for the remainder of this work.

Leakage of the value at the selected position `r`. This is the element being loaded depending on the random number `r` and subsequently stored at the position determined by the round iterator i. In contrast to the two previous values, the domain here is always $[0 : n-1]$. This leakage corresponds to the final permuted sequence, i.e. the leakage captured in the first loop iteration corresponds to the value ending up at the last position, the second leakage corresponds to the second-to-last position and so on. We refer to this value as `s2` for the remainder of this work.

Combined Leakage of `r` *and* i. Both `r` and i are used as address for elements of the shuffled sequence. It seems therefore plausible to get leakage on a combination of those two values such as the Hamming distance of `r` and i per iteration.

Combined Leakage of the elements to be swapped in the sequence. Just as `r` and i are used as addresses for elements in sequence, the respective elements `s1` and `s2` are swapped in their position. Likely, this will in reality result in one element being overwritten by the other thus also resulting in some kind of combined leakage.

3.3 Leakages from ML-DSA's NTT

As outlined above, the only state-of-the-art attack on ML-DSA's NTT uses information from multiple NTT layers to recover the entire secret polynomial `s1`. [7] furthermore states that their attack against a shuffled NTT instance in ML-KEM/Kyber gets more complicated with increasing size of the layer, respectively the shuffled block within one layer, up to the point where they are unable to directly apply their attack. As the first layer of the forward transform has the largest shuffling block size in their terminology (i.e. the entire layer) and has from an attacker's perspective the most advantageous properties, we opted to focus our attack solely on the first layer. For ML-DSA, this first layer consists of 128 butterfly operations, each computing on two input coefficients, one from the lower half of the input polynomial and one from the upper half. As discussed above, we do not know the order in which these butterfly operations are executed in a shuffled implementation but we assume here that we are able to obtain some kind of leakage on both coefficients per butterfly but do as of yet not know to which lower and upper coefficients in the polynomial this leakage belongs.

3.4 The Measurement Setup

We evaluated our results on STM32F303 microcontrollers with one device serving as profiling device and two other devices as attack devices. All traces were captured on the ChipWhisperer CW308 platform using an 8-bit Picoscope 6402D with a sampling rate of 2.5 GS/s. The microcontrollers were running at 10 MHz and the oscilloscope was clock synchronized with the controllers. We measured the current draw of the device over a Shunt of 12 Ω. We captured a dataset of 100.000 traces for profiling and first evaluation of the accuracy templating approaches and leakage models. Each attack dataset consists of 1000 measurements of the same key, each taken with fresh randomness for the shuffle. Each measurement contains a full execution of the Fisher-Yates algorithm including rejection sampling assuming that there is a sufficiently large buffer of randomness available from which further samples can be drawn in case of a rejection. While the rejection sampling routine in FY is in principle unbounded in the amount of randomness required, we simplify our attack at this point as we consider the generation of random bits out of scope for this work. We measured the shuffled NTT seperately with the previously generated shuffling sequence as an input due to issues with the total measurement length. We stress that, though the measurement is split in two, measuring an entire execution of Fisher-Yates respectively the first NTT-layer via the power side-channel aims to be as close to real attack scenario as feasible with our equipment.

3.5 Preprocessing of the Measurements

The rejection sampling routine discards a random number `r` if it does not fit in the required interval for the current iteration. Each discard leads to misalignments after the first iteration of the for-loop in the algorithm due to the fact that additional random numbers need to be loaded and checked until one is within the desired range. To successfully model the leakages discussed above, we therefore need to detect such rejections and realign the measurement accordingly. We hence first need to determine whether one or multiple rejections happened for a certain iteration of the algorithm.

Detecting rejections This can be achieved by building a template-based distinguisher between a rejection and the subsequent swap when no rejection happens and applying this distinguisher sequentially for each loop iteration. To this end, we require two datasets for profiling, one without rejections for the current iteration and one with one or more rejections for the current iteration. For a sufficiently large SNR, by inspection of traces or otherwise by inspecting the assembly on our profiling device, we are able to determine the number of clock cycles corresponding to a single iteration as well as to an additional rejection, thus allowing us to iterate over the trace and to remove the number of clock cycles corresponding to one rejection for the profiling dataset. Based on this modeling we can then iteratively detect whether one or more rejections happened in the

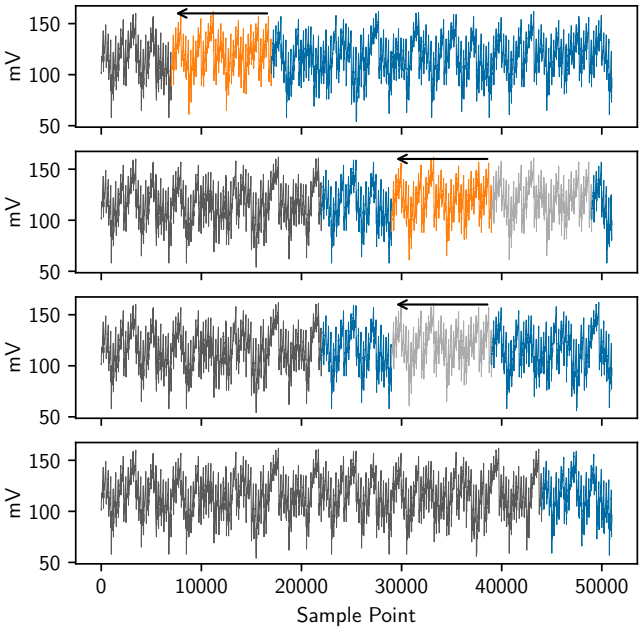

Fig. 5. Iterative removal of samples belonging to a rejection in a FY-trace

current iteration on our attack device and remove the number of clock cycles corresponding to one rejection from the measurement. This process can be repeated recursively within one iteration thus allowing us to remove an arbitrary number of rejections while having to only distinguish two cases - no rejection or at least one rejection. Given that all rejections were correctly identified and removed, the captured attack traces should again be aligned for the next iteration of the loop in FY, allowing us to repeat the process. We illustrate this procedure with a sample trace in Figure 5 where remove one rejection in the first plot and one at a time two rejections from the subsequent iteration in the second and third plot. After applying this operation to all iterations we ideally end up with a set of traces without any rejections in the trace and thus with aligned swap operations for each iteration. We tested this approach cross-device for validation purposes and found that we detect all rejections correctly for around 90% of the measurements thus resulting in around 10% measurements where subsequent attack steps will likely fail due to misalignment. Distinguishing these measurements is not possible in a real-world scenario, so these measurements are still passed into the attack pipeline.

3.6 Hints obtained from the Fisher-Yates Algorithm

With the profiling dataset freed of the clock cycles corresponding to rejections and the attack dataset ideally also free of rejections based on the iteratively built and applied templates above, we can now continue the attack by iterating over these "new" measurement traces. Figure 6 summarizes the six types of hints generated on a toy example of length 8.

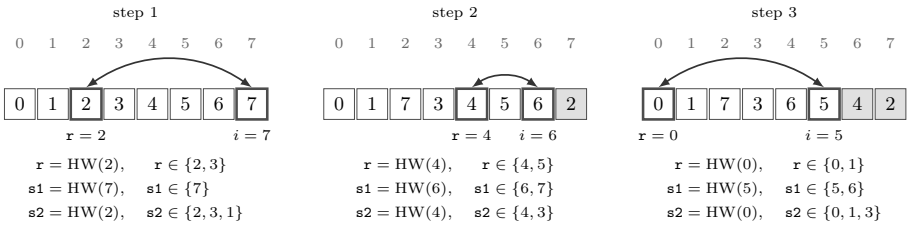

Fig. 6. Example for hints from a FY shuffle of length 8. `r` is the Hamming weight of the random index. `s1` and `s2` are the Hamming weights of the value being replaced and the value being picked up, respectively. For each step we get three candidate-list hints containing the possible domain of `r`, `s1`, `s2` (Integer-valued).

Modeling per Loop-Iteration of the algorithm. We first build templates for the three single-value leakage points outlined above (`r`, `s1`, `s2`). As discussed in Section 3.2, the number of classes here varies with each iteration leading us to build the models also per iteration as the on average smaller number of classes to distinguish leads to superior results in comparison to building models with a larger amount of training data by using all iterations at once. Subsequently, we apply these templates to our realigned attack dataset.

Chosen Leakage Models. We built the above mentioned models for Hamming weight (HW) leakage as well as for Identity (i.e. Integer value) leakage of each value. Unfortunately, in contrast to what we expected, we observed little to no Hamming-Distance leakage for the possible leakage points discussed above, leading us to omit the Hamming-Distance from our subsequent evaluation.

HW Hints. Here, we built templates for the eight possible Hamming-Weight classes and subsequently provided only the HW with the maximum likelihood in addition to the apriori determined standard deviation of the underlying distribution for this assumed HW. A set of HW hints for one full execution of the algorithm thus contains three lists, for `r`, `s1` and `s2` of 127 HWs each.

Candidate-List Hints. As discussed above, the possible intervals here vary, but at maximum we need to build templates and distinguish the 128 classes associated with $[0 : 127]$. From applying these templates we obtain a probability for each

possible value with the sum of all probabilities being one. In order to get this leakage into a form consumable by our solver, we opted to build a list of possible integer values for each attacked variable in each loop iteration of the algorithm. The length of this candidate-list is determined by a threshold for the cumulative probability of all entries in the list thus excluding a varying number of the most unlikely candidates. In addition to this candidate-list hint, we also provide the probability associated with each included value (later referenced as the posterior P_t).

3.7 Hints Obtained from the First Layer of ML-DSA's NTT

For the first layer of the NTT of ML-DSA's secret polynomial $\mathbf{s}_1$, we base our attack on the analysis in [6]. Here the authors distinguish between the lower and upper 128 coefficients of $\mathbf{s}_1$, as the upper 128 coefficients are multiplied by a twiddle factor first and get subsequently reduced, resulting in significantly more side-channel compared to the lower 128 coefficients. We therefore also split our attack in two stages, first attacking the upper half for which opted to directly distinguish five classes for the five different possible values per coefficient in ML-DSA-44 with the possible range $[-2:2]$. Put differently, we get

$$c'_{j'+128} \quad \forall\, j' \in [0:127].$$

Note however, that we do not yet know the permutation used in our shuffling countermeasure such that we can not map j' back to j. However, we can utilize the knowledge about $c'_{j'+127}$ to improve our attack on $c'_{j'}$ as both are used in the same butterfly operation and thus at the same point in time in our measurements. We can build templates conditioned on the knowledge of c'_{j+128}, i.e.:

$$c'_{j'} = \max P(\texttt{leak}(c'_{j'})|c'_{j'+128})$$

The main advantage here is that this allows us to include the addition and subtraction results from the butterfly operation into our template thus increasing the amount of leaking instructions in our software we can observe and extract from. We further exploit the fact that the HW of -2 and -1 differs significantly from the HW of 0, 1, and 2 thus allowing for an efficient distinguisher between negative and non-negative coefficients. In total this allows us to build templates distinguishing -2 and -1 separately from 0, 1, and 2 and dependent on the corresponding coefficient from the upper half, allowing for a better accuracy. From here, we refer to these values with their class labels $[0:4]$ which are obtained by adding two each coefficient.

3.8 Measured Leakages and Resulting Hints for FY and the first layer of ML-DSA's NTT

For the HW hints, we assess the quality of our measurements by the standard deviation of the perceived HWs for a fixed true HW, as done in [15,5,7], i.e. we

assume that the perceived HW h' of a true weight h follows

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

The corresponding standard deviations measured with for our attack on FY can be found in Table 1.

To assess the quality of our candidate-list hints, we provide the average list length and well as the rank of the correct solution within this list as metrics. Due to the nature of the solver we use to retrieve the correct permuted sequence from these hints, we need each list to contain the correct value for this position. We thus use a relatively high threshold for the lists to include the correct solution with a sufficiently high probability.

Hint	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7
`r`	0	0.15	0.15	0.15	0.15	0.15	0.15	0
`s1`	0.16	0.18	0.18	0.16	0.16	0.16	0.18	0.40
`s2`	0.16	0.18	0.18	0.17	0.18	0.18	0.25	0.35

Table 1. Per-weight standard deviations of the Hamming weight hints

Still, this approach yields approximately 89% *un*usable hint sets in a cross-device setting, resulting in a total of $\approx 10\%$ usable hint sets taking measurements with incorrectly detected rejections from above also into account. Again, this number has been obtained with prior knowledge of the computed data as there is no practical way of distinguishing usable from unusable hint sets solely based on the measurements. The resulting list lengths and thresholds are summarized in Table 2.

Hint	Rank in List	Avg. List Length	List Cutoff Threshold
`r`	0.79	5.7	0.9999999
`s1`	3.79	27.9	0.99999
`s2`	6.04	40.1	0.9999999

Table 2. Accuracy of the candidate-list hints

For the first layer of ML-DSA's NTT, we achieved perfect accuracy for the upper half of input coefficients also in the cross-device setting. Put differently, we know per butterfly which coefficient was used as second input, but we do not know which position in the upper half this coefficient belongs to. Still, for the given butterfly this allows us to attack the first input with knowledge of the second input. We furthermore achieved perfect accuracy on distinguishing a negative from a non-negative first input thus also allowing us to apply the strategy described above of building separate templates to distinguish within the

domains $[-2:-1]$ and $[0:2]$. We summarize these results in Table 3 with overall accuracy, i.e. fraction of hints with the correct value at the highest position.

Hint	Value	Neg./Non. neg	Value w. knowledge of other input
Lower Half $\mathbf{s}_1$	71.4%	100%	91.9%
Upper Half $\mathbf{s}_1$	100%	-	-

Table 3. Accuracy of the hints obtained of the first layer of ML-DSA's NTT

Please note that we observed systematic misclassifications for these lower coefficients in our cross-device setting which can be attributed to the challenge of porting our templates between devices. A discussion as well as possible solutions for this can be found in [13].

4 Recovery of One Shuffled Sequence

This section is concerned with the reconstruction of a single shuffled sequence of length $L = 128$ on the basis of the Hamming weight (HW) and candidate-list hints.

As input we have 6 hints *per step* of the shuffle: Three HWs `HW(r)`, `HW(s1)` and `HW(s1)`, which are the weight of the randomly selected index, the element being replaced and the element being picked up, respectively. We assume that these HW readings are affected by Gaussian noise with some per-value standard deviation $\sigma_{\{\texttt{r},\texttt{s1},\texttt{s2}\}}$ as given in Table 1. For each type of hint we also have the candidate lists `cand_r`, `cand_s1` and `cand_s2` which contain all admissible assignments for this step. We assume that the correct value is always contained within these candidate lists as discussed in section 3.6. Associated with each candidate in each list, we also have the posterior probability of this value, summing to 1 over all candidates. Table 4 gives as an example a set of hints given to the solver from a real measurement, together with the hard and soft information used in the solver as discussed in the next section.

The task now is to reconstruct the most likely shuffled sequence given the hints. Since the σ-values of some HW readings are quite large and the candidate list posteriors may not be exact, the most *likely* sequence is often not the true shuffle, just a more likely answer *given the hints*. We are therefore also interested in constructing a ranked list of the k most likely shuffles.

4.1 CP-SAT Modeling

We use the CP-SAT solver of Google's open-source software suite OR-Tools [4] to encode both the hard constraints and the soft objective.

[4] https://developers.google.com/optimization

	r	s1	s2
HW({r,s1,s2})	1	7	6
cand_{r,s1,s2} (posterior P_t)	2 (0.71) 3 (0.29)	127 (0.65) 125 (0.19) 119 (0.10) 123 (0.05) 126 (0.01)	119 (0.39) 126 (0.35) 123 (0.16) 125 (0.10) 121 (0.00)
domain	$[0:114]$	$[114:127]$	$[0:127]$
hard constraint	$v \in \text{candidates} \cap \text{domain}$		
soft term for candidate v	$\log P_t(v) + \texttt{HW_FLOOR}(\texttt{HW(\{r,s1,s2\})}, \texttt{HW}(v))$		

Table 4. An exemplary input to the solver at step 13 ($i = 114$). The true values at this step are r $= 2$, s1 $= 123$, s2 $= 119$.

Recall that the HW leakage is modeled as the true value plus some additive Gaussian noise, so each observation obs can be written as

$$\texttt{obs} = \text{clamp}_{\{0,\ldots,7\}}(\text{round}(\texttt{true_HW} + \mathcal{N}(0, \sigma^2)). \tag{1}$$

The likelihood $P(\texttt{obs}|\texttt{true_HW})$ of observing obs given that the actual weight was true_HW is therefore

$$P(\texttt{obs}|\texttt{true_HW}) = \Phi\left(\frac{d+0.5}{\sigma}\right) - \Phi\left(\frac{d-0.5}{\sigma}\right)$$

where d is the (integer-valued) difference between the observed and true value,σ is the standard deviation of true_HW from the modeling in Eq.1 and $\Phi(z)$ is the CDF of the normal distribution. For obs $= 0$ and obs $= 7$ we use the one-sided tails (the two bins are open-ended since every value below 0.5 gets mapped to 0 due to clamping, same for the upper part above 6.5).

We start by precomputing the 8×8 log-likelihood matrix HW_FLOOR for each stream (r, s1, s2) as follows:

$$\texttt{HW_LL}[\texttt{obs}][\texttt{true}] = \log\left(P(\texttt{obs}|\texttt{true_HW})\right)$$

$$\texttt{HW_FLOOR}[\texttt{obs}][\texttt{true}] = \begin{cases} \log(10^{-6}) & \text{if } \texttt{HW_LL}[\texttt{obs}][\texttt{true}] = -\infty \text{ and } \sigma > 0 \\ \texttt{HW_LL}[\texttt{obs}][\texttt{true}] & \text{else} \end{cases}$$

This is to prevent a floating-point underflow in instances where σ is small but nonzero and d is large, meaning that an unlikely measurement will be penalized heavily by the solver but not be treated as impossible (as opposed to the cases where $\sigma = 0$, which should be treated as hard constraints).

Next, we build the SAT encoding. The shuffling problem can be characterized naively by $128 \times 127 \approx 16000$ variables $v_{i,j}$, where each variable represents the possible values of position i in shuffling step j. Since each step of the shuffling only touches on at most two positions, most of the variables will be copies of

each other. We use variable aliasing to resolve this issue by keeping a list of variable references and only creating a fresh variable when necessary. Based on the average number of candidate solutions for the drawn index (i.e. the number of elements in `cand_r` list at this step, cf. Table 3) we only need around 7 new variables per step, leading to an overall variable count of ≈ 800. We call these variables the *state* variables. In addition to the state variables, we also generate the same amount of *choice* variables, one new boolean variable per candidate position of `r`, per step. Finally, we create three 'cost' variables per step, one each for `r, s1, s2` encoding the soft information. Their sum over all 128 steps will be the objective to optimize. Table 4.1 shows the created variables for one exemplary step (same as in the above example of Table 4).

variable	meaning
`j13_2`	$\mathtt{r} = 2$
`j13_3`	$\mathtt{r} = 3$
`A13_114`	value at 114 after
`A13_2`	value at 2 after
`A13_3`	value at 3 after
`c1_13`	`s1` cost
`c2_13`	`s2` cost

constraint	role
$\mathtt{j13_2} + \mathtt{j13_3} = 1$	exactly one r
$\mathtt{j13_2} \Rightarrow \mathtt{A13_114} = \text{old}[2]$	swap if $\mathtt{r} = 2$
$\mathtt{j13_2} \Rightarrow \mathtt{A13_2} = \text{old}[114]$	
$\neg\mathtt{j13_2} \Rightarrow \mathtt{A13_2} = \text{old}[2]$	else unchanged
$\mathtt{j13_3} \Rightarrow \mathtt{A13_114} = \text{old}[3]$	swap if $\mathtt{r} = 3$
$\mathtt{j13_3} \Rightarrow \mathtt{A13_3} = \text{old}[114]$	
$\neg\mathtt{j13_3} \Rightarrow \mathtt{A13_3} = \text{old}[3]$	else unchanged
$\text{old}[114] \in \mathtt{cand_s1} \cap [114, 127]$	`s1` membership
$\mathtt{A13_114} \in \mathtt{cand_s2}$	`s2` membership
$\mathtt{c1_13} = \text{S1COST}[\text{old}[114]]$	cost lookup
$\mathtt{c2_13} = \text{S2COST}[\mathtt{A13_114}]$	cost lookup
$\text{obj} \mathrel{+}= \mathtt{c1_13} + \mathtt{c2_13} + w_2\,\mathtt{j13_2} + w_3\,\mathtt{j13_3}$	

Table 5. Left: variables created at the step ($i = 114$, $\mathtt{r} \in \{2, 3\}$). Right: constraints added or updated during the same step.

Per-step constraints. For each step we encode the following constraints: First, there should be exactly one `r` chosen per step (i.e. one of the two boolean variables created is True). Second, the values of `s1` and `s2` must be within the candidate list and allowed domain (e.g. larger than i for `s1`). Third, the swap itself is encoded – if `r` is chosen, the values at position i and `r` are exchanged, else position `r` keeps its value. Finally, we look up the two costs for `s1` and `s2` from the `HW_FLOOR` table at the value of i before and after the swap, respectively, and update the objective function. Aside from the per-step constraints, we also add global constraints inherent to FY: the `AllDiff` constraint on the final state and the domains of `r` and `s1`. These are redundant given the other constraints but we found experimentally that they propagate well and make the model converge faster.

The set of admissible solutions given only the hard constraints (i.e. the candidate lists) grows exponentially with the length of the shuffled sequence. At the

target length $L = 128$ the size of this admissible set is difficult to compute, but we believe it to be in the order of $10^{20} - 10^{30}$, so impossible to enumerate by any means. In order to retrieve the most likely shuffled sequences, we need to incorporate the soft information, namely the HW hints and the posterior probabilities of the list elements, directly into the solver.

The objective function is the scaled loss of the HW hints and the posterior probability of the observation, summed across all steps and all hint streams:

$$\text{objective} = \sum_{i=0}^{L-2} \sum_{\texttt{s1,s2,index}} \lfloor \texttt{scale} \cdot (\texttt{HW_FLOOR}[\texttt{obs}][v] + \log P_t(v) \rceil, \qquad (2)$$

where v is the value of the hint at step i (e.g. $\texttt{r}_i$ for the random index hint at step i). $\lfloor \cdot \rceil$ denotes rounding to the nearest integer and is required since the CP-SAT objective can only have integer coefficients. Setting the scaling parameter to 1000 moves a log-likelihood value of, say, $-0.3354...$ up to $-335.4..$, which is then rounded to the nearest integer -335. Put differently, by setting `scale` to 10^x, x decimal digits of the log-likelihood are preserved. Experimentally, we found that `scale` $= 1000$ provides a good trade-off between precision and performance.

4.2 Enumerating the top-k solutions.

The CP-SAT solver cannot by itself enumerate the most likely solutions given the hints, it can only enumerate a solution set for a model *without* an objective, where it would list all possible assignments in an unordered fashion. It can, however, produce the most likely (MAP) solution to the model, even providing a proof of optimality, meaning that the returned solution is the maximum of the objective function under the given constraints. We can bootstrap this to enumerate all solutions in the following way: First, query the solver with the initial modeling and constraints. It returns the MAP solution given the provided objective function. We add the recovered sequence as a new NOT-clause to the modeling, disallowing the exact same assignment to be produced again and requery the solver. Now, it will produce the guaranteed second-best sequence, which we can again add as a blocking clause. We proceed in this fashion until the k most likely solutions are produced.

5 Mapping Shuffled Sequences Back to NTT Coefficients

In this section we describe the reconstruction of one NTT polynomial $\mathbf{s}_1$, represented by its coefficients $c_0 - c_{256}$, from a list of candidate shuffles. We recall that we map the coefficients from $[-2 : 2]$ to their class labels $[0 : 4]$ for our discussion.

For any given measurement, we start by enumerating the top-k shuffles given the hints associated with this measurement. As an additional input, we have

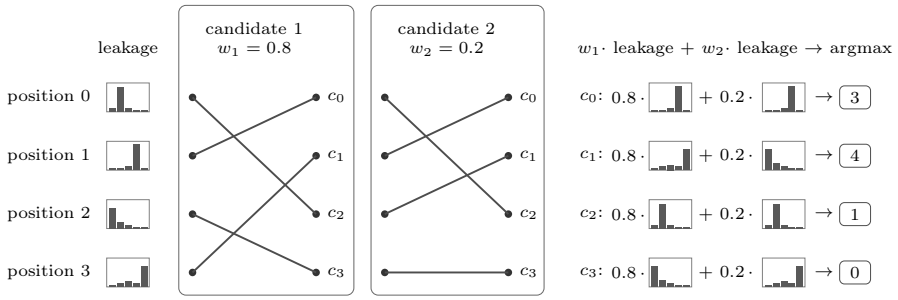

Fig. 7. Toy example of the coefficient reconstruction. Each of the top-k shuffles suggests one mapping of leakage positions to coefficients. The weight w_i of each candidate reflects the confidence that this shuffle is correct and is used to weigh the leakage. The coefficient sequence is obtained by computing the argmax over $\{0, \ldots, 4\}$ for all coefficients.

the coefficient leakage as described in section 3.7, which is one list per coefficient value containing the probability for each assignment, summing to one. The hints are split into two halves, one corresponding to the lower half of measured positions and one to the upper.

We for now focus on the reconstruction of the upper half since it tends to have more precise hints and discuss in the following section how the reconstruction of the upper half can facilitate the reconstruction of the other, since the two are linked by the same shuffle (cf. Figure 4).

Each candidate in the top-k enumerated results suggests one mapping of the measured leakage *positions* to the upper coefficients $c_{128} - c_{255}$, giving a prediction about the coefficient values based on the leakage values. Each candidate should ideally contribute to this prediction proportional to how likely the candidate was given the hints. The scores attached to each solution returned by the solver are with respect to the objective function 2 and not normalized and still scaled to fit into the integer-valued objective. We therefore re-score each of the obtained sequences according to the same loss function, without the scaling and rounding, to obtain a value `score_k` each, and then compute the softmax over all scores within the top-k as

$$w_k = \frac{\exp(\texttt{score_k} - \texttt{s_max})}{\sum_m \exp(\texttt{score_m} - \texttt{s_max})} \tag{3}$$

which assigns to the top-k candidates weights w_i that sum to one. Next, we build a 128×5 array where each row belongs to one coefficient and each column to one 'hypothesis' $\in \{0, \ldots 4\}$. As an example, for the top sequence in the first measurement which has a confidence weight w_1: for each position j from 0 to 127, read off where this sequence suggests position j maps to. To this row in the array add $w_1 \cdot$ `key_leakage_upper`. This is repeated for every sequence, adding all measurements into the same array. Finally, computing the argmax on each

row gives the most likely value for each coefficient of the polynomial. A toy example illustrating the process is given in Figure 5.

This method of reconstructing the coefficients is quite robust to wrong shuffles in the candidate lists, and also works in cases where the true shuffle is not even in the candidate list. An interesting observation is also that there exist different shuffles that map to the same sequence of coefficients, since their alphabet is quite small. It is entirely possible that even if the true shuffle was not within the enumerated list, the sequences that are contained are similar enough to 'vote' correctly on most positions and still recover the coefficients correctly, we observe this effect in Section 7.

6 Putting It All Together

In this section we describe the full end-to-end attack from measurements to the recovery of all NTT coefficients, highlighting the parameters that need to be decided in each step. Figure 8 shows an overview of the full attack pipeline.

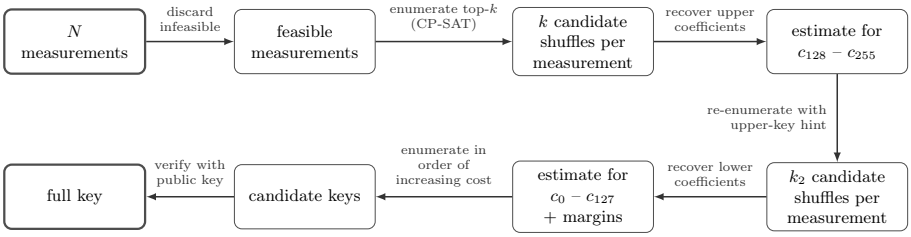

Fig. 8. The full attack pipeline, from N measurements to the full key.

We start from N measurements as obtained from Section 3.8. We build a CP-SAT instance for only the hard constraints of each measurement as described in Section 4 in order to decide which measurements are feasible to use for recovery. If the hints derived from any measurements generate for example candidate lists that disagree with each other or are not compatible with *any* FY shuffle, they are discarded in this step. This decision is very quick to make, costing only a few seconds for the entire measurement file. On average $10\% - 20\%$ of measurements taken were usable, these feasible measurements proceed to the next step. At this point we need to highlight that there are two types of measurements that the solver will mark as feasible: For both kinds, the model is satisfiable, i.e., there exists an assignment that fulfills all constraints. But, for some of these measurements the true sequence is not admissible given the model, i.e., there exists *some* solutions, but the the true sequence is not one of them. As an attacker, we cannot distinguish between these two cases, so both proceed to the next step.

For each feasible measurement, we rebuild the SAT-model, this time with the additional soft information and the objective function given in Eq. 2. We

enumerate the k most likely sequences for each measurement as described in subsection 4.2, each of which receives a weighted score through the softmax function described in Eq.3. In theory, we could now map back to the NTT coefficients directly, 'unshuffling' the coefficient leakage as prescribed by the top-k shuffles and computing the most likely coefficients for both the upper and the lower part. However, as discussed in Section 3.8, the leakage from the lower coefficients is far worse than from the upper ones. While the upper coefficients can almost always be correctly reconstructed in this step, the lower coefficients are far noisier, and thus the reconstruction is far less robust to sequences in the top-k that are 'very different', in some sense, to the true shuffle. At this point, we can exploit the fact that the lower and upper coefficients were affected by the same shuffle and use the low-noise leakage of the upper coefficients to better reconstruct the lower ones.

To increase the accuracy of the reconstruction of the lower coefficients, we thus re-run the CP-SAT solver with an additional soft constraint: every shuffle returned should agree with the presumably correctly recovered upper coefficients. As an example, if we assume the upper coefficients have been recovered correctly and the value of coefficient c_{40} is '3' each sequence that would most likely result in another value being put in coefficient c_{40} (and thus also c_{168}) should be penalized. More concretely, we adapt the objective function `obj` from before (2) to

$$\text{objective} = \texttt{obj} + \sum_{p=0}^{L-1} \log\left(\texttt{leakage_upper}[\hat{c}_{\pi(p)+128}]\right) \tag{4}$$

i.e., one term per position in the final sequence, which measures how well this position agrees with the recovered upper coefficients. $\pi(p)$ is the permutation defined by the shuffle currently being scored, and $\hat{c}_{\pi(p)+128}$ is the value of that position's corresponding upper coefficient. Using this refined objective function, we again enumerate the top-k_2 most likely shuffles, where k_2 can be chosen independently of k. These all have the property that they agree with the upper coefficients, so all that is left to do is recovering the bottom coefficients as described in section 5.

When evaluating this pipeline, we found that this fixes most of the coefficients of the lower part but the quality of the lower leakage is still not good enough to recover all coefficients most of the time. The number of wrong coefficients is however quite small, so we propose a final enumeration step to recover the entire key. In the final step of the coefficient recovery, we read off the key as the argmax over $\{0, \ldots, 4\}$ for all positions, but in actuality we have at each position a list of accumulated probabilities per coefficients (cf. Figure 5). We can use this to enumerate keys in order of decreasing likelihood: associate to each coefficient-assignment combination a *cost* value that reflects the cost of moving from the argmax assignment a to this new assignment v:

$$\text{cost}(c, v) = \text{E}[c][a] - \text{E}[c][v]$$

where E is the accumulated evidence table from the right-hand side of Figure 5. We found that in practice only few positions need to be enumerated in this fashion, which is feasible since the cost of testing a key is cheap.

7 Results and Analysis

This section reports results for each stage of the attack. We begin with the recovery of a single shuffle, before turning to the end-to-end recovery of the secret key We used five datasets from two different devices for this analysis, with 1000 measurements each, all for different secret keys.

7.1 Recovery of a Single Shuffle

We first consider the recovery of the true shuffled sequence from a given set of measurements. Recall that for each measurement we can enumerate the certified top-k solutions with respect to the objective function 2.

We first investigate how often the true shuffled sequence is contained among the top-k results returned by the solver. This is in some sense an indicator of the quality of the hints, in that noisy hints will often lead to many other solutions with a higher objective score, i.e. matching the hints better than the true sequence does.

Figure 9 shows the probability that the true shuffle is within the k highest ranked solutions returned by the solver, as a function of k, i.e., *all* entries of the shuffled sequence are correctly recovered. This curve was obtained over 196 out of 1000 measurements from device 1, marked by the solver as feasible. As an example, a value of 0.21 at $k = 50$ means that in 21 percent of feasible measurements, the true shuffle is within the first 50 results.

Also shown as a dashed line is the fraction of admissible shuffles, i.e., the ones that were passed through as 'feasible' by the solver and for which the true sequence is a valid solution given the hints. This is only provided as a reference, since an attacker could not classify whether a feasible measurement can admit the true shuffle or not, but it indicates the ceiling that any solver can hope to reach. We found that in approximately 20% of feasible measurements, the true shuffle was among the first 50 results. For 10%, already the top-10 candidates contain the true shuffle. This set includes also the roughly 44% of measurements for which the true solution cannot possibly be found, so as a reference we also give the orange curve that only considers the 'achievable' measurements.

Another view is given in Table 6, which lists the number of measurements required to obtain the true shuffle in *at least* one of them, as a function of k. As an example, to be 99% sure that the true shuffle has been recovered, we need 16 measurements with the top $k = 100$ candidates each.

Since the cost of obtaining additional measurements is quite low, this makes the attack on the FY shuffle very practical. We also want to stress that even in cases where the true shuffle is not recovered, most of the positions in the shuffle are recovered correctly in all cases (meaning that the top-k candidates

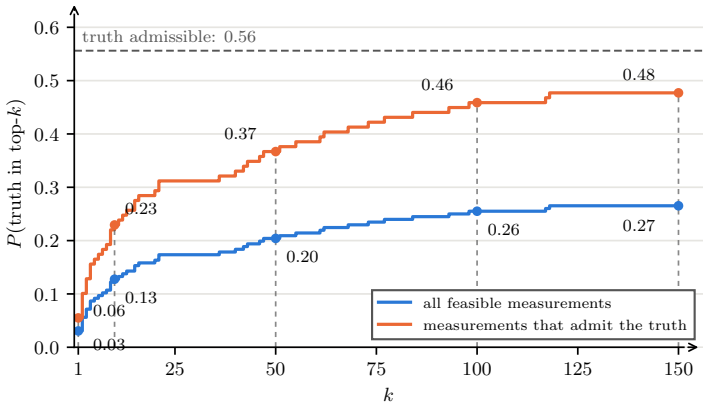

Fig. 9. Probability of the true shuffle being in the top-k enumerated results, computed over 196 feasible measurements (blue) and the subset of 109 feasible measurements that admit the true sequence. marked by the solver as feasible. The dashed line represents the fraction of measurements for which the truth is admissible.

are always very similar). In the next section we will show that it is not strictly necessary to have the true shuffle among the enumerated candidates, since the aggregation is robust to outliers that are wrong in only few positions.

7.2 Recovery of the polynomial $\mathbf{s_1}$

In this section we discuss the success rate of the end-to-end attack described in Figure 8. Recall that we first enumerate k shuffles per measurement, use them to derive the upper coefficients, and then derive the lower coefficients with the additional information from the upper coefficients. In case the polynomial obtained in this way leads to a wrong key, we start an additional enumeration step in order of decreasing likelihood.

We first report the accuracy of reconstructing the upper and lower coefficients. Figure 10 shows the fraction of correctly recovered coefficients as a function of the number of measurements m. We selected these three datasets in order to not overcrowd the figure and because they show different behaviors in the lower coefficients.

We first note that the 128 upper coefficients can be recovered from as few as $5-10$ measurements in all datasets. We can clearly see the lower accuracy of the lower leakage, for which some coefficients can not be reconstructed, no matter the number of additional measurements. As an example, device 1, key 1 saturates early, because it contains two coefficients that cannot ever be recovered. As discussed above, this is due to systematic misclassifications caused by porting a template from one device to another. These errors can e.g. be avoided with the techniques described in [13]. Even without this fix, we can still hope to enumerate these few remaining coefficients as shown in the final table of this section.

		target probability q				
k	p_k	0.5	0.8	0.9	0.95	0.99
1	0.031	23	52	75	97	149
3	0.071	10	22	32	41	63
10	0.128	6	12	17	22	34
50	0.204	4	8	11	14	21
100	0.255	3	6	8	11	16
150	0.265	3	6	8	10	15

Table 6. Number of feasible measurements m needed such that, with probability at least q, the true shuffle of at least one of them is among its top-k solutions. p_k is the probability that a single feasible measurement has its true shuffle among its top-k solutions (Figure 9). The probability that this holds for none of m measurements is $(1-p_k)^m$, so $m = \lceil \log(1-q)/\log(1-p_k) \rceil$. Infeasible measurements are discarded by the solver beforehand, so m counts feasible ones only.

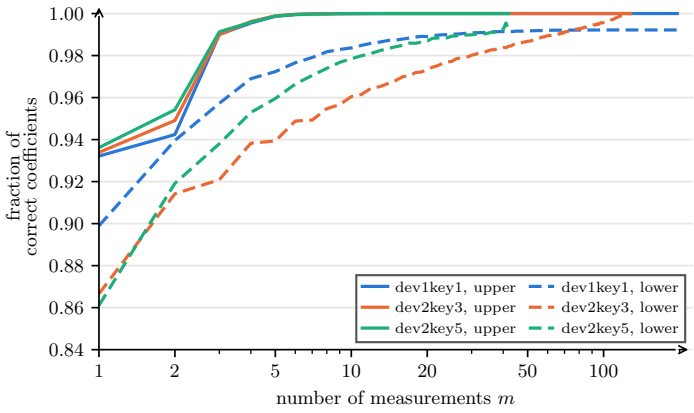

Fig. 10. Correctly recovered coefficients with increasing number of measurements. Solid lines: upper coefficients $c_{128} - c_{255}$, dashed lines: lower coefficients $c_0 - c_{127}$.

Somewhat surprisingly, we used only the top $k = 3$ candidates of each measurement for the reconstruction of the upper part and $k_2 = 10$ candidates for the reconstruction of the lower half. Increasing these parameters further did *not* lead to an increase in reconstructed positions, since both halves are either fully recovered or at their theoretical ceiling.

Another view is given in Figure 11, which shows the probability that m measurements are sufficient to reach the theoretical ceiling, i.e. reconstruct all coefficients that the leakage allows for, plotted for the same measurements as above.

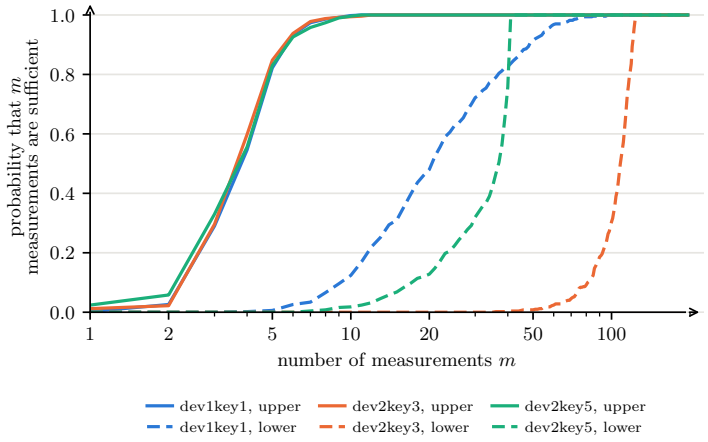

Fig. 11. Probability that m measurements are sufficient to reconstruct all coefficients up to their theoretical ceiling

Finally, we check the enumeration cost in those cases where not all coefficients could be directly reconstructed. In some sense, this count quantifies how 'confidently wrong' the leakage was across all coefficients. If the true polynomial is only different in one position, but this position is wrong with very confident leakage information, it will take long to get there by enumeration since we enumerate in order of decreasing likelihood, not number of correct positions. Counterintuitively, there can be reconstructions where two coefficients are wrong, and it only takes 4 tries to reach them by enumeration, since they are at the same time the most unsure positions.

Table 7 shows a summary of all device measurements. We can see that in all but one setting, the full key can be recovered with very low enumeration effort. Note that device 1, key 2 has the same amount of wrongly predicted in the lower part as device 2 in its second key, but the enumeration cost is vastly different. This is because the measurement of device 1 has positions that are confidently wrong, owing to the porting error described above. Perhaps surprisingly, we can

device	1	1	2	2	2
key	1	2	3	4	5
infeasible (discarded)	804	805	814	933	949
feasible	196	195	186	67	51
true shuffle in the top-k	14	12	10	8	6
used for the lower key	196	194	185	67	51
upper coefficients recovered	128	128	128	128	128
lower coefficients recovered	127	124	126	124	128
total	255/256	252/256	254/256	252/256	256/256
candidates to the true key	46	$> 2 \cdot 10^6$	4	648	1

Table 7. Summary of the coefficient reconstruction for both devices. The bottom row gives the number of candidate keys that need to be enumerated in order to recover the true key. Computed over 1000 measurements each for $k = 3$ and $k_2 = 10$

see that only very few of the measurements that produced the upper coefficients even included the true shuffled sequence in their top-3 enumerated results, highlighting the robustness of the method.

8 Conclusion

Summary Within this work, we proposed a side-channel leakage based attack on the Fisher-Yates algorithm. We attacked a software implementation of the algorithm used to permute the execution order of butterfly operations within ML-DSA's NTT. We managed to successfully remove timing variances due to the underlying rejection sampling routine and generated hints from the recovered leakage feasible for a constraint solver. With these hints, we were able to generate a ranked list of possible permuted sequences containing the correct sequence with reasonably high probability and to estimate how many measurements are required for a certain number of candidates per measurement, in order to have the correct shuffle at least once contained. If we e.g. want get the correct shuffle with 99% probability at least once for a list length of 10 we require 34 feasible measurements. With the knowledge of these possible sequences and additional leakage on the shuffled operations from ML-DSA's NTT, we were finally able to recover one $\mathbf{s}_1$ polynomial of a ML-DSA-44 secret key in four out of five cases from 1000 measured traces containing between 196 and 51 feasible measurements up to a point where it can be directly enumerated.

Outlook This work shows a practical attack on Fisher-Yates based shuffling as a countermeasure, and opens up many further open questions. On one hand, the attack strategies outlined here are suboptimal as we currently retrieve leakage from our implementation, model it to a certain type of hint and loose information on this way. We expect significant improvements in the performance of the

attack given that one interleaves the two processes more closely and e.g. uses information about structural positional differences for the hints or dependencies of the removal of a rejection on the hint quality also within the solver. It should also be noted here that the limitation of this attack is rather in the quality of the leakage on the NTT than in recovering the permutation from FY such that improving the portability of the templates as mentioned above should significantly improve the attack.

On the other hand, the natural question arises how the attack path opened by this type of attack can be closed. Here we want to stress that the algorithm used to generate permuted sequences must also be protected against side-channel attacks though this increases the cost of the otherwise low-cost countermeasure shuffling. As outlined above, [23] proposes a protected implementation of FY. Here it would be of interest whether this attack can be adapted to such a protected implementation.

Declaration of Generative AI Claude Code (Anthropic) was used to generate boilerplate code such as data parsers, but did not generate any of the main code of the measurement or the reconstruction. Further, all figures (excluding data plots) were generated by Claude based on human input (drawings) and feedback. No text or technical result was produced by a generative AI. The authors take full responsibility for all content within the paper.

References

1. Backlund, L., Ngo, K., Gärtner, J., Dubrova, E.: Secret key recovery attacks on masked and shuffled implementations of CRYSTALS-Kyber and Saber. Cryptology ePrint Archive, Report 2022/1692 (2022), https://eprint.iacr.org/2022/1692
2. Bruinderink, L.G., Pessl, P.: Differential fault attacks on deterministic lattice signatures. IACR TCHES **2018**(3), 21–43 (2018). https://doi.org/10.13154/tches.v2018.i3.21-43, https://tches.iacr.org/index.php/TCHES/article/view/7267
3. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: Kaliski, Jr., B.S., Koç, Çetin Kaya., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Berlin, Heidelberg (Aug 2003). https://doi.org/10.1007/3-540-36400-5_3
4. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation **19**(90), 297–301 (1965). https://doi.org/10.1090/s0025-5718-1965-0178586-1
5. Hamburg, M., Hermelink, J., Primas, R., Samardjiska, S., Schamberger, T., Streit, S., Strieder, E., van Vredendaal, C.: Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. IACR TCHES **2021**(4), 88–113 (2021). https://doi.org/10.46586/tches.v2021.i4.88-113, https://tches.iacr.org/index.php/TCHES/article/view/9061
6. Han, J., Lee, T., Kwon, J., Lee, J., Kim, I.J., Cho, J., Han, D.G., Sim, B.Y.: Single-trace attack on nist round 3 candidate dilithium using machine learning-based profiling. IEEE Access **9**, 166283–166292 (2021). https://doi.org/10.1109/ACCESS.2021.3135600

7. Hermelink, J., Streit, S., Strieder, E., Thieme, K.: Adapting belief propagation to counter shuffling of NTTs. IACR TCHES **2023**(1), 60–88 (2023). https://doi.org/10.46586/tches.v2023.i1.60-88
8. Karabulut, E., Alkim, E., Aysu, A.: Single-trace side-channel attacks on ω-small polynomial sampling: With applications to NTRU, NTRU prime, and CRYSTALS-DILITHIUM. Cryptology ePrint Archive, Report 2022/494 (2022), https://eprint.iacr.org/2022/494
9. Krausz, M., Land, G., Richter-Brockmann, J., Güneysu, T.: A holistic approach towards side-channel secure fixed-weight polynomial sampling. In: Boldyreva, A., Kolesnikov, V. (eds.) PKC 2023, Part II. LNCS, vol. 13941, pp. 94–124. Springer, Cham (May 2023). https://doi.org/10.1007/978-3-031-31371-4_4
10. Mangard, S.: Hardware countermeasures against dpa – a statistical analysis of their effectiveness. In: Okamoto, T. (ed.) Topics in Cryptology – CT-RSA 2004. pp. 222–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
11. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: Revealing the secrets of smart cards. Springer (2007)
12. Module-Lattice-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 204, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.204
13. Peng, E.C.Y., Kuhn, M.G.: Adaptive template attacks on the kyber binomial sampler. IACR TCHES **2025**(3), 470–492 (2025). https://doi.org/10.46586/tches.v2025.i3.470-492
14. Pessl, P.: Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In: Dunkelman, O., Sanadhya, S.K. (eds.) INDOCRYPT 2016. LNCS, vol. 10095, pp. 153–170. Springer, Cham (Dec 2016). https://doi.org/10.1007/978-3-319-49890-4_9
15. Pessl, P., Primas, R.: More practical single-trace attacks on the number theoretic transform. In: Schwabe, P., Thériault, N. (eds.) LATINCRYPT 2019. LNCS, vol. 11774, pp. 130–149. Springer, Cham (Oct 2019). https://doi.org/10.1007/978-3-030-30530-7_7
16. Primas, R., Pessl, P., Mangard, S.: Single-trace side-channel attacks on masked lattice-based encryption. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 513–533. Springer, Cham (Sep 2017). https://doi.org/10.1007/978-3-319-66787-4_25
17. Ravi, P., Chattopadhyay, A., Baksi, A.: Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. Cryptology ePrint Archive, Report 2022/737 (2022), https://eprint.iacr.org/2022/737
18. Ravi, P., Jhanwar, M.P., Howe, J., Chattopadhyay, A., Bhasin, S.: Side-channel assisted existential forgery attack on Dilithium - A NIST PQC candidate. Cryptology ePrint Archive, Report 2018/821 (2018), https://eprint.iacr.org/2018/821
19. Ravi, P., Poussier, R., Bhasin, S., Chattopadhyay, A.: On configurable SCA countermeasures against single trace attacks for the NTT - A performance evaluation study over Kyber and Dilithium on the ARM Cortex-M4. Cryptology ePrint Archive, Report 2020/1038 (2020), https://eprint.iacr.org/2020/1038
20. Ravi, P., Poussier, R., Bhasin, S., Chattopadhyay, A.: On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT: A Performance Evaluation Study over Kyber and Dilithium on the ARM Cortex-M4, pp. 123–146. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-66626-2_7

21. Roy, S.S., Reparaz, O., Vercauteren, F., Verbauwhede, I.: Compact and side channel secure discrete Gaussian sampling. Cryptology ePrint Archive, Report 2014/591 (2014), https://eprint.iacr.org/2014/591
22. Sande, G., Gentlemen, W.M.: Fast Fourier Transforms-for Fun and Profit . In: Managing Requirements Knowledge, International Workshop on. vol. 1, p. 563. IEEE Computer Society (1996). https://doi.org/10.1109/AFIPS.1966.83, https://doi.ieeecomputersociety.org/10.1109/AFIPS.1966.83
23. Sendrier, N.: Secure sampling of constant-weight words – application to BIKE. Cryptology ePrint Archive, Report 2021/1631 (2021), https://eprint.iacr.org/2021/1631
24. Nist special publication 800-90a revision 1 - recommendation for random number generation using deterministic random bit generators. National Institute of Standards and Technology, U.S. Department of Commerce (Jun 2015). https://doi.org/10.6028/NIST.SP.800-90Ar1
25. Standaert, F.X., Archambeau, C.: Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 411–425. Springer, Berlin, Heidelberg (Aug 2008). https://doi.org/10.1007/978-3-540-85053-3_26
26. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against side-channel attacks: A comprehensive study with cautionary note. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 740–757. Springer, Berlin, Heidelberg (Dec 2012). https://doi.org/10.1007/978-3-642-34961-4_44