Side-Channel Attacks on Lattice-Based KEMs Are Not Prevented by Higher-Order Masking

Kalle Ngo, Ruize Wang, Elena Dubrova and Nils Paulsrud

KTH Royal Institute of Technology, Stockholm, Sweden
{kngo,ruize,dubrova,nilspa}@kth.se

Abstract. In this paper, we present the first side-channel attack on a higher-order masked implementation of an IND-CCA secure lattice-based key encapsulation mechanism (KEM). Our attack exploits a vulnerability in the procedure for the arithmetic to Boolean conversion which we discovered. On the example of Saber KEM, we demonstrate successful message and secret key recovery attacks on the second- and third-order masked implementations running on a different device than the profiling one. In our experiments, we use the latest publicly available higher-order masked implementation of Saber KEM in which all known vulnerabilities are patched. The presented approach is not specific to Saber and can be potentially applied to other lattice-based PKE and KEM algorithms, including CRYSTALS-Kyber which has been recently selected for standardization by NIST.

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

1 Introduction

Masking is a well-known countermeasure against power/EM side-channel analysis [CJRR99]. A k-order masking partitions any sensitive variable x into $k+1$ shares, $x_1, x_2, \ldots, x_{k+1}$, such that $x = x_1 \circ x_2 \circ \ldots \circ x_{k+1}$, and executes all operations separately on the shares. The operator "$\circ$" depends on the type of masking, e.g. in arithmetic masking "$\circ$" is equal to "$+$" and in Boolean masking "$\circ$" is "$\oplus$" .

In theory, performing operations separately on the shares $x_1, x_2, \ldots, x_{k+1}$ should prevent the leakage of side-channel information related to x since computations do not directly involve x. Instead, the leakage is linked to the shares $x_1, x_2, \ldots, x_{k+1}$. 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 random masks $r_1, r_2, \ldots, r_k$ to k shares and computing the last share as $x - (r_1 + r_2 + \ldots + r_k)$ for arithmetic masking or $x \oplus r_1 \oplus r_2 \oplus \ldots \oplus r_k$ for Boolean masking.

However, it has been shown that, in practice, a first-order masked implementation can be broken by side-channel analysis. In the attack presented in [NDGJ21], the sensitive variable is recovered directly, without explicitly extracting random masks at each execution. Apart from its simplicity, such an approach enables profiling on traces captured from the device under attack (because it does not need to know masks). Previous attacks on masked implementations of LWE/LWR-based PKEs/KEMs [RBRC20] required a controllable profiling device which can be re-programmed to the implementation with known masks. Profiling on the device under attack helps maximize the neural network's prediction accuracy.

The side-channel attack in [NDGJ21] was mounted on a first-order masked software implementation of Saber KEM which, at that time, was the only NIST round 3 candidate

having a publicly available protected implementation. In response to vulnerabilities discovered in [NDGJ21] and other related worksa new, higher-order masked implementation of Saber KEM has been recently released [KDB+22] in which the procedures with known leakage points are re-implemented.

Our contributions: In this paper, we show that the higher-order masked implementation of Saber KEM presented in [KDB+22] has an exploitable vulnerability in its arithmetic to Boolean conversion procedure `A2B_bitsliced_msg()`. This vulnerability allows us to extend the message recovery method from [NDGJ21] to higher-order masked implementations. We demonstrate sucessfull message and secret key recovery attacks on the second- and third-order masked implementations. To the best of our knowledge, no side-channel attacks on a higher-order masked implementation of a lattice-based PKE or KEM algorithm has been reported until now.

The presented approach is not specific to Saber KEM and can be applied to other lattice-based PKE and KEM algorithms using a similar implementation of `A2B_bitsliced_msg()` procedure, including CRYSTALS-Kyber [S+20] which has been recently selected for standardization by NIST [NIS16]. We use Saber for our experiments because its higher-order masked implementation is publicly available.[1]

The rest of this paper is organized as follows. Section 2 reviews previous work related to side-channel analysis of lattice-based PKE/KEMs. Section 3 gives a background on the Saber algorithm. Section 4 describes the new vulnerability exploited in our attacks. Sections 5 and 6 present the profiling and attack stages, respectively. Section 7 shows experimental results. Section 8 concludes the paper and discusses future work.

2 Previous work

In this section, we describe previous side-channel attacks on lattice-based PKE and KEM algorithms.

Since the beginning of NIST post-quantum cryptorgaphy (PQC) standardization process in 2016 [NIS16], timing, power and electromagnetic (EM) emanations-based side-channel attacks on implementations of lattice-based PKE/KEMs have received a lot of attention. This is due to the fact that three out of four round 3 candidates of the NIST PQC standardization process are based on lattice problems: an NTRU-based scheme NTRU [C+20], a Learning With Errors (LWE)-based scheme Kyber [S+20], and a Learning With Rounding (LWR)-based scheme Saber [D+20]. Lattice problems are believed to be difficult for large-scale quantum computers.

In [SKL+20], message recovery attacks using a single power trace from an unprotected encapsulation part of round 3 candidates CRYSTALS-Kyber and Saber as well as round 3 alternate candidate FrodoKEM were presented. In [MBM+22], attacks using correlation power analysis are presented, targeting the polynomial multiplication in unprotected implementations of all lattice-based finalists in the NIST PQC standardization process. In [BDH+21], side-channel attacks on two implementations of masked polynomial comparison, applied to Kyber, are presented.

In [RSRCB20], a near field EM message recovery attack on Kyber using a vulnerability in the Fujisaki-Okamoto (FO) transform is demonstrated. In [XPSR+21], a secret key recovery attack on an unprotected Kyber using near field EM side-channels was demonstrated. In [RBRC20], near field EM secret key recovery attacks on unprotected and protected implementations of NIST PQC round 3 candidates CRYSTALS-Kyber, Saber and round 3 alternate candidate FrodoKEM, as well as some round 2 candidates, were presented. It was shown how masked implementations can be broken by attacking each share individually. The resistance of an unprotected Saber KEM to amplitude-modulated EM emanations has

[1] A higher-order masked implementation of Kyber has been presented at CHES'2021 [BGR+21], however, it is not publicly available.

been investigated in [WND22b] and [WND22a]. In [GJN20], a timing attack on FrodoKEM was presented.

In [NDGJ21], a message and secret key recovery attack on a first-order masked implementation of Saber KEM through the use of deep learning-based power analysis was demonstrated. In [NDJ21], it was shown that it is possible to recover the secret key from Saber KEM, even if masking is complemented with a shuffling countermeasure. An attack using a method similar to [NDGJ21] on a first-order masked implementation of Kyber was presented in [WCCL22], targeting the message encoding vulnerability found in [SKL+20].

In [UXT+21], power/near field EM secret key recovery attacks on KEMs were described, including Kyber, Saber, and NTRU. These attacks use side-channel leakage during the re-encryption step of decapsulation as a plaintext-checking oracle that determines whether the results of the PKE decryption and the reference plaintext are equivalent.

3 Saber KEM algorithm

Saber [D+20] is one of the candidates of the NIST PQC standardization process which made it all the way to the third round. However, it has not been selected for standardization after the third round, and will not continue into the fourth round [NIS16]. The winner among candidates for key-establishment is CRYSTALS-Kyber [S+20].

The security of Saber is based on the hardness of the Module Learning with Rounding problem (MLWR). The core of Saber is an IND-CPA secure encryption scheme, Saber.PKE, and which is followed by an IND-CCA secure key encapsulation mechanism, Saber.KEM. The latter is created by transforming Saber.PKE into Saber.KEM using a variation of the FO transform [FO99]. Figs. 1 and 2 describe algorithms Saber.PKE and Saber.KEM, respectively. We follow the notation of [NDGJ21].

Let $\mathbb{Z}_q$ denote the ring of integers modulo a positive integer q and R_q the quotient ring $\mathbb{Z}_q[X]/(X^n+1)$. The rank of the module and the rounding modulus are denoted by p and l, respectively. The positive integers q, p, and T are chosen to be a power of 2, i.e., $q = 2^{\epsilon_q}$, $p = 2^{\epsilon_p}$, and $T = 2^{\epsilon_T}$, respectively. Saber uses parameters $n = 256$, $l = 3$, $q = 2^{13}$, $p = 2^{10}$, and $T = 2^4$.

Let $\mathcal{U}$ denote the uniform distribution and β_μ the centered binomial distribution with parameter μ, where μ is an even positive integer. Saber uses $\mu = 8$. The samples of β_μ are in the interval $[-\mu/2, \mu/2]$ and its probability mass function is given by $P[x|x \leftarrow \beta_\mu] = \frac{\mu!}{(\mu/2+x)!(\mu/2-x)!}2^{-\mu}$, where $x \leftarrow \beta_\mu$ stands for sampling from β_μ. The term $\beta_u(R_q^{l\times k}; r)$ induces 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 hash functions $\mathcal{F}$ and $\mathcal{H}$ are realized using SHA3-256, and the hash function $\mathcal{G}$ is realized using SHA3-512. An extendable output function gen is used to generate a pseudorandom matrix $\mathbf{A} \in R_q^{l\times l}$ from a seed $seed_{\mathbf{A}}$. It is implemented by SHAKE-128.

The bitwise right shift operation is denoted by "$\gg$". It extends to polynomials and matrices by applying it coefficient-wise. In order to implement rounding operations by a simple bit shift, Saber uses two constant polynomials $h_1 \in R_q$ and $h_2 \in R_q$ with all coefficients fixed to $2^{\epsilon_q-\epsilon_p-1}$ and $2^{\epsilon_p-2} - 2^{\epsilon_p-\epsilon_T-1} + 2^{\epsilon_q-\epsilon_p-1}$, respectively, and one constant vector $\mathbf{h} \in R_q^{l\times 1}$ in which each polynomial equals to h_1.

Due to specific features of its design: power-of-two moduli q, p and T, and limited noise sampling of LWR, Saber can be efficiently masked. Due to the former, modular reductions are basically free. The latter implies that only the secret key has to be sampled securely.

Saber.PKE.KeyGen()

1: $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $\mathbf{A} = \mathsf{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
3: $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: $\mathbf{s} \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
5: $\mathbf{b} = ((\mathbf{A}^T\mathbf{s} + \mathbf{h}) \mod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6: **return** $(pk \doteq (seed_{\mathbf{A}}, \mathbf{b}), sk \doteq \mathbf{s})$

Saber.PKE.Dec($\mathbf{s}, (c_m, \mathbf{b}')$)

1: $v = \mathbf{b}'^T(\mathbf{s} \mod p) \in R_p$
2: $m' = ((v + h_2 - 2^{\epsilon_p - \epsilon_T} c_m) \mod p) \gg (\epsilon_p - 1) \in R_2$
3: **return** m'

Saber.PKE.Enc($(seed_{\mathbf{A}}, \mathbf{b}), m; r$)

1: $\mathbf{A} = \mathsf{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
2: **if** r is not specified **then**
3: $\quad r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: **end if**
5: $\mathbf{s}' \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
6: $\mathbf{b}' = ((\mathbf{A}\mathbf{s}' + \mathbf{h}) \mod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
7: $v' = \mathbf{b}^T(\mathbf{s}' \mod p) \in R_p$
8: $c_m = ((v' + h_1 - 2^{\epsilon_p - 1} m) \mod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
9: **return** $(c \doteq (c_m, \mathbf{b}'))$

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

Saber.KEM.KeyGen()

1: $(seed_{\mathbf{A}}, \mathbf{b}, \mathbf{s}) =$ Saber.PKE.KeyGen()
2: $pk = (seed_{\mathbf{A}}, \mathbf{b})$
3: $pkh = \mathcal{F}(pk)$
4: $z \leftarrow \mathcal{U}(\{0,1\}^{256})$
5: **return** $(pk \doteq (seed_{\mathbf{A}}, \mathbf{b}), sk \doteq (z, pkh, pk, \mathbf{s}))$

Saber.KEM.Decaps($(z, pkh, pk, \mathbf{s})$,c)

1: $m' =$ Saber.PKE.Dec($\mathbf{s}, c$)
2: $(\hat{K}', r') = \mathcal{G}(pkh, m')$
3: $c' =$ Saber.PKE.Enc($pk, m'; r'$)
4: **if** $c = c'$ **then**
5: $\quad$ **return** $K = \mathcal{H}(\hat{K}', c)$
6: **else**
7: $\quad$ **return** $K = \mathcal{H}(z, c)$
8: **end if**

Saber.KEM.Encaps($(seed_{\mathbf{A}}, \mathbf{b})$)

1: $m \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
3: $c =$ Saber.PKE.Enc($pk, m; r$)
4: $K = \mathcal{H}(\hat{K}, c)$
5: **return** (c, K)

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

4 Vulnerability exploited in the attack

In this section, we first describe known vulnerabilities in software implementations of LWE/LWR-based PKE/KEMs and then present the new vulnerability which we discovered in the arithmetic to Boolean conversion procedure `A2B_bitsliced_msg()` of the higher-order masked implementation of Saber KEM from [KDB+22].

4.1 Known vulnerabilities

In previous work, a number of vulnerabilities have been discovered in software implementations of LWE/LWR-based PKE/KEMs including incremental-storage vulnerability [RBRC20], weakness of re-encryption operation in FO transform [UXT+21] and weakness of polynomial multiplication [MBM+22].

In [RBRC20], it was shown that two procedures in some non-masked implementations of LWE/LWR-based PKE/KEMs contain an exploitable incremental-storage vulnerability. The first one is message decoding function (line 2 of Saber.PKE.Dec() at Fig. 1) in which each polynomial coefficient is iteratively mapped into the corresponding message bit. Since the message is computed one bit at a time, an attacker can recover the bit by building a distinguisher for '0's and '1's.

The second one is `POL2MSG()` procedure, shown in Fig. 3. It iterates over an array of values encoded as single bits and saves them into a packed array of bytes. As one can see

```
void POL2MSG(uint16 *msg_unpacked, char *msg_dec)
int i, j;
 1: for (j = 0; i < 32; i++) do
 2:    msg_dec[j] = 0;
 3:    for (i = 0; i < 8; i++) do
 4:       msg_dec[j] |= (msg_unpacked[j*8+1]«i);
 5:    end for
 6: end for
```

```
0x0800493e      adds    r1, 0x1f
0x08004940      strb    r4, [r2, 1]
0x08004942      ldrh    r3, [r0]   ; arg1
0x08004944      lsrs    r3, r3, 0xd
0x08004946      strb    r3, [r2, 1]
0x08004948      ldrh    r5, [r0, 2] ; arg1
0x0800494a      lsrs    r5, r5, 0xd
0x0800494c      orr.w   r3, r3, r5, lsl 1
0x08004950      strb    r3, [r2, 1]
0x08004952      ldrh    r5, [r0, 4] ; arg1
0x08004954      lsrs    r5, r5, 0xd
0x08004956      orr.w   r3, r3, r5, lsl 2
0x0800495a      strb    r3, [r2, 1]
0x0800495c      ldrh    r5, [r0, 6] ; arg1
0x0800495e      lsrs    r5, r5, 0xd
0x08004960      orr.w   r3, r3, r5, lsl 3
0x08004964      strb    r3, [r2, 1]
0x08004966      ldrh    r5, [r0, 8] ; arg1
0x08004968      lsrs    r5, r5, 0xd
0x0800496a      orr.w   r3, r3, r5, lsl 4
0x0800496e      strb    r3, [r2, 1]
0x08004970      ldrh    r5, [r0, 0xa] ; arg1
0x08004972      lsrs    r5, r5, 0xd
0x08004974      orr.w   r3, r3, r5, lsl 5
0x08004978      sxtb    r3, r3
0x0800497a      strb    r3, [r2, 1]
0x0800497c      ldrh    r5, [r0, 0xc] ; arg1
0x0800497e      lsrs    r5, r5, 0xd
0x08004980      orr.w   r3, r3, r5, lsl 6
0x08004984      sxtb    r3, r3
0x08004986      strb    r3, [r2, 1]
0x08004988      ldrh    r5, [r0, 0xe] ; arg1
0x0800498a      lsrs    r5, r5, 0xd
0x0800498c      orr.w   r3, r3, r5, lsl 7
0x08004990      strb    r3, [r2, 1]!
0x08004994      cmp     r1, r2
0x08004996      add.w   r0, r0, 0x10 ; arg1
0x0800499a      bne     0x8004940
0x0800499c      pop     {r4, r5}
```

Figure 3: C and assembly codes of the original `POL2MSG()` implementation from [BDK+21] using bit-wise storage in memory.

```
void POLmsg2BS(uint8 bytes[N], uint16 data[256])
int i, j;
uint8 byte;
 1: for (j = 0; i < 32; i++) do
 2:    byte = 0;
 3:    for (i = 0; i < 8; i++) do
 4:       for (k = 0; i < 32; k++) do
 5:          byte |= ((data[j*8+i] & 0x01 « i);
 6:       end for
 7:       bytes[j] = byte
 8:    end for
 9: end for
```

```
0x080034ba      add.w   r5, r1, 0x200 ; arg2
0x080034be      ldrh    r4, [r1, 4] ; arg2
0x080034c0      ldrh.w  r8, [r1, 2] ; arg2
0x080034c4      ldrh    r2, [r1]   ; arg2
0x080034c6      ldrh    r3, [r1, 6] ; arg2
0x080034c8      ldrh.w  ip, [r1, 8] ; arg2
0x080034cc      ldrh.w  lr, [r1, 0xa] ; arg2
0x080034d0      ldrh    r7, [r1, 0xc] ; arg2
0x080034d2      ldrh    r6, [r1, 0xe] ; arg2
0x080034d4      and     r4, r4, 1
0x080034d8      lsls    r4, r4, 2
0x080034da      and     r8, r8, 1
0x080034de      orr.w   r4, r4, r8, lsl 1
0x080034e2      and     r2, r2, 1
0x080034e6      orrs    r2, r4
0x080034e8      and     r3, r3, 1
0x080034ec      orr.w   r3, r2, r3, lsl 3
0x080034f0      and     ip, ip, 1
0x080034f4      orr.w   r3, r3, ip, lsl 4
0x080034f8      and     lr, lr, 1
0x080034fc      orr.w   r3, r3, lr, lsl 5
0x08003500      and     r7, r7, 1
0x08003504      orr.w   r3, r3, r7, lsl 6
0x08003508      and     r6, r6, 1
0x0800350c      adds    r1, 0x10
0x0800350e      orr.w   r3, r3, r6, lsl 7
0x08003512      cmp     r5, r1
0x08003514      strb    r3, [r0, 1]!
0x08003518      bne     0x80034be
0x0800351a      pop.w   {r4, r5, r6, r7, r8, pc}
0x0800351e      nop
```

Figure 4: C and assembly codes of the improved `POLmsg2BS()` implementation from [BDK+21] and [KDB+22] using byte-wise storage in memory.

```
void A2B_bitsliced_msg(uint16 msg[SHARES][N],
uint16 vp[N][SHARES])
uint32 v1_BS[SHARES][SABER_EP];
uint32 v2_BS[SHARES][SABER_EP];
 1: for (i = 0; i < N/32; i+= 1) do
 2:    pack_bitslice(SHARES,SABER_EP,v1_BS,&vp[32*i]
 3:    A2B_bitsliced(SHARES,SABER_EP,v2_BS,v1_BS);
 4:    for (j=0; i<SHARES; j++) do
 5:       for (k = 0; i<32; k++) do
 6:          msg[j][32*i+k] =
             (v2_BS[j][SABER_EP-1]»k)&1;
 7:       end for
 8:    end for
 9: end for
```

```
0x08005ebe      ldr     r5, [var_5ch]
0x08005ec0      movs    r4, 0
0x08005ec2      add     r3, var_e0h
0x08005ec4      add     r3, r4
0x08005ec6      mov     r1, r5
0x08005ec8      ldr     r0, [var_24h_2]
0x08005eca      movs    r3, 0
0x08005ecc      lsr.w   r2, r0, r3
0x08005ed0      adds    r3, 1
0x08005ed2      and     r2, r2, 1
0x08005ed6      cmp     r3, 0x20    ; loc.Abu0
0x08005ed8      strh    r2, [r1, 2]!
0x08005edc      bne     0x8005ecc
0x08005ede      adds    r4, 0x28
0x08005ee0      cmp     r4, 0x78    ; loc.Ama0
0x08005ee2      add.w   r5, r5, 0x200
0x08005ee6      bne     0x8005ec2
0x08005ee8      ldr     r2, [var_5ch]
0x08005eea      ldr     r3, [var_60h]
```

Figure 5: C and assembly codes of `A2B_bitsliced_msg()` implementation from [KDB+22] using bit-wise storage in memory.

```
void InnerProdDec_masked_HO(uint8 m[SHARES][SABER_KEYBYTES], uint8
ct[SABER_BYTES_CCA_DEC], uint16 s[SHARES][L][N])
uint16 v[SHARES][N], vtemp[SHARES][N];
uint16 bp[N];
uint16 (*cm) = bp;
 1: for (i = 0; i < N; i++) do
 2:    BS2POLp(&ct[i * SABER_POLYCOMPRESSBYTES], cm);
 3:    /* polynomial multiplication */
 4: end for
 5: BS2POLT(ct+SABER_POLYVECCOMPRESSBYTES, cm);
 6: for (i = 0; i < N; i++) do
 7:    v[0][i] += h2-(cm[i]<<(SABER_EP-SABER_ET));
 8: end for
 9: for (j = 0; j < SHARES; j++) do
10:    for (i = 0; i < N; i++) do
11:       vtemp[i][j] = (v[j][i]&(SABER_P-1));
12:    end for
13: end for
14: A2B_bitsliced_msg(v, vtemp);
15: for (i = 0; i < SHARES; i++) do
16:    POLmsg2BS(m[i],v[i]);
17: end for
```

Figure 6: C code of `InnerProdDec_masked_HO()` procedure of Saber.PKE.Dec algorithm which calls `A2B_bitsliced_msg()` [KDB+22].

from the assembly code in Fig. 3, the inner loop is unrolled into eight store byte (`strb`) ARM instructions. Each of these *store* instructions updates the memory with a single new bit of information, resulting in bit-wise leakage. This makes message recovery by side-channel analysis easy.

In [NDGJ21] it was shown that `POL2MSG()` vulnerability can also be exploited in the first-order masked implementation of Saber KEM from [BDK+21]. In addition, it was demonstrated that `poly_A2A()` primitive designed in [BDK+21] for masked logical shifting on arithmetic shares contains an exploitable vulnerability.

In the subsequently released version of the first-order masked implementation of Saber KEM by Van Beirendonck et al. [BDK+21] and the higher-order masked implementation of Saber KEM by Kundu et al. [KDB+22], `POL2MSG()` was patched by re-implementing it to accumulate the message bits in a register before writing the entire byte to memory, as shown in Fig. 4. This results in a byte-dependent leakage, making side-channel analysis more difficult. Additionally, in the implementation from [KDB+22], `poly_A2A()` was replaced by the `A2B_bitsliced_msg()` procedure, shown in Fig. 5, which performs arithmetic to Boolean conversion of shares in a bitsliced fashion.

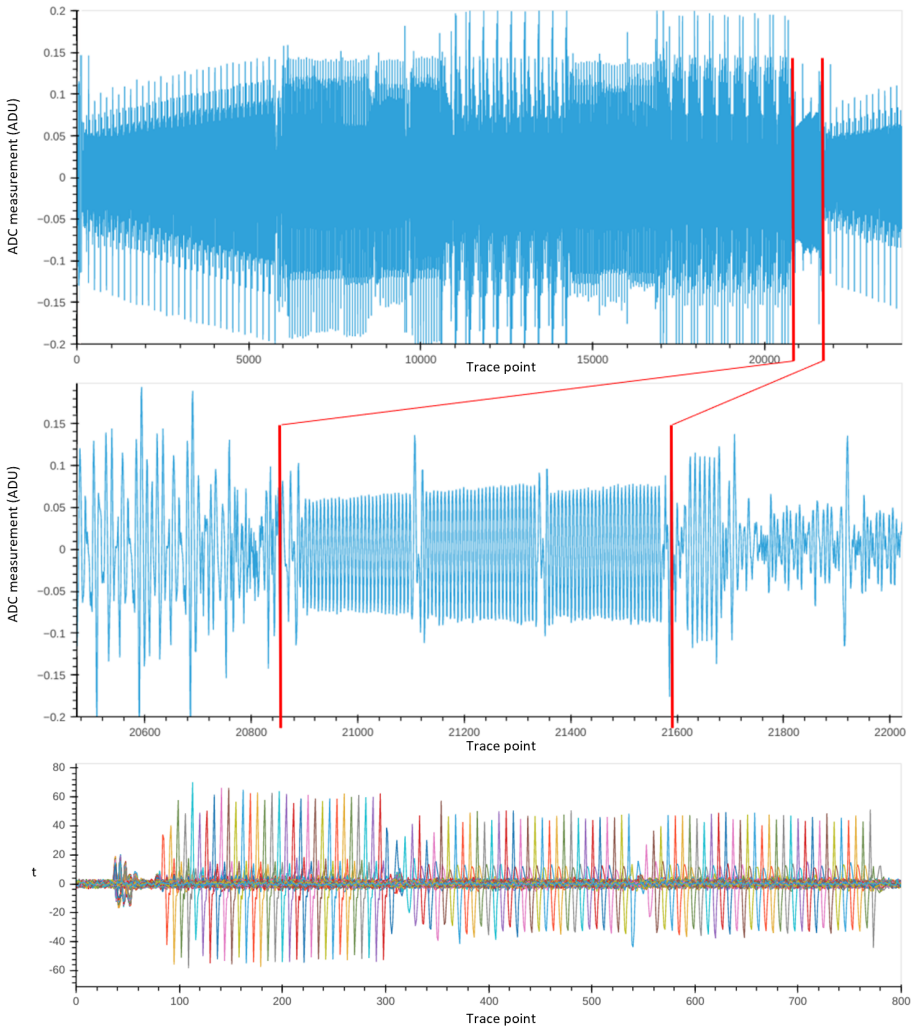

Figure 7: An average power trace representing the first iteration of `A2B_bitsliced_msg()` in the second-order masked implementation (top), a zoomed-in view of the segment in which three 32-bit shares are stored in memory bit-by-bit after conversion (middle), and t-test results for 10K traces with known masks (bottom).

4.2 New vulnerability

We discovered an exploitable vulnerability in `A2B_bitsliced_msg()` procedure of the higher-order masked implementation of Saber KEM from [KDB+22]. `A2B_bitsliced_msg()` is called by the procedure `InnerProdDec_masked_HO()` of Saber.PKE.Dec() algorithm during the decapsulation (see line 1 of Saber.KEM.Decaps() in Fig. 2). The C code of `InnerProdDec_masked_HO()` is shown in Fig. 6, where the call to `A2B_bitsliced_msg()` is marked in red.

Due to the bitsliced organization, `A2B_bitsliced_msg()` is executed in eight 32-bit iterations, as shown in the C code in Fig. 5. Fig. 7 (top) shows a power trace representing the first iteration of `A2B_bitsliced_msg()` in the second-order masked implementation. This trace is obtained by averaging 10K traces captured for ciphertexts encrypting random messages.

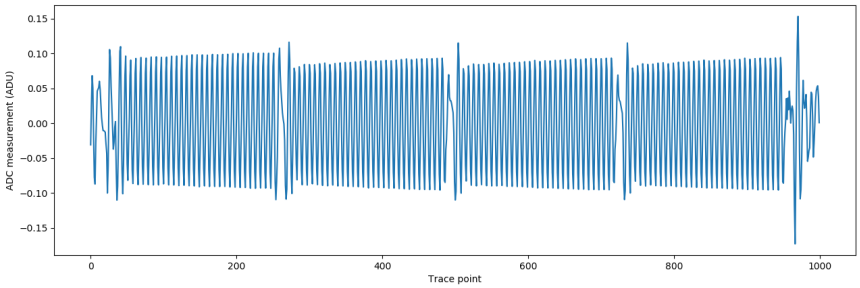

Figure 8: An average power trace representing a segment of the first iteration of `A2B_bitsliced_msg()` in the third-order masked implementation.

At the end of each iteration, when the arithmetic to Boolean conversion of shares is completed, the shares are copied from the intermediate variables into the message data structure in memory one bit at a time (see line 6 of Fig. 5). This is the location of the new vulnerability. Fig. 7 (middle) gives a zoomed-in view of the segment of trace where the lines 4-8 of the C code of `A2B_bitsliced_msg()` at Fig. 5 are executed during the first iteration. One can count 32 peaks in each of the three shares, corresponding to the storage of individual bits in memory. By running Welch's t-test [Wel47] on traces with known masks, we can verify that there are noticeable differences in measurements representing '0' and '1' bits, see Fig. 7 (bottom). The t-test scores of some bits reach 70. High t-test scores mean that '0's and '1's can be distinguished. Therefore, a neural network classifier capable of computing the $k+1$-argument XOR of the extracted bit values of $k+1$ shares should be able to recover the message from traces of a k-order masked implementation.

It is known that neural networks are capable of learning the XOR operation [GBC16, p. 166]. It is also known that neural networks can be trained to distinguish between '0' and '1' at a given bit location in a first-order masked implementation [NDGJ21]. In the second-order implementation, we can find the location of any given bit by finding the first peak of each share and measuring the distance between the peaks corresponding to bits. For example, in Fig. 7 (middle) the distance between the peaks is 7. This allows us to select a correct segment of trace for training and testing of neural networks without knowing the value of masks. For the third-order masked implementation, power traces look similarly (see Fig. 8) except that the shares are four and the distance between the peaks corresponding to bits is 8.

Note that the `A2B_bitsliced_msg()` procedure, which is our attack point, is not specific for the Saber KEM implementation from [KDB+22]. Any LWE/LWR-based PKE or KEM algorithm using a similar implementation of `A2B_bitsliced_msg()` can be attacked in a similar way. The arithmetic to Boolean conversion implemented by `A2B_bitsliced_msg()` is designed in [DBV22] as a general method for both, higher-order masked Saber and higher-order masked CRYSTALS-Kyber algorithms. A similar to `A2B_bitsliced_msg()` procedure is called `A2B_keepbitsliced()` in the implementation of [DBV22].

It is also important to mention that, not only `A2B_bitsliced_msg()`, but also *any* other procedure which stores bits of a sensitive variable in memory one-by-one may potentially contain an exploitable vulnerability. As an example, in Fig. 9 we show C and assembly codes of `masked_poly_tomsg()` procedure of CRYSTALS-Kyber implementation from [HKL+22]. The lines marked in red may contain an exploitable vulnerability since the shares are processed bit-wise.

```
void masked_poly_tomsg(uint8 msg[2][32], uint16
poly[2][256])
uint16 c[2];
 1: for (j = 0; i < 32; i++) do
 2:    byte = 0;
 3:    for (int i = 0; i < 32; i++) do
 4:       for (int j = 0; i < 8; j++) do
 5:          ...  Processing ...
 6:          msg[0][i] += ((c[0]»15)&1)«j;
 7:          msg[1][i] += ((c[1]»15)&1)«j;
 8:       end for
 9:    end for
10: end for
```

```
0x08006b24      mov r1, r5
0x08006b26      uxth r0, r3
0x08006b28      bl A2B_convert      ; dbg.A2B_convert
0x08006b2c      lsrs r5, r5, 0xf
0x08006b2e      ldrb.w r1, [fp]
0x08006b32      ldrb.w r3, [arg_20h]
0x08006b36      ubfx r2, r0, 0xf, 1
0x08006b3a      lsl.w r2, r2, sb
0x08006b3e      lsl.w r5, r5, sb
0x08006b42      add.w sb, sb, 1
0x08006b46      add r2, r1
0x08006b48      add r3, r5
0x08006b4a      cmp.w sb, 8         ; loc.mDo0
0x08006b4e      strb.w r2, [fp]
0x08006b52      strb.w r3, [arg_20h]
0x08006b56      bne 0x8006a64
0x08006b58      ldr r3, [sp]
0x08006b5a      adds r3, 0x10
```

Figure 9: C and assembly codes of `masked_poly_tomsg()` procedure from the first-order masked CRYSTALS-Kyber implementation from [HKL+22] using bit-wise storage to memory. The lines marked in red may contain an exploitable vulnerability.

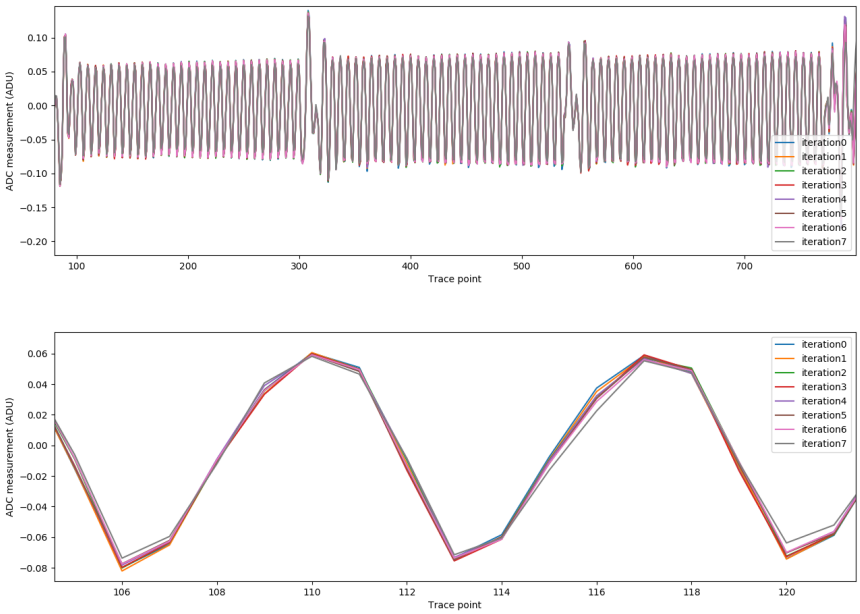

Figure 10: The comparison of average power traces of all eight iterations of `A2B_bitsliced_msg()` in the second-order masked implementation (top) and zoomed-in view of two bits (bottom).

5 Profiling stage

Our overall profiling strategy is similar to the one of [NDGJ21]. However, there are also differences which we highlight in this section.

5.1 Neural network type and input data shape

Let $\mathbb{R}$ to denote the set of real numbers and $\mathbb{I}$ denote a subset of $\mathbb{R}$ with elements within the interval [0,1], $\mathbb{I} := \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$.

We use the method introduced in [NDGJ21] in which neural networks are trained to extract messages directly, without explicitly retrieving the random masks. Message bits

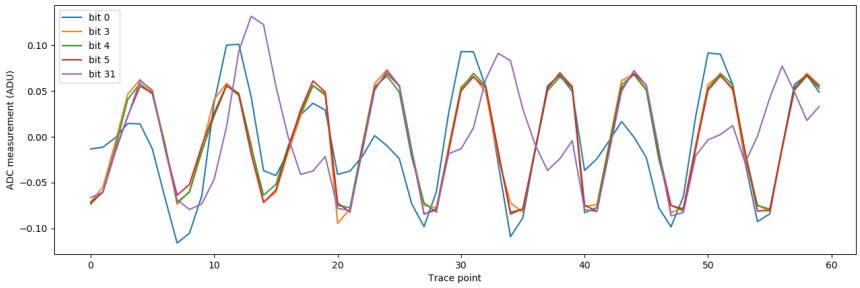

Figure 11: The comparison of average power traces of five different bits during the execution of `A2B_bitsliced_msg()` in the second-order masked implementation.

values '0' and '1' are used as labels for traces. To train a neural network $\mathcal{N}_j : \mathbb{R}^{|\mathcal{T}|} \rightarrow \mathbb{I}$ capable of predicting a given message bit j, each trace $\mathcal{T}$ in the training set $\boldsymbol{\mathcal{T}}$ is assigned a label $l(\mathcal{T}) = m[j]$, where $m[j]$ is the jth bit of the message m contained in the input ciphertext c which is decapsulated when $\mathcal{T}$ is captured.

Each $\mathcal{T} \in \boldsymbol{\mathcal{T}}$ contains $k+1$ intervals representing a given bit of $k+1$ shares $x_1, \ldots, x_{k+1}$ of a k-order masked implementation. Fig. 12 show traces which we give as input to neural networks for the second-order (top left) and third-order (top right) masked implementations, respectively. We select the intervals containing a given bit in each share x_i and concatenate them. The red dashed lines show the borders where the shares are concatenated. Note that, if a very large training set is available, one can also give a full trace to the network and let it figure out where the bits of shares are. However, we prefer to select and concatenate manually because it allows us to increase the size of the training set "for free" by applying the cut-and-join technique from [NDGJ21].

5.2 Cut-and-join technique

The cut-and-join technique [NDGJ21] composes the training set as a union of trace intervals corresponding to the processing of n different bits. This makes it possible increasing the size of the training set by a factor of n without having to capture n times as many traces. For example, for the third-order masked implementation, it takes us 12 hour to capture 100K traces. By cut-and-joining on 30 bits, we can get a training set of size 3M which would take us 15 days to capture otherwise.

The cut-and-join technique works best on implementations in which trace intervals corresponding to different bits look the same. As we mentioned in Section 4, due to the bitsliced organization `A2B_bitsliced_msg()` performs the conversion in eight iterations, in blocks of 32-bits. In Fig. 10 we compare the traces of all iterations in the second-order masked implementation. Fig. 10 (bottom) gives a zoomed-in view of two bits. One can see that their shape is very similar. Furthermore, within each iteration, the shape of 32 peaks corresponding to 32 bits is similar as well, except for the first and the last bits. The first and the last bits of each iteration are special, see Fig. 11, because their previous and next instructions, respectively, are different from the ones of other bits processed within the same iteration. Due to the ARM Cortex-M4 CPU's three-stage pipeline, the next instruction starts being processed before the previous instruction has finished. As a result, the power consumption during the processing of the first and the last bits of each iteration differs from the power consumption during the processing of other bits.

The similarity of traces of all eight iterations makes it possible to use traces of the first iteration for training universal neural networks which can recover message bits from other

Table 1: The MLP architecture used for message recovery attack on the second-order masked implementation. For the third-order masked implementation, the input size is 80 and the width of all layers, except the output one, is doubled.

Layer type	(Input, output) shape	# Parameters
Batch Normalization 1	(60, 60)	240
Dense 1	(60, 128)	7808
Batch Normalization 2	(128, 128)	512
ReLU	(128, 128)	0
Dense 2	(128, 32)	4128
Batch Normalization 3	(32, 32)	128
ReLU	(32, 32)	0
Dense 3	(32, 16)	528
Batch Normalization 4	(16, 16)	64
ReLU	(16, 16)	0
Dense 4	(16, 1)	17
Softmax	(1, 1)	0

iterations as well. This is one of the differences from the attack in [NDGJ21] which uses intervals representing all message bits for training. Our models can recover bits from the positions they have never seen during training.

We would like to mention that, most likely, the prediction accuracy of neural networks would be even higher if one would train a specialized model $\mathcal{N}_i$ for each message bit $i \in \{0, 1, \ldots, 255\}$ using the training set of size equal to the size of our training set after cut-and-join. However, unless access to device under attack is very limited, it may not be worthwhile spending 30 times more time on trace acquisition and training.

5.3 Standardization

From Fig. 10 (bottom) we can see that traces of eight iterations of `A2B_bitsliced_msg()` have some small shifts along y-axis. To smooth them, we apply standardization to traces (also known as *variance scaling*) [ZC18]. This is another difference from the attack in [NDGJ21] which does not use scaling[2].

Given a set of traces $\boldsymbol{\mathcal{T}}$ with elements $\mathcal{T} = (\tau_1, \ldots, \tau_{|\mathcal{T}|})$, each $\mathcal{T} \in \boldsymbol{\mathcal{T}}$ is standardized to $\mathcal{T}' = (\tau'_1, \ldots, \tau'_{|\mathcal{T}|})$ such that, for all $i \in \{1, \ldots, |\mathcal{T}|\}$:

$$\tau'_i = \frac{\tau_i - \mu_i}{\sigma_i},$$

where and μ_i and σ_i are the mean and standard deviation of the elements of $\boldsymbol{\mathcal{T}}$ at the ith data point.

5.4 Neural network architecture and hyperparameter selection

Table 1 shows the architecture of the multilayer perceptron (MLP) networks which we use for message recovery attack on the second-order masked implementation (same as in [NDGJ21]). For the third-order masked implementation, the network's input size is 80 and the width of all layers, except the output one, is doubled.

During training, we use NAdam optimizer [Doz16], which is an extension of RMSprop with Nesterov momentum, with a learning rate of 0.01 and numerical stability constant *epsilon*=1e-8. Binary cross-entropy is used as a loss function to evaluate the network classification error. The training is run for a maximum of 200 epochs, with a batch size of

[2]Scaling is not so important for the attack in [NDGJ21] because they give trace intervals representing all bits to neural networks during training.

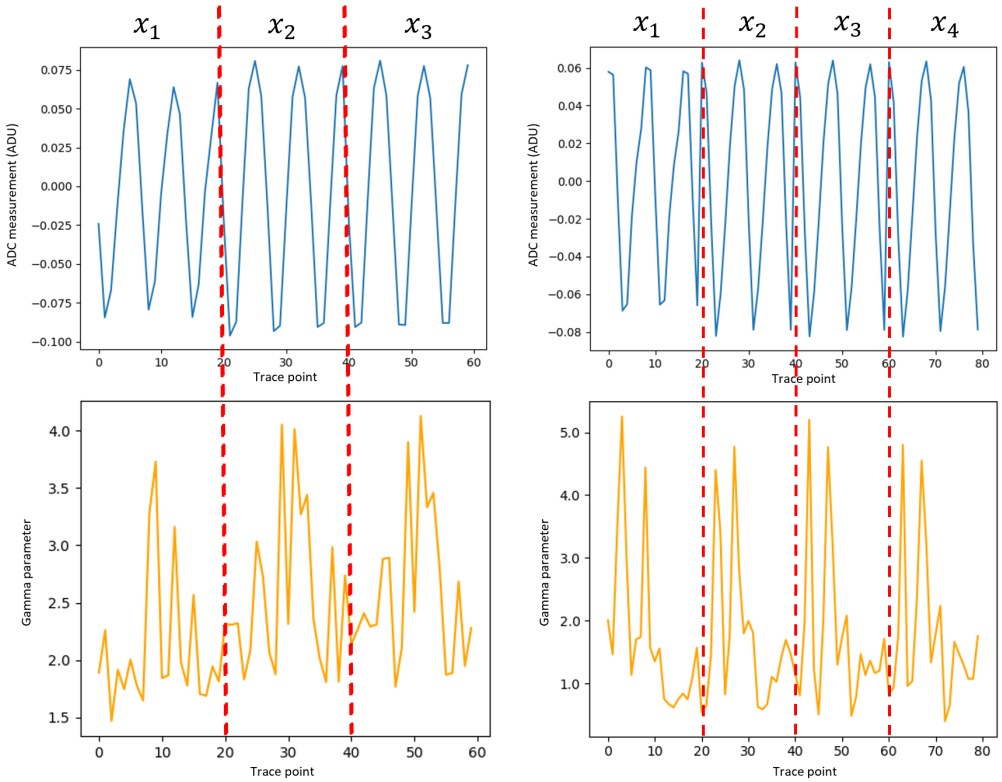

Figure 12: Segments of traces given as input to neural networks for the second-order (top left) and third-order (top right) masked implementations and the corresponding gamma parameters of the first Batch Normalization layer of a model after training.

1024 for training from scratch and 256 for fine-tuning from a pre-trained model. We use early stopping with patience 20. 70% of the training set is used for training, and 30% is reserved for validation. Only the model with the highest validation accuracy is saved.

5.5 Training from scratch vs pre-training

In our experiments with the second-order masked implementation, neural networks were easily learning from scratch. However, we spent some time figuring out how to get networks to learn for the third-order masked implementation. Training from scratch on cut-and-joined traces, or on traces for individual bits, consistently resulted in a random guess accuracy.

Finally, we found that neural networks start learning if we cut-and-join a small numbers of bits first. Fig. 13 (left) shows the training and validation accuracies of a neural network trained on cut-and-joined traces of bits 2-7, 600K traces in total. The validation accuracy flattens around 0.93. We then continued training on the cut-and-joined traces of bits 1 and 8-30, by loading the pre-trained model $\mathcal{N}_{2-7}$ as the starting state. Fig. 13 (right) shows that the validation accuracy increased above 97% as a result. Similarly, we trained models $\mathcal{N}_0$ and $\mathcal{N}_{31}$ starting from the pre-trained model $\mathcal{N}_{2-7}$.

We analysed weights of the resulting models to check if they correctly identify the position of leakage points in four shares. Fig. 12 (bottom, right) shows gamma parameters of the first Batch Normalization layer of a model after training. We can clearly see

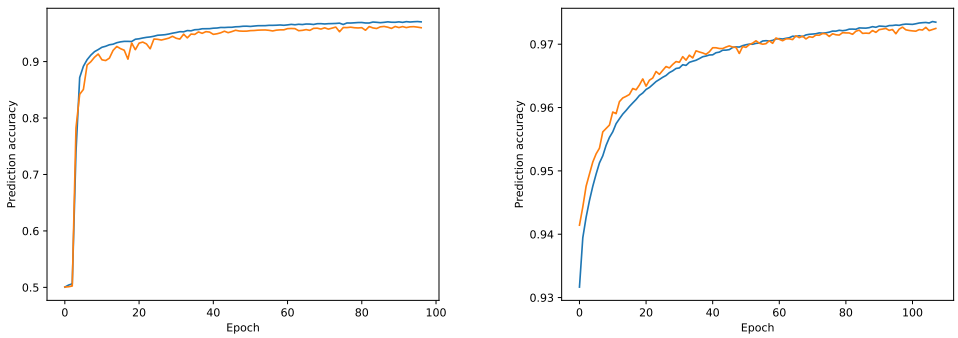

Figure 13: The training (blue) and validation (orange) accuracies of neural networks trained on traces from the third-order masked implementation for two cases: training from scratch (left) and training from a pre-trained model (right).

four groups of peaks corresponding to the leakage points in each share. This might be an indication that the model recovers the individual share bits and then XORs them together to get the final message bit. As we mentioned before, neural networks are capable of realizing the two-argument XOR operation [GBC16, p. 166]. Given that the two-argument XOR can be realized, the complexity of realizing the $k+1$-argument XOR, $x = x_1 \oplus x_2 \oplus \ldots \oplus x_{k+1}$, grows linearly in the number of arguments. Therefore, it is not surprising that the complexity of a deep learning-based side-channel attack grows linearly in the number of shares.

6 Attack stage

In this section, we describe the message, session key and secret key recovery methods used in our attacks. We also quantify the probability to recover the complete secret key as a function of the probability to recover a single message bit. The latter helps us decide how many times each measurement should be repeated for a successful attack.

6.1 Message recovery

Let m be a message to be recovered and $c = (\boldsymbol{c_m}, \mathbf{b}')$ be a properly generated ciphertext which contains m. To recover m, we use the device under attack to decapsulate c and capture the corresponding power trace $\mathcal{T}$. Then we locate in $\mathcal{T}$ the segment corresponding to storage of shares in memory at the end of each iteration of `A2B_bitsliced_msg()` (marked by red lines in Fig. 7 (middle)) and, for each bit $i \in \{0, 1, \ldots, 255\}$, locate ith bit of each share. The segments containing these bits are extracted and concatenated. In our experiments, we use the segments of size $p = 20$.

The resulting $p \times (k+1)$-point trace is given as input to the MLP models trained at the profiling stage. The models $\mathcal{N}_0$ and $\mathcal{N}_{31}$ are used to recover the bits $32b$ and $32b+31$, respectively, for $b \in \{0, 1, \ldots, 7\}$. The model $\mathcal{N}_{1-30}$ is used to recover the bits $32b+j$ for all $j \in \{1, 2, \ldots, 30\}$ and $b \in \{0, 1, 2, \ldots, 7\}$.

6.2 Session key recovery

A successful recovery of the message from a properly generated ciphertext c trivially implies the session key recovery, since the session key can be derived as $K = \mathcal{H}(\hat{K}', c)$ where

Table 2: The mapping of bits of eight messages into secret key coefficients [NDGJ21].

Coef. of $\mathbf{s}$	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

$(\hat{K}', r') = \mathcal{G}(pkh, m)$ (see lines 2 and 4 of Saber.KEM.Encaps()).

6.3 Secret key recovery

For secret key recovery, we use the ECC-based method of [NDGJ21] based on recovering the secret key from 24 chosen ciphertexts. Although the method of [NDGJ21] uses twice as many ciphertexts as methods presented in [RBRC20, RSRCB20], it is able to correct single-bit errors and detect double-bit errors in every coefficient of the secret key. This is a great advantage since perfect message recovery from a different device is difficult. The methods [RBRC20, RSRCB20] cannot correct any errors.

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

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

where the pairs (k_0, k_1) are defined in Table 2. The 768 coefficients of the secret key are mapped into codewords of the $[8, 4, 4]_2$ extended Hamming code composed from the bits of eight messages. The first group of 256 secret key coefficients is derived from the messages recovered from $c_1, \ldots, c_8$, the second group of 256 coefficients - from the messages recovered from $c_9, \ldots, c_{16}$, and the third group of 256 coefficients - from the messages recovered from $c_{17}, \ldots, c_{24}$.

For each chosen ciphertext c_i, $i \in \{1, \ldots, 24\}$, we capture N traces from the device under attack. As our experimental results show, repeating the same measurement more than once is necessary for handling errors which are beyond the ECC capacity. Using the $24 \times N$ traces, we recover 24 messages encrypted in the chosen ciphertexts as described in the previous section. Finally, the secret key is derived from the recovered messages using the mapping in Table 2.

Next we quantify the probability to recover the complete secret key of Saber KEM as a function of the probability to recover a single message bit. The expression *enumeration degree* K used in the sequel means that 9^K enumerations are required.

Property 1. If p is the probability to recover a message bit correctly and bit errors are mutually independent, then the sucess probability to recover the complete secret key of Saber KEM with enumeration degree K using the ECC method of [NDGJ21] is given by:

$$\mathcal{P}_{\mathbf{s}} = \sum_{k=0}^{K} \binom{768}{k} \mathcal{P}_d^k (\mathcal{P}_c + \mathcal{P}_p)^{768-k}, \tag{1}$$

where

$\mathcal{P}_p = p^8$ is the probability to recover a secret key coefficient correctly,

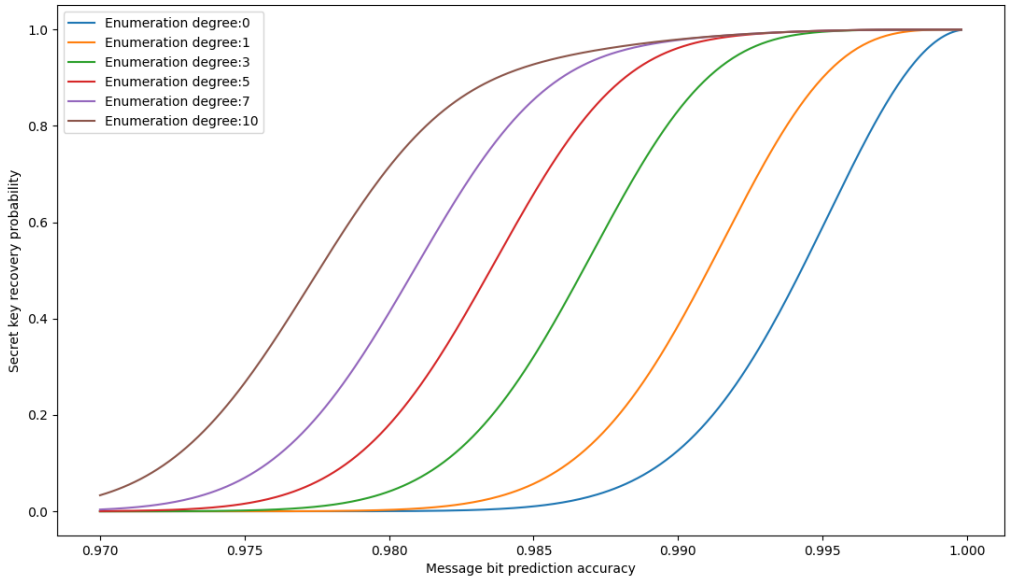

Figure 14: Secret key recovery probability computed using eq.(1).

$\mathcal{P}_c = 8(1-p)p^7$ is the probability to correct an error in a secret key coefficient by ECC, $\binom{8}{2}(1-p)^2p^6 < \mathcal{P}_d < 1-\mathcal{P}_p-\mathcal{P}_c$ is the probability to detect an error in a secret key coefficient by ECC.

Proof. The enumeration of degree K can handle at most K detected errors. The rest of secret key coefficients should be either predicted correctly, or corrected by the ECC. The probability of k detected errors is $\mathcal{P}_d^k$ and the probability of $(768-k)$ correct predictions is $(\mathcal{P}_c+\mathcal{P}_p)^{768-k}$. The success probability of secret key recovery is a sum of products $\mathcal{P}_d^k(\mathcal{P}_c+\mathcal{P}_p)^{768-k}$ for all possible combinations $\binom{768}{k}$.

The upper bound on $\mathcal{P}_d$ is given by $1-\mathcal{P}_p-\mathcal{P}_c$ because the probability of undetected errors is higher than zero. The lower bound on $\mathcal{P}_d$ is given by $\binom{8}{2}(1-p)^2p^6$ because the ECC can detect two errors in the predicted eight message bits and, if there are more than two errors, the ECC can detect them only if the Hamming distance between the predicted eight message bits and the message bits in Table 2 is larger than 1. □

Fig. 14 shows how the success probability of secret key recovery grows for different enumeration degrees.

7 Experimental results

This section describes the equipment we use for capturing traces and presents the results of message and secret key recovery attacks on the second- and third-order masked implementations of Saber KEM.

7.1 Measurement setup

Our measurement setup is shown in Fig. 15. It consists of the ChipWhisperer-Lite board, the CW308 UFO board and two CW308T-STM32F4 target boards, D_P and D_A.

The ChipWhisperer-Lite board [New] is used to measure power consumption and control the communication between the target device and the computer. It uses a synchronous

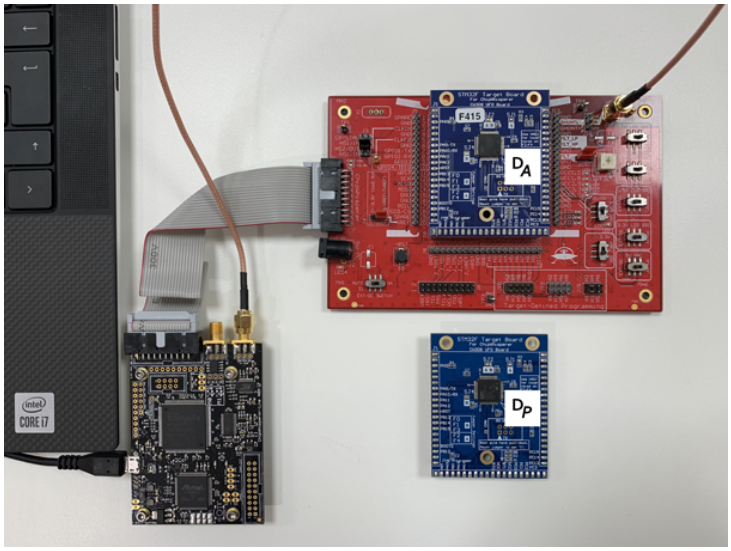

Figure 15: Equipment used for trace acquisition.

capture method which produces well-synchronized traces and reduces the required sample rate and the data storage. The maximum sampling rate of the ChipWhisperer-Lite board is 105 MS/sec and its buffer size is 24,400 samples.

The CW308 UFO board is a generic 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 device is programmed to the C implementation of higher-order masked Saber from [KDB+22]. The implementation is compiled with `arm-none-eabi-gcc` using the compiler optimization level -O3 (recommended default). This is the highest optimization level which is typically the most difficult to break by side-channel analysis [SKL+20]. The target board is run at 24 MHz and sampled at 24 MHz (1 pt/clock cycle).

The device D_P is used for capturing 100K traces for training neural networks at the profiling stage. The traces are captured for ciphertexts encrypting random messages and random secret keys. Each trace represents the execution of the first iterations on `A2B_bitsliced_msg()` procedure shown as Fig. 7 (top).

The device D_A is used for capturing test traces for the attack stage. The traces are captured with repetitions $N \in \{1, 3, 5, 7, 9\}$ for chosen ciphertexts and a fixed secret key. The execution of the eight iterations of `A2B_bitsliced_msg()` are recorded.

All traces in the training and test sets are pre-processed using standardization, as described in Section 5.3.

Table 3 shows the amount of time it takes us to capture the training and test sets. For test traces, the capture time is limited by the communication baud rate of 38,000 bps (sending chosen ciphertexts). For training traces, it is limited by the microprocessor clock frequency of 24 MHz. Note that, if profiling is done on D_P, the capture time from D_A defines the required access time to device under attack. Otherwise, the access time to device under attack is the sum of the training and test set capture times.

7.2 Message recovery attack

At the profiling stage, we used 100K traces captured from the profiling device, D_P, to train neural networks for recovering message bits. As described in Section 5, we train neural networks on the segment of traces corresponding to the first iterations of

Table 3: Time for capturing the training and test sets.

Test set captured from D_A	# Shares	Time for capturing 24 traces with N repetitions (mins)				
		1	3	5	7	9
	3	01:16	03:02	04:47	06:33	08:19
	4	01:37	04:05	06:32	09:01	11:29

Training set captured from D_P	# Shares	Time for capturing 100K traces
	3	8.82 hrs
	4	12.13 hrs

Table 4: Empirical probability to recover a message bit using traces captured without repetitions from the second-order masked implementation on D_P.

2nd-order D_P	Bit position in byte								**average**
Byte	0	1	2	3	4	5	6	7	
0	0.951	0.919	0.981	0.985	0.990	0.986	0.988	0.991	0.974
1	0.984	0.981	0.989	0.991	0.991	0.990	0.987	0.982	0.987
2	0.985	0.980	0.985	0.992	0.982	0.986	0.985	0.984	0.985
3	0.983	0.984	0.982	0.988	0.986	0.989	0.951	0.966	0.979
average	0.976	0.966	0.984	0.989	0.987	0.988	0.978	0.981	**0.981**

Table 5: Empirical probability to recover a message bit using traces captured without repetitions from the second-order masked implementation on D_A.

2nd-order D_A	Bit position in byte								**average**
Byte	0	1	2	3	4	5	6	7	
0	0.925	0.896	0.954	0.971	0.946	0.975	0.958	0.979	0.951
1	0.975	0.979	0.971	0.958	0.975	0.967	0.963	0.954	0.968
2	0.967	0.988	0.963	0.971	0.967	0.975	0.950	0.979	0.970
3	0.967	0.958	0.954	0.967	0.971	0.979	0.929	0.887	0.952
average	0.9585	0.9553	0.9605	0.967	0.965	0.974	0.950	0.949	**0.960**

Table 6: Empirical probability to recover a message bit using traces captured without repetitions from the third-order masked implementation on D_P.

3rd-order D_P	Bit position in byte								**average**
Byte	0	1	2	3	4	5	6	7	
0	0.954	0.981	0.960	0.954	0.979	0.979	0.980	0.977	0.971
1	0.983	0.985	0.984	0.983	0.977	0.984	0.986	0.983	0.983
2	0.982	0.980	0.971	0.975	0.980	0.976	0.967	0.976	0.976
3	0.977	0.977	0.976	0.974	0.979	0.976	0.938	0.945	0.968
average	0.974	0.981	0.973	0.972	0.979	0.979	0.968	0.970	**0.974**

`A2B_bitsliced_msg()` procedure. Thanks to the similarity of traces of all iterations, and standardization of traces, the model trained on the ith message bit of the first iteration is also capable to recover the bits $32b + i$, for all $i \in \{0, 1, \ldots, 31\}$ and $b \in \{1, 2, \ldots, 7\}$.

We train one universal model for bits 1 to 30, $\mathcal{N}_{1-30}$, and two specialized models for the corner bits 0 and 31, $\mathcal{N}_0$ and $\mathcal{N}_{31}$, respectively. For each of the three types, we trained nine different models to be used in an ensemble at the attack stage. The ensemble approach [GBC16] is well-known to be beneficial for side-channel attacks. The models for the bits 0 and 31 were trained on 100K traces. The models for the bits 1 to 30 were trained on 3M traces composed using the cut-and-join technique of [NDJ21].

Tables 4 and 5 list the average empirical probabilities to recover a message bit from

Table 7: Empirical probability to recover a message bit using traces captured without repetitions from the third-order masked implementation on D_A.

3rd-order D_A	Bit position in byte								**average**
Byte	0	1	2	3	4	5	6	7	
0	0.850	0.971	0.942	0.938	0.971	0.975	0.963	0.950	0.945
1	0.967	0.963	0.950	0.971	0.975	0.975	0.946	0.979	0.966
2	0.908	0.942	0.975	0.971	0.967	0.912	0.954	0.929	0.945
3	0.983	0.925	0.963	0.954	0.946	0.946	0.958	0.679	0.919
average	0.927	0.950	0.957	0.958	0.965	0.952	0.955	0.884	**0.944**

Table 8: Empirical probability to recover a message bit using traces captured with N repetitions from D_A.

Implementation	Number of repetitions N			
	3	5	7	9
2nd-order	0.982	0.993	0.996	0.998
3rd-order	0.974	0.988	0.990	0.996

traces captured without repetitions from the second-order masked implementation running on the same as profiling device and on a different device, respectively, for the first 32 message bits. Tables 6 and 7 present the corresponding numbers for the third-order masked implementation. We include test results for the profiling device to illustrate the impact of intra-device variability on neural network's prediction accuracy.

As expected, the third-order masked implementation is more difficult to break that the second-order one. However, the average drop in prediction accuracy is only 0.7-1.6%. This justifies our hypothesis that the complexity of the attack grows linearly with the number of shares.

We can also see from the tables that, when traces are captured without repetitions, the models cannot predict message bits with a sufficiently high accuracy. An accuracy of at least 0.9973% per bit is required to recover a complete message with the probability higher than 50%, since $0.9974^{256} = 0.5005$. For this reason, in the secret key recovery attack presented in the next section, we use the ECC-based approach from [NDGJ21] which can correct single-bit errors and detect double-bit errors in every coefficient of the secret key. In addition, we raise the number of repetitions of the same measurement to $N = 9$. This helps us boost the average probability of recovering a message bit from a different device to 0.998 for the second-order masked implementation, see Table 8. From Fig. 14 we can see that, for the average message bit prediction accuracy of 0.998, we may expect to recover the secret key with the enumeration degree $K = 1$, i.e. using 9 enumerations only.

7.3 Secret key recovery attack

The secret key recovery attack follows the procedure presented in Section 6.3. First, the messages $m_1, \ldots, m_{24}$ contained in the chosen ciphertexts $c_1, \ldots, c_{24}$ are recovered from traces captured with N repetitions. Then, 768 coefficients of the secret key are derived from these messages using the mapping in Table 2.

Tables 9 and 10 show the results for the second- and third-order masked implementations running on the device under attack D_A, respectively for different number of repetitions $N \in \{1, 3, 5, 7, 9\}$. There are four possible outcomes. The first one is when the ground truth secret key coefficient and the recovered coefficient agree (the column "No errors"). In the second case, there is one error in the eight message bits from which the coefficient

Table 9: The empirical distribution of errors in 768 coefficients of the secret key recovered using $24 \times N$ traces captured from the second-order masked implementation on D_A.

2nd-order D_A	# Correct predictions		# Errors	
N	No errors	Errors corrected by ECC	Detected errors	Undetected errors
1	424	232	99	13
3	670	83	12	3
5	731	32	5	0
7	746	22	0	0
9	757	10	1	0

Table 10: The empirical distribution of errors in 768 coefficients of the secret key recovered using $24 \times N$ traces captured from the third-order masked implementation on D_A.

3rd-order D_A	# Correct predictions		# Errors	
N	No errors	Errors corrected by ECC	Detected errors	Undetected errors
1	384	275	98	11
3	613	113	18	3
5	714	43	10	1
7	724	33	9	2
9	751	13	4	0

is derived according to the mapping in Table 2. The ECC corrects this error (the 3rd column). The third case is when more than one error is detected by the ECC in the eight message bits and this combination of bits is not in Table 2. The ECC detects these faults (the 4th column). The fourth case is when the eight message bits are combined as shown in Table 2, but the recovered coefficient differs from the ground truth secret key coefficient (the last column).

In the fourth case, the key recovery attack fails since the location of the incorrectly recovered coefficient is unknown. Contrary, in the third case, if the number of detected errors is small, they can be fixed since the relation between the public key and the secret key is known:

$$\mathbf{b} = ((\mathbf{A}^T\mathbf{s} + \mathbf{h}) \mod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}.$$

So, one can either employ a post-processing step with lattice reduction, or use enumeration. The complexity of enumeration is 9^K, where K is the number of detected errors.

We can see from Tables 9 and 10 that, for the second-order masked implementation, the secret key can be successfully recovered with $N = 5$ and 9^5 enumerations[3]. For the third-order masked implementation, $N = 9$ and 9^4 enumerations are required for key recovery. This means that, if profiling is done on a different device D_P, the required access time to device under attack is 4.47 min and 11.29 min for the second- and third-order masked implementations, respectively (see Table 3).

8 Conclusion

We demonstrated the first side-channel attack on a higher-order masked Saber KEM implementation which exploits a vulnerability in the `A2B_bitsliced_msg()` procedure for the arithmetic to Boolean conversion. The presented attack is not specific for Saber KEM.

[3]We believe that the inconsistency of $N = 7$ and $N = 9$ cases in Table 9 is due to the small size of our test set.

It can potentially be applied to any LWE/LWR-based PKE/KEM algorithm which uses a similar to `A2B_bitsliced_msg()` implementation of the arithmetic to Boolean conversion. Furthermore, other procedures which perform bit-wise storage of sensitive variables in memory might also be vulnerable.

Our work shows that even a higher-order masking may fail to protect software implementations of cryptogrpahic algorithms from side-channel attacks. We believe that masking is more suitable for protecting hardware implementations, where the shares are processed in parallel. For software implementations, one may try to increase the attack difficulty by accumulating n bits of a sensitive variable into a register before writing them in memory. The more bits are accumulated, the harder it is to extract the sensitive variable. Note, however, that accumulating a byte is not enough because 2^8 a small number and thus the multiple-bit injection method from [WND22a] can potentially be applied to extract bytes. A rule of thumb is that any n for which a good 2^n-class neural network model can be trained is too small. As with the secret key size, the bound on the required size of n is likely to increase with further advances in deep learning.

Future work includes analysing hardware implementations of LWE/LWR-based PKE/KEMs and designing deep learning-resistant countermeasures.

9 Acknowledgments

This work was supported in part by the Swedish Civil Contingencies Agency (Grant No. 2020-11632) and the Swedish Research Council (Grant No. 2018-04482).

References

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. Cryptology ePrint Archive, Paper 2021/104, 2021. https://eprint.iacr.org/2021/104.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(2):1–26, 2021.

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

[C+20] C. Chen et al. NTRU algorithm specifications and supporting documentation, 2020. https://csrc.nist.gov/projects/postquantum-cryptography/round-3-submissions.

[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.

[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.

[DBV22] Jan-Pieter D'Anvers, Michiel Van Beirendonck, and Ingrid Verbauwhede. Revisiting higher-order masked comparison for lattice-based cryptography: Algorithms and bit-sliced implementations. Cryptology ePrint Archive, Paper 2022/110, 2022. https://eprint.iacr.org/2022/110.

[Doz16] Timothy Dozat. Incorporating Nesterov momentum into Adam. 2016.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.

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

[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. Cryptology ePrint Archive, Paper 2020/743, 2020. https://eprint.iacr.org/2020/743.

[HKL+22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. First-order masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2022/058, 2022. https://eprint.iacr.org/2022/058.

[KDB+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked Saber. Cryptology ePrint Archive, Paper 2022/389, 2022. https://eprint.iacr.org/2022/389.

[MBM+22] Catinca Mujdei, Arthur Beckers, Jose Maria Bermudo Mera, Angshuman Karmakar, Lennert Wouters, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. Cryptology ePrint Archive, Paper 2022/474, 2022. https://eprint.iacr.org/2022/474.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure Saber KEM implementation. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pages 676–707, 2021.

[NDJ21] Kalle Ngo, Elena Dubrova, and Thomas Johansson. Breaking masked and shuffled CCA secure Saber KEM by power analysis. In *Proc. of the 5th Workshop on Attacks and Solutions in Hardware Security*, pages 51–61, 2021.

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

[NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[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.

[RSRCB20] 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, Jun. 2020.

[S+20] P. Schwabe et al. CRYSTALS-Kyber algorithm specifications and supporting documentation, 2020. https://csrc.nist.gov/projects/postquantum-cryptography/round-3-submissions.

[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.

[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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):296–322, Nov. 2021.

[WCCL22] Jian Wang, Weiqiong Cao, Hua Chen, and Haoyuan Li. Practical side-channel attack on masked message encoding in latticed-based KEM. Cryptology ePrint Archive, Paper 2022/859, 2022. https://eprint.iacr.org/2022/859.

[Wel47] Bernard L Welch. The generalization of 'student's'problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[WND22a] Ruize Wang, Kalle Ngo, and Elena Dubrova. Making biased DL models work: Message and key recovery attacks on Saber using amplitude-modulated EM emanations. Cryptology ePrint Archive, Paper 2022/852, 2022. https://eprint.iacr.org/2022/852.

[WND22b] Ruize Wang, Kalle Ngo, and Elena Dubrova. Side-channel analysis of Saber KEM using amplitude-modulated EM emanations. In *Proc. of 25th Euromicro Conference on Digital System Design (DSD)*, 2022. https://eprint.iacr.org/2022/807.

[XPSR+21] Zhuang Xu, Owen Michael Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of Kyber. *IEEE Transactions on Computers*, pages 1–1, 2021.

[ZC18] Alice Zheng and Amanda Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. O'Reilly Media, Inc., 2018.