A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM

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

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

Abstract. In this paper, we present the first side-channel attack on a first-order masked implementation of IND-CCA secure Saber KEM. We show how to recover both the session key and the long-term secret key from 16 traces by deep learning-based power analysis without explicitly extracting the random mask at each execution. Since the presented method is not dependent on the mask, we can improve success probability by combining score vectors of multiple traces captured for the same ciphertext. This is an important advantage over previous attacks on LWE/LWR-based KEMs, which must rely on a single trace. Another advantage is that the presented method does not require a profiling device with deactivated countermeasure, or known secret key. Thus, if a device under attack is accessible, it can be used for profiling. This typically maximizes the classification accuracy of deep learning models. In addition, we discovered a leakage point in the primitive for masked logical shifting on arithmetic shares which has not been known before. We also present a new approach for secret key recovery, using maps from error-correcting codes. This approach can compensate for some errors in the recovered message.

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

1 Introduction

Public-key cryptographic schemes in current use depend on the intractability of specific mathematical problems such as integer factorization or the discrete logarithm problem. However, it is known that when large-scale quantum computers become a reality, factoring and discrete log can be efficiently solved using the Shor algorithm [Sho99]. Even if it will take many years until large-scale quantum computers are available, the need for long term security (what we protect today must remain secure also in 10 years from now) makes this an issue that needs immediate attention.

In response to this situation, the National Institute of Standards and Technology (NIST) started a few years ago a project for standardizing post-quantum cryptographic primitives (NIST PQ standardization project). The candidate primitives in this project rely on problems that are not known to be solvable by a quantum computer. The two most common areas for such problems are lattices problems and decoding problems for error correcting codes. In round 1, security was the main focus in evaluation, whereas round 2 considered implementation aspects to a larger extent. The project recently entered round 3, where it is expected that security in relation to side-channel attacks will have a larger focus.

As mentioned, lattice-based cryptography is perhaps the most promising areas in post-quantum crypto. The remaining candidates in round 3 are split into two subsets, the finalists and the alternates. Among the finalists for the primitive key encapsulation mechanism (KEM), 3 out of 4 finalists are lattice-based (and one more among the alternates).

Among lattice-based schemes one may further split into several categories: NTRU-based schemes with finalist NTRU [C+20]; Learning With Errors (LWE)-based schemes with finalist Kyber [S+20]; and the Learning With Rounding (LWR)-based schemes with finalist Saber [D+20]. The hardness in these problems comes from inserting unknown noise into otherwise linear equations.

Side-channel attacks were introduced by Kocher [KJJ99] and are today considered as the main threat against implementations of cryptographic algorithms, in particular for applications in embedded devices. Side-channel attacks exploit information obtained from physically measurable alternative channels and the most common ones are timing measurements and the measured power consumption of a device. Side-channel attacks and the corresponding countermeasures have been a major area of research for many years now, often targeting cryptographic standards. A more recent sub-area is the investigation of side-channel attacks for post-quantum cryptography. This is getting increasing attention in the research community, in particular in connection with the NIST PQ standardization project. The analysis and protection against side-channel attacks for the round 3 finalist candidates is an urgent area to explore.

The first and most basic form of side-channel analysis and protection is obtained by considering the timing channel, simply measuring the execution time of software implementations of the cryptographic algorithms. The general protection method is to make implementations such that they all run in *constant time*. This is today a standard assumption for software implementations. The timing channel can be extended to consider cache-timing attacks, where time variation due to memory management in the executing device is considered. A typical example of an exploit is the use of look-up tables.

Even with constant time implementations and avoiding implementation weaknesses such as the use of look-up tables, a software implementation is still vulnerable to attacks if power measurements from the CPU can be used. Additional protection measures need to be considered and the main tools are such techniques as masking and shuffling.

A fully side-channel protected implementation of a lattice-based cryptosystem was first to proposed in [RRVV15] followed by [RdCR+16], based on masking. It should be noted that masking involves doing linear operations twice, whereas non-linear operations calls for more complex solutions which decrease the speed even more. The masked implementation in [RRVV15] increases the number of CPU cycles on an ARM Cortex-M4 by a factor more than 5 compared to a non-protected implementation.

Whereas these protection attempts consider Chosen-Plaintext Attack (CPA)-secure lattice schemes, it is more interesting to consider to secure primitives designed to withstand Chosen-Ciphertext Attacks (CCA). CCA secure primitives are usually obtained through a transform and a CPA secure primitive. The most common transformation is the Fujisaki-Okamoto (FO) transform or some variation of it [HHK17]. The CCA-transform is itself susceptible to side-channel attacks and should be masked [RRCB20]. Examples of recent masked implementations are: [OSPG18] of a KEM similar to NewHope; and [BBE+18, MGTF19, GR19] on different lattice-based signature schemes.

Narrowing in on the NIST round 3 finalists, only the candidate Saber has an associated protected software implementation available [BDK+20]. Saber is a Module-LWR-based KEM that is a finalist in the third round of the NIST PQ standardization project. LWR means that noise is added through rounding instead of adding explicit error terms as for LWE.

In [BDK+20] the authors construct a first-order masked implementation of the Saber CCA-secure decapsulation algorithm that comes with an overhead factor of only 2.5 compared to the unmasked implementation. It is claimed that this side-channel secure version can be built with relatively simple building blocks compared to other candidates, resulting in a small overhead for side-channel protection. The masked implementation of Saber is based on masked logical shifting on arithmetic shares and a masked binomial sampler. The work includes experimental validation of the implementation to confirm suppression of side-channel leakage on the Cortex-M4 general-purpose processor.

Side-channel attacks on the unprotected implementations of NIST PQ standardization project candidates have been considered in some recent papers. In [SKL$^+$20] a message recovery attack (session key recovery) was described using a single trace 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 EM side-channel approach. In particular, a mechanism of turning a message recovery attack to a secret key recovery attack was proposed, giving a secret key recovery using e.g. 184 traces for 98% success rate. In [GJN20] similar ideas for timing attacks were considered.

In the very recently posted paper [RBRC20][1], 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 and because only a single trace is available for each unknown mask value, the success rate in message bit recovery for such a two-step approach may be far from 100%.

Contributions: In this paper, we present the first side-channel attack on a masked implementation of IND-CCA secure Saber KEM. It does not require a profiling device with deactivated countermeasure as in previous attacks on masked implementations of LWE/LWR-based KEMs [RBRC20, SKL$^+$20]. We show how to recover both the session key and the long-term secret key, by deep learning-based power analysis using a small number of traces without explicitly extracting the random mask at each execution. Since the presented method is independent of the mask, we can do error correction by capturing multiple traces for the same input. This is an important advantage over the attacks in [RBRC20, SKL$^+$20] which *must* succeed from a single trace. The independence of the mask also makes it possible to use the device under attack for capturing traces for the profiling stage, if it is accessible for a sufficiently long time. This typically maximizes classification accuracy of deep learning models created at the profiling stage. We also present a new approach for secret key recovery, using maps from error-correcting codes. This approach can compensate for some errors in the recovered message.

The remainder of this paper is organized as follows. In Section 2 we give the necessary background both on Saber and on the use of deep learning in side-channel attacks. In Section 3 we describe the main part of the work, which is a message recovery attack on the decryption/decapsulation algorithm. In Section 4 we subsequently show how an attack recovering the long-term secret key can be done, using the message recovery attack from the previous section. Section 5 concludes the paper and describes future work.

2 Background

This section provides background information on the Saber algorithm, the masked implementation of Saber from [BDK$^+$20], profiled side-channel attacks, and Test Vector Leakage Assessment (TVLA).

2.1 SABER algorithm

Saber [D$^+$20] is a finalist candidate in the NIST PQ standardization project, where the security is based on the hardness of the Module Learning with Rounding problem (MLWR).

[1]The design of the attack and most of the experimental work in this paper were done before the posting of [RBRC20].

```
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: $r = \mathcal{U}(\{0,1\}^{256})$
4: $\mathbf{s} = \beta_\mu(R_q^{l\times 1}; r)$
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:     $r = \mathcal{U}(\{0,1\}^{256})$
4: end if
5: $\mathbf{s}' = \beta_\mu(R_q^{l\times 1}; r)$
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 from [D+20].

It starts with an IND-CPA secure encryption scheme, Saber.PKE, and then presents an IND-CCA secure key encapsulation mechanism (KEM), Saber.KEM, which is transformed from Saber.PKE through a version of the FO transform. Algorithms Saber.PKE and Saber.KEM are described in Fig. 1 and 2, respectively.

We now introduce some notations used in the description of Saber. Let $\mathbb{Z}_q$ denote the ring of integers modulo a positive integer q and R_q the quotient ring $\mathbb{Z}_q[X]/(X^n + 1)$. Saber sets $n = 256$. The rank of the module is denoted by l and it increases for a higher security level.

In Saber, 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. We use $x \leftarrow \chi(S)$ to denote sampling from χ, if χ is a distribution over a set S. The notation S can be omitted, i.e., we write $x \leftarrow \chi$, if there is no ambiguity.

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

The bitwise right shift operation is denoted by $\gg$ and can be extended to polynomials and matrices by applying it coefficient-wise. Saber also includes three constants to efficient implement rounding operations by a simple bit shift, i.e., two constant polynomials $h_1 \in R_q$ and $h_2 \in R_q$ with all coefficients set 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 $\mathbf{h} \in R_q^{l\times 1}$ with each polynomial set equal to h_1.

Three parameter sets (see Table 1) are proposed in the round 3 Saber document, i.e., LightSaber, Saber, and FireSaber, aiming for the security levels of NIST-I, NIST-III, and NIST-V, respectively. These parameter sets achieve decryption failure probabilities bounded by 2^{-120}, 2^{-136}, and 2^{-165}, respectively. For a more detailed description of the different parts of Saber, we refer to the design document [D+20].

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

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 $(c, K)$

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:     return $K = \mathcal{H}(\hat{K}', c)$
6: else
7:     return $K = \mathcal{H}(z, c)$
8: end if
```

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

Table 1: Proposed parameters of round-3 Saber.

	l	n	q	p	T	μ	security	p_{fail}
LightSaber	2	256	2^{13}	2^{10}	2^3	10	NIST-I	2^{-120}
Saber	3	256	2^{13}	2^{10}	2^4	8	NIST-III	2^{-136}
FireSaber	4	256	2^{13}	2^{10}	2^6	6	NIST-V	2^{-165}

2.2 Masked Saber KEM

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 "$\oplus$" in 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 moduli 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 and an adapted masked binomial sampler from [SPOG19]. A particular attention is devoted in [BDK+20] to the protection of the decapsulation algorithm, Saber.KEM.Decaps(), since it involves operations with the long-term secret key $\mathbf{s}$.

2.3 Profiled side-channel attacks

A profiled side-channel attack is performed in two stages: profiling and attack. Profiling can be done by creating a template [APSQ06, CPM+18, HGA+19], or training a model, e.g. an artificial neural network [MPP16, CDP17, KPH+19, BFD20].

If artificial neural networks are used, then at the profiling stage a network is trained to learn the leakage "profile" of the target device for all possible values of the sensitive variable. The training is done using a large number of traces captured from the profiling device, which are labelled according to the selected leakage model (e.g. Hamming weight, Hamming distance, identity, etc). Afterwards, at the attack stage, the trained network is used to classify traces captured from the device under attack (which may be the same or different from the profiling device).

2.4 Test vector leakage assessment

The Test Vector Leakage Assessment (TVLA) introduced by Goodwill et al. [GJJR11] is a popular statistical technique which is used as a metric for evaluating side-channel leakage and as a tool for feature extraction from side-channel measurements [RJJ+18, RRCB20, SKL+20].

TVLA applies the Welch's t-test to find differences between two sets of side-channel measurements. The t-test takes a sample from each of the two sets and establishes whether they differ by assuming a null hypothesis that the means of two sets are equal.

The TVLA of two sets of measurements $\mathcal{T}_0$ and $\mathcal{T}_1$ is carried out as follows:

$$TVLA = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}}}$$

where μ_i, σ_i and n_i stand for mean, standard deviation and cardinality of the set $\mathcal{T}_i$, for $i \in \{0, 1\}$. The null hypothesis is rejected with a confidence of 99.9999% only if the absolute value of the t-test score is greater than 4.5 [GJJR11]. A rejected null hypothesis means that the two data sets are noticeably different and thus might leak some information.

In this work, we use TVLA for a posteriori analysis of side-channel measurements.

3 Message recovery attack

First, we present an attack which recovers a message from traces captured during the execution of Saber.KEM.Decaps() by the device under attack. Later, in Section 4.2, we show how both the long-term secret key can be extracted from a few recovered message.

3.1 Main idea

Side-channel attacks aiming to extract a secret $\mathcal{S}$ from a set of side-channel measurements $\mathcal{T}$ captured from a masked implementation of an algorithm $\mathcal{A}$ face two problems:

1. How to find points in $\mathcal{T}$ which leak information about $\mathcal{S}$?

2. How to recover $\mathcal{S}$ without knowing the value of the mask at each execution of $\mathcal{A}$?

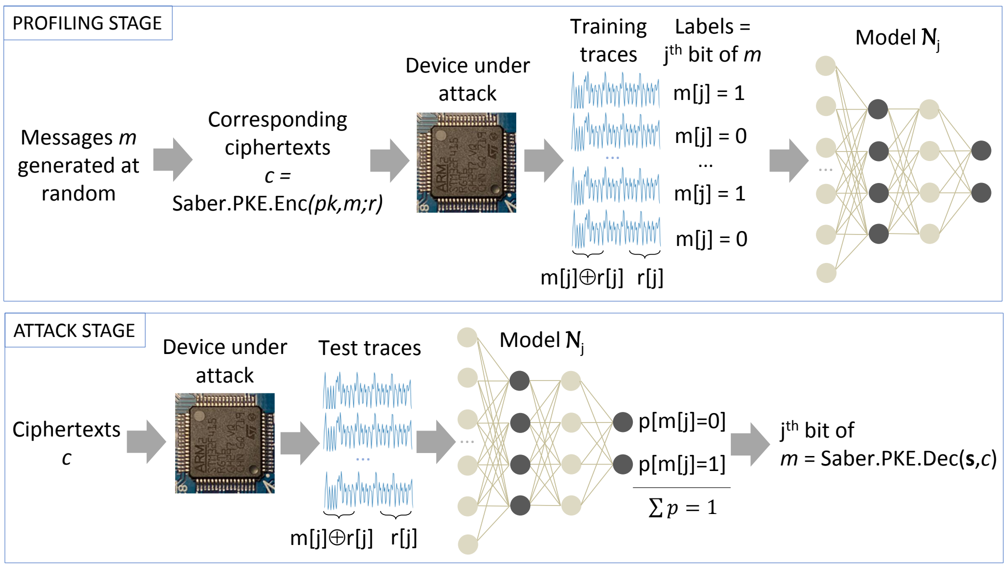

Figure 3: Profiling and attack stages of the presented message recovery attack.

3.1.1 Finding leakage points

The attacks on non-masked implementations [RRCB20, SKL+20] solve the problem (1) by identifying leakage points in side-channel measurements using, for example, TVLA [GJJR11], or Correlation Power Analysis (CPA) [BCO04]. However, such an approach does not apply to masked implementations because the value of the random mask at each execution of $\mathcal{A}$ is unknown.

Previous works addressing masked LWE/LWR-based KEMs [RBRC20, SKL+20] suggest to solve the problem (1) by first deactivating the masking countermeasure, or fixing the mask to a constant, and then finding leakage points as in the non-masked case. However, in order to deactivate the countermeasure, or fix the mask, one requires the implementation source code of the algorithm under attack. The source code may be proprietary. In addition, a modified source code may be optimized differently by the compiler due to the changes made to deactivate the countermeasure. This might change the shape of power traces, as we show in Section 3.5.4.

We solve the problem (1) using a deep learning method which works without explicitly extracting the random mask at each execution. We first hypothesize an approximate location of the leakage point in a trace based on knowledge of the algorithm under attack and through experience gained in power analysis of its non-masked implementations. Then, we verify each hypothesis by training a deep learning model on an interval of trace covering the selected point. If the model learns with a high accuracy, the point is assumed to leak. Otherwise, we shift the interval window and repeat the training. If all shift attempts fail, the hypothesis is rejected.

3.1.2 Recovering message without knowing the mask

In the previous work on LWE/LWR-based KEMs, the problem (2) is addressed by either constructing a template [RBRC20], or training a deep learning model [SKL+20] on power/EM traces from a profiling device in which the masking countermeasure is deactivated, or the mask r is fixed to a constant. Profiling aims to distinguish the difference between cases when a message bit value of "0" is processed at the leakage point; from the case when a message bit takes the value of "1".

During the attack, traces captured from a device under attack are given as input to the template/model to separately recover the jth bit of the shares $m \oplus r$ and r, for all $j \in \{0, 1, \ldots, 255\}$. Finally, the message is computed as $m = r \oplus (m \oplus r)$. Note that such two-phase attacks *must* recover the message from a single power/EM trace because a new random mask r is generated for each execution of the algorithm.

We show that it is possible to train an accurate deep learning model capable to recover message bits from a masked LWE/LWR-based KEM directly, *without explicitly knowing or manipulating the value of the mask*. At the profiling stage, a model for the bit j, $\mathcal{N}_j$, is trained on traces containing jth bits of both shares, $m[j] \oplus r[j]$ and $r[j]$, and labelled by the value of the message bit $m[j]$ (see Fig. 3). At the attack stage, the model $\mathcal{N}_j$ takes an interval of trace containing $m[j] \oplus r[j]$ and $r[j]$, and performs processing equivalent to recognizing their values from the shape of power traces and doing a logic operation, XOR, on them. A similar strategy is used in [MPP16] for extracting the secret key from a masked implementation of AES except that we use the message bit values as a leakage model, while in [MPP16] the Hamming weight of S-Box output is used as a leakage model.

Another difference is that, since public-key encryption is performed using the public key, for LWE/LWR-based KEMs we can pre-compute a set of ciphertexts corresponding to any set of messages (random or chosen). Therefore, if the device under attack is accessible, we can use it to capture training traces for the profiling stage. This is clearly not possible in the case of symmetric encryption algorithms since they use the secret key for both encryption and decryption. Using the device under attack for profiling is advantageous because the deep learning model's classification accuracy does not deteriorate due to differences in training and test traces caused by manufacturing process variation [WBFD19].

An advantage of our attack over the two-phase attacks on masked LWE/LWR-based KEMs [RBRC20, SKL$^+$20] is that we can improve message recovery probability by combining score vectors of multiple traces captured for the same ciphertext. The two-phase attacks must rely on a single trace.

3.2 Assumptions

We assume that the adversary knows that the device under attack implements the Saber algorithm and has physical access to the device under attack to capture power traces. We consider two scenarios:

1: The access time is sufficient to capture both, training traces for the profiling stage and test traces for the attack stage. In this case, the device under attack can be used for profiling.

2: The access time is sufficient to capture test traces for the attack stage only. In this case, a different device, identical to the device under attack is required for profiling.

3.3 Trace acquisition

Next, we describe equipment for trace acquisition and how leakage points are located.

3.3.1 Equipment

Our measurement setup is shown in Fig. 4. It consists of the ChipWhisperer-Lite board, the CW308 UFO board and CW308T-STM32F4 target board.

The ChipWhisperer is a hardware security evaluation toolkit based on a low-cost open hardware platform and an open-source software [New]. The ChipWhisperer-Lite board can be used to measure power consumption and control the communication between the target device and the computer. The power is measured over the shunt resistor placed between the power supply and the target device. ChipWhisperer-Lite uses a synchronous capture

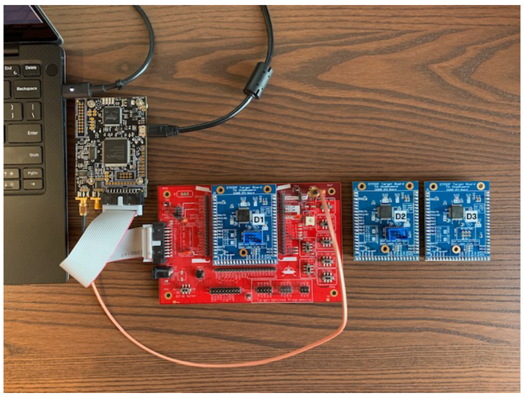

Figure 4: Equipment for trace acquisition.

method which greatly improves the synchronization of traces and reduces the required sample rate and the data storage. The maximum sampling rate of ChipWhisperer-Lite board is 105 MS/sec and the 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 masked Saber from [BDK+20]. The implementation is compiled with `arm-none-eabi-gcc` using the highest compiler optimization level -O3 (recommended default) which is typically the most difficult to break by side-channel analysis [SKL+20].

The target board is run at 24 MHz and sampled either at 24 MHz (1 pt/clock cycle), or at 96 MHz (4 pt/clock cycle) depending on the experiment.

3.3.2 Locating leakage points

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 (line 2 of Saber.PKE.Decryt() at Fig. 1) iteratively maps each polynomial coefficient into a corresponding message bit, thus computing the decrypted message one bit at a time.

It was observed in [RBRC20] that, in a non-masked implementations of the decoding function (see `indcpa_kem_dec()` at Fig. 6), there are two points containing exploitable Incremental-Storage vulnerability. The first one is at line 8 of `indcpa_kem_dec()` where the message bits $m[j]$ are computed and stored in a 16-bit memory location `v[i]` in an unpacked fashion. Since `v[i]` can take only two possible values, 0 or 1, an attacker can recover the message bit $m[j]$ by distinguishing between 0 and 1. The second point, located at line 4 of `POL2MSG()` procedure where the decoded message bits are packed into a byte array in memory.

By examining the masked implementation of the decoding function from [BDK+20] shown as `indcpa_kem_dec_masked()` in Fig. 6, we can see that the same procedure `POL2MSG()` as in `indcpa_kem_dec()` is used for packing the decrypted message shares. As we demonstrate in Section 3.5, Incremental-Storage vulnerability can still be exploited with our deep learning approach despite the message being partitioned into shares. Furthermore, we found that `poly_A2A()`, which is a primitive designed in [BDK+20] for masked logical shifting on arithmetic shares, also contains a point with an exploitable Incremental-Storage

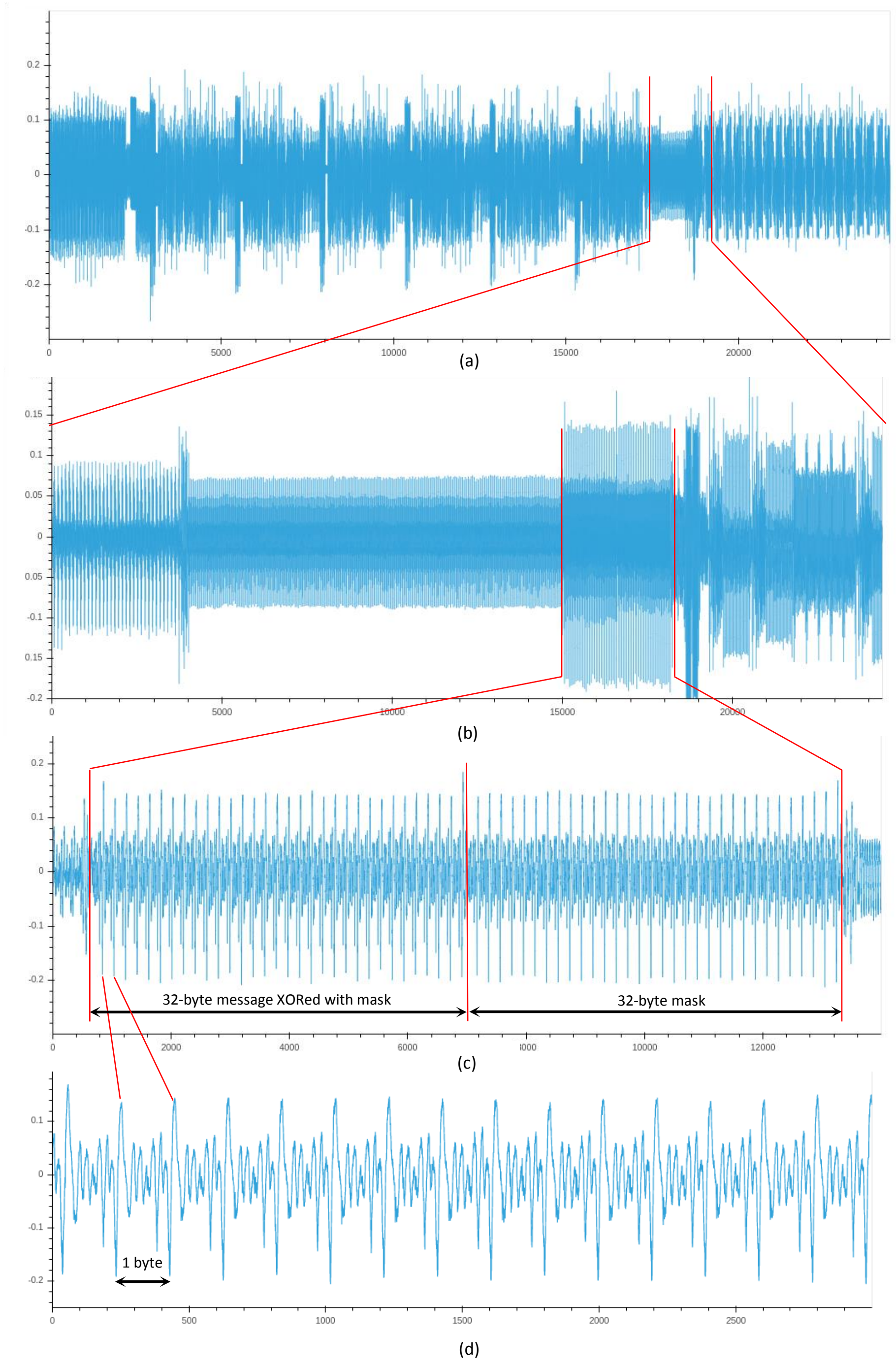

Figure 5: Locating `POL2MSG()`: (a) The initial part of Saber.KEM.Decaps() sampled 1 pt/clock cycle with decimation 15; (b) Zoomed part of (a) sampled 1 pt/clock cycle with decimation 1; (c) Part of (b) representing two `POL2MSG()` executions sampled 4 pt/clock cycle with decimation 1; (d) Zoomed part of (c) containing the first 15 bytes of message XORed with mask.

```
void indcpa_kem_dec(char *sk,
char *ct, char m[])
uint16_t v[N];
uint16_t sksv[K][N];
BS2POLVECq(sk,sksv);
SABER_un_pack(&ct, v);
for (i = 0; i < N; ++i) do
  v[i] = h2-(v[i]≪(EP-ET));
end for
VectorMul(ciphertext,sksv,v);
for (i = 0; i < N; ++i) do
  v[i] = (v[i]&(P-1))≫(EP-1);
end for
/* pack decrypted message */
POL2MSG(v,m);
```

```
void POL2MSG(uint16_t *v, chair *m)
for (j = 0; j < BYTES; j++) do
  m[j] = 0;
  for (i = 0; i < 8; i++) do
    m[j] = m[j]|(v[8*j+i]≪i);
  end for
end for
```

```
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
```

Figure 6: C code of non-masked and masked implementations of Saber.PKE.Dec() from [BDK+20].

vulnerability. This leakage point was not known before.

First we explain how we located the position of `POL2MSG()` in traces. Fig. 5(a) shows a trace representing the initial part of Saber.KEM.Decaps(). The trace is obtained by averaging 50,000 traces captured for random ciphertexts. Since `POL2MSG()` packs the message bits into a byte array, we expect its trace to look like a block of repeating, similar patterns. The segment of Fig. 5(a) marked by two red lines is a possible candidate. Fig. 5(b) shows its zoomed version. By further zooming into the interval of Fig. 5(b) marked by the red lines, we can distinguish 64 repeating patterns representing the processing of bytes. Fig. 5(d) gives a more detailed view of one byte processing; it takes 196 samples.

Since `poly_A2A()` is executed immediately before `POL2MSG()`, we can hypothesize it is located in the interval between the points 4,000 and 15,000 in Fig 5(b).

Next we describe how we used deep learning to check if these two intervals contain exploitable leakage points.

3.4 Profiling and attack stages

Let $\boldsymbol{m} = \{m_1, m_2, \ldots, m_t\}$, where $m_i \in \{0,1\}^{256}$, for $i \in \{1, \ldots, t\}$, be a set of messages selected at random. Let $\boldsymbol{c} = \{c_1, c_2, \ldots, c_t\}$, be the set of corresponding ciphertexts $c_i =$ Saber.PKE.Enc$(pk, m_i; r_i)$. Let $\mathcal{T}_i \in \mathbb{R}^l$ be a trace captured from the profiling device D_{pro} during the execution of Saber.KEM.Decaps() with c_i as input, where l is the trace size.

A pseudocode in Fig. 7 describes the profiling and attack stages of the presented message recovery attack. At the profiling stage, a neural network model $\mathcal{N}_j : \mathbb{R}^l \rightarrow \mathbb{I}^2$, where $\mathbb{I} = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$, is trained for a selected message bit j. A network $\mathcal{N}_j$ maps a trace $\mathcal{T}_i \in \mathbb{R}^l$ into a score vector $S_{j,i} = \mathcal{N}_j(\mathcal{T}_i) \in \mathbb{I}^2$ whose elements $s_{j,i,k}$ represent

```
ProfilingStage($D_{pro}, t, l, j$) /* # of training traces $t$, trace size $l$, bit position $j$ */
 1: $\boldsymbol{m} = \{m_i \in \{0,1\}^{256} \mid m_i$ is selected at random, $\forall i \in \{1,\ldots,t\}\}$
 2: $\boldsymbol{c} = \{c_i \in \{0,1\}^{34816} \mid c_i =$ Saber.PKE.Enc$(pk, m_i; r_i), \forall i \in \{1,\ldots,t\}\}$
 3: $\boldsymbol{\mathcal{T}} = \{\mathcal{T}_i \in \mathbb{R}^l \mid \mathcal{T}_i \Leftarrow D_{pro}$[Saber.KEM.Decaps$(c_i)$], $\forall i \in \{1,\ldots,t\}\}$
 4: $L_j = \{l_j(\mathcal{T}_i) \in \{0,1\} \mid l_j(\mathcal{T}_i) = m_i[j], \forall i \in \{1,\ldots,t\}\}$
 5: Train $\mathcal{N}_j : \mathbb{R}^l \to \mathbb{I}^2$ on $(\boldsymbol{\mathcal{T}}, L_j)$
 6: return $\mathcal{N}_j$

AttackStage($D_{att}, \mathcal{N}_j, d, l$)
 1: Select $c \in \{0,1\}^{34816}$
 2: for each $i \in \{0,1,\ldots,d\}$ do
 3:     $\hat{\boldsymbol{\mathcal{T}}} = \{\hat{\mathcal{T}}_i \in \mathbb{R}^l \mid \hat{\mathcal{T}}_i \Leftarrow D_{att}$[Saber.KEM.Decaps$(c)$], $\forall i \in \{1,\ldots,d\}\}$
 4: end for
 5: for each $i \in \{0,1,\ldots,d\}$ do
 6:     $S_{j,i} = \mathcal{N}_j(\hat{\mathcal{T}}_i)$
 7:     $m_i[j] = \arg\max_{k \in \{0,1\}} s_{j,i,k}$
 8: end for
 9: $m[j] = \mathrm{maj}_{i=1}^{d}\, m_i[j]$
10: return $m[j]$
```

Figure 7: Description of the profiling and attack stages of the presented message recovery attack on masked Saber PKE; D_{pro} is the profiling device and D_{att} is the device under attack (can be the same instance).

Table 2: The MLP architecture used in the message recovery attack; l = 7,000 and 600 for `POL2MSG()` and `poly_A2A()` leakage points, respectively.

Layer type	(Input, output) shape	# Parameters
Batch Normalization 1	(l, l)	28000
Dense 1	(l, 4)	28004
Batch Normalization 2	(4, 4)	16
ReLU	(4, 4)	0
Dense 2	(4, 2)	10
Softmax	(2, 2)	0
Total parameters:	56,030	
Trainable parameters:	42,022	

the probability that the jth bit of m_i, $m_i[j]$, is equal to $k = \{0,1\}$ in trace $\mathcal{T}_i$. We use the multilayer perceptron (MLP) architecture listed in Table 2.

To train a network $\mathcal{N}_j$, each trace $\mathcal{T}_i$ in the training set $\boldsymbol{\mathcal{T}}$ is assigned a label $l_j(\mathcal{T}_i) = m_i[j]$, for $i \in \{1,\ldots,t\}$. We use 70% of the set $\boldsymbol{\mathcal{T}}$ for training and 30% for validation. Categorical cross-entropy is used as a loss function. No input normalization is applied. Nadam optimizer (an extension of RMSprop with Nesterov momentum) with the learning rate 0.0001 and numerical stability constant epsilon=1e-08 is used. The training is carried out for a maximum of 100 epochs with batch size 10 and early stopping.

Since the MLP architecture in Table 2 is very simple, the training time is short, e.g. for $|\boldsymbol{\mathcal{T}}| = 100,000$ and trace size $l = 7,000$, training a model takes less than 5 minutes on average on a quad-core 4.0 GHz PC.

At the attack stage, d traces $\hat{\boldsymbol{\mathcal{T}}} = \{\hat{\mathcal{T}}_1, \ldots, \hat{\mathcal{T}}_d\}$ are captured from the device under attack D_{att} for the same ciphertext c. To recover a message bit $m[j]$, the model $\mathcal{N}_j$ is used to compute score vectors $S_{j,i} = \mathcal{N}_j(\hat{\mathcal{T}}_i)$ for every $i \in \{0,1,\ldots,d\}$. The most likely value of

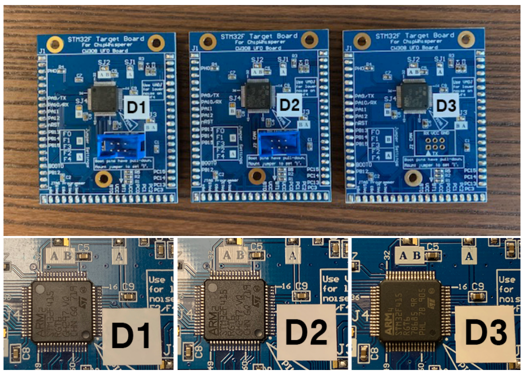

Figure 8: Three CW308T-STM32F4 boards used in the experiments.

$m[j]$ is the decided for each trace $\hat{\mathcal{T}}_i$ as

$$m_i[j] = \arg\max_{k\in\{0,1\}} s_{j,i,k}$$

and d results are combined by majority voting as

$$m[j] = \text{maj}_{i=1}^{d}\, m_i[j]$$

where $\text{maj}_{i=1}^{d}$ stands for a d-ary majority operation.

3.5 Experimental results

In the experiments, we use three CW303 ARM devices shown in Fig. 8. The device D_1 for profiling and all three devices for testing. One can see that D_1 and D_2 look similar. They are acquired from the same chip vendor. The device D_3 is visually different from the others, it is acquired from a different chip vendor. We test on multiple boards to investigate how the classification accuracy of the models trained on D_1 is affected by manufacturing process variation. It is known to be an issue for advances technologies [WBFD19].

Using the equipment and the method described in Section 3.3, we captured from D_1 a large set of traces $\mathcal{T}$ of size 100,000 for training the MLP models. We also captured from each of D_1, D_2 and D_3 a smaller set of traces $\hat{\mathcal{T}}$ of size 1,000 for testing the models.

Throughout the section, we use p_j and $p_{j,d}$ to denote the probability to recover a specific message bit $j \in \{0,\ldots,255\}$ from a single trace and n traces, respectively. Similarly, we use p_m and $p_{m,d}$ to denote the probability to recover a complete message from a single trace and d traces, respectively.

3.5.1 Single-trace attack

1. Using `POL2MSG()` leakage point Table 3 lists expected probabilities to recover a message bit $m[j]$ from a single trace from a test set $\hat{\mathcal{T}}$ using the MLP model $\mathcal{N}_j$ trained on the 7,000-point interval shown in Fig. 9, for $j \in \{0,\ldots,7\}$ and $|\hat{\mathcal{T}}| = 1,000$. The size $l = 7,000$ is chosen by dividing the trace in Fig. 9 by the number for shares, dividing the result by the number of bytes in one share, and adding to the share size the byte size plus some safety margin.

As expected, the models $\mathcal{N}_j$ best classify the test traces captured from the same device as the one used for profiling, D_1. For D_2 the difference in the classification accuracy is 1.2% on average. For D_3 it is to 2.7% on average.

We can also see that the expected message recovery probabilities differ for different bits. The bit $m[7]$ is the most difficult. To understand the reasons for this, we perform a posteriori TVLA analysis described in Section 3.5.3.

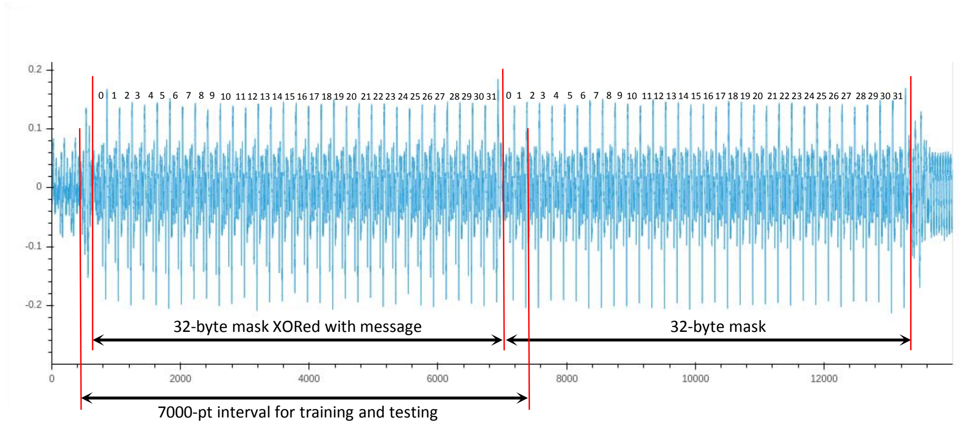

Figure 9: The 7,000-point trace used as an input to $\mathcal{N}_j$ for all $j \in \{0, \ldots, 7\}$ in the attack using `POL2MSG()` leakage point sampled 4 pt/clock cycle.

Table 3: Probability p_j to recover $m[j]$ from a single trace using `POL2MSG()` leakage point.

Device	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	**average**
D_1	0.998	0.998	0.993	0.992	0.989	0.988	0.985	0.953	0.987
D_2	0.994	0.989	0.986	0.959	0.978	0.962	0.985	0.945	0.975
D_3	0.984	0.985	0.988	0.963	0.972	0.991	0.975	0.819	0.960
average	0.992	0.990	0.989	0.971	0.979	0.980	0.982	0.906	0.974

2. Using `poly_A2A()` leakage point Table 3 lists expected probabilities to recover a message bit $m[j]$ from a single trace from a test set $\hat{\mathcal{T}}$ using the MLP model $\mathcal{N}_j$ trained on the 600-point interval shown in Fig. 10, for $j \in \{0, \ldots, 7\}$ and $|\hat{\mathcal{T}}| = 1,000$. The size $l = 600$ is selected by dividing the interval shown in Fig. 10 by the number of bytes and adding to the result some safety margin.

We can see that `poly_A2A()` leaks less than `POL2MSG()`. Another difference is that, for `poly_A2A()`, the expected probabilities for all bits except $m[0]$ are similar. The bit 0 is more difficult to recover. However, later in Section 3.5.3 we show that $m[0]$ is a special case for the byte 0. For other bytes, $m[0]$ is not different from the rest.

Clearly, it is possible to exploit both leakage points simultaneously by combining score vectors of the corresponding models.

3.5.2 Multiple-trace attack

If classification errors are mutually independent, the probability to recover a message bit j from d traces captured for the same ciphertext can be estimated as [Dub13, p. 64]:

$$p_{j,d} = \sum_{i=\lceil d/2 \rceil}^{d} \binom{d}{i} p_j^i (1-p_j)^{d-i},$$

where p_j is the probability to recover a message bit j from a single trace and d is odd.

Assuming that the expected probabilities for the other message bytes are the same as the ones for the first byte, the probability to recover a 256-bit message from d traces can be computed as $p_{m,d} = (\prod_{j=0}^{7} p_{j,d})^{32}$. Table 5 lists the minimal number of traces d required to achieve $p_{m,d} > 99.9\%$ for D_1, D_2 and D_3, respectively, for the case when both leakage points are used simultaneously.

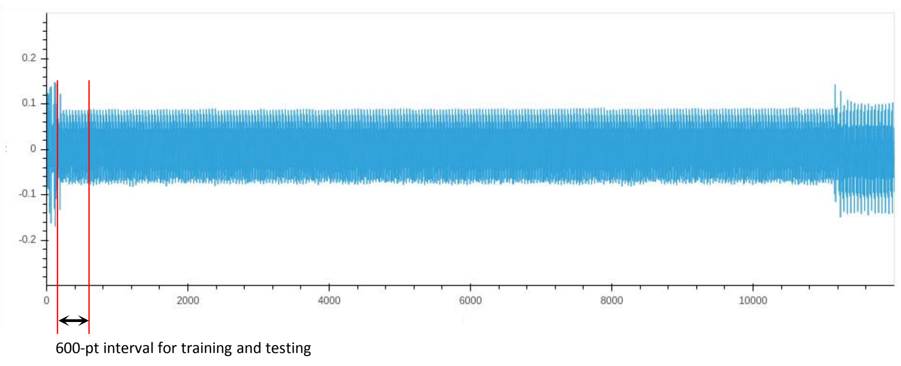

Figure 10: The 600-point trace used as an input to $\mathcal{N}_j$ for all $j \in \{0, \ldots, 7\}$ in the attack using `poly_A2A()` leakage point sampled 1 pt/clock cycle.

Table 4: Expected probability to recover a message bit from a single trace using `poly_A2A()` leakage point.

Device	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	**average**
D_1	0.845	0.970	0.959	0.905	0.948	0.960	0.953	0.972	0.939
D_2	0.828	0.962	0.942	0.945	0.920	0.919	0.950	0.950	0.927
D_3	0.848	0.900	0.941	0.884	0.949	0.905	0.914	0.947	0.911
average	0.840	0.944	0.947	0.912	0.939	0.928	0.939	0.956	0.926

Table 5: Probability $p_{m,d}$ to recover a 256-bit message from d traces captured for the same ciphertext.

Device	d	$p_{m,d}$
D_1	9	0.99921
D_2	11	0.99955
D_3	11	0.99961

3.5.3 A posteriori TVLA analysis

To understand why some bits are more difficult to extract than others, we perform a posteriori TVLA analysis of side-channel measurements. This section shows results for the first byte of the share $m \oplus r$.

For each $j \in \{0, \ldots, 7\}$, we partition the training set $\mathcal{T}$ into two sets containing traces in which $m_i[j] \oplus r_i[j] = k$ when c_i is applied as input:

$$\mathcal{T}_k = \{\mathcal{T}_i \in \mathcal{T} \mid m_i[j] \oplus r_i[j] = k\},$$

for $k \in \{0, 1\}$ and $i \in \{1, \ldots, t\}$, where $r_i[j]$ denotes jth bit of the mask r_i.

Fig. 11 shows the results. Fig. 12 shows zoomed intervals [100:600] and [11100:11500] of Fig. 11 representing the processing of the first byte of the share $m \oplus r$ by `poly_A2A()` and `POL2MSG()`, respectively.

From the t-test results we can see that overall `POL2MSG()` leaks greater than `poly_A2A()`. This is also apparent from Tables 3 and 4. For `POL2MSG()`, the shape of t-test plots is different for every bit of a byte (see Fig. 13). Possibly the peaks represent the storage of the decoded message bit $m[j]$ into a byte array in memory and the addition (OR) of the current content of the byte array with the following $7 - j$ bits, $j \in \{0, \ldots, 7\}$. Note that our method of labelling training traces is "memoryless", a label for $\mathcal{N}_j$ is determined by the bit $m[j]$ only and not by the bits $m[0], \ldots, [j - 1]$ already stored in the byte array

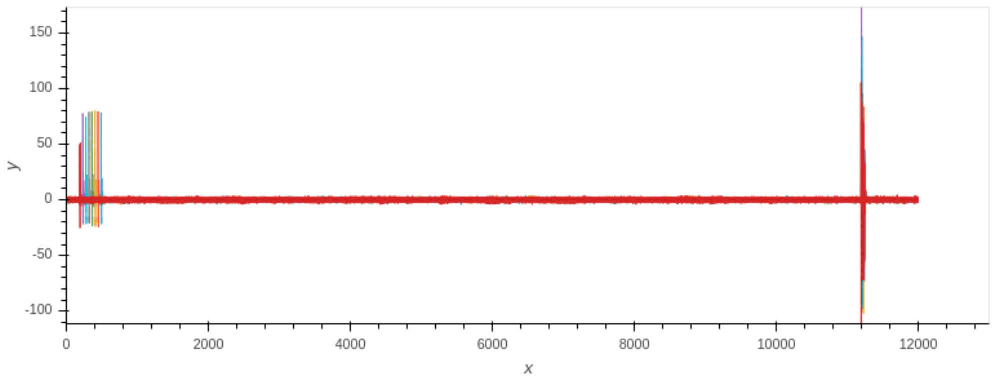

Figure 11: The t-test results for the first byte of the share $m \oplus r$.

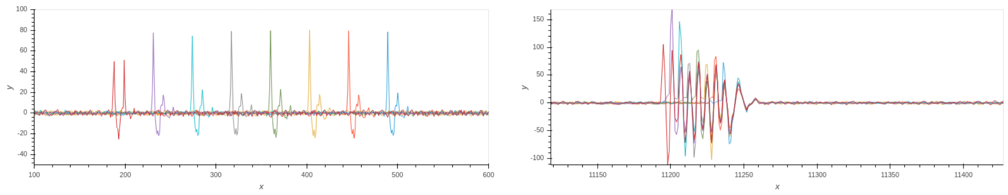

Figure 12: The intervals [100:600] and [11100:11500] of Fig. 11 representing the processing of the first byte of the share $m \oplus r$ by `poly_A2A()` and `POL2MSG()`, respectively.

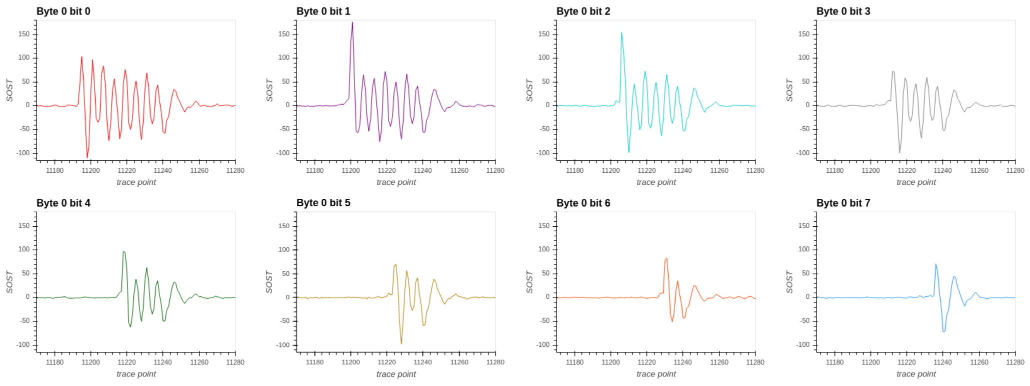

Figure 13: The t-test results for `POL2MSG()` leakage point separately for each bit.

by the time j is processed, as in [RBRC20]. While we might be losing in classification accuracy for $m[7]$, we need to train eight models only to recover all bits of a byte. In the Hamming weight based-method [RBRC20], 44 templates are required to recover all bits of a byte.

Tables 6 and 7 compare sum of squared pairwise t-differences (SOST) values of the first four bytes of the share $m \oplus r$ for `POL2MSG()` and `poly_A2A()` leakage points. We can see that, on average, SOST of different bytes do not differ significantly. The byte 0 is more difficult than other bytes for `POL2MSG()`. The bit 0 of byte 0 is a specially difficult case for `poly_A2A()`.

Table 6: SOST of the first 4 message bytes for `POL2MSG()`.

	byte 0	byte 1	byte 2	byte 3	**average**
bit 0	224.5	407.4	550.3	425.4	401.9
bit 1	266.3	386.1	493.8	385.6	382.9
bit 2	185.9	237.5	218.1	214.1	213.9
bit 3	293.4	211.4	236.8	307.5	262.2
bit 4	224.6	206.3	177.4	255.5	215.9
bit 5	189.8	177.9	185.8	187.8	185.3
bit 6	135.6	225.8	180.5	136.9	169.7
bit 7	117.3	118.8	109.5	107.9	113.3
average	204.6	246.4	269.0	252.5	243.1

Table 7: SOST of the first 4 message bytes for `poly_A2A()`.

	byte 0	byte 1	byte 2	byte 3	**average**
bit 0	51.34	78.04	77.19	76.40	70.74
bit 1	77.70	76.96	79.38	77.54	77.90
bit 2	74.66	78.60	79.16	77.13	77.39
bit 3	79.17	79.73	76.34	77.00	78.06
bit 4	79.69	79.24	75.95	74.31	77.30
bit 5	80.24	79.09	76.45	77.32	78.27
bit 6	79.57	79.71	75.25	79.51	78.51
bit 7	78.47	78.61	76.23	78.55	77.96
average	75.10	78.75	76.99	77.22	77.02

3.5.4 Effect of masking countermeasure deactivation

In this section, we show a recompiled reference source code of masked Saber, in which masking countermeasure is deactivated, may produce very different power traces from the original source code.

The trace shown in Fig.9 represents two consecutive executions of `POL2MSG()` in `indcpa_kem_dec_masked()` without any modifications. The trace in Fig.14 represents two consecutive executions of `POL2MSG()` in `indcpa_kem_dec_masked()` with the mask fixed to 0. We can see that, in the latter case, one execution of `POL2MSG()` is twice as short than in the former. As a result, a model trained on traces in Fig. 14 is likely to fail in recovering the message from the traces in Fig. 9.

Traces in the two cases differ because the compiler applied a different optimization strategy due to the changes made to deactivate the countermeasure.

4 Recovering the long-term secret key

The session key can be derived directly from the recovered message and the public key. In this section, we show how to recover the long-term secret key using maps from error-correcting codes.

For IND-CCA-secure lattice-base schemes, [RRCB20] proposes a secret key recovery attack through the analysis of recovered messages; this approach however assumes that the message is perfectly recovered. Perfect message recovery is not trivial. The previous solutions to achieve a very high message recovery probability by using multiple traces might require a large number of traces in a combination with masking.

We next describe a new secret key recovery attack that compensates for the errors in

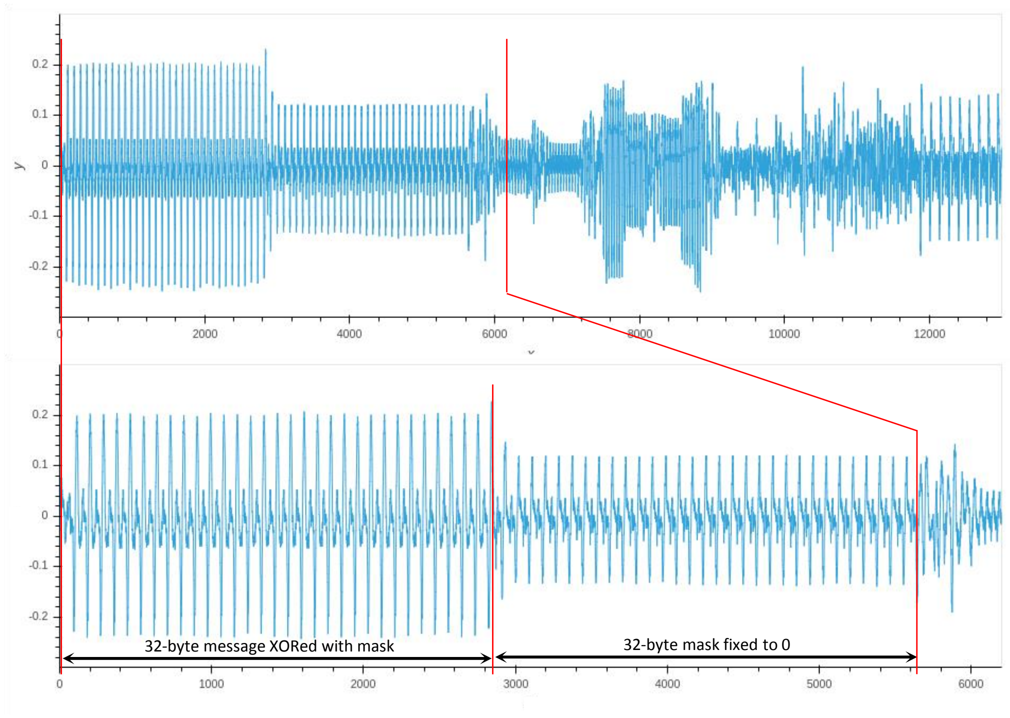

Figure 14: The execution of `POL2MSG()` with the mask fixed to 0 sampled 4 pt/clock cycle.

the recovered message. We start with a basic version that is optimized in terms of the sample complexity, i.e., the required number of traces. This basic version may encounter a large failure probability if the success probability in the message-recovery attack is low. We thus propose novel improvements using error correcting codes and other observations to reduce the noise occurring in the process of recovering the secret bits.

We focus on LightSaber, observing that the other two parameter sets (see Table 1) can be attacked in a similar manner.

4.1 The basic version of secret key recovery

Following the basic idea in [RRCB20], we prepare ciphertexts $(c_m, \mathbf{b}')$ where $c_m = k_0 \sum_{i=0}^{255} x^i \in R_T$ and $\mathbf{b}' = (k_1, 0) \in R_p^{2\times 1}$. Then, the decryption algorithm computes

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

Thus, the i-th bit in m', denoted by $m[i]$, is a function of the tuple $(k_0, k_1, \mathbf{s}[i])$, where $\mathbf{s}[i]$ is the i-th coefficient in the secret $\mathbf{s}$. Let the constant H be $2^{\epsilon_p-2} - 2^{\epsilon_p-\epsilon_T-1} + 2^{\epsilon_q-\epsilon_p-1}$. The decryption algorithm computes

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

If we use the message-recovery attack described in the previous section to recover the message bit $m[i]$, then partial secret information of $\mathbf{s}[i]$ is known. Using the decision table as shown in Table 8, we could recover the first 256 positions of $\mathbf{s}$ with four queries, when perfect message recovery is assumed. Then we could prepare ciphertexts $(c_m, \mathbf{b}')$ where $c_m = k_0 \sum_{i=0}^{255} x^i \in R_T$ and $\mathbf{b}' = (0, k_1) \in R_p^{2\times 1}$ to recover the remaining 256 positions of $\mathbf{s}$. In summary, one needs *eight* traces for LightSaber.

Table 8: Chosen pairs of (k_1, k_0) to determine $\mathbf{s}[i]$ based on $m[i]$ for LightSaber without error correction. (X: $m[i] = 1$; O: $m[i] = 0$)

Secret	(k_1, k_0)			
Coeff.	(60,2)	(191,6)	(303,3)	(532,2)
-5	X	X	O	O
-4	X	X	X	X
-3	X	X	X	O
-2	X	O	O	X
-1	X	O	X	O
0	X	O	X	X
1	O	X	O	O
2	O	X	O	X
3	O	O	X	X
4	O	O	O	O
5	O	O	O	X

Table 9: Chosen pairs of (k_1, k_0) to determine $\mathbf{s}[i]$ based on $m[i]$ for LightSaber with $[8, 4, 4]_2$ extended Hamming codes. (X: $m[i] = 1$; O: $m[i] = 0$)

Secret	(k_1, k_0)							
Coeff.	(143,2)	(219,1)	(542,2)	(956,1)	(286,2)	(365,1)	(453,2)	(35,1)
-5	O	X	O	O	X	O	X	X
-4	O	O	X	O	X	X	O	X
-3	X	O	O	O	O	X	X	X
-2	X	X	X	O	O	O	O	X
-1	X	X	O	O	X	X	O	O
0	X	O	X	O	X	O	X	O
1	O	O	O	O	O	O	O	O
2	O	O	O	X	X	X	X	O
3	O	X	X	X	X	O	O	O
4	X	X	O	X	O	O	X	O
5	X	O	X	X	O	X	O	O

This attack version works in the most optimistic setting. In the real case, the attack success probability could be very low, given an insufficient success probability in the message recovery attack. For instance, as shown in Table 4, the lowest probability of recovering one message bit for a D_3 device is only 0.947, resulting in a probability of 0.804 for recovering a single position in the secret key. Thus, the chance of getting all the secret coefficients correct is negligible.

4.2 New improvements

We now describe the improved key-recovery attack, consisting of three novel techniques.

Employing extended Hamming codes. The $[8, 4, 4]_2$ extended Hamming codes with code length 8, dimension 4, and minimum distance 4 can correct one error and detect the event that two errors occur. We design a new decision table shown in Table 9, mapping the secret coefficient $\mathbf{s}[i]$ to a codeword of the $[8, 4, 4]_2$ extended Hamming code. We will show that the error correcting capability of this $[8, 4, 4]_2$ extended Hamming code is sufficient for our attack; one may need to employ low-rate codes with larger minimum distance if a less accurate message-recovery is assumed.

Mapping a secret coefficient to various message bits. Since the leakage quality varies for different bits in a message byte (see Table 6), we could map a secret coefficient to several message bits to reduce the largest error probability for recovering such a coefficient. We prepare ciphertexts $(c_m, \mathbf{b}')$ where $c_m = k_0 \sum_{i=0}^{255} x^i \in R_T$ and $\mathbf{b}' = (k_1 x^j, 0) \in R_p^{2\times 1}$ for the j-th trace with $0 \le j \le 7$.
The decryption algorithm computes

$$m[i] = \begin{cases} ((k_1 \cdot (\mathbf{s}[i-j] \mod p) + H - 2^{\epsilon_p - \epsilon_T} k_0) \mod p) \gg (\epsilon_p - 1) & \text{if } i \ge j, \\ (-(k_1 \cdot (\mathbf{s}[i-j+256] \mod p) + H - 2^{\epsilon_p - \epsilon_T} k_0) \mod p) \gg (\epsilon_p - 1) & \text{otherwise.} \end{cases}$$

Thus, the decision table in Table 9 can be employed to recover $\mathbf{s}[i]$ for $i \in [0, 1, .., 248]$ and similar approaches for $i \in [256, 257, .., 504]$, where recovering each $\mathbf{s}[i]$ is connected to recovering eight consecutive message bits with different index modulo 8. For simplicity, we could mark the 14 positions of $\mathbf{s}[i]$ for $i \in [249, 250, .., 255]$ or $i \in [505, 506, .., 511]$ as erasures and recover them in a later step.

Using lattice reduction algorithms for post-processing. Since the connection between the public key and the secret key, i.e.,

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

is publicly known, one could employ a post-processing step with lattice reduction algorithms to fully recover the secret key $\mathbf{s}$.

Now we assume for a restricted adversary model with a D_3 device (see Section 3.5), so the adversity has limited access to the targeted device. For the j-th bit in a message byte, we use the leakage point `POL2MSG()` or `poly_A2A()` that maximizes the probability p_j from a single trace for $j \in \{0, 1, .., 7\}$. The notation p_j is defined in Section 3.5. We know from Table 3 and 4 that

$$(p_0, p_1, .., p_7) = (0.984, 0.985, 0.988, 0.963, 0.972, 0.991, 0.975, 0.947).$$

Similar to the assumption in Section 3.5.2, we assume that the expected probabilities for the other message bytes are not worse than the ones for the first byte. Since we connect each $\mathbf{s}[i]$ to the recovery of eight consecutive message bits with different index modulo 8, the probabilities of having less than one error, two errors, and no less than three errors are 0.9856, 0.0138, and 0.0006, respectively. With the $[8, 4, 4]_2$ extended Hamming codes, one error can be corrected and two errors can be detected with the corresponding secret coefficient marked as an erasure; only the last case will introduce an erroneous secret coefficient. We only need to recover $512 - 14 = 498$ positions, so the expected number of failures of recovering $\mathbf{s}[i]$ is about $498 \times 0.0006 \approx 0.3$. We also have about $14 + 498 \times 0.0138 < 21$ positions marked as erasures, but we know the coefficients are small since they are generated from the centered binomial distribution β_μ. The resulting lattice problem is easy to solve since the dimension of the new lattice is quite small.

In summary, one could recover the long-term key using only 16 traces even from a D_3 device which is different from the profiling one.

Note that the improved key-recovery attack from noisy messages has significance beyond evaluating the security of masked Saber. We leave the investigation of its wide applications for future work.

5 Conclusion

We presented a side-channel attack on a masked implementation of Saber which recovers the session key and the long-term secret key using 16 traces for LightSaber. Our approach has several advantages over previously proposed ones, including a possibility to increase success probability by analyzing multiple traces and using the device under attack for profiling. We discovered a previously unknown leakage point in the primitive for masked logical shifting on arithmetic shares. We also described a new approach for secret key recovery which can compensate for some errors in the recovered message.

Future work includes assessing higher-order masking schemes and combined ones, as well as designing deep learning-resistant countermeasures for LWE/LWR-based PKE/KEMs.

6 Acknowledgments

We are indebted to the authors of [BDK+20] who generously shared with us their source code of the masked Saber. The first and second authors would also like to thank Can Aknesil and Yang Yu for their help with scripts and experiments.

This work was supported in part by the Swedish Civil Contingencies Agency (Grants 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.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[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.

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

[Dub13] Elena Dubrova. *Fault-Tolerant Design.* Springer, 2013.

[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.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[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.

[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.

[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.

[RJJ$^+$18] P. Ravi, B. Jungk, D. Jap, Z. Najm, and S. Bhasin. Feature selection methods for non-profiled side-channel attacks on ecc. In *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pages 1–5, 2018.

[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.

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

[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.

[WBFD19] Huanyu Wang, Martin Brisfors, Sebastian Forsmark, and Elena Dubrova. How diversity affects deep-learning side-channel attacks. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2019.

[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 âĂę.