IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 4, pp. 676–707. DOI:10.46586/tches.v2021.i4.676-707

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

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

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

Abstract. In this paper, we present a 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 24 traces using a deep neural network created at the profiling stage. The proposed message recovery approach learns a higher-order model directly, without explicitly extracting random masks at each execution. This eliminates the need for a fully controllable profiling device which is required in previous attacks on masked implementations of LWE/LWR-based PKEs/KEMs. We also present a new secret key recovery approach based on maps from error-correcting codes that can compensate for some errors in the recovered message. In addition, we discovered a previously unknown leakage point in the primitive for masked logical shifting on arithmetic shares.

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 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 lattice 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 area 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

Licensed under Creative Commons License CC-BY 4.0.
Received: 2021-04-15 Accepted: 2021-06-15 Published: 2021-08-11

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. 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 and 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. 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 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 call for more complex solutions which decrease the speed even more. The masked implementation approach in [RRVV15] increases the number of CPU cycles on an ARM Cortex-M4 by a factor of more than 5 compared to a non-protected implementation, according to [BDK+20, p. 2].

Whereas these protection attempts consider Chosen-Plaintext Attack (CPA)-secure lattice schemes, it is more interesting to consider 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, at the time of writing, 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 [RBRC20b][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. **Contributions:** In this paper, we present a side-channel attack on a masked implementation of IND-CCA secure Saber KEM. We demonstrate how deep learning-based power analysis can be used to recover both the session key and the long-term secret key from a small number of traces. The presented message recovery approach learns a higher-order model directly, without explicitly extracting random masks at each execution. This eliminates the need for a fully controllable profiling device required in previous attacks on masked implementations of LWE/LWR-based PKEs/KEMs [RBRC20b, SKL+20]. 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). 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

[1]The design of the attack and most of the experimental work in this paper were done before the posting of [RBRC20b].

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 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 from [D+20].

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].

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

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}

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. "$+$" is arithmetic masking and "$\oplus$" is Boolean masking.

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

A challenge in masking lattice-based cryptosystems is the integration of bit-wise operations with arithmetic masking which requires methods for secure conversion between masked representations. Saber can be efficiently masked due to specific features of its design: power-of-two 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 labeled 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 that 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 that recovers a message from traces captured during the execution of Saber.KEM.Decaps() by the device under attack. Later, in Section 4.3.2, we show how both, the session key and the long-term secret key, can be extracted from the recovered messages.

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 $\mathcal{A}$?

3.1.1 Finding points of interest

The attacks on non-masked implementations [RRCB20, SKL+20, RBRC20a] solve the problem (1) by identifying points of interest 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 [RBRC20b, SKL+20] suggest solving the problem (1) by first deactivating the masking countermeasure, or fixing the mask to a constant, and then finding points of interest 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.

We solve the problem (1) using a deep learning method that works without explicitly extracting the random mask at each execution. We first hypothesize an approximate location of the point of interest 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 [RBRC20b], 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 between cases when a message bit value of '0' is processed at the point of interest 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 $m = r \oplus (m \oplus r)$.

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 is trained on traces containing jth bits of both shares, r and $m \oplus r$, and labelled by the value of the jth message bit. At the attack stage, the model takes an interval of the trace containing jth bits of both shares, 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 in this case the deep learning model's classification accuracy does not deteriorate due to differences in training and test traces caused by manufacturing process variation [WBFD19].

3.2 Trace acquisition

Next, we describe equipment for trace acquisition and how the points of interest are located.

3.2.1 Equipment

Our measurement setup is shown in Fig. 3. 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 method which greatly improves the synchronization of 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 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

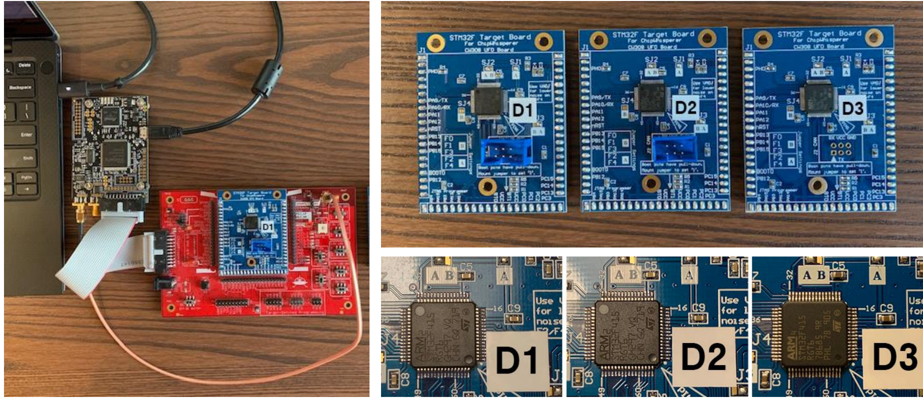

Figure 3: Equipment for trace acquisition and three boards used in the exeriments.

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 at 24 MHz (1 pt/clock cycle).

3.2.2 Locating points of interest

In previous work, a number of vulnerabilities were discovered in the non-masked LWE/LWR-based PKE/KEMs [ACLZ20, SKL+20, RRCB20, RBRC20b]. One is *Incremental-Storage* vulnerability resulting from an incremental update of the decrypted message in memory during message decoding [RBRC20b]. The decoding function (line 2 of Saber.PKE.Decrypt() 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 [RBRC20b] that, in a non-masked implementations of the decoding function (see `indcpa_kem_dec()` at Fig. 4), 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. 4, we found 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. We also 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 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(). This trace represents an average of 50K traces captured for different ciphertexts selected at random. 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. By measuring the distance between the peaks we can find that the processing of one byte

```
void indcpa_kem_dec(char *sk, char *ct,
char m[])
uint16_t v[N];
uint16_t sksv[K][N];
 1: BS2POLVECq(sk,sksv);
 2: SABER_un_pack(&ct, v);
 3: for (i = 0; i < N; ++i) do
 4:    v[i] = h2-(v[i]«(EP-ET));
 5: end for
 6: VectorMul(ciphertext,sksv,v);
 7: for (i = 0; i < N; ++i) do
 8:    v[i] = (v[i]&(P-1))»(EP-1);
 9: end for
    /* pack decrypted message */
10: POL2MSG(v,m);

void POL2MSG(uint16_t *v, chair *m)
 1: for (j = 0; j < BYTES; j++) do
 2:    m[j] = 0;
 3:    for (i = 0; i < 8; i++) do
 4:       m[j] = m[j]|(v[8*j+i]«i);
 5:    end for
 6: 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};
 1: SABER_un_pack(&ct,v1);
 2: for (i = 0; i < N; i++) do
 3:    v1[i] = h2-(v1[i]«(EP-ET));
 4: end for
 5: BS2POLVEC(ct,pksv,P);
 6: InnerProd(pksv,sksv1,P-1,v1);
 7: InnerProd(pksv,sksv2,P-1,v2);
 8: poly_A2A(v1,v2);
 9: POL2MSG(v1,m1);
10: POL2MSG(v2,m2);

void poly_A2A(uint16_t A[N], uint16_t R[N])
uint32_t A, R;
 1: for (i = 0; i < N; i++) do
 2:    A = A[i]; R = R[i];
 3:    ...  /* processing */
 4:    A[i] = A; R[i] = R;
 5: end for
```

Figure 4: C code of non-masked and masked implementations of Saber.PKE.Dec() from [BDK+20].

takes 49 points. Throughout the paper, we call this distance *byte offset* of `POL2MSG()`.

Since `poly_A2A()` is executed immediately before `POL2MSG()`, it is located in the interval approx. between the points 4,000 and 15,000 in Fig 5(b). By zooming in, we can distinguish 256 repeating patterns representing the processing of bits. Fig. 6 shows the first 15 bits of `poly_A2A()`. The distance between the peaks is 43 points. Throughout the paper, we call this distance *bit offset* of `poly_A2A()`.

From Fig. 5(b) we can also find the starting points of `POL2MSG()` and `poly_A2A()` procedures, referred to as *initial offsets*, and the distance between the processing of corresponding bytes in the two shares of `POL2MSG()`, referred to as *share offset*. For the traces in Fig. 6(b), the share offset of `POL2MSG()` is 1583. This offset is large because `POL2MSG()` first packs all bytes of one share, and then packs all bytes of the other share (see lines 9 and 10 of `indcpa_kem_dec()` at Fig. 4). Contrary, for `poly_A2A()`, the corresponding bits of shares, $A[i]$ and $R[i]$, are processed one after another (see line 4 of `poly_A2A()` in Fig. 4). Therefore, for `poly_A2A()`, there is no offset between the shares.

We would like to stress that the ability to deduce the exact time of byte/bit processing from the shape of averaged traces is very useful for the analysis. Apart from enabling us to locate `POL2MSG()` and `poly_A2A()` procedures, this information allows us to decrease the time required for capturing a sufficiently large training set from the device under attack by a factor of 32 (for `POL2MSG()`) or 256 (for `poly_A2A()`).

3.3 Adversary model

The adversary can be anyone who has equipment for power analysis and expertise in side-channel attacks and deep learning.

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

1. The access time is sufficient to capture traces for both profiling and attack stages. In this case, the device under attack is used in profiling. Profiling on traces from the

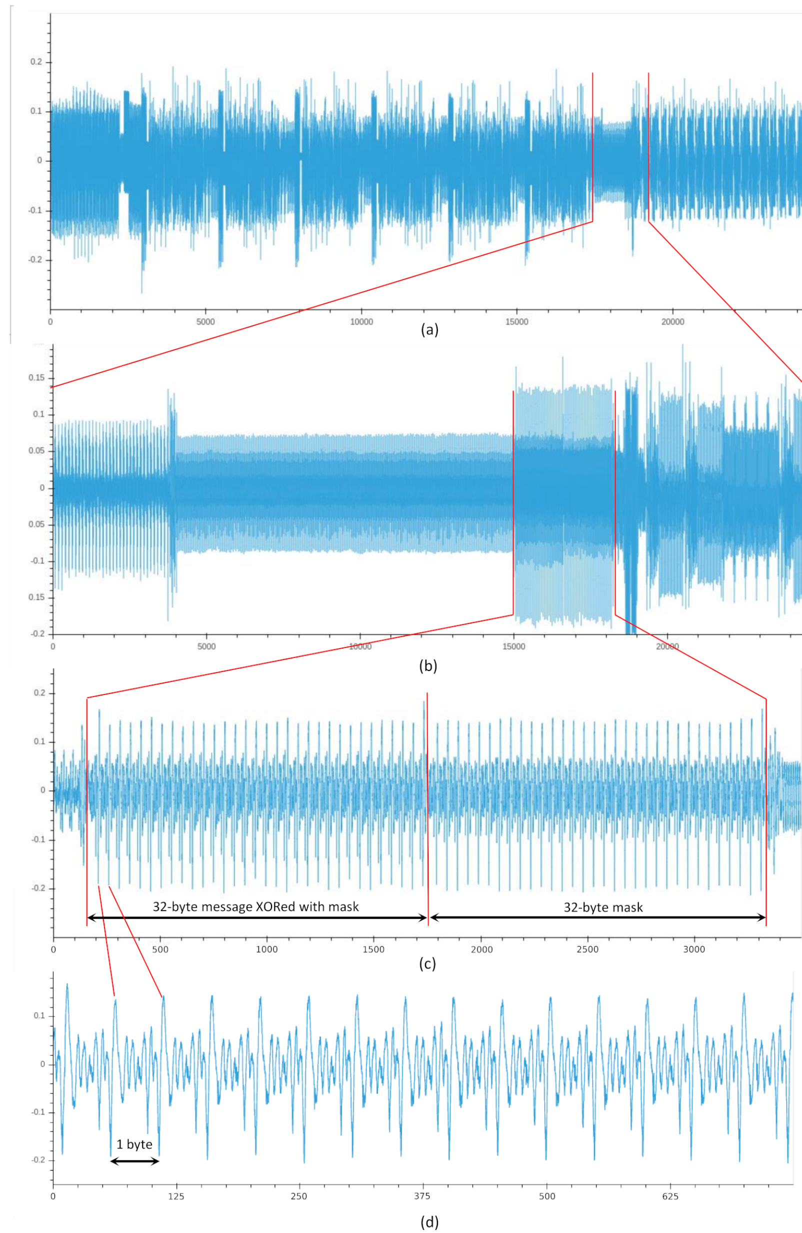

Figure 5: (a) The initial part of Saber.KEM.Decaps() sampled with decimation 15; (b) An interval containing `poly_A2A(v1,v2)`, `POL2MSG(v1,m1)` and `POL2MSG(v2,m2)`; (c) Two `POL2MSG()` shares; (d) The first 15 bytes of the first share.

device under attack typically maximizes the classification accuracy of deep learning models. In our experiments, we use device D_1 for profiling and attack in this scenario.

2. The access time is sufficient to capture traces for the attack only. In this case, a device similar to the device under attack is used for profiling. In our experiments, we use device D_1 for profiling and devices D_2 and D_3 for the attack in this scenario.

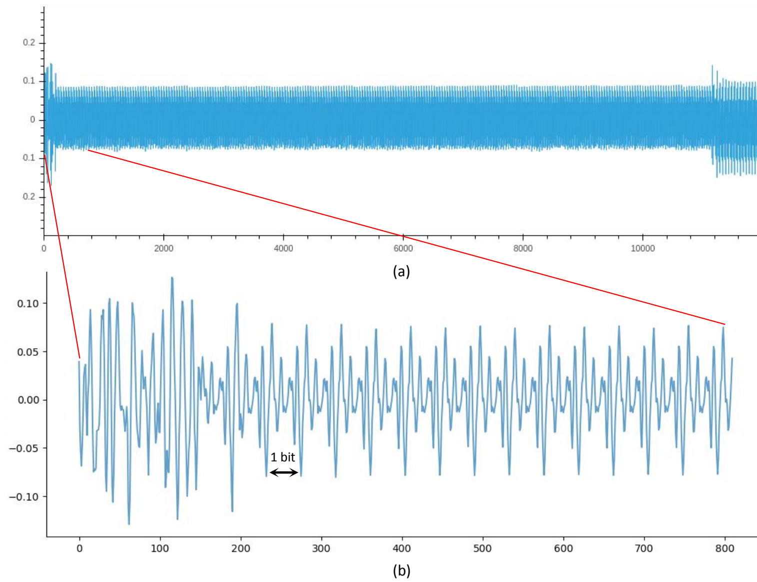

Figure 6: (a) The complete procedure `poly_A2A()`; (b) The first 15 bits of `poly_A2A()`.

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 $\mathcal{T}_i \in \mathbb{R}^u$ denote a trace captured from a device during the execution of Saber.KEM.Decaps() with c_i as input, where u is the trace size.

3.4.1 Profiling stage

The aim of profiling is to construct neural network models capable of recovering all message bits. A neural network $\mathcal{N}_j : \mathbb{R}^u \to \mathbb{I}$, maps the trace $\mathcal{T}_i$ into a score $s_{j,i} = \mathcal{N}_j(\mathcal{T}_i) \in \mathbb{I}$ representing the probability that the jth bit of m_i, $m_i[j]$, is 1 in $\mathcal{T}_i$, where $\mathbb{I} = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$. To train a network for a given bit j, each trace in the training set $\boldsymbol{\mathcal{T}} = \{\mathcal{T}_1, \ldots, \mathcal{T}_t\}$ is assigned a label $l(\mathcal{T}_i) = m_i[j]$, for $i \in \{1, \ldots, t\}$.

We train models for more that one bit at the same time by cutting intervals corresponding to bytes/bits and taking their union. In this way the training set can be expanded by a factor of 32 (for bytes) or 256 (for bits), saving the trace acquisition time proportionally. For instance, by cutting into bits, we can compose a 1,024K set from a 4K set. It takes less than 45 min to capture the latter and 8 days to capture the former. So, with the cutting technique, the usage of device under attack as a profiling device becomes possible in practice.

The `POL2MSG()` procedure processes different bits of a byte differently when it packs the decoded message into a byte array in memory. The current content of the byte array (initially zero) is subsequently ORed with the bits which arrive one-by-one. Therefore, traces representing the processing of bits in different byte positions look differently. For this reason, for `POL2MSG()`, we train eight models, $\mathcal{N}_0^{p2m}, \ldots, \mathcal{N}_7^{p2m}$, one per each bit position of a byte. The pseudocode at Fig. 7 describes the main steps.

Note that our method for labelling training traces is "memoryless", e.g. a label for $\mathcal{N}_j^{p2m}$ is determined by the bit $m[j]$ only and does not depend on the bits $m[0], \ldots, [j-1]$ already stored in the byte array by the time j is processed, $j \in \{0, \ldots, 7\}$. As a result, we train only eight models for `POL2MSG()`. Previous attacks exploiting `POL2MSG()`, such as

```
TrainModels(D_pro, t, u, v) /* D_pro is profiling device, t is the number of traces, u is the trace
size, v is the neural network input size */
 1: for each bit ∈ {0, 1, ..., 7} do
 2:     (𝓣, 𝓛) = GetTrainingTraces(D_pro, t, u, v, bit)
 3:     Train N_bit^p2m : ℝ^v → 𝕀 on (𝓣, 𝓛)
 4: end for
 5: return N_0^p2m, ..., N_7^p2m

GetTrainingTraces(D_pro, t, u, v, bit)
 1: m = {m_i ∈ {0,1}^256 | m_i is selected at random, ∀i ∈ {1, ..., t}}
 2: c = {c_i ∈ {0,1}^8704 | c_i = Saber.PKE.Enc(pk, m_i; r_i), ∀i ∈ {1, ..., t}}
 3: 𝓣_init = {T_i ∈ ℝ^u | T_i ⇐ D_pro[Saber.KEM.Decaps(c_i)], ∀i ∈ {1, ..., t}}
 4: Determine initial offset α, share offset β, and byte offset γ from 𝓣_init
 5: 𝓣 = ∅, 𝓛 = ∅
 6: for each byte ∈ {0, 1, ..., 31} do
 7:     points = SelectPoI(v, byte, bit, α, β, γ)
 8:     𝓣 = 𝓣 ∪ 𝓣_init[:, points]
 9:     𝓛 = 𝓛 ∪ {l(T_i) ∈ {0,1} | l(T_i) = m_i[8*byte+bit], ∀i ∈ {1, ..., t}}
10: end for
11: return (𝓣, 𝓛)

SelectPoI(v, byte, bit, α, β, γ) /* Selects points of interest */
 1: start_1 = α + byte * γ, stop_1 = start_1 + v/2
 2: start_2 = start_1 + β, stop_2 = start_2 + v/2
 3: points = append(start_1 : stop_1, start_2 : stop_2)
 4: return points

RecoverMessage(D_attack, N_0^p2m, ..., N_7^p2m, u, v, c, α, β, γ) /*D_attack is device under attack */
 1: T̂ ⇐ D_attack[Saber.KEM.Decaps(c)], T̂ ∈ ℝ^u
 2: for each j ∈ {0, 1, ..., 256} do
 3:     byte = j mod 32, bit = j mod 8
 4:     points = SelectPoI(v, byte, bit, α, β, γ)
 5:     s_j = N_bit^p2m(T̂[points])
 6:     m[j] = 1 if s_j > 0.5, else m[j] = 0
 7: end for
 8: return m = (m[0], ..., m[255])
```

Figure 7: Pseudocode of the message recovery attack using `POL2MSG()`.

Table 2: The MLP architecture used in the message recovery attack. For $\mathcal{N}_0^{p2m}, .., \mathcal{N}_7^{p2m}$ and $\mathcal{N}^{a2a}$, the input size is $v = 128$. For the models using both points, $v = 256$.

Layer type	(Input, output) shape	# Parameters
Batch Normalization 1	(v, v)	$4v$
Dense 1	(v, 128)	$64(v + 1)$
Batch Normalization 2	(128, 128)	256
ReLU	(128, 128)	0
Dense 2	(128, 32)	2080
Batch Normalization 2	(32, 32)	128
ReLU	(32, 32)	0
Dense 3	(32, 16)	528
Batch Normalization 2	(16, 16)	64
ReLU	(16, 16)	0
Dense 4	(16, 1)	17
Softmax	(1, 1)	0

the Hamming weight based-method in [RBRC20b], require 44 templates.

The `poly_A2A()` procedure processes all message bits in the same way during their storage in a memory. Thus, traces representing the execution of `poly_A2A()` look identically for all message bits except for the first and the last. The first and the last bits are special because their previous and next instructions, respectively, are different from for the ones of other bits. Since in Cortex-M4 CPU the next instruction starts being processed before the previous instruction has finished, the power consumption during the processing of the first and the last bits differs from the power consumption during the processing of other bits.

The similarity of bit processing makes it possible to train a single model, $\mathcal{N}^{a2a}$, capable of recovering all message bits (excluding the first and the last) with a high accuracy. The steps of the message recovery attack based on `poly_A2A()` are the same as the ones in Fig. 7 except that the **for**-loops in TrainModels() and GetTrainingTraces() are run over all 256 bits and in SelectAttackPoints() the points of interest are selected as $points = append(start : stop)$, with $start = \alpha + byte * 8 * \gamma + bit * \gamma$, $stop = start + v/2$, where α is the initial offset for `poly_A2A()`, and γ is the bit offset for `poly_A2A()`.

To train on both `POL2MSG()` and `poly_A2A()`, the intervals representing points of interest of `POL2MSG()` and `poly_A2A()` are appended. The eight models are trained following the same steps as in the pseudocode in Fig. 7.

3.4.2 Neural network architecture

In all experiments, we use the multilayer perceptron (MLP) architecture shown in Table 2. We selected it using the grid search algorithm [GBC16] which trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter. The combination that yields the best validation set error is chosen as the best. The list of values to search over was selected on a logarithmic scale, e.g. a learning rate taken within the set $\{0.1, 0.01, 10^{-3}, 10^{-4}, 10^{-5}\}$, and size of dense layers taken within the set $\{4, 8, 2^4, 2^5, 2^6, 2^7, 2^8\}$. We tried adding convolutional layers in early experiments, however, this brought no improvement and slowed down the training. Since we use ChipWhisperer to capture traces, the resulting trace sets are perfectly synchronized. We believe that this is the reason why MLPs are sufficient for our case.

During training, we use binary cross-entropy as a loss function. No input normalization is applied. Nadam optimizer (an extension of RMSprop with Nesterov momentum) with the learning rate 0.001 and numerical stability constant epsilon=1e-08 is used. The training is carried out for a maximum of 300 epochs with batch size 32 and early stopping. 70% of the training set is used for training and 30% for validation.

3.4.3 Attack stage

Let $\hat{\boldsymbol{\mathcal{T}}} = \{\hat{\mathcal{T}}_1, \dots, \hat{\mathcal{T}}_w\}$ be a set of traces captured from the device under attack D_{attack} for the ciphertexts $c_1, \dots, c_w$.

To recover the message from a trace $\hat{\mathcal{T}}_i \in \hat{\boldsymbol{\mathcal{T}}}$ using the models $\mathcal{N}_0^{p2m}, \mathcal{N}_1^{p2m}, \dots, \mathcal{N}_7^{p2m}$, the score of each message bit $j \in \{0, 1, \dots, 255\}$ is computed as $s_{j,i} = \mathcal{N}_{j \bmod 8}^{p2m}(\hat{\mathcal{T}}_i)$. The most likely value of the jth bit is then decided by rounding the score as:

$$m_i[j] = \begin{cases} 1 & if \ s_{j,i} > 0.5 \\ 0 & otherwise. \end{cases}$$

To recover the message from a trace $\mathcal{T}_i \in \hat{\boldsymbol{\mathcal{T}}}$ using the model $\mathcal{N}^{a2a}$, the score of each message bit $j \in \{0, 1, \dots, 255\}$ is computed as $s_{j,i} = \mathcal{N}^{a2a}(\hat{\mathcal{T}}_i)$ and $m_i[j]$ is decided in the same way as above.

For the models trained on both points, the message bits are recovered similarly to the `POL2MSG()` case.

3.5 Experimental results

In the experiments, we use three CW303 ARM devices shown in Fig. 3. The device D_1 is used for profiling and all three devices for testing. From the zoomed chip photos in Fig. 3 one can see that D_1 and D_2 look similarly. They are acquired from the same chip vendor. The device D_3 looks differently from the other two. It is acquired from a different chip vendor. We test on different types of boards to investigate how the classification accuracy of the models trained on D_1 is affected by manufacturing process variation.

Using the equipment described in Section 3.2, we captured from D_1 a set of 50K traces $\boldsymbol{\mathcal{T}}_{init}$ containing the executions of both `poly_A2A()` and `POL2MSG()` procedures (14.5K data points in each trace). The traces were captured for random ciphertexts and random secret keys. The $\boldsymbol{\mathcal{T}}_{init}$ was used to construct the 4K $\times$ 256 = 1.024M training set for the model $\mathcal{N}^{a2a}$ and the 50K $\times$ 32 = 1.6M training sets for the models $\mathcal{N}_0^{p2m}, \mathcal{N}_1^{p2m}, \dots, \mathcal{N}_7^{p2m}$ and the models using both points.

For testing, we captured from each of D_1, D_2 and D_3 a set of 1K traces for random ciphertexts and a fixed secret key.

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

3.5.1 Single-trace message recovery attack

Tables 3, Table 4 and Table 5 list empirical probabilities to recover the first 8 message bits from a single trace using the models trained on `POL2MSG()`, `poly_A2A()`, and both points, respectively, for 1000 trials. The complete tables for all message bits are shown in the Appendix.

We can see that the success rates differ for different bits. For `POL2MSG()`, the bit 7 is the most difficult to recover. For `poly_A2A()`, the bit 0 is the most difficult to recover. We explain the reasons for this in Section 3.5.3, after TVLA analysis.

Since difficult bits are in the different positions for `POL2MSG()` and `poly_A2A()`, when both points are used, the average success rate is maximized.

For the devices D_1 and D_2, which are similar, the success rates are comparable. For D_3, the success rate is by a few percent smaller than the one of D_1.

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

Device	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	**average**
D_1	0.996	0.995	0.997	0.996	0.995	0.993	0.995	0.881	0.981
D_2	0.989	0.989	0.993	0.992	0.994	0.997	0.993	0.851	0.975
D_3	0.996	0.999	0.997	0.991	0.998	0.997	0.986	0.743	0.963
average	0.994	0.994	0.996	0.993	0.996	0.996	0.991	0.825	0.973

Table 4: Empirical probability p_j to recover $m[j]$ from a single trace using `poly_A2A()`.

Device	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	**average**
D_1	0.845	0.992	0.997	0.994	0.999	0.998	0.995	0.998	0.977
D_2	0.836	0.992	0.995	0.992	0.995	0.993	0.996	0.998	0.975
D_3	0.810	0.880	0.889	0.859	0.928	0.939	0.933	0.923	0.895
average	0.830	0.955	0.960	0.948	0.974	0.977	0.975	0.973	0.949

Table 5: Empirical probability p_j to recover $m[j]$ from a single trace using both points.

Device	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	**average**
D_1	0.993	0.999	0.998	1.000	0.997	0.998	0.999	0.995	0.997
D_2	0.987	0.998	0.999	0.999	0.997	0.998	0.998	0.999	0.997
D_3	0.982	0.966	0.976	0.962	0.966	0.954	0.968	0.941	0.964
average	0.987	0.988	0.991	0.987	0.987	0.983	0.988	0.978	0.986

3.5.2 N-trace message recovery attack

If classification errors are mutually independent, the probability to recover a message bit j from N traces captured for the same ciphertext can be estimated as $p_{j,N} = \sum_{i=\lceil N/2 \rceil}^{N} \binom{N}{i} p_j^i (1-p_j)^{N-i}$, where p_j is the probability to recover a message bit j from a single trace and N is odd [Dub13, p. 64].

Assuming that the classification errors are mutually independent, the probability to recover the complete 256-bit message from N traces can be estimated as $p_{m,N} = (\prod_{j=0}^{255} p_{j,N})$.

3.5.3 A posteriori TVLA analysis

To understand why some bits are more difficult to recover than others, we perform *a posteriori* TVLA analysis of side-channel measurements. This section shows results for the first four bytes 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:

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

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

Fig. 8 shows the results. Fig. 9 shows zoomed intervals [100:600] and [11100:11500] of Fig. 8 representing the processing of the first byte of the share $m \oplus r$ by `poly_A2A()` and `POL2MSG()`, respectively.

From the t-test results in Fig. 10 we can see that, for `POL2MSG()`, the shape of t-test plots is different for every bit of a byte. The peaks possibly 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, $j \in \{0, \ldots, 7\}$. For `poly_A2A()`, t-test plots for all bits look similar except for the first bit.

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()`. We can see that, for

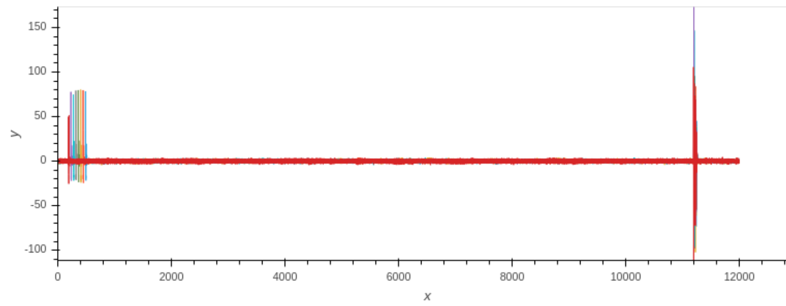

Figure 8: The t-test results for the first byte of the share $m \oplus r$.

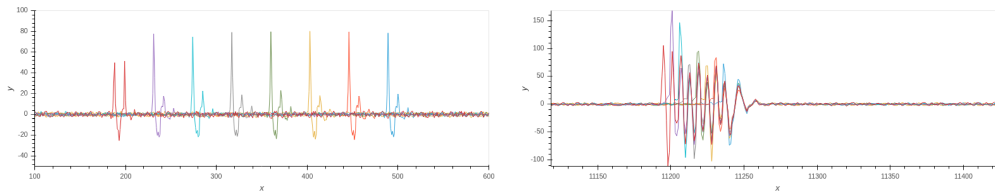

Figure 9: The intervals [100:600] and [11100:11500] of Fig. 8 representing the processing of the first byte of the share $m \oplus r$ by `poly_A2A()` and `POL2MSG()`, respectively.

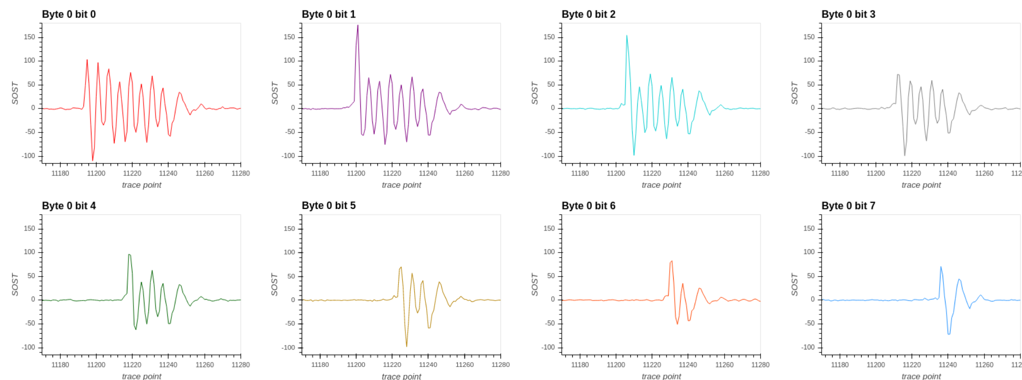

Figure 10: The t-test results for `POL2MSG()` separately for each bit.

`poly_A2A()`, the SOST of the first bit (51.34) is nearly 30% smaller than the average SOST of other bits (77.02). For `POL2MSG()`, the SOST of all bits 7 (113.3 on average) is less than a half of the average SOST of all bits (243.1). The bits with a smaller SOST are typically more difficult to recover.

4 Key recovery attack

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 (ECC).

For IND-CCA-secure lattice-base schemes, [RBRC20b, RRCB20] proposes a secret key recovery attack through the analysis of recovered messages; this approach however assumes that the messages are recovered perfectly. If the message is recovered with errors, the attack fails. Clearly, occasional errors are unavoidable. The previous solutions increase the

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

success rate of message recovery by repeating the measurements N times and averaging traces to reduce the signal-to-noise ratio (SNR), e.g. $N = 5$ is used in [RBRC20b] to achieve 98.24% message recovery probability. However, this increases the number of traces required for the attack proportionally.

As an alternative, in this section, we present a new ECC-based secret key recovery approach that compensates for some errors in the recovered message. We start with a basic version that is optimized in terms of the required number of traces. This basic version may have a high failure probability if the success probability of the message recovery attack is low. Then we present novel improvements based on ECCs and other techniques to reduce the errors occurring in the process of recovering the message bits.

We focus on Saber, observing that FireSaber and LightSaber (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 choose ciphertexts $(c_m, \mathbf{b}')$ where $c_m = k_0 \sum_{i=0}^{255} x^i \in R_T$ and $\mathbf{b}' = (k_1, 0, 0) \in R_p^{3\times 1}$. 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

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

If we use the message recovery attack presented in the previous section to recover the message bit $m[i]$, then the partial secret information of $\mathbf{s}[i]$ is known. Using the decision table as shown in Table 8, we 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

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

Secret	(k_1, k_0)			
Coeff.	(28, 4)	(819,2)	(295,6)	(522,4)
-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

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

Secret	(k_1, k_0)							
Coeff.	(186,0)	(293,7)	(311,7)	(615,2)	(613,2)	(890,4)	(903,4)	(199,0)
-4	O	X	X	X	X	O	O	O
-3	X	X	X	O	O	O	O	X
-2	X	O	O	X	X	O	O	X
-1	O	O	O	O	O	O	O	O
0	O	X	X	O	O	X	X	O
1	O	O	O	X	X	X	X	O
2	X	O	O	O	O	X	X	X
3	X	X	X	X	X	X	X	X
4	X	X	O	X	O	O	X	O

$c_m = k_0 \sum_{i=0}^{255} x^i \in R_T$ and $\mathbf{b}' = (0, k_1, 0) \in R_p^{3\times1}$ (or $\mathbf{b}' = (0, 0, k_1) \in R_p^{3\times1}$) to recover the next (or last) 256 positions of $\mathbf{s}$. In summary, one needs 12 traces for Saber.

This attack version works in the most optimistic setting. In practice, the key recovery attack success rate could be very low if the message is recovered with many errors. For instance, as shown in Table 5, the lowest probability of recovering one message bit for device D_2 is only 0.987, resulting in a probability of 0.949 for recovering a single position in the secret key. Thus, the chance of getting all 768 secret coefficients correctly is negligible.

4.2 New improvements

We now describe the improved key recovery attack, consisting of the following 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 two errors. 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 case; one may need to employ low-rate codes with a larger minimum distance if the expected success probability of message recovery is lower.

Including a post-processing step. Since the connection between the public key and the secret key, i.e.,

$$\mathbf{b} = ((\mathbf{A}^T\mathbf{s} + \mathbf{h}) \mod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times1},$$

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

Table 10: Empirical distribution of errors in the ECC-based key recovery attack based on $[8,4,4]_2$ extended Hamming code (100 trials).

Device	N	# cases of all errors detected	# undetected errors in k positions, #"E"				
			$k=1$	$k=2$	$k=3$	$k=4$	...
D_1	1	100	0	0	0	0	...
D_2	1	98	2	0	0	0	...
D_3	9	86	13	1	0	0	...

Table 11: Empirical distribution of erasures in the ECC-based key recovery attack based on $[8,4,4]_2$ extended Hamming code (100 trials).

Device	N	# cases of no erasures	# erasures in k positions, #"?"												
			k=1	2	3	4	5	6	7	8	9	10	11	12	...
D_1	1	64	24	12	0	0	0	0	0	0	0	0	0	0	...
D_2	1	34	28	17	13	5	1	0	0	0	1	0	1	0	...
D_3	9	9	29	30	16	7	5	3	1	0	0	0	0	0	...

4.3 Experimental results

We have implemented different versions of the new key recovery attack on the masked SABER implementation [BDK+20].

4.3.1 ECC-based approach

The results of ECC-based key recovery attacks using the $[8,4,4]_2$ extended Hamming code are summarized in Tables 10 and 11.

We aim to recover all of 768 secret coefficients. For each secret coefficient, the decoding algorithm of the employed extended Hamming code may either detect two bit errors in the message or output a codeword that is not listed in Table 9. For both cases, we know that the decoding is not correct, so we assign a question mark "?" to this coefficient and call it an *erasure*.

We performed 100 trials and counted the number of undetected decoding errors and the number erasures. We use the notation #"E" (#"?") to denote the number of decoding errors (erasures) collected. As discussed in the adversary model section, we treat D_1 as the device under attack for the scenario when access time is sufficient to capture traces for the profiling and attack stages. Otherwise, we consider D_2 and D_3 as devices under attack and profile on D_1.

For the device D_1, we can see from Table 10 that there are no undetected errors among 100 trials. Furthermore, Table 11 shows that the number of erasures is bounded by 2 among 100 trials. So, the full secret can be easily recovered by enumerating 2 positions. As each secret coefficient is sampled as an integer in $[-4,4]$, we only need to enumerate $9^2 = 81$ possibilities. Thus, the empirical success rate of the key recovery attack is 100% for the device D_1.

Similarly, as shown in Tables 10 and 11, for the device D_2, 98 out of 100 trials have no undetected errors and in 97 out of 100 trials, the number of erasures is bounded by 4. Thus, the full secret key can be recovered by enumeration of 9^4 possibilities. Therefore, for the device D_2, the empirical success rate is at least 95%.

For the device D_3, majority voting is required to correct message bits. Tables 10 and 11 show results for $N = 9$. We can see that in 86 out of 100 trials there are no undetected errors and in 96 out of 100 trials the number of erasures is bounded by 5. Thus, for the device D_3, the empirical success rate is at least 82% with enumeration 9^5. The success probability can be increased by using a larger N.

In summary, using only 24 traces we can recover the long term key from the profiling

device D_1 and the similar to profiling device D_2 with a high probability. In our experiments, it takes less than *one minute* to collect such 24 traces in the online phase[2]. For the different from profiling device D_3, we need $9 \times 24 = 216$ traces to recover the key with a high probability.

4.3.2 Approach without ECC

We also implemented the basic version of key recovery attack without ECC presented in Sect. 4.1. It is less successful than the ECC-based attack; even for D_1, it fails all 100 trials. If the majority voting with $N = 3(5)$ is used to correct message bits, then, for D_1, the success rate increases to 52% (99%).

For D_2, the success rate remains 0% even for $N = 5$. Note that $12 \times 5 = 60$ traces are required for the attack without ECC with $N = 5$, while only 24 traces are required an the ECC-based attack with $N = 1$.

For D_3 the success rate of the attack without ECC is 0% for $N = 9$.

5 Conclusion

We demonstrated a side-channel attack on a masked Saber implementation that uses 24 traces to recover the session key and the long-term secret key. Our message recovery approach has several advantages over previous proposals, including the ability to profile on the device under attack. We discovered a previously unknown leakage point in the primitive for masked logical shifting on arithmetic shares. We also presented a new approach for secret key recovery that can compensate for some errors in the recovered message.

We would like to point out that the presented ECC-based key recovery attack has significance beyond evaluating the security of masked Saber and could be applied to side channel attacks on other LWE/LWR-based PKE/KEM implementations.

All our traces, models, scripts, and a video of a live demo of the key recover attack are available at

`http://drive.google.com/drive/folders/1jO0yD3--s-772nt7ZmFwe88wGcyzKAiZ`

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.

[2]The time required for the attack is determined by the baud rate of the communication channel between host computer and the implementation. In our case it is 38400 bps.

[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.

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

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

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 359–386, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

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

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

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

[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.

[RBRC20a] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by drop you break the rock - exploiting generic vulnerabilities in lattice-based pke/kems using em-based physical attacks. Cryptology ePrint Archive, Report 2020/549, 2020. https://eprint.iacr.org/2020/549.

[RBRC20b] 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.

7 Appendix

This appendix presents tables of empirical probabilities of message bit recovery using `poly_A2A()`, `POL2MSG()` and both points (for 1000 trials).

Table 12: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using `POL2MSG()`. Average results for 1K traces from device D_1 (profiling).

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.996	0.995	0.997	0.996	0.995	0.993	0.995	0.881	0.981
1	0.996	0.993	0.993	0.991	0.993	0.994	0.996	0.970	0.991
2	0.996	0.995	0.994	0.994	0.996	0.996	0.995	0.975	0.993
3	0.995	0.994	0.996	0.995	0.997	0.996	0.998	0.965	0.992
4	0.996	0.995	0.995	0.997	0.999	0.995	0.996	0.958	0.991
5	0.995	0.996	0.999	0.994	0.998	0.996	0.993	0.966	0.992
6	0.995	0.996	0.996	0.995	0.996	0.999	0.994	0.961	0.991
7	0.995	0.997	0.996	0.995	0.995	0.994	0.995	0.963	0.991
8	0.994	0.992	0.995	0.993	0.996	0.995	0.997	0.965	0.991
9	0.996	0.995	0.997	0.994	0.998	0.995	0.998	0.963	0.992
10	0.994	0.996	0.999	0.997	0.995	0.998	0.995	0.940	0.989
11	0.996	0.997	0.995	0.994	0.997	0.997	0.998	0.967	0.993
12	0.994	0.997	0.998	0.996	0.996	0.995	0.995	0.953	0.991
13	0.998	0.997	0.998	0.996	0.995	0.997	0.993	0.962	0.992
14	0.997	0.996	0.993	0.996	0.994	0.997	0.995	0.948	0.990
15	0.995	0.997	0.995	1.000	0.996	0.993	0.997	0.963	0.992
16	0.997	0.998	0.996	0.997	0.996	0.997	0.994	0.950	0.991
17	0.997	0.998	0.995	0.995	0.994	0.993	0.994	0.977	0.993
18	0.996	0.998	0.995	0.997	0.996	0.997	0.992	0.970	0.993
19	0.999	0.997	0.996	0.996	0.997	0.993	0.997	0.971	0.993
20	0.995	0.998	0.998	0.995	0.994	0.991	0.996	0.961	0.991
21	0.996	0.995	0.997	0.995	0.993	0.995	0.996	0.960	0.991
22	0.998	0.996	0.994	0.996	0.995	0.994	0.998	0.960	0.991
23	0.997	0.997	0.996	0.996	0.996	0.993	0.997	0.963	0.992
24	0.997	0.997	0.997	0.995	0.994	0.997	0.996	0.973	0.993
25	0.994	0.996	0.999	0.993	0.997	0.995	0.996	0.963	0.992
26	0.994	0.997	0.993	0.994	0.995	0.993	0.994	0.952	0.989
27	0.995	0.995	0.995	0.996	0.994	0.995	0.996	0.976	0.993
28	0.995	0.994	0.993	0.995	0.996	0.995	0.994	0.965	0.991
29	0.995	0.998	0.995	0.999	0.997	0.995	0.995	0.963	0.992
30	0.993	0.995	0.997	0.997	0.995	0.995	0.996	0.964	0.992
31	0.994	0.998	0.996	0.995	0.996	0.999	0.994	0.884	0.982
average	0.996	0.996	0.996	0.995	0.996	0.995	0.995	0.958	**0.9909**

Table 13: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using `POL2MSG()`. Average results for 1K traces from device D_2.

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.989	0.989	0.993	0.992	0.994	0.997	0.993	0.851	0.975
1	0.983	0.988	0.994	0.996	0.992	0.994	0.994	0.966	0.988
2	0.996	0.996	0.994	0.996	0.995	0.995	0.991	0.975	0.992
3	0.995	0.994	0.997	0.993	0.996	0.992	0.997	0.974	0.992
4	0.992	0.996	0.998	0.995	0.997	0.999	0.993	0.968	0.992
5	0.996	0.995	0.996	0.996	0.993	0.993	0.997	0.981	0.993
6	0.995	0.996	0.994	0.996	0.994	0.997	0.995	0.960	0.991
7	0.993	0.995	0.992	0.992	0.996	0.992	0.993	0.973	0.991
8	0.993	0.997	0.989	0.993	0.991	0.993	0.996	0.953	0.988
9	0.992	0.997	0.993	0.997	0.994	0.995	0.995	0.946	0.989
10	0.993	0.996	0.995	0.998	0.997	0.996	0.996	0.944	0.989
11	0.993	0.996	0.991	0.992	0.995	0.991	0.997	0.959	0.989
12	0.995	0.993	0.995	0.995	0.995	0.995	0.995	0.946	0.989
13	0.992	0.999	0.997	0.992	0.997	0.994	0.993	0.965	0.991
14	0.993	0.996	0.995	0.994	0.996	0.990	0.995	0.952	0.989
15	0.994	0.993	0.994	0.995	0.998	0.997	0.992	0.969	0.992
16	0.989	0.994	0.992	0.995	0.996	0.993	0.994	0.946	0.987
17	0.994	0.994	0.993	0.997	0.996	0.996	0.996	0.969	0.992
18	0.996	0.995	0.997	0.995	0.995	0.991	0.997	0.961	0.991
19	0.994	0.997	0.994	0.993	0.996	0.991	0.993	0.967	0.991
20	0.994	0.995	0.996	0.994	0.994	0.994	0.994	0.962	0.990
21	0.995	0.994	0.994	0.996	0.995	0.995	0.997	0.971	0.992
22	0.995	0.992	0.993	0.991	0.994	0.994	0.994	0.962	0.989
23	0.992	0.992	0.994	0.994	0.993	0.991	0.996	0.963	0.989
24	0.993	0.991	0.992	0.997	0.993	0.995	0.995	0.962	0.990
25	0.995	0.992	0.996	0.992	0.996	0.993	0.993	0.966	0.990
26	0.994	0.993	0.993	0.998	0.993	0.995	0.994	0.919	0.985
27	0.996	0.995	0.996	0.994	0.997	0.996	0.991	0.958	0.990
28	0.992	0.993	0.994	0.993	0.994	0.999	0.994	0.964	0.990
29	0.995	0.993	0.991	0.996	0.997	0.992	0.994	0.973	0.991
30	0.994	0.996	0.995	0.994	0.996	0.994	0.995	0.935	0.987
31	0.995	0.994	0.993	0.993	0.992	0.988	0.993	0.902	0.981
average	0.993	0.994	0.994	0.994	0.995	0.994	0.994	0.955	**0.9893**

Table 14: Empirical probability to recover the message bit $m[8i+j]$ from a single trace using `POL2MSG()`. Average results for 1K traces from device D_3.

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.996	0.999	0.997	0.991	0.998	0.997	0.986	0.743	0.963
1	0.995	0.996	0.994	0.999	0.998	0.997	0.980	0.700	0.957
2	0.995	0.999	0.999	0.999	0.999	0.998	0.987	0.699	0.959
3	0.997	0.998	1.000	0.998	1.000	0.998	0.988	0.693	0.959
4	0.997	0.997	1.000	0.996	0.998	0.997	0.986	0.758	0.966
5	0.995	0.984	1.000	0.997	0.996	0.995	0.979	0.739	0.961
6	0.997	0.990	0.999	0.999	0.996	0.998	0.992	0.751	0.965
7	0.996	0.998	0.998	1.000	0.998	0.994	0.991	0.747	0.965
8	0.998	0.994	0.996	0.986	0.997	0.997	0.986	0.650	0.951
9	0.994	0.992	0.996	0.995	1.000	0.996	0.976	0.695	0.956
10	0.995	0.993	0.998	0.996	0.997	0.998	0.987	0.622	0.948
11	0.997	0.997	0.998	0.996	0.996	0.996	0.984	0.677	0.955
12	0.996	0.997	0.998	0.995	1.000	0.997	0.988	0.688	0.957
13	0.996	0.992	0.998	0.995	0.994	0.996	0.983	0.751	0.963
14	0.994	0.992	0.997	0.996	0.998	0.995	0.988	0.731	0.961
15	0.998	0.999	0.999	0.998	0.999	0.996	0.994	0.744	0.966
16	0.996	0.996	0.999	0.994	0.992	0.998	0.984	0.666	0.953
17	0.997	0.997	0.997	0.995	0.996	0.997	0.962	0.707	0.956
18	0.992	0.997	0.995	0.999	0.998	1.000	0.977	0.654	0.952
19	0.995	0.998	0.997	0.996	0.996	0.996	0.982	0.703	0.958
20	0.997	0.994	0.997	0.998	0.997	0.996	0.979	0.732	0.961
21	0.995	0.995	0.997	0.997	0.996	0.992	0.978	0.767	0.965
22	0.988	0.990	0.994	0.999	0.999	0.995	0.986	0.727	0.960
23	0.993	0.998	0.999	0.996	0.996	0.994	0.988	0.790	0.969
24	0.993	0.986	0.999	0.985	0.999	0.998	0.982	0.669	0.951
25	0.996	0.997	0.997	0.996	0.994	0.997	0.980	0.667	0.953
26	0.998	0.998	0.995	0.993	0.998	0.998	0.987	0.629	0.950
27	0.996	0.997	0.998	0.993	0.996	0.999	0.973	0.661	0.952
28	0.994	0.999	0.996	0.996	0.994	0.998	0.987	0.666	0.954
29	0.999	0.998	0.999	0.996	0.995	0.995	0.987	0.716	0.961
30	0.991	0.998	0.999	0.996	0.999	0.996	0.987	0.701	0.958
31	0.995	0.996	0.995	0.997	0.999	0.994	0.978	0.808	0.970
average	0.995	0.995	0.997	0.996	0.997	0.997	0.983	0.708	**0.9586**

Table 15: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using `poly_A2A()`. Average results for 1K traces from device D_1 (profiling).

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.845	0.992	0.997	0.994	0.999	0.998	0.995	0.998	0.977
1	0.997	0.999	0.998	0.997	0.996	0.998	0.995	0.997	0.997
2	0.998	0.999	0.998	0.999	0.999	0.999	0.999	0.999	0.999
3	0.997	0.999	0.999	0.996	0.994	0.997	0.997	0.999	0.997
4	1.000	0.998	0.999	0.998	0.999	0.998	0.999	0.997	0.998
5	0.998	0.998	0.999	0.999	0.997	0.997	0.997	0.998	0.998
6	0.999	0.997	0.998	0.997	0.997	0.997	0.996	0.998	0.997
7	0.998	0.998	0.998	0.993	0.999	0.997	0.997	0.998	0.997
8	0.999	0.996	1.000	0.995	0.999	0.999	0.997	0.999	0.998
9	0.999	0.996	0.999	0.998	0.996	0.999	0.997	0.999	0.998
10	0.998	0.995	0.998	0.998	0.997	0.999	0.998	0.997	0.997
11	0.998	0.997	0.999	0.996	0.996	0.998	0.998	0.998	0.998
12	0.996	0.997	0.999	0.998	0.998	1.000	0.997	0.996	0.998
13	1.000	0.995	0.998	0.997	0.998	1.000	1.000	0.999	0.998
14	0.997	0.997	0.996	0.998	0.996	0.996	0.998	0.996	0.997
15	0.995	0.997	0.996	0.992	0.999	0.998	0.999	0.998	0.997
16	0.998	0.997	0.995	0.999	0.994	0.998	0.994	0.998	0.997
17	0.999	0.998	0.999	0.998	0.992	0.997	0.991	0.998	0.996
18	0.995	0.996	0.997	0.996	0.996	0.999	0.995	0.999	0.997
19	0.997	0.998	0.996	0.999	0.997	0.997	0.998	0.998	0.998
20	0.999	0.996	0.998	0.995	0.996	0.997	0.997	0.998	0.997
21	0.999	0.998	0.997	0.997	0.994	0.997	0.996	0.996	0.997
22	0.996	0.994	0.996	0.997	0.996	1.000	0.998	0.995	0.996
23	0.997	0.998	0.995	0.994	0.997	0.997	0.992	0.993	0.995
24	0.996	0.997	0.997	0.994	0.998	0.999	0.995	0.997	0.997
25	0.994	0.996	0.994	0.998	0.998	0.998	0.996	0.998	0.997
26	0.995	0.997	0.994	0.993	0.997	0.998	0.998	0.997	0.996
27	0.995	0.998	0.998	0.995	0.996	0.996	0.999	0.999	0.997
28	0.995	0.997	0.996	0.997	0.998	0.994	0.997	0.995	0.996
29	0.995	0.997	0.997	0.998	0.996	0.996	0.997	0.999	0.997
30	0.992	0.996	0.995	0.992	0.995	0.997	0.997	0.997	0.995
31	0.995	0.997	0.995	0.997	0.994	0.997	0.895	0.666	0.942
average	0.992	0.997	0.997	0.996	0.997	0.998	0.994	0.987	**0.9947**

Table 16: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using `poly_A2A()`. Average results for 1K traces from device D_2.

D_2	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.836	0.992	0.995	0.992	0.995	0.993	0.996	0.998	0.975
1	0.996	0.998	0.997	0.991	0.994	0.993	0.996	0.998	0.995
2	0.998	0.994	0.998	0.996	0.998	0.990	0.992	0.989	0.994
3	0.991	0.997	0.991	0.994	0.996	0.997	0.997	0.994	0.995
4	0.994	0.990	0.991	0.992	0.994	0.990	0.996	0.991	0.992
5	0.996	0.999	0.994	0.997	0.999	0.993	0.997	0.994	0.996
6	0.997	0.994	0.995	0.995	0.993	0.993	0.996	0.993	0.995
7	0.995	0.998	0.992	0.986	0.995	0.994	0.999	0.994	0.994
8	0.998	0.996	0.998	0.996	0.994	0.994	0.991	0.995	0.995
9	0.988	0.992	0.993	0.994	0.995	0.992	0.993	0.992	0.992
10	0.993	0.992	0.988	0.993	0.991	0.991	0.992	0.986	0.991
11	0.990	0.990	0.994	0.989	0.996	0.993	0.994	0.993	0.992
12	0.992	0.994	0.992	0.990	0.995	0.996	0.998	0.997	0.994
13	0.994	0.995	0.994	0.995	0.995	0.992	0.991	0.994	0.994
14	0.993	0.994	0.993	0.991	0.992	0.991	0.992	0.988	0.992
15	0.992	0.993	0.993	0.989	0.988	0.994	0.996	0.994	0.992
16	0.989	0.993	0.995	0.992	0.993	0.991	0.998	0.989	0.992
17	0.993	0.992	0.993	0.995	0.993	0.996	0.989	0.993	0.993
18	0.989	0.993	0.993	0.988	0.994	0.991	0.988	0.992	0.991
19	0.992	0.994	0.990	0.992	0.991	0.992	0.992	0.992	0.992
20	0.992	0.992	0.993	0.993	0.993	0.994	0.997	0.997	0.994
21	0.994	0.993	0.994	0.995	0.990	0.990	0.990	0.991	0.992
22	0.997	0.990	0.994	0.991	0.990	0.992	0.993	0.989	0.992
23	0.995	0.995	0.991	0.992	0.990	0.995	0.991	0.997	0.993
24	0.991	0.995	0.993	0.992	0.989	0.996	0.994	0.986	0.992
25	0.994	0.995	0.993	0.992	0.989	0.991	0.992	0.996	0.993
26	0.989	0.991	0.992	0.990	0.990	0.992	0.992	0.992	0.991
27	0.989	0.991	0.992	0.992	0.994	0.994	0.992	0.991	0.992
28	0.993	0.987	0.990	0.984	0.989	0.988	0.996	0.993	0.990
29	0.989	0.992	0.991	0.995	0.989	0.992	0.990	0.992	0.991
30	0.989	0.988	0.989	0.993	0.994	0.991	0.989	0.991	0.990
31	0.994	0.990	0.994	0.991	0.997	0.992	0.857	0.704	0.940
average	0.988	0.993	0.993	0.992	0.993	0.993	0.989	0.984	**0.9906**

Table 17: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using `poly_A2A()`. Average results for 1K traces from device D_3.

D_3	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.810	0.880	0.889	0.859	0.928	0.939	0.933	0.923	0.895
1	0.931	0.900	0.947	0.903	0.933	0.927	0.923	0.916	0.923
2	0.934	0.924	0.944	0.899	0.928	0.882	0.933	0.885	0.916
3	0.905	0.900	0.897	0.864	0.928	0.904	0.928	0.883	0.901
4	0.910	0.882	0.901	0.862	0.945	0.927	0.950	0.932	0.914
5	0.954	0.927	0.934	0.903	0.952	0.935	0.943	0.925	0.934
6	0.941	0.927	0.940	0.903	0.930	0.901	0.937	0.888	0.921
7	0.921	0.908	0.911	0.886	0.940	0.918	0.927	0.893	0.913
8	0.934	0.923	0.921	0.908	0.942	0.940	0.943	0.914	0.928
9	0.925	0.924	0.921	0.882	0.939	0.933	0.927	0.904	0.919
10	0.936	0.922	0.916	0.871	0.930	0.891	0.934	0.907	0.913
11	0.918	0.902	0.885	0.874	0.927	0.882	0.922	0.897	0.901
12	0.918	0.900	0.906	0.864	0.967	0.916	0.940	0.930	0.918
13	0.947	0.912	0.939	0.908	0.962	0.947	0.965	0.924	0.938
14	0.929	0.928	0.943	0.890	0.938	0.888	0.944	0.902	0.920
15	0.928	0.908	0.917	0.883	0.944	0.915	0.933	0.900	0.916
16	0.924	0.915	0.912	0.907	0.951	0.941	0.955	0.945	0.931
17	0.948	0.946	0.953	0.902	0.945	0.942	0.943	0.925	0.938
18	0.945	0.924	0.946	0.903	0.940	0.915	0.929	0.911	0.927
19	0.912	0.910	0.930	0.899	0.938	0.911	0.942	0.898	0.917
20	0.926	0.891	0.929	0.880	0.961	0.942	0.969	0.944	0.930
21	0.936	0.951	0.954	0.908	0.963	0.946	0.955	0.949	0.945
22	0.968	0.948	0.950	0.939	0.940	0.913	0.938	0.926	0.940
23	0.930	0.937	0.943	0.911	0.957	0.932	0.955	0.909	0.934
24	0.933	0.936	0.944	0.921	0.960	0.938	0.950	0.949	0.941
25	0.944	0.933	0.938	0.923	0.956	0.938	0.941	0.924	0.937
26	0.949	0.936	0.959	0.919	0.944	0.930	0.936	0.915	0.936
27	0.947	0.921	0.935	0.913	0.953	0.920	0.952	0.928	0.934
28	0.938	0.925	0.926	0.901	0.960	0.944	0.965	0.951	0.939
29	0.943	0.948	0.957	0.922	0.969	0.950	0.960	0.948	0.950
30	0.952	0.946	0.945	0.937	0.963	0.941	0.946	0.940	0.946
31	0.950	0.937	0.933	0.898	0.943	0.934	0.905	0.650	0.894
average	0.931	0.921	0.930	0.898	0.946	0.924	0.941	0.910	**0.9253**

Table 18: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using both points. Average results for 1K traces from device D_1 (profiling).

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.993	0.999	0.998	1.000	0.997	0.998	0.999	0.995	0.997
1	0.999	0.999	1.000	1.000	1.000	0.999	0.998	0.998	0.999
2	0.999	0.998	1.000	1.000	0.999	0.999	0.999	1.000	0.999
3	1.000	0.999	1.000	0.998	0.998	1.000	0.999	0.998	0.999
4	0.999	0.998	0.998	0.999	1.000	0.999	0.999	0.998	0.999
5	1.000	0.998	1.000	0.999	0.998	0.999	0.999	0.999	0.999
6	0.998	0.997	0.998	0.999	0.999	1.000	0.998	1.000	0.999
7	0.999	0.998	1.000	0.997	0.999	0.999	0.999	0.999	0.999
8	0.998	1.000	1.000	0.998	0.997	0.999	0.999	0.999	0.999
9	0.999	1.000	0.999	0.998	1.000	0.998	0.999	0.999	0.999
10	0.999	0.999	1.000	0.999	0.999	1.000	0.999	0.998	0.999
11	0.998	0.998	0.999	0.997	0.999	0.999	0.999	0.998	0.998
12	0.999	0.998	0.999	0.998	0.999	1.000	0.997	0.997	0.998
13	0.999	0.999	0.999	0.998	0.997	1.000	1.000	0.998	0.999
14	1.000	0.998	0.997	1.000	0.999	1.000	0.998	0.997	0.999
15	0.998	0.999	0.997	1.000	0.998	0.999	0.998	0.999	0.998
16	0.999	0.999	0.999	0.999	0.999	0.997	0.997	0.999	0.998
17	0.999	0.999	0.999	0.997	0.999	0.998	0.997	0.997	0.998
18	1.000	0.996	0.997	0.999	0.996	0.998	0.999	1.000	0.998
19	0.999	0.999	1.000	0.998	0.999	0.998	0.999	0.996	0.998
20	0.998	0.998	1.000	0.999	0.996	0.995	0.998	0.997	0.998
21	1.000	0.997	0.998	0.999	0.996	0.998	0.998	0.996	0.998
22	1.000	0.998	0.997	0.998	0.998	0.997	0.997	0.997	0.998
23	0.999	0.997	0.995	0.998	0.999	0.998	0.997	0.993	0.997
24	1.000	0.996	0.997	0.997	0.998	0.998	0.995	0.998	0.997
25	0.996	0.997	0.999	0.996	1.000	0.997	0.997	0.998	0.997
26	0.997	0.999	0.996	0.999	0.997	0.997	0.998	0.998	0.998
27	1.000	0.998	0.999	0.997	0.998	0.996	0.997	0.999	0.998
28	0.998	0.998	0.997	0.997	0.999	0.999	0.998	0.999	0.998
29	0.998	0.997	0.997	0.999	0.997	0.997	0.999	0.996	0.998
30	0.996	1.000	0.999	0.999	0.996	0.997	0.999	0.996	0.998
31	0.997	0.998	0.997	0.996	0.999	0.998	0.999	0.952	0.992
average	0.999	0.998	0.998	0.998	0.998	0.998	0.998	0.996	**0.9981**

Table 19: Empirical probability to recover the message bit $m[8i+j]$ from a single trace using both points. Average results for 1K traces from device D_2.

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.987	0.998	0.999	0.999	0.997	0.998	0.998	0.999	0.997
1	0.994	1.000	0.998	0.999	0.999	0.996	0.999	1.000	0.998
2	0.997	0.999	0.999	0.999	0.996	0.996	0.999	0.997	0.998
3	0.999	1.000	0.998	1.000	0.996	0.999	0.997	0.998	0.998
4	0.999	0.999	0.999	0.995	0.999	0.996	0.998	0.997	0.998
5	0.998	0.998	1.000	0.997	0.999	0.996	0.998	0.995	0.998
6	0.998	1.000	0.996	0.997	0.997	0.998	0.997	0.997	0.997
7	1.000	1.000	0.996	0.998	0.999	0.997	0.996	0.997	0.998
8	0.998	0.996	1.000	0.999	0.999	0.997	0.997	0.995	0.998
9	0.994	0.997	0.998	0.998	0.996	0.998	0.994	0.995	0.996
10	0.998	0.996	0.999	0.996	0.996	0.999	0.997	0.997	0.997
11	0.996	0.996	0.997	0.999	0.997	0.997	0.997	0.996	0.997
12	0.998	0.998	0.997	0.995	0.998	0.996	0.999	0.996	0.997
13	0.997	0.995	0.998	0.997	0.995	0.995	0.995	0.997	0.996
14	0.995	0.995	0.995	0.991	0.995	0.997	0.996	0.995	0.995
15	0.996	0.995	0.995	0.994	0.997	0.997	0.996	0.995	0.996
16	0.995	0.996	0.996	0.996	0.997	0.994	0.995	0.995	0.996
17	0.996	0.996	0.994	0.996	0.994	0.995	0.994	0.991	0.994
18	0.996	0.996	0.996	0.994	0.992	0.994	0.993	0.996	0.995
19	0.997	0.995	0.996	0.994	0.993	0.992	0.993	0.991	0.994
20	0.997	0.996	0.996	0.994	0.996	0.996	0.998	0.991	0.995
21	0.996	0.995	0.996	0.995	0.995	0.992	0.991	0.993	0.994
22	0.996	0.996	0.994	0.996	0.993	0.995	0.994	0.991	0.994
23	0.997	0.994	0.996	0.993	0.991	0.995	0.994	0.996	0.994
24	0.994	0.999	0.993	0.995	0.992	0.995	0.993	0.988	0.994
25	0.996	0.998	0.997	0.993	0.993	0.992	0.993	0.992	0.994
26	0.995	0.994	0.992	0.995	0.989	0.994	0.996	0.992	0.993
27	0.998	0.995	0.995	0.992	0.996	0.997	0.997	0.993	0.995
28	0.996	0.993	0.995	0.989	0.994	0.997	0.995	0.993	0.994
29	0.995	0.993	0.995	0.997	0.997	0.993	0.996	0.995	0.995
30	0.995	0.990	0.995	0.995	0.994	0.993	0.995	0.996	0.994
31	0.996	0.996	0.993	0.993	0.993	0.991	0.996	0.936	0.987
average	0.996	0.996	0.996	0.996	0.995	0.996	0.996	0.993	**0.9955**

Table 20: Empirical probability to recover the message bit $m[8i + j]$ from a single trace using both points. Average results for 1K traces from device D_3.

D_1	j								
i	0	1	2	3	4	5	6	7	**average**
0	0.982	0.966	0.976	0.962	0.966	0.954	0.968	0.941	0.964
1	0.981	0.941	0.991	0.978	0.968	0.961	0.976	0.941	0.967
2	0.990	0.958	0.988	0.971	0.968	0.953	0.966	0.923	0.965
3	0.992	0.952	0.977	0.966	0.963	0.958	0.973	0.918	0.962
4	0.988	0.947	0.979	0.959	0.970	0.963	0.980	0.964	0.969
5	0.991	0.963	0.990	0.987	0.976	0.976	0.982	0.945	0.976
6	0.992	0.977	0.992	0.985	0.973	0.958	0.975	0.928	0.972
7	0.989	0.966	0.985	0.969	0.973	0.960	0.975	0.927	0.968
8	0.993	0.959	0.989	0.956	0.973	0.970	0.980	0.939	0.970
9	0.990	0.953	0.982	0.957	0.980	0.974	0.976	0.939	0.969
10	0.987	0.959	0.988	0.965	0.963	0.954	0.975	0.943	0.967
11	0.989	0.962	0.978	0.955	0.963	0.953	0.971	0.928	0.962
12	0.987	0.941	0.981	0.956	0.987	0.966	0.975	0.950	0.968
13	0.993	0.955	0.991	0.969	0.989	0.976	0.990	0.946	0.976
14	0.985	0.966	0.992	0.978	0.969	0.963	0.977	0.932	0.970
15	0.993	0.960	0.990	0.962	0.977	0.968	0.985	0.932	0.971
16	0.995	0.955	0.987	0.973	0.973	0.977	0.982	0.971	0.977
17	0.991	0.976	0.988	0.967	0.981	0.981	0.984	0.947	0.977
18	0.988	0.964	0.990	0.980	0.960	0.976	0.977	0.932	0.971
19	0.984	0.969	0.985	0.964	0.971	0.972	0.975	0.923	0.968
20	0.987	0.948	0.982	0.974	0.977	0.976	0.992	0.954	0.974
21	0.984	0.977	0.986	0.976	0.981	0.981	0.991	0.960	0.979
22	0.985	0.986	0.990	0.987	0.962	0.970	0.973	0.947	0.975
23	0.989	0.976	0.989	0.978	0.985	0.967	0.986	0.935	0.976
24	0.991	0.977	0.987	0.972	0.983	0.985	0.984	0.961	0.980
25	0.994	0.966	0.993	0.981	0.985	0.983	0.976	0.948	0.978
26	0.991	0.971	0.993	0.973	0.976	0.980	0.980	0.945	0.976
27	0.990	0.968	0.987	0.970	0.981	0.977	0.991	0.955	0.977
28	0.989	0.957	0.992	0.967	0.978	0.981	0.990	0.962	0.977
29	0.996	0.982	0.995	0.985	0.985	0.983	0.990	0.967	0.985
30	0.991	0.982	0.989	0.983	0.979	0.983	0.988	0.955	0.981
31	0.997	0.975	0.989	0.984	0.981	0.977	0.975	0.831	0.964
average	0.989	0.964	0.987	0.972	0.975	0.971	0.980	0.940	**0.9723**