Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis

Kalle Ngo[1], Elena Dubrova[1] and Thomas Johansson[2]

[1] Royal Institute of Technology (KTH), Stockholm, Sweden
{kngo,dubrova}@kth.se
[2] Lund University, Lund, Sweden
thomas.johansson@eit.lth.se

Abstract. In this paper, we show that a software implementation of CCA secure Saber KEM protected by first-order masking and shuffling can be broken by deep learning-based power analysis. Using an ensemble of deep neural networks created at the profiling stage, we can recover the session key and the long-term secret key from $257 \times N$ and $24 \times 257 \times N$ traces, respectively, where N is the number of repetitions of the same measurement. The value of N depends on the implementation, environmental factors, acquisition noise, etc.; in our experiments $N = 15$ is enough to succeed. The neural networks are trained on a combination of 80% of traces from the profiling device with a known shuffling order and 20% of traces from the device under attack captured for all-0 and all-1 messages. "Spicing" the training set with traces from the device under attack helps minimize the negative effect of device variability.

Keywords: Public-key cryptography · post-quantum cryptography · Saber KEM · LWE/LWR-based KEM · side-channel attack · power analysis

1 Introduction

Public-key cryptographic schemes used today depend on the intractability of certain mathematical problems that are known to be efficiently solved with a large-scale quantum computer [Sho99]. Even if it will take many years until this is a reality, the need for long term security makes it urgent to investigate new solutions.

To address this need, the National Institute of Standards and Technology (NIST) started some years ago a project for standardizing post-quantum cryptographic primitives, NIST PQ. Candidate primitives rely on problems that are not known to be targets for a quantum computer, such as lattices problems and decoding problems in Hamming metric. In rounds 1 and 2, security and implementation aspects were the main focus in evaluation. The project has entered round 3, where security in relation to side-channel attacks is a main topic of investigation.

Previous Work: The first side-channel protected implementation of a lattice-based cryptosystem was proposed in [RRVV15] followed by [RdCR+16], based on masking. Masking involves doing linear operations twice, whereas non-linear operations needs more complex solutions decreasing the speed substantially. The implementation approach in [RRVV15] increases the number of CPU cycles on an ARM Cortex-M4 by a factor more than 5 compared to a standard implementation, see [BDK+20, p. 2].

These protected implementations focus on *Chosen-Plaintext Attack* (CPA)-secure lattice schemes, but more relevant are secure primitives designed to withstand *Chosen-Ciphertext Attacks* (CCA). CCA secure primitives are usually obtained from a CPA secure primitive using a transform, such as the *Fujisaki-Okamoto* (FO) transform or some variation of it [HHK17]. The CCA-transform is itself susceptible to side-channel attacks and should be protected [RRCB20]. Examples of recent masked implementations are: [OSPG18] of a KEM similar to NewHope; and [BBE+18, MGTF19, GR19] being lattice-based signature schemes.

For the NIST round 3 finalists, only the candidate Saber has a protected software implementation available [BDK+20]. It utilizes a first-order masking of the Saber CCA-secure decapsulation algorithm with an overhead factor of only 2.5 compared to an unmasked implementation.

Side-channel attacks on unprotected implementations of NIST PQ project candidates have appeared in some recent papers. In [SKL+20] a message recovery attack was described on the unprotected encapsulation part of some of the round 3 candidates; in [RRCB20] side-channel attacks on several round 2 candidates were described; in [XPRO] unprotected Kyber was attacked as a case study, using an electromagnetic (EM) side-channel approach. A way of turning a message recovery attack to a secret key recovery attack was proposed, using e.g. 184 traces for 98% success rate. In [UXT+21] another power/EM-based secret key recovery attack on some round 3 candidates KEMs was presented. In [GJN20] similar ideas were used for timing attacks.

In [RBRC20], the authors improve the key recovery attacks on unprotected implementations of three NIST PQ finalists, including Saber. They also discuss how to attack masked implementations by attacking shares individually. However, no actual attack on masked Saber is performed. Finally, in [NDGJ21] a higher-order side-channel attack on a the masked implementation of the CCA secure Saber KEM is demonstrated. It recovers both the session key and the long-term secret key using a deep neural network trained at the profiling stage. The secret key recovery attack requires 24 traces.

Contributions: In this paper, we present the first side-channel attack on a masked and shuffled implementation of CCA secure Saber KEM. Until now, these countermeasures combined together were believed to provide an adequate protection against power and EM analysis.

We show how to recover the session key and the long-term secret key by deep learning-based power analysis from $257 \times N$ and $24 \times 257 \times N$ traces, respectively, captured using the execution of the decapsulation algorithm, where N is the number of repetitions of the same measurement. The value of N depends on the implementation, environmental factors, acquisition noise, etc.; in our experiments, $N = 15$ is enough to succeed. Following the method presented in [NDGJ21], our deep neural networks learn a higher-order model directly, without explicitly extracting random masks at each execution. However, since we attack an implementation in which the message bits are shuffled, it is not possible to directly recover the message from a signed trace, as in [NDGJ21]. Only the message Hamming weight (HW) can be derived. To find the order of message bits, traces for 256 additional decapsulations have to be captured and analyzed for each chosen chiphertext (hence ×257).

We quantify the success rate of the message HW recovery as a function of the success rate of a single message bit recovery and show that the latter should be of the order of 0.999 to recover the message HW with a high probability. To increase the success rate of a single message bit recovery, we introduce a novel approach for training neural networks which uses a combination of traces from the profiling device with a known shuffling order and traces from the device under attack captured for all-0 and all-1 messages. We also use an ensemble of models to increase the success rate of message recovery from the derived HWs.

The remainder of this paper is organized as follows. Section 2 gives the necessary background on Saber algorithm and profiled side-channel attacks. Section 3 describes the implementation of masked and shuffled Saber KEM which is used in our experiments. Section 4 presents equipment for trace acquisition. Section 5 shows how points of interest

```
Saber.PKE.KeyGen()
    $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
    $\mathbf{A} = \mathsf{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
    $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
    $\mathbf{s} \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
    $\mathbf{b} = ((\mathbf{A}^T \mathbf{s} + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
    return $(pk \doteq (seed_{\mathbf{A}}, \mathbf{b}), sk \doteq \mathbf{s})$

Saber.PKE.Dec($\mathbf{s}, (\boldsymbol{c_m}, \mathbf{b}')$)
    $v = \mathbf{b}'^T(\mathbf{s} \bmod p) \in R_p$
    $m' = ((v + h_2 - 2^{\epsilon_p - \epsilon_T} \boldsymbol{c_m}) \bmod p) \gg (\epsilon_p - 1) \in R_2$
    return $m'$

Saber.PKE.Enc($(seed_{\mathbf{A}}, \mathbf{b}), m; r$)
    $\mathbf{A} = \mathsf{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
    if r is not specified then
        $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
    end if
    $\mathbf{s}' \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
    $\mathbf{b}' = ((\mathbf{A}\mathbf{s}' + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
    $v' = \mathbf{b}^T(\mathbf{s}' \bmod p) \in R_p$
    $\boldsymbol{c_m} = ((v' + h_1 - 2^{\epsilon_p - 1} m) \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
    return $(c = (\boldsymbol{c_m}, \mathbf{b}'))$
```

Figure 1: Description of Saber.PKE from [D+20].

```
Saber.KEM.KeyGen()
    $(seed_{\mathbf{A}}, \mathbf{b}, \mathbf{s})$ = Saber.PKE.KeyGen()
    $pk = (seed_{\mathbf{A}}, \mathbf{b})$
    $pkh = \mathcal{F}(pk)$
    $z \leftarrow \mathcal{U}(\{0,1\}^{256})$
    return $(pk = (seed_{\mathbf{A}}, \mathbf{b}), sk = (z, pkh, pk, \mathbf{s}))$

Saber.KEM.Encaps($(seed_{\mathbf{A}}, \mathbf{b})$)
    $m \leftarrow \mathcal{U}(\{0,1\}^{256})$
    $(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
    $c$ = Saber.PKE.Enc($pk, m; r$)
    $K = \mathcal{H}(\hat{K}, c)$
    return $(c, K)$

Saber.KEM.Decaps($(z, pkh, pk, \mathbf{s})$,c)
    $m'$ = Saber.PKE.Dec($\mathbf{s}, c$)
    $(\hat{K}', r') = \mathcal{G}(pkh, m')$
    $c'$ = Saber.PKE.Enc($pk, m'; r'$)
    if $c = c'$ then
        return $K = \mathcal{H}(\hat{K}', c)$
    else
        return $K = \mathcal{H}(z, c)$
    end if
```

Figure 2: Description of Saber.KEM from [D+20].

are located in side-channel measurements. Sections 6 and 7 describe the profiling and the attack stages, respectively. Section 8 summarizes the experimental results. Section 9 concludes the paper and describes future work.

2 Background

This section briefly describes Saber and profiled side-channel attacks. A more detailed description of Saber can be found in [D+20].

2.1 Saber design description

Saber is a package of cryptographic algorithms whose security relies on the hardness of the Module Learning With Rounding problem (Mod-LWR) [D+20]. It contains a CPA-secure public key encryption scheme, Saber.PKE, and a CCA-secure key encapsulation mechanism, Saber.KEM, based on a post-quantum version of the Fujisaki-Okamoto transform.

Pseudocodes of Saber.KEM and Saber.PKE are shown in Fig. 1 and 2, respectively. We follow the notation of [NDGJ21].

Let $\mathbb{Z}_q$ be the ring of integers modulo q and R_q be the quotient ring $\mathbb{Z}_q[X]/(X^n + 1)$. The rank of the module is denoted by l. The rounding modulus is denoted by p.

The notation $x \leftarrow \chi(S)$ stands for denote sampling x according to a distribution χ over a set S. The uniform distribution is denoted by $\mathcal{U}$. The centered binomial distribution with parameter μ is denoted by β_μ, where μ is an even positive integer. The term $\beta_\mu(R_q^{l \times k}; r)$

generates a matrix in $R_q^{l \times k}$ where the coefficients of polynomials in R_q are sampled in a deterministic manner from β_μ using seed r.

The functions $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$ are SHA3-256, SHA3-512 and SHA3-256 hash functions, respectively. The gen is an extendable output function which is used to generate a pseudorandom matrix $\mathbf{A} \in R_q^{l \times l}$ from $seed_{\mathbf{A}}$. It is instantiated with SHAKE-128.

The bitwise right shift operation is denoted by "$\gg$". It is extendable to polynomials and matrices by performing the shift coefficient-wise. To allow for an efficient implementation, Saber design uses power of two moduli q, p, and T, namely $q = 2^{\epsilon_q}$, $p = 2^{\epsilon_p}$, and $T = 2^{\epsilon_T}$. In order to implement rounding operations by a simple bit shift, three constants are used: polynomials $h_1 \in R_q$ and $h_2 \in R_q$ with all coefficients being $2^{\epsilon_q - \epsilon_p - 1}$ and $2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_T - 1} + 2^{\epsilon_q - \epsilon_p - 1}$, respectively, and a constant vector $\mathbf{h} \in R_q^{l \times 1}$ in which each polynomial is equal to h_1.

In the round 3 Saber document [D+20], three sets of parameters are proposed for the security levels of NIST-I, NIST-III, and NIST-V: LightSaber, Saber and FireSaber, respectively. All results presented in this paper are for Saber, but it is trivial to extend them to the other versions. Saber uses $n = 256$, $l = 3$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^4$, and $\mu = 8$. Its decryption failure probability is bounded by 2^{-136}.

2.2 Profiled side-channel attacks

Side-channel attacks can be carried out in two settings: profiled and non-profiled. *Profiled* attacks first learn a leakage profile of the targeted cryptographic algorithm's implementation using a device similar to the device under attack, called *profiling device*. The profiling can be done by creating a template [APSQ06, CPM+18, HGA+19], or training a neural network model [MPP16, CDP17, KPH+19, BFD20]. Then, the resulting template/model is used to recover the secret variable, e.g. the key, from the device under attack [MPP16]. *Non-profiled* attacks attack directly [Tim18].

Profiled side-channel attacks typically assume that:

1. The attacker has at least one profiling device similar to the device under attack which runs the same implementation.
2. The attacker has a full control over the profiling device.
3. The attacker has a direct physical access to the device under attack to measure side-channel signals for chosen inputs.

3 Implementation of masked and shuffled Saber

All experiments presented in this paper are performed on a masked and shuffled implementation of Saber which we created ourselves. To the best of our knowledge, no implementation protected by both countermeasures are available at present.

We used the masked implementation of Saber presented in [BDK+20] as a base and added shuffling on the top as described below.

3.1 Masking

Masking is a well-known countermeasure against power/EM analysis [CJRR99]. *First-order* masking protects against attacks leveraging information in the first-order statistical moment. A first-order masking partitions any sensitive variable x into two shares, x_1 and x_2, such that $x = x_1 \circ x_2$, and executes all operations separately on the shares. The operator "$\circ$" depends on the type of masking, e.g. it is "$+$" is arithmetic masking and "$\oplus$" is Boolean masking.

Carrying out operations on the shares x_1 and x_2 prevents leakage of side-channel information related to x as computations do not explicitly involve x. Instead, x_1 and x_2 are linked to the leakage. Since the shares are randomized at each execution of the algorithm, they are not expected to contain exploitable information about x. The randomization is usually done by assigning a random mask r to one share and computing the other share as $x - r$ for arithmetic masking or $x \oplus r$ for Boolean masking.

A challenge in masking lattice-based cryptosystems is the integration of bit-wise operations with arithmetic masking which requires methods for secure conversion between masked representations. Saber can be efficiently masked due to specific features of its design: power-of-two modulo q, p and T, and limited noise sampling of LWR. Due to the former, modular reductions are basically free. The latter implies that only the secret key $\mathbf{s}$ has to be sampled securely. In contrast, LWE-based schemes also need to securely sample two additional error vectors.

Masking duplicates most linear operations, but requires more complex routines for non-linear operations. The first-order masked implementation of Saber presented [BDK+20] uses a custom primitive for masked logical shifting on arithmetic shares, called `poly_A2A()`, and an adapted masked binomial sampler from [SPOG19]. Particular attention is devoted in [BDK+20] to the protection of the decapsulation algorithm since it involves operations with the long-term secret key $\mathbf{s}$. At its first step (see Saber.KEM.Decaps() in Fig. 2) the decapsulation algorithm calls Saber.PKE.Dec() to decrypt the input ciphertext c. Fig. 3 shows the implementation of Saber.PKE.Dec() from [BDK+20] called `indcpa_kem_dec_masked()`.

3.2 Shuffling

Shuffling is another well-known countermeasure against power/EM analysis. We use the modernized version of the Fisher-Yates (FY) algorithm [Dur64] which generates a random permutation of a finite sequence. The generated sequence is used as the loop iterator to index the inner loop function's data processing. This effectively scrambles the order in which the elements of an array are processed as opposed the linear sequence of a non-shuffled loop. Shuffling makes power analysis and neural network training significantly more difficult as this removes the linear correlation of index sequence with time.

Figure 3 shows our masked and shuffled implementation of Saber.PKE.Dec() called `indcpa_kem_dec_masked_and_shuffled()`.

We implement bitwise shuffling of a 256-bit message in the primitive `poly_A2A()` by calling the `FY_Gen()` function to randomly permute a list of the same length (see `poly_A2A_shuffled()`). The shuffled values (in the range from 0 to 255) are then subsequently referenced at the start of every loop iteration, resulting in randomized execution order.

In a similar way, we implement bytewise shuffling of a message in the procedure `POL2MSG()` (see `POL2MSG_shuffled()`) by calling the `FY_Gen()` to randomly permute a list of the length equal to the number of bytes, 32.

3.3 Known vulnerabilities

In previous work, a number of vulnerabilities were discovered in the non-masked LWE/LWR-based PKE/KEMs [ACLZ20, SKL+20, RRCB20, RBRC20]. One is Incremental-Storage vulnerability resulting from an incremental update of the decrypted message in memory during message decoding [RBRC20]. The decoding function iteratively maps each polynomial coefficient into a corresponding message bit, thus computing the decrypted message one bit at a time.

It was further observed in [RBRC20] that a non-masked implementations of the decoding function contains two points with exploitable Incremental-Storage vulnerability. The first

```
void indcpa_kem_dec_masked(uint16_t
sksv1[], uint16_t sksv2[], char *ct,
char m1[], char m2[])
uint16_t pksv[K][N];
uint16_t v1[N]={0}, v2[N]={0};

SABER_un_pack(&ct,v1);
for (i = 0; i < N; i++) do
  v1[i] = h2-(v1[i]≪(EP-ET));
end for
BS2POLVEC(ct,pksv,P);
InnerProd(pksv,sksv1,P-1,v1);
InnerProd(pksv,sksv2,P-1,v2);
poly_A2A(v1,v2);
POL2MSG(v1,m1);
POL2MSG(v2,m2);
```

```
void poly_A2A(uint16_t A[N], uint16_t
R[N])
uint32_t A, R;

for (i = 0; i < N; i++) do
  A = A[i]; R = R[i];
   ... /* processing */
  A[i] = A; R[i] = R;
end for
```

```
void FY_Gen(uint8_t* fylist, int max)

for (i = 0; i < max; i++) do
  fylist[i] = i
end for
for (i = max-1; i >= 0; i--) do
  int index = rand() % max;
  uint8_t temp = fylist[index];
  fylist[index] = fylist[i];
  fylist[i] = temp;
end for
```

```
void indcpa_kem_dec_masked_and_shuffled
(uint16_t sksv1[], uint16_t sksv2[],
char *ct, char m1[], char m2[])
uint16_t pksv[K][N];
uint16_t v1[N]={0}, v2[N]={0};

SABER_un_pack(&ct,v1);
for (i = 0; i < N; i++) do
  v1[i] = h2-(v1[i]≪(EP-ET));
end for
BS2POLVEC(ct,pksv,P);
InnerProd(pksv,sksv1,P-1,v1);
InnerProd(pksv,sksv2,P-1,v2);
poly_A2A_shuffled(v1,v2);
POL2MSG_shuffled(v1,m1);
POL2MSG_shuffled(v2,m2);
```

```
void poly_A2A_shuffled(uint16_t A[N],
uint16_t R[N])
uint8_t fylist[256];
uint32_t A, R;

FY_Gen(fylist, 256);
for (i = 0; i < N; i++) do
  y = fylist[i]
  A = A[y]; R = R[y];
   ... /* processing */
  A[y] = A; R[y] = R;
end for
```

```
void POL2MSG_shuffled(uint16_t *v, chair
*m)
uint8_t fylist[32];
FY_Gen(fylist, 32);
for (j = 0; j < BYTES; j++) do
  y = fylist[j]
  m[y] = 0;
  for (i = 0; i < 8; i++) do
    m[y] = m[y]|(v[8*y+i]≪i);
  end for
end for
```

Figure 3: The masked implementation of Saber.PKE.Dec() from [BDK+20] (left) and the presented masked and shuffled implementation of Saber.PKE.Dec() (right).

one is where the message bits are computed and stored in a 16-bit memory location in an unpacked fashion. Since the memory location can take only two possible values, 0 or 1, an attacker can recover the message bit by distinguishing between 0 and 1. The second point is in `POL2MSG()` procedure where the decoded message bits are packed into a byte array in memory.

In [NDGJ21] it was demonstrated that, despite partitioning the message into two shares in a first-order masked implementation of Saber, the leakage point in `POL2MSG()` procedure can still be exploited. In addition, a new the leakage point in `poly_A2A()` procedure was discovered (highlighted in red in Figure 3). The attacks presented in this paper are based on the corresponding point in `poly_A2A_shuffled()`) (highlighted in red in Figure 3).

4 Equipment for trace acquisition

The equipment we use for trace acquisition consists of the ChipWhis- perer-Lite board, the CW308 UFO board and two CW308T-STM32F4 target boards.

The ChipWhisperer is a hardware security evaluation toolkit based on a low-cost open hardware platform and an open-source software [New]. It can be used to measure power consumption and to make communication between the target device and the computer easier. Power is measured over a shunt resistor connected between the power supply and the target device. ChipWhisperer-Lite employs a synchronous capture method, which greatly improves trace synchronization while also lowering the required sample rate and data storage.

The CW308 UFO board is a general-purpose platform for evaluating multiple targets [CW3]. The target board is plugged into a dedicated U connector.

The target board CW308T-STM32F4 contains a 32-bit ARM Cortex-M4 CPU with STM32F415-RGT6 device. The board operates at 24 MHz and it is sampled at 24 MHz, i.e. 1 point per clock cycle.

In our experiments, the Cortex-M4 is programmed with the masked and shuffled Saber implementation described in the previous section. The implementation is compiled with `arm-none-eabi-gcc` at the highest level of compiler optimization -O3 (recommended default) which is typically the most difficult to break by side-channel analysis [SKL$^+$20].

5 Locating points of interest

The attacks on unprotected implementations of LWE/LWR-based KEMs [RRCB20, SKL$^+$20] typically locate leakage points in side-channel measurements using techniques such as Test Vector Leakage Assessment (TVLA) [GJJR11], or Correlation Power Analysis (CPA). However, such a method is not applicable to a protected implementation since masked implementations change random masks for each execution and shuffled implementations change shuffling order for each execution.

In this section, we describe our method for locating points of interest in a masked and shuffled implementation of Saber. Fig. 4(a) shows a power trace obtained by averaging 50K measurement made during the execution of Saber.KEM.Decaps() for random ciphertexts. We can clearly see different blocks with different structure. Our aim is `poly_A2A()` procedure which processes 256 message bits one-by-one. The segment of Fig. 4(a) marked by two red lines is a possible candidate. By zooming in, see Fig. 4(b) and (c), one can verify that the number of repeating peaks is indeed 256.

By measuring the distance between the peaks, we can find that the processing of one bit by `poly_A2A()` takes 51 points. This parameter is referred to as *bit_offset* in the sequel. Since for `poly_A2A()` the shares $A[i]$ and $R[i]$ are processed immediately following each other (see line 4 of `poly_A2A()` in Fig. 3), *bit_offset* contains both shares.

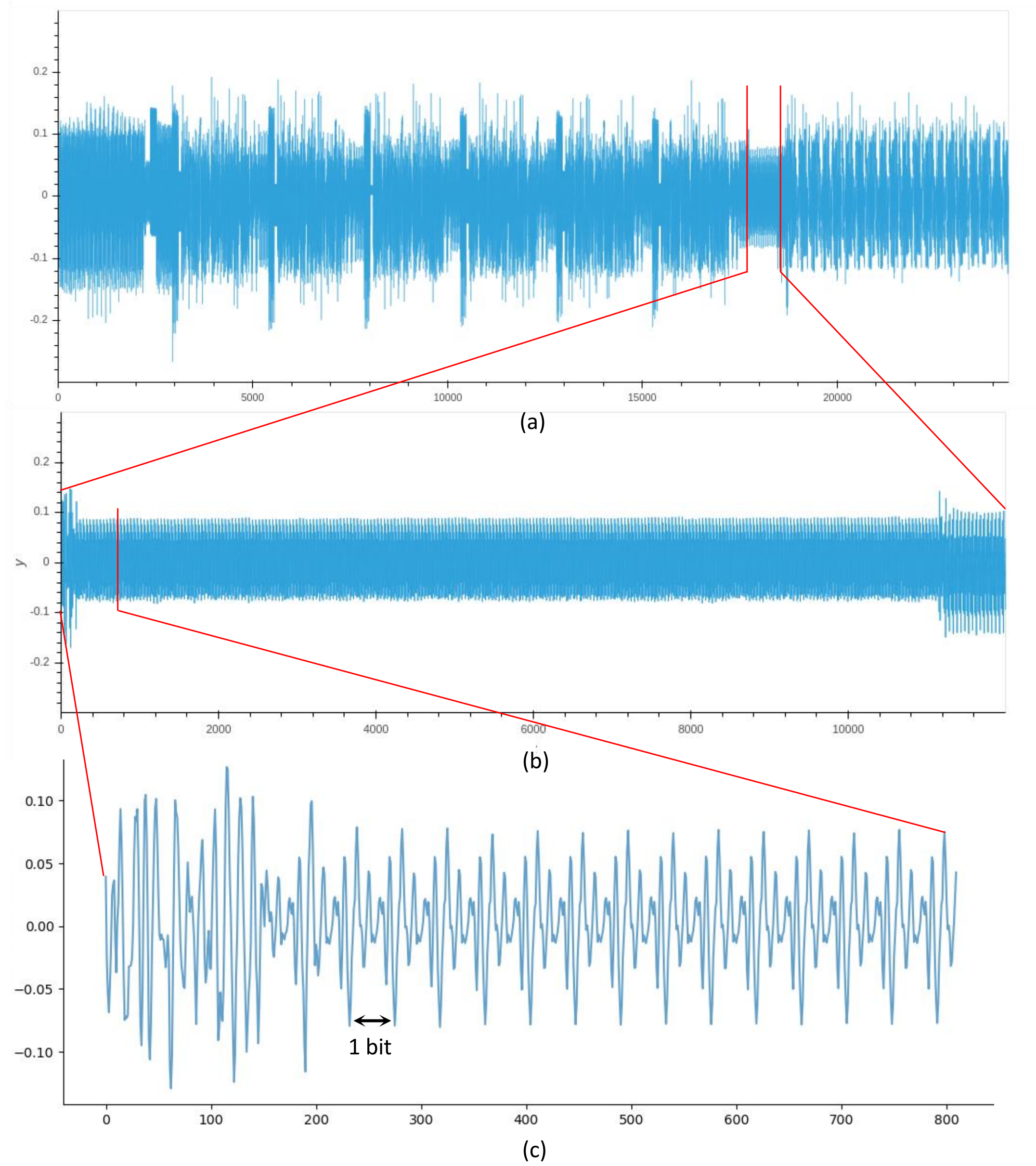

Figure 4: (a) A power trace representing the execution of the first step of Saber.KEM.Decaps() (average of 50K measurements sampled with decimation 15); (b) A detailed view of `poly_A2A(v1,v2)` (sampled with decimation 1); (d) The first 15 bits of `poly_A2A(v1,v2)`.

By locating the first peak, we can find the starting point of `poly_A2A()` procedure. This parameter is referred to as *offset*. Note that we do not need to know neither the value of a random mask, nor the shuffling order to compute the *offset* and *bit_offset*.

Table 1: The MLP architecture.

Layer type	(Input, output) shape	# Parameters
Batch Normalization 1	(90, 90)	360
Dense 1	(90, 128)	11648
Batch Normalization 2	(128, 128)	512
ReLU	(p128, 128)	0
Dense 2	(128, 32)	4128
Batch Normalization 2	(32, 32)	128
ReLU	(32, 32)	0
Dense 3	(32, 16)	528
Batch Normalization 2	(16, 16)	64
ReLU	(16, 16)	0
Dense 4	(16, 1)	17
Softmax	(1, 1)	0
Total parameters:	17,385	
Trainable parameters:	16,853	

6 Profiling stage

The aim of profiling is to construct a neural network model capable of distinguishing between the message bit values ‘0’ and ‘1’. At the attack stage, we use this model to count the number of ‘1’s in the message is order to determine its HW.

We use neural networks with a multilayer perceptron (MLP) architecture shown in Table 1. It is the same as the one in [NDGJ21] except for the input size. This architecture was selected using the grid search algorithm [GBC16] which trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter.

During training, we use *Nadam* optimizer, which is an extension of RMSprop with Nesterov momentum, with a learning rate of 0.001 and a numerical stability constant epsilon=1e-08. Binary cross-entropy is used as a loss function. The training is run for a maximum of 100 epochs, with a batch size of 128 and an early stopping. 70% of the training set is used for training, and 30% is used for validation.

Unlike [NDGJ21] where eight models were trained, one for each bit position of a byte, we train a single model capable of recovering all message bits. This is accomplished by composing the training set as a union of trace intervals corresponding to individual bit processing. As a result, we get a universal model which has “learned” features for all 256 bits. Using a cut-and-join technique like this, we can increase the size of the training set by a factor of 256 without having to capture 256 times as many traces. For example, the 2M training set used in our experiments is composed from 7.8K captured traces. On an ARM Cortex-M4 running at 24MHz, it takes less than 17 minutes to capture the latter and 3 days to capture the former.

The cut-and-join technique is applicable to `poly_A2A()` leakage point because `poly_A2A()` procedure processes all message bits in the same way during their storage in memory. Thus, traces representing the execution of `poly_A2A()` appear identical for all message bits except the first and last, as we can see from Fig 5. Because of the Cortex-M4’s three-stage pipeline, the next instruction begins before the previous instruction has finished. As a result, the power consumed during the processing of the first and the last bits differs from the power consumed during the processing of other bits.

Similarly to [NDGJ21], we defeat masking by training models on traces containing the bits of both shares labelled by the value of the corresponding message bit. Thus, the models are capable of recovering the message bits directly, without explicitly extracting the mask. However, since the message bits are also shuffled in our case, we cannot train on

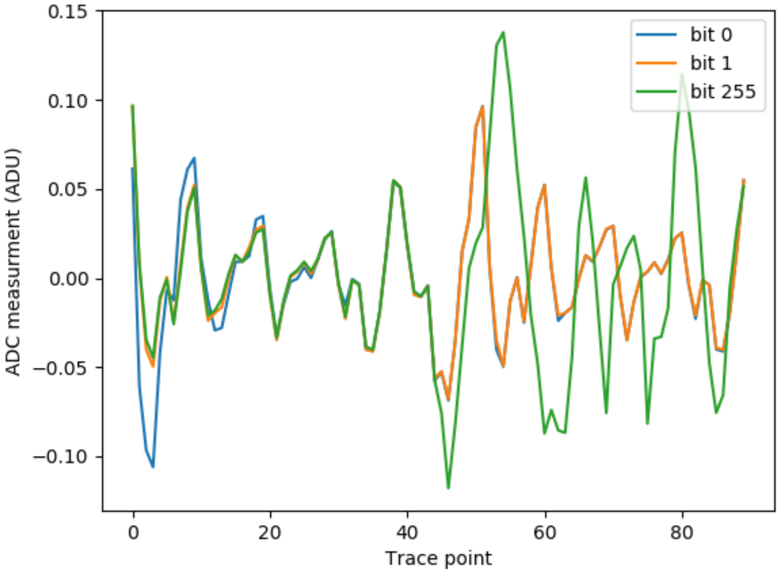

Figure 5: Power traces depicting the precessing of message bits 0, 1 and 255 by `poly_A2A()` (average of 10K measurements). The traces for the remaining bits have the same shape as bit 1.

traces captured from the device under attack for random messages, as in [NDGJ21] because the order of bits (and thus training labels) is unknown. Instead, we train on a combination of traces from the profiling device running an implementation with deactivated shuffling, and traces from the device under attack captured for all-0 and all-1 messages. Obviously, the labels of all bits are the same for all-0 and all-1 messages. However, as we show in the experimental results section, training on only all-0 and all-1 messages does not produce good results. Traces of all-0 and all-1 messages do not allow the neural network to learn all possible features due to the above-mentioned impact of the previous and next instructions on power consumption.

The pseudocode of the profiling algorithm is shown in Fig. 6. TrainModel() takes as input the number of traces to be captured, τ, the neural network's input size, *in_size*, and a parameter $k \in \mathbb{I}$, $\mathbb{I} = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$, which defines which fraction of traces is captured from the profiling device, D_p. For example, $k = 0.8$ means that 80% of traces are from D_p. The rest of traces is captured from the device under attack, D_a, for all-0 and all-1 messages in equal parts, $r = (1 - k)/2$.

At step 1, ComposeTrainingSet() procedure is called to create a set of training traces, $\boldsymbol{\mathcal{T}}$, and the corresponding set of labels, $\boldsymbol{\mathcal{L}}$. In ComposeTrainingSet(), $k{*}\tau$ messages are selected at random and encrypted by a fixed public key[1]. The profiling device D_p, which is running an implementation with deactivated shuffling, is used to decapsulate the resulting set of ciphertexts. During its execution, the power traces are captured.

Similarly, $\tau{*}r$ all-0 and $\tau{*}r$ all-1 messages are generated and encrypted. The device under attack D_a is used to decapsulate the resulting ciphertexts, and the power traces are captured (step 4-8).

Next, the initial offset, *offset*, and the distance between the message bits in $\boldsymbol{\mathcal{T}'}$, *bit_offset*, are determined as described in Section 5. Finally, the cut-and-join technique is used to divide $\boldsymbol{\mathcal{T}'}$ into intervals representing individual message bit processing and to

[1] It is also possible to train with different keys. This does not affect the outcome.

```
TrainModel($\tau$, $in\_size$, $k$)
$(\boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{L}})$ = ComposeTrainingSet($\tau$, $in\_size$, $k$)
Train $\mathcal{NN} : \mathbb{R}^{in\_size} \rightarrow \mathbb{I}$ on $(\boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{L}})$
return $\mathcal{NN}$

ComposeTrainingSet($\tau$, $in\_size, k$)
$\boldsymbol{m} = \emptyset, \boldsymbol{\mathcal{T}'} = \emptyset$
$\boldsymbol{M} = \{0,1\}^{256}$
$(\boldsymbol{m}, \boldsymbol{\mathcal{T}'})$ = CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \boldsymbol{\mathcal{T}'}, D_p, 1, k{*}\tau$)
$r = (1-k)/2$
$\boldsymbol{M} = \{0\}^{256}$
$(\boldsymbol{m}, \boldsymbol{\mathcal{T}'})$ = CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \boldsymbol{\mathcal{T}'}, D_a, k{*}\tau + 1, r{*}\tau$)
$\boldsymbol{M} = \{1\}^{256}$
$(\boldsymbol{m}, \boldsymbol{\mathcal{T}'})$ = CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \boldsymbol{\mathcal{T}'}, D_a, (k+r){*}\tau + 1, r{*}\tau$)
Determine initial $offset$ and $bit\_offset$ from $\boldsymbol{\mathcal{T}'}$
$\boldsymbol{\mathcal{T}} = \emptyset,\ \boldsymbol{\mathcal{L}} = \emptyset$
for each $b \in \{0, 1, \ldots, 255\}$ do
    $start$ = $offset$ + $b{*}bit\_offset$
    $stop$ = $start$ + $in\_size$
    $\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{T}} \cup \boldsymbol{\mathcal{T}'}[:, start\!:\!stop]$
    $\boldsymbol{\mathcal{L}} = \boldsymbol{\mathcal{L}} \cup \{\ l(\mathcal{T}_i) \in \{0,1\} \mid l(\mathcal{T}_i) = m_i[b], \forall i \in \{1, \ldots, \tau\}\}$
end for
return $(\boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{L}})$

CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \boldsymbol{\mathcal{T}'}, D, a, b$)
for each $i \in \{a, a+1, \ldots, a+b\}$ do
    $m_i \leftarrow \mathcal{U}(\boldsymbol{M})$
    $\boldsymbol{m} = \boldsymbol{m} \cup \{m_i\}$
    $r_i \leftarrow \mathcal{U}(\{0,1\}^{256})$
    $pk_i \leftarrow \mathcal{U}(\{0,1\}^{256})$
    $c_i$ = Saber.PKE.Enc($pk_i, m_i; r_i$)
    $\mathcal{T}_i \Leftarrow D$[Saber.KEM.Decaps($c_i$)]
    $\boldsymbol{\mathcal{T}'} = \boldsymbol{\mathcal{T}'} \cup \{\mathcal{T}_i\}$
end for
return $(\boldsymbol{m}, \boldsymbol{\mathcal{T}'})$
```

Figure 6: Profiling algorithm.

Table 2: Pairs (k_1, k_0) which are used to derive secret key coefficients $\mathbf{s}[i]$ from eight message bits [NDGJ21].

$\mathbf{s}[i]$	The message bit value for the pair (k_1, k_0)							
	(186,0)	(293,7)	(311,7)	(615,2)	(613,2)	(890,4)	(903,4)	(199,0)
-4	0	1	1	1	1	0	0	0
-3	1	1	1	0	0	0	0	1
-2	1	0	0	1	1	0	0	1
-1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	0
1	0	0	0	1	1	1	1	0
2	1	0	0	0	0	1	1	1
3	1	1	1	1	1	1	1	1
4	1	1	0	1	0	0	1	0

generate the set of labels $\mathcal{L}$ containing the corresponding message bit values.

7 Attack Stage

To defeat the combined masked and shuffled countermeasures, we make use of the existing key and message recovery techniques presented in [NDGJ21] and [RBRC20] for masked-only and shuffled-only LWE/LWR-based KEMs, respectively, and introduce two new algorithms.

In this section, we outline the main steps of the proposed secret and session key recovery approaches, then describe the key and message recovery techniques from from [NDGJ21] and [RBRC20], and finally present the new algorithms.

7.1 Main steps

Secret key recovery:

1. Construct 24 chosen ciphertexts $c_1, \ldots, c_{24}$ as described in Section 7.2.
2. For each c_i, $i \in \{1, \ldots, 24\}$, construct 256 ciphertexts $c_{i_0}, \ldots, c_{i_{255}}$ such that c_{i_j} decrypts to $m'_{i_j} = \text{Saber.PKE.Dec}(\mathbf{s}, c_{i_j})$ which is equal to the message $m'_i = \text{Saber.PKE.Dec}(\mathbf{s}, c_i)$ with the jth bit is flipped, for $j \in \{0, \ldots, 255\}$. The procedure is described in Section 7.3.
3. For each of 24×257 resulting ciphertexts, acquire a power trace during the decapsulation of the ciphertext by the device under attack. Repeat N times for each ciphertext.
4. Use the acquired $24 \times 257 \times N$ power traces to recover the messages m'_i contained in the ciphertexts c_i, for all $i \in \{1, \ldots, 24\}$, using RecoverMessage() algorithm presented in Section 7.4.
5. Derive the secret key from the 24 recovered messages $m'_1, \ldots, m'_{24}$ as described in Section 7.2.

Session key recovery: Assume that the adversary has a properly generated ciphertext c which is decapsulated by the device under attack. The adversary follows the steps (2)-(5) of the secret key recovery algorithm described above to extract the message m' contained in c from $257 \times N$ power traces. Given m', he/she computes $(\hat{K}', r') = \mathcal{G}(pkh, m')$ and gets the session key as $K = \mathcal{H}(\hat{K}', c)$.

7.2 Chosen ciphertext construction

In [NDGJ21] an approach based on error-correcting codes (ECC) was introduced to recover the secret key from masked Saber. We use the same chosen ciphertexts as in [NDGJ21] for recovering the secret key from masked and shuffled Saber.

The ciphertexts are constructed as $c_j = (\boldsymbol{c_m}, \mathbf{b}')$ where $\boldsymbol{c_m} = k_0 \sum_{i=0}^{255} x^i \in R_T$ and

$$\mathbf{b}' = \begin{cases} (k_1, 0, 0) \in R_p^{3\times 1} & \text{for} \quad j = \{1, \ldots, 8\}, \\ (0, k_1, 0) \in R_p^{3\times 1} & \text{for} \quad j = \{9, \ldots, 16\}, \\ (0, 0, k_1) \in R_p^{3\times 1} & \text{for} \quad j = \{17, \ldots, 24\}, \end{cases}$$

where the pairs (k_0, k_1) are listed in Table 2. In this table, the ith coefficient of the secret key $\mathbf{s}$, $\mathbf{s}[i]$, is mapped into a codeword of the $[8, 4, 4]_2$ extended Hamming code composed from the eight message bits. The first 256 secret key coefficients are derived from messages recovered from $c_1, \ldots, c_8$, the second 256 coefficients - from $c_9, \ldots, c_{16}$, and the last 256 coefficients - from $c_{17}, \ldots, c_{24}$.

The approach in [NDGJ21] works because decryption of $(\boldsymbol{c_m}, \mathbf{b}')$ yields the message

$$m' = ((\mathbf{b}'^T(\mathbf{s} \mod p) + h_2 - 2^{\epsilon_p - \epsilon_T}\boldsymbol{c_m}) \mod p) \gg (\epsilon_p - 1) \in R_2,$$

whose ith bit, $m'[i]$, is a function of the triple $(k_0, k_1, \mathbf{s}[i])$:

$$m'[i] = ((k_1 \cdot (\mathbf{s}[i] \mod p) + H - 2^{\epsilon_p - \epsilon_T} k_0) \mod p) \gg (\epsilon_p - 1), \tag{1}$$

where $H = 2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_T - 1} + 2^{\epsilon_q - \epsilon_p - 1}$. Thus, $m'[i]$ leaks information about $\mathbf{s}[i]$.

7.3 Bit-flip technique

In [RBRC20] a technique called *bit-flip* was introduced to recover the message m' contained in ciphertext c which is decapsulated by the device under attack implementing a shuffled LWE/LWR-based KEM algorithm. We use a "fuzzy" version of this technique, presented in Section 7.4, for recovering messages contained in 24 chosen ciphertexts which are decapsulated by the device under attack implementing masked and shuffled Saber.

Given a ciphertext $c = (\boldsymbol{c_m}, \mathbf{b}')$, the bit-flip technique [RBRC20] constructs 256 ciphertexts c_j, $j \in \{0, \ldots, 255\}$, in which the value of the center of the integer ring $\mathbb{Z}_q$ is subtracted from the jth coefficient of $\boldsymbol{c_m}$. Since the message polynomial is only additively hidden within the ciphertext, this results in a ciphertext decrypting m'_j which is equal $m' = \text{Saber.PKE.Dec}(\mathbf{s}, c)$ with jth bit flipped.

For c and each c_j, a side-channel HW classifier is applied to find the HW of m' and each m'_j, for $j \in \{0, \ldots, 255\}$. In [RBRC20], the HW classifier is constructed by the template approach. From the obtained HWs, the message m' is recovered bit-by-bit as follows:

$$m'[i] = \begin{cases} 0 & \text{if} \quad HW(m'_j[i]) = HW(m'[i]) + 1 \\ 1 & \text{if} \quad HW(m'_j[i]) = HW(m'[i]) - 1. \end{cases} \tag{2}$$

7.4 Message HW recovery algorithm

In this section, we present the algorithm RecoverHW() which we use to recover HW of messages contained in 24 chosen ciphertexts and their bit-flipped versions. Its pseudo-code is shown in Fig. 7.

RecoverHW() takes as input the neural network trained at the profiling stage, $\mathcal{NN}$, the neural network's input size, *in_size*, the initial offset, *offset*, the distance between the bits, *bit_offset*, the ciphertext c for which the message HW has to be recovered, and the degree of repetition of the same measurement, N.

```
RecoverHW($\mathcal{NN}$,$in\_size$, $offset$, $bit\_offset$, $c$, $N$)
 1: for each $i \in \{1, \dots, N\}$ do
 2:     $\hat{\mathcal{T}}_i \Leftarrow D_a$[Saber.KEM.Decaps($c$)]
 3:     $HW_i = 0$
 4:     for each $b \in \{0, \dots, 255\}$ do
 5:         $start = offset + b * bit\_offset$
 6:         $stop = start + in\_size$
 7:         $s_b = \mathcal{NN}(\hat{\mathcal{T}}[start:stop])$
 8:         if $s_b > 0.5$ then
 9:             $HW_i = HW_i + 1$
10:         end if
11:     end for
12: end for
13: $N'$ = RemoveOutliers($HW_1, \dots, HW_N$)
14: $HW$ = Median($HW_1, \dots, HW_{N'}$)
15: return $HW$
```

Figure 7: Message HW recovery algorithm.

First, N trials are performed to recover the HW of the message m' contained in c. The device under attack is used to decapsulate c, and a power trace $\hat{\mathcal{T}}_i$ is captured during its execution (step 2). The interval corresponding to the processing of b in $\hat{\mathcal{T}}_i$ is located based on *offset* and *bit_offset* for each of the 256 bit positions $b \in \{0, 1, \dots, 255\}$ (representing the message bits in an unknown shuffled order) (steps 5-6). This interval is fed into the neural network $\mathcal{NN}$ trained during the profiling stage to determine whether the message bit in position b has a value of 0’ or 1’. If the resulting score s_b is greater than 0.5 (i.e. ‘1’ has a higher probability), the HW is incremented. Otherwise, the HW is not changed.

The HW is then determined by first removing the outliers and then computing the median of the remaining HWs (steps 13-14). An outlier is defined as a HW that differs from the median HW by more than 10%. We explored a variety of combining methods. The one we present consistently outperforms others in our experiments.

7.5 Message recovery algorithm

In this section, we present a ‘fuzzy” version of the bit-flip technique, RecoverMessage(). We construct 256 ciphertexts containing bit-flipped messages in the same way as in the original method [RBRC20]. However, we take a different approach to deciding the final message bit values. We also quantify the success rate of message HW recovery as a function of the success rate of single bit recovery.

The pseudo-code is shown in Fig. 8. RecoverMessage() takes as input the same parameters as RecoverHW() algorithm. First, the HW of m' contained in c is recovered by calling RecoverHW(). Then, the following loop is repeated 256 times: For each $i \in \{0, \dots, 255\}$, the ciphertext c_i is constructed using BitFlip(), i.e. the value of the center of $\mathbb{Z}_q$ is subtracted from the ith coefficient of c. The HW of m'_i contained in c_i is recovered by calling RecoverHW(). If $HW(m'_i) > HW(m')$, the ith bit of m' is assigned ‘0’. If $HW(m'_i) < HW(m')$, the ith bit of m' is assigned ‘1’. Otherwise the ith bit of m' is assigned ‘2’ to indicate that the bit is not recovered correctly. In the experiments, we call this case a *detectable error*.

Next we quantify the probability to recover the message HW as a function of the probability to recover the single message bit. The property below assumes that the message is balanced, i.e. has equal number of ‘1’s and ‘0’s.

```
RecoverMessage($\mathcal{NN}$, $in\_size$, $offset$, $bit\_offset$, $c$, $N$)
$HW(m') = \text{RecoverHW}(\mathcal{NN}, in\_size, offset, bit\_offset, c, N)$
for each $i \in \{0, \ldots, 255\}$ do
    $c_i = \text{BitFlip}(c, i)$
    $HW(m'_i) = \text{RecoverHW}(\mathcal{NN}, in\_size, offset, bit\_offset, c_i, N)$
    if $HW(m'_i) > HW(m')$ then
        $m'[i] = 0$
    else
        if $HW(m'_i) < HW(m')$ then
            $m'[i] = 1$
        else
            $m'[i] = 2$ /* error detected */
        end if
    end if
end for
return $m'$
```

Figure 8: Message recovery algorithm.

Property 1. Let m be a balanced n-bit binary message. If p is the success rate of single bit recovery and bit errors are mutually independent events, then the success rate of message HW recovery is given by:

$$p_{HW} = \sum_{i=0}^{n/2} \binom{n/2}{i}^2 p^{n-2i}(1-p)^{2i} \tag{3}$$

Proof. The proof is based on the fact that, if, for any $0 \leq k \leq n/2$, k message bits change as $0 \rightarrow 1$ and other k message bits change as $1 \rightarrow 0$, then the message HW does not change.

A n-bit balanced binary message has $n/2$ '0's and $n/2$ '1's. There are $\binom{n/2}{k}$ choices to select k elements from a set of size $n/2$. Thus, for a fixed k, the number of possible $2k$-bit errors in which k bits flip in one direction and the rest of bits flip in another direction is $\binom{n/2}{k}^2$. Since the probability of a $2k$-bit error in an n-bit message is $p^{n-2k}(1-p)^{2k}$, we get (3). □

Using Property 1 we can estimate the success rate of single bit recovery required to recover the message HW. Table 3 lists some examples. According to the table, the success rate of single bit recovery should be of the order of 0.999 to recover the message HW with a high probability.

Table 3: Success rate of 256-bit message HW recovery.

p	p_{HW}
0.99	0.279883
0.999	0.879911
0.9999	0.987280
0.99999	0.998719

8 Experimental results

In the experiments, we use two identical CW303 ARM devices, D_P and D_A. D_P is the profiling device. We have complete control over D_P, which means we can reload it with a

Table 4: The impact of training set composition on message recovery success rate.

Training	# Detected (d) and undetected (u) errors in the test set																					
set	0		1		2		3		4		5		6		7		8		9		**Average**	
$D_P : D_A$	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u
0:100	7	98	4	101	5	104	6	102	4	111	8	100	7	102	0	113	10	86	12	75	6.3	99.2
20:80	3	138	0	127	0	125	0	126	0	126	0	129	1	129	0	114	4	124	3	133	1.1	127.1
50:50	0	142	0	128	0	126	0	126	0	126	0	131	0	130	0	114	0	138	0	144	0	130.5
80:20	1	1	2	3	0	0	0	2	2	3	0	1	0	0	2	5	0	4	0	2	0.7	2.1
100:0	0	1	1	4	2	2	1	4	3	5	0	4	1	4	2	24	0	1	1	0	1.1	4.9

different implementation, change its secret key, etc. D_A is the device that is being attacked. We use D_A to capture traces for key recovery and a part of traces for training.

8.1 Message recovery

In this section we evaluate the impact of training set composition on the success rate of RecoverMessage() algorithm.

We trained MLP models on trace sets of size 2M with varying proportions of D_P and D_A traces, denoted by $D_P : D_A$. We tried five cases: $D_P : D_A = \{0{:}100, 20{:}80, 50{:}50, 80{:}20, 100{:}0\}$. The notation $x : y$ means that $x\%$ of traces are from D_P and $y\%$ are from D_A.

For each fraction $D_P : D_A = x : y$, we trained ten models with the architecture in Table 1 using TrainModel() with input parameters $\tau = 2\text{M}$ and $k = x/100$ and selected the best.

We tested the models on ten different ciphertexts created by encrypting a random message with a random public key. To recover the message, $257 \times N = 5140$ traces from D_A were captured for each ciphertext, for $N = 20$.

Table 4 lists the number of detected and undetected errors for each of the ten test sets. Recall that detected errors are those for which RecoverMessage() returns "2" as the message bit value. The ability to detect errors is very useful since e detected bit errors can be handled by enumerating 2^e possible choices, computing $(\hat{K}', r') = \mathcal{G}(pkh, m')$ and then checking if $c = \text{Saber.PKE.Enc}(pk, m'; r')$.

We can see from Table 4 that the model trained on a combination of 80% of traces from D_P and 20% of traces from D_A produces the best results. It seems that including a small number of traces from the device under attack helps to mitigate the negative effect of device variability on classification accuracy. The next best option is to train on all of the traces from the profiling device. Its primary advantage is that profiling can be completed prior to the attack. However, it is worth noting that, thanks to the cut-and-join technique, we only need to capture 1.5K traces from D_A to contribute 20% of traces to the 2M training set, which takes less than 4 min. As a result, composing the training set as 80:20 has no significant effect on physical access time to the device under attack.

Table 4 also shows that a single model trained on an 80:20 training set has no undetected errors in only 2 out of 10 cases. It is possible to improve the success rate by increasing the value of N; for example, for $N = 40$, there are no undetected errors in all cases.

However, a larger N increases capture time for attack traces, which is undesirable. Thus, in the experiments that follow, rather than increasing N, we use an ensemble of models to improve the success rate of message recovery. The ensemble approach increases training time, but this is not as critical as increasing physical access time to D_A.

Table 5: Success rate of key recovery (average for 10 tests)

N	# Models in ensemble, k	# Errors		Attack time	
		Detected	Undetected	Post-proc.	Capture
20	11	0.8	0	32.9 min	5.6 h 123,360 traces
	7	1.6	0	22.3 min	
	5	3.8	0	15.2 min	
	3	11.5	0.2	8.9 min	
15	11	7	0	25.9 min	4.2 h 92,520 traces
	7	8.2	0	18.2 min	
	5	12.4	0	12.5 min	
	3	29.7	2.4	7.2 min	
10	11	21.8	0.4	19.4 min	3.2 h 61,680 traces
	7	28.6	1.1	12.4 min	
	5	36.3	1.9	8.9 min	
	3	62.6	3.8	5.4 min	

8.2 Secret key recovery

To evaluate the success rate of the secret key recovery attack, we captured ten test sets of $24 \times 257 \times N$ traces representing the decapsulation of ciphertexts constructed following steps 1-3 of the procedure in Section 7.1, for $N = 10, 15$ and 20. Each test set was captured for a different secret key.

To recover the secret messages contained in the ciphertexts, we use an ensemble of models trained on a combination of 80% of traces from D_P and 20% of traces from D_A. The ensemble method is known to be useful in side-channel analysis [WD20, PCP20]. Table 5 shows the results for ensembles of size $k \in \{3, 5, 7, 11\}$.

The output of an ensemble of k models is obtained as follows. For each $j \in \{0, 1, \ldots, 255\}$, models that result in $m'[j] = 2$ (i.e. detected error) are excluded from voting, and then the mean of the $m'[j]$s produced by the remaining models is computed. If the mean is $\leq$ 0.5, the jth message bit is set to '0'; otherwise it is set to '1'. Finally, the secret key is derived from the 24 recovered messages using the procedure in Section 7.1, step 5.

Since we use the ECC-based method [NDGJ21] which is able to correct single errors and detects one additional error in the recovered message, we can mark the positions of the detected incorrect key coefficients for later enumeration. With d detected incorrect key coefficients, 9^d enumerations are required to find the true key. For example, for $N = 20$ and $V = 11$, $9^{0.8} \approx 6$ enumerations are required.

Undetected errors are positions that are not handled by the ECC. We determine them by comparing the recovered key to the true key. One can see from the Table 5 that, for $N \geq 15$ and $k \geq 5$, there are no undetected errors. Certainly, the values of N and k may vary depending on the implementation, environmental conditions, acquisition method, etc.

The last two columns of Table 5 show the time required for capturing traces and post-processing. Note that capture requires physical access to the device under attack, whereas post-processing does not.

9 Conclusion

We demonstrated that it is possible to break a masked and shuffled implementation of Saber KEM. These countermeasures, when combined, were thought to provide adequate protection against power/EM analysis. The presented message and key recovery attacks are not specific to Saber and might be applicable to other LWE/LWR-based PKE/KEMs.

Future work includes designing stronger countermeasures for LWE/LWR-based PKE/KEMs.

10 Acknowledgments

We are grateful to the authors of [BDK+20], who generously shared the source code of the masked Saber with us.

This work was supported in part by the Swedish Civil Contingencies Agency (Grant No. 2020-11632), the Swedish Research Council (Grants No. 2018-04482 and 2019-04166), the Swedish Foundation for Strategic Research (Grant No. RIT17-0005) and the Wallenberg Autonomous Systems and Software Program (WASP).

References

[ACLZ20] D. Amiet, A. Curiger, L. Leuenberger, and P. Zbinden. Defeating NewHope with a single trace. In *International Conference on Post-Quantum Cryptography*, pages 189–205. Springer, 2020.

[APSQ06] C. Archambeau, E. Peeters, F. X. Standaert, and J. J. Quisquater. Template attacks in principal subspaces. In *Cryptographic Hardware and Embedded Systems*, pages 1–14, 2006.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. Cryptology ePrint Archive, Report 2020/733, 2020. https://eprint.iacr.org/2020/733.

[BFD20] Martin Brisfors, Sebastian Forsmark, and Elena Dubrova. How deep learning helps compromising USIM. In *Proc. of the 19th Smart Card Research and Advanced Application Conference (CARDIS'2020)*, Nov. 2020.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 45–68, 2017.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CPM+18] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 163–177, 2018.

[CW3] CW308 UFO Target. https://wiki.newae.com/CW308_UFO_Target.

[D$^+$20] J. D'Anvers et al. Saber algorithm specifications and supporting documentation. *https://csrc.nist.gov/projects/postquantum-cryptography/round-3-submissions*, 2020.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, July 1964.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sideâĂŘchannel resistance validation. In *NIST Mon-Invasive Attack Testing Workshop*, 2011.

[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. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 359–386, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qtesla. In *International Conference on Smart Card Research and Advanced Applications*, pages 74–91. Springer, 2019.

[HGA$^+$19] C. Hoffman, C. Gebotys, D. F. Aranha, M. Cortes, and G. AraÃžjo. Circumventing uniqueness of XOR arbiter PUFs. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 222–229, 2019.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[KPH$^+$19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. Unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):148–179, May 2019.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 344–362, Bogota, Colombia, June 5–7, 2019. Springer, Heidelberg, Germany.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 3–26, Cham, 2016. Springer International Publishing.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure saber KEM. *IACR Trans. on CHES*, 2021(4), 2021.

[New] NewAE Technology Inc. Chipwhisperer. https://newae.com/tools/chipwhisperer.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure masked Ring-LWE implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):142–174, 2018. https://tches.iacr.org/index.php/TCHES/article/view/836.

[PCP20] Guilherme Perin, ÅĄukasz Chmielewski, and Stjepan Picek. Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):337–364, Aug. 2020.

[RBRC20] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) nist pqc candidates for practical message recovery and key recovery attacks. Cryptology ePrint Archive, Report 2020/1559, 2020. https://eprint.iacr.org/2020/1559.

[RdCR$^+$16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-lwe masking. In *Post-Quantum Cryptography*, pages 233–244. Springer, 2016.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):307–335, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8592.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 683–702. Springer, 2015.

[Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[SKL$^+$20] Bo-Yeon Sim, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Taeho Lee, Jaeseung Han, Hyojin Yoon, Jihoon Cho, and Dong-Guk Han. Single-trace attacks on the message encoding of lattice-based kems. Cryptology ePrint Archive, Report 2020/992, 2020. https://eprint.iacr.org/2020/992.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography – PKC 2019*, pages 534–564, Cham, 2019. Springer International Publishing.

[Tim18] Benjamin Timon. Non-profiled deep learning-based side-channel attacks. IACR Cryptology ePrint Archive, Report 2018/196, 2018.

[UXT$^+$21] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. Cryptology ePrint Archive, Report 2021/849, 2021. https://eprint.iacr.org/2021/849.

[WD20] Huanyu Wang and Elena Dubrova. Tandem deep learning side-channel attack against FPGA implementation of AES. In *Proc. of IEEE International Symposium on Smart Electronic Systems (iSES)*, pages 147–150, 2020.

[XPRO] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David Oswald. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. Technical report, Cryptology ePrint Archive, Report 2020/912, 2020. https://eprint.iacr.org.