Breaking Masked Implementations of the Clyde-Cipher by Means of Side-Channel Analysis

A Report on the CHES Challenge Side-Channel Contest 2020

Aron Gohr[1], Friederike Laus[2] and Werner Schindler[2]

[1] aron.gohr@gmail.com
[2] Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn, Germany
firstname.lastname@bsi.bund.de

Abstract. In this paper we present our solution to the CHES Challenge 2020, the task of which it was to break masked hardware respective software implementations of the lightweight cipher Clyde by means of side-channel analysis. We target the secret cipher state after processing of the first S-box layer. Using the provided trace data we obtain a strongly biased posterior distribution for the secret-shared cipher state at the targeted point; this enables us to see exploitable biases even *before* the secret sharing based masking. These biases on the unshared state can be evaluated one S-box at a time and combined across traces, which enables us to recover likely key hypotheses S-box by S-box.
In order to see the shared cipher state, we employ a deep neural network similar to the one used by Gohr, Jacob and Schindler to solve the CHES 2018 AES challenge. We modify their architecture to predict the exact bit sequence of the secret-shared cipher state. We find that convergence of training on this task is unsatisfying with the standard encoding of the shared cipher state and therefore introduce a different encoding of the prediction target, which we call the *scattershot encoding*. In order to further investigate how exactly the scattershot encoding helps to solve the task at hand, we construct a simple synthetic task where convergence problems very similar to those we observed in our side-channel task appear with the naive target data encoding but disappear with the scattershot encoding.
We complete our analysis by showing results that we obtained with a "classical" method (as opposed to an AI-based method), namely the stochastic approach, that we generalize for this purpose first to the setting of shared keys. We show that the neural network draws on a much broader set of features, which may partially explain why the neural-network based approach massively outperforms the stochastic approach. On the other hand, the stochastic approach provides insights into properties of the implementation, in particular the observation that the S-boxes behave very different regarding the easiness respective hardness of their prediction.

Keywords: Lightweight cryptography · Clyde-cipher · Side-channel analysis · Countermeasures · Masking · Secret-sharing · ISW-Multiplication · Deep neural network · Residual neural network · Stochastic approach · CHES Challenge 2020

1 Introduction

Nowadays, there are several emerging areas such as the Internet of Things, healthcare, distributed control systems, sensor networks or cyber physical systems, in which highly-constrained devices are interconnected, typically communicating wirelessly with one another, and working in concert to accomplish some task. This poses new challenges to the cryptographic community, as these devices process more and more sensitive data while being

themselves usually not very well protected and thus easily accessible. Furthermore, because the majority of current cryptographic algorithms was designed for desktop/server environments, many of these algorithms do not fit into constrained devices whose computational power is limited.
This raises the need for new cryptographic algorithms that on the one hand can be operated under limited resources and on the other hand can be protected against side-channel and fault attacks without loosing too much of its performance. It further served the National Institute of Standards and Technology (NIST) in August 2018 as motivation for a competition searching for so-called *lightweight* ciphers that fulfill the demands for efficiency and side-channel resistance mentioned above [14]. NIST received 57 submissions to be considered for standardization. After the initial review of the submissions, 56 were selected as Round 1 candidates, out of which 32 were selected to continue to round 2. On March 29, 2021, NIST announced ten finalists that are currently standardized.
One of the second round candidates is the *Spook* authenticated encryption scheme with associated data (AEAD) [4]. Its block cipher is the so called *Clyde* cipher that follows a tweakable LS-design consisting of 6 rounds. Clyde can efficiently be protected against side-channel attacks by Boolean masking and the aim of the CHES challenge 2020 was to examine the effectiveness of this countermeasure. For that purpose several challenges varying in the amount of masking were posted prior to the CHES conference 2020. A team of eight colleagues[1] at BSI worked on the software challenges and won all the prizes that were finally awarded.

Main Contributions In this paper, we provide a detailed description as well as a further analysis of our solution to the different challenges. Using a deep neural network, we were able to overcome the masking countermeasure by extracting a lot of information on the unmasked cipher state already from a single trace. This required seeing all bits of the masked internal state with very high bias, which proved difficult, since some bits of the internal state were much more challenging to learn for our neural networks than others. Indeed, our attempts to directly predict some of these bits seemed to not converge to results better than random guessing.
In order to achieve convergence for all bits of the targeted internal state variable, we represent the internal state in a way which is designed to achieve the following objectives:

- The chosen state representation allows for full and efficient reconstruction of the standard representation of the internal state as a bit-vector, even in the presence of noise (i.e. taking a target state $s \in \mathbb{F}_2^{128d}$ with alternative representation $v \in \mathbb{R}^n$, any $v' \in \mathbb{R}^n$ with small $\|v - v'\|_2$ should suffice to reconstruct s without much error).

- The chosen state representation should make it difficult to reduce loss by making progress on the prediction of easy bits in isolation. Instead, progress on predicting any part of the target vector should be correlated with progress on the whole prediction problem.

We achieve this by a surprisingly simple method, namely by sending the target values through a *random linear map* before training. Given a list of training traces X and a list of target internal state values Y, we hence essentially replace $y_i \in Y$ with $y_i' := Ay_i$, where multiplication is performed over $\mathbb{Z}$ and A is a random binary matrix. We then try to learn to predict y_i' from $x_i \in X$ by minimizing standard mean squared error loss. When using the trained model on new power traces, we simply multiply the model output with the Moore-Penrose pseudoinverse A^+ of A to obtain predictions for each bit of the target state.

[1] Members of the BSI-team (in alphabetical order): Tobias Damm, Aron Gohr, Sven Jacob, Dominik Klein, Friederike Laus, Natalie Peter, Werner Schindler, Vivien Thiel.

We call this method of sending the target values that our model is intended to ultimately predict through a random linear function the *scattershot encoding*.
That a simple random transformation of the target values helps to obtain a better model is to some extent surprising, since a linear post-processing step of this type is in principle easy to represent for a neural network. This makes the scattershot encoding worthy to be studied on its own. We show that convergence problems similar to those observed with the side-channel task under consideration appear also in a synthetic problem, namely when trying to teach a neural network to compute a simple $\mathbb{F}_2$-linear function, and that the scattershot encoding helps achieve uniform convergence at that task as well.
Having obtained models for all of the software challenges of the CHES 2020 side-channel contest, we afterwards study their performance. We find that the easiest, 3-share challenge can be solved with around 20-30 traces, whereas a few ten thousand traces are required for the hardest 8-share challenge. Our analyses also reveal that different nibbles of the unshared internal state still represent vastly different levels of difficulty to our neural network, which is unexpected, since the implementation runs all nibbles in parallel using the same code.
Finally, we complete our findings by applying the stochastic approach as a "classical" statistical profiled method to the problem at hand. We find that our neural network and the stochastic approach consider the same parts of the internal state to be difficult to predict; however, the performance of the neural network is vastly superior. An further analysis of the leakage the two approaches exploit might serve as an explanation for this behavior, as it turns out that the AI-based approach is able to extract information from a much larger range of the trace.

Related Work Deep learning recently entered the field of side-channel analysis and turned out to be a powerful tool that enlarges an attackers' capabilities compared to other techniques. For a recent overview on the subject we refer to [15] and the references therein.
The neural network architecture used in our deep learning approach was first introduced in [8] to break a protected AES implementation. In [8], the side-channel attack extracted the Hamming weights of all AES subkeys. Afterwards, a equation solving stage was used to perform a limited amount of error correction on the extracted Hamming weights while simultaneously deriving the full key values from the Hamming weight guesses. This work uses the same network architecture, but cannot use any of the subsequent post-processing ideas used in [8]. There are multiple reasons for this: the Clyde-128 key schedule does not impose significant constraints on the values we might recover the side-channel data made available in the contest does not cover the full round number of the cipher anyway and we never get to see any unmasked internal state in the Clyde attack.
Our solution to the CHES challenge presented in this paper combines a signal extraction stage relying heavily on a neural network with classical elements, such as the choice of a suitable leakage target variable and leakage model as well as a manually designed key extraction stage that combines leakage over many traces to derive the target key.
In general, the question whether AI-based methods are superior over classical statistical methods is of great interest, and we complete our analysis of the strength of the neural network based leakage extraction stage by developing an alternative solution based on the stochastic approach. Combining classical and machine learning based techniques in order to achieve the best possible exploitation of the available signal is common in state-of-the-art side-channel attacks, since it is expected that some parts of optimal attacks will be difficult for neural networks to execute. Accordingly, there is a significant amount of recent research that is aimed at combining the strengths of deep learning and classical methods in side-channel analysis; for instance, a first approach that combines the stochastic approach with deep learning methods has recently been proposed in [18].
Another attack on the same data set as used in the present paper has been proposed in [5].

The authors essentially show that a template attack can reach an overall similar level of performance as our attack if it uses a deep understanding of the implementation. We show that results comparable to theirs can be achieved by a neural network based attack that exploits only the knowledge of the cipher state at one well-chosen point of the cipher's execution if a good representation of the training target is chosen.
Other notable recent work considered the possibility of combining classical template attacks with deep learning based preprocessing of input traces, achieving competitive results on several datasets at relatively low training cost [17].

Outline The outline of the remaining paper is as follows: First, we describe in Section 2 the side-channel contest of the CHES challenge 2020 before we detail the Clyde-128 cipher and its masked implementation in Section 3. Then, in Section 4 we present our solution to the CHES challenge based on a residual neural network and provide in Section 5 further insights gained with the stochastic approach. Finally, conclusions and an outline of future work are given in Section 6.

2 CHES Challenge 2020 Side-Channel Contest

2.1 Objectives

The CHES challenge 2020 consisted in a side-channel contest whose objective it was to use side-channel analysis to break masked implementations of the Clyde-128 cipher [4]. The challenge involved a total of seven sub-challenges. Four of the challenges targeted a software implementation of Clyde-128 protected by an efficient variant of ISW-masking proposed by Goudarzi and Rouvain [9] with 3, 4, 6 and 8 shares. The remaining three challenges targeted a hardware implementation of Clyde-128 using the glitch-resistant ISW-masking developed by Cassiers et al. [6]. This paper focuses completely on the software challenges. In the context of this contest, breaking an implementation means deriving from trace and tweak information a simple ranking of all possible keys such that the expected rank of the correct key is below 2^{32}.

2.2 Data Sets and Evaluation Framework

For each challenge, the organizers collected and published large random- and fixed-key datasets. The random-key datasets for the software challenge consisted of 200,000 power traces covering the first half of the first round of the execution of Clyde-128 on the all-zero message block with varying tweak values; tweaks, keys and shared keys were also given. The fixed-key datasets contained only the trace and tweak information. Additionally, the organizers provided the source code of the corresponding implementations, proof-of-concept attacks on the weakest software and hardware targets, documentation explaining the main ideas behind the implementations and an evaluation environment based on the container tool singularity [13]. Submissions were tested against unknown test data held by the organizers.

3 Clyde-128 Cipher

In this section, we briefly describe the Clyde-128 cipher, before we detail its masked software implementation in the context of the CHES challenge 2020. The description of the cipher is based on [4], with slightly adapted notation, while the description of the masked implementation is based on code inspection.

3.1 TLS Design

The Clyde-128 cipher is a tweakable block cipher that is part of Spook, an algorithm for authenticated encryption with associated data (AEAD) and a second-round candidate of the NIST Lightweight Cryptography competition[2]. It relies on a tweakable LS-design (TLS-design) [11, 10] and works on $n = (s \cdot l)$-bit states, where $s = 4$ denotes the size of the S-box and $l = 32$ the size of the L-box. The full cipher state is referred to as $x \in \mathbb{F}_2^{s \times l}$, a row state is denoted as $x[i, \cdot]$, $i = 0, \ldots, s-1$, and a column state as $x[\cdot, j]$, $j = 0, \ldots, l-1$. In the following, we will simultaneously use bitmatrices $x \in \mathbb{F}_2^{s \times l}$ as well as rowwise reshaped bitvectors $B \in \mathbb{F}_2^{s \cdot l}$, which are related via

$$x[i, j] = B[i \cdot l + j].$$

The basic TLS-design is summarized in Algorithm 1 and next, we describe the different building blocks in more details.

Algorithm 1 Clyde-128 TLS-design with s-bit S-box and $2l$-bit L-box.

Input: plaintext $p \in \mathbb{F}_2^{s \times l}$, tweakey $\mathrm{TK} \in \mathbb{F}_2^{s \times l}$, s-bit S-box S, $2l$-bit modified L-box L', where $s = 4$ and $l = 32$.
Output: ciphertext matrix $x \in \mathbb{F}_2^{s \times l}$
$x \leftarrow p \oplus \mathrm{TK}(0)$
for $\sigma = 0, \ldots, N_s - 1$ **do**
  for $\rho = 0, 1$ **do**
    $r = 2\sigma + \rho$
    for $j = 0, \ldots, l-1$ **do**
      $x[\cdot, j] = S\big(x[\cdot, j]\big)$
    end for
    for $i = 0, \ldots, \frac{s}{2} - 1$ **do**
      $\big(x[2i, \cdot], x[2i+1, \cdot]\big) = L'\big(x[2i, \cdot], x[2i+1, \cdot]\big)$
    end for
    $x[\cdot, 0] \leftarrow x[\cdot, 0] \oplus W(r)$
  end for
  $x \leftarrow x \oplus \mathrm{TK}(\sigma + 1)$
end for

Tweakey The tweakey scheduling algorithm for the n-bit key $k \in \mathbb{F}_2^n$ and the n-bit tweak $T \in \mathbb{F}_2^n$ reads as follows: First, the tweak is divided into $\frac{n}{2}$-bit halves t_0 and t_1, that is, $T = t_0 || t_1$. Then, the tweakey depends on the remainder of the round index i by division by three as

$$\begin{aligned} \mathrm{TK}(3i) &= k \oplus (t_0 || t_1) \\ \mathrm{TK}(3i+1) &= k \oplus (t_0 \oplus t_1 || t_1) \\ \mathrm{TK}(3i+2) &= k \oplus (t_0 || t_0 \oplus t_1). \end{aligned}$$

***S*-Box** The 4-bit S-box is a variant of the S-box proposed in [3] and can efficiently be computed using four AND-gates and four XOR-gates as follows: Let $x \in \mathbb{F}_2^4$ be a 4-bit

[2]For further information, see https://csrc.nist.gov/projects/lightweight-cryptography

word, then $y = S(x)$ is given by

$$\begin{aligned} y[1] &= \big(x[0] \odot x[1]\big) \oplus x[2], \\ y[0] &= \big(x[3] \odot x[0]\big) \oplus x[1], \\ y[3] &= \big(y[1] \odot x[3]\big) \oplus x[0], \\ y[2] &= \big(y[0] \odot y[1]\big) \oplus x[3]. \end{aligned}$$

In fact, the for-loop in lines 7-9 of Algorithm 1 for the computation of the S-box is not necessary, but for a full Clyde-state $x \in \mathbb{F}_2^{4\times 32}$, the S-box can be computed as

$$\begin{aligned} y[1,\cdot] &= \big(x[0,\cdot] \odot x[1,\cdot]\big) \oplus x[2,\cdot], \\ y[0,\cdot] &= \big(x[3,\cdot] \odot x[0,\cdot]\big) \oplus x[1,\cdot], \\ y[3,\cdot] &= \big(y[1,\cdot] \odot x[3,\cdot]\big) \oplus x[0,\cdot], \\ y[2,\cdot] &= \big(y[0,\cdot] \odot y[1,\cdot]\big) \oplus x[3,\cdot]. \end{aligned} \tag{1}$$

L-Box The modified L-box L' acts on pairs of 32-bit words $(x, y) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$ as

$$(a, b) = L'(x, y) = \begin{pmatrix} \mathrm{circ}(\texttt{0xec045008}) \cdot x^{\mathrm{T}} \oplus \mathrm{circ}(\texttt{0x36000f60}) \cdot y^{\mathrm{T}} \\ \mathrm{circ}(\texttt{0x1b0007b0}) \cdot x^{\mathrm{T}} \oplus \mathrm{circ}(\texttt{0xec045008}) \cdot y^{\mathrm{T}} \end{pmatrix},$$

where $\mathrm{circ}(c)$ denotes a circulant matrix with first row c in hexadecimal notation, so that $c = \sum_{i=0}^{31} c_i 2^i$ corresponds to the row vector $(c_0, \ldots, c_{31})$. The L-box can efficiently be implemented as well using six word-level (left) rotations and six 32-bit XORs per word:

$$\begin{aligned} a &= x \oplus \mathrm{rot}(x, 12), \\ b &= y \oplus \mathrm{rot}(y, 12), \\ a &= a \oplus \mathrm{rot}(a, 3), \\ b &= b \oplus \mathrm{rot}(b, 3), \\ a &= a \oplus \mathrm{rot}(x, 17), \\ b &= b \oplus \mathrm{rot}(y, 17), \\ c &= a \oplus \mathrm{rot}(a, 31), \\ d &= b \oplus \mathrm{rot}(b, 31), \\ a &= a \oplus \mathrm{rot}(d, 26), \\ b &= b \oplus \mathrm{rot}(c, 25), \\ a &= a \oplus \mathrm{rot}(c, 15), \\ b &= b \oplus \mathrm{rot}(d, 15). \end{aligned} \tag{2}$$

Also the L-box can be computed in parallel for the two row pairs $\big(x[0,\cdot], x[1,\cdot]\big)$ and $\big(x[2,\cdot], x[3,\cdot]\big)$.

Round Constants Finally, the round constants, which can be computed using a 4-bit LSFR, are given by four bits that are XORed with the first column of the Clyde-state. They read for the different rounds $r = 0, \ldots, 11$ as

$r = 0$	$(1,0,0,0)$	$r = 1$	$(0,1,0,0)$	$r = 2$	$(0,0,1,0)$	$r = 3$	$(0,0,0,1)$
$r = 4$	$(1,1,0,0)$	$r = 5$	$(0,1,1,0)$	$r = 6$	$(0,0,1,1)$	$r = 7$	$(1,1,0,1)$
$r = 8$	$(1,0,1,0)$	$r = 9$	$(0,1,0,1)$	$r = 10$	$(1,1,1,0)$	$r = 11$	$(0,1,1,1)$.

3.2 Masked Implementation

In the context of the CHES challenge 2020, the implementation of the Clyde-128 cipher is masked according to [9] by decomposing a sensitive variable $x \in \mathbb{F}_2^{s\times l}$ into d shares $(x_0, \ldots, x_{d-1})$ such that

$$x = x_0 \oplus \cdots \oplus x_{d-1}.$$

In the sequel, we add the shares as a third dimension to the data and denote masked versions of a variable/operator with $\tilde{\cdot}$, e.g. $\tilde{x} \in \mathbb{F}_2^{s\times l\times d}$ denotes the masked version of $x \in \mathbb{F}_2^{s\times l}$ and for $(i, j) \in \{0, \ldots, s-1\} \times \{0, \ldots, l-1\}$ it holds

$$x[i, j] = \bigoplus_{\ell=0}^{d-1} \tilde{x}[i, j, \ell].$$

The masked Clyde-128 algorithm is summarized in Algorithm 2.

Algorithm 2 Masked Clyde-128 Algorithm.

1: **Input:** plaintext $p \in \mathbb{F}_2^{s\times l}$, tweak $T \in \mathbb{F}_2^{s\times l}$, shared key $\tilde{k} \in \mathbb{F}_2^{s\times l\times d}$, masked s-bit S-box $\tilde{S}$, masked $2l$-bit modified L-box $\tilde{L}'$, where $s = 4$ and $l = 32$.
2: **Output:** ciphertext matrix $x \in \mathbb{F}_2^{s\times l}$
3: $\tilde{x} \leftarrow \tilde{k}$
4: $\tilde{x}[\cdot, \cdot, 0] \leftarrow \tilde{x}[\cdot, \cdot, 0] \oplus T(0)$
5: $\tilde{x}[\cdot, \cdot, 0] \leftarrow \tilde{x}[\cdot, \cdot, 0] \oplus p$
6: **for** $\sigma = 0, \ldots, 5$ **do**
7: $\quad \tilde{x} = \tilde{S}\big(\tilde{x}, \texttt{refresh} = 1\big)$
8: $\quad \tilde{x} = \tilde{L}'(\tilde{x})$
9: $\quad \tilde{x}[\cdot, 0, 0] \leftarrow \tilde{x}[\cdot, 0, 0] \oplus W(2\sigma)$
10: $\quad \tilde{x} = \tilde{S}\big(\tilde{x}, \texttt{refresh} = 0\big)$
11: $\quad \tilde{x} = \tilde{L}'(\tilde{x})$
12: $\quad \tilde{x}[\cdot, 0, 0] \leftarrow \tilde{x}[\cdot, 0, 0] \oplus W(2\sigma + 1)$
13: $\quad \tilde{x} = \tilde{x} \oplus \tilde{k}$
14: $\quad \tilde{x}[\cdot, \cdot, 0] = \tilde{x}[\cdot, \cdot, 0] \oplus T(\sigma + 1)$
15: **end for**
16: $x = \tilde{x}[\cdot, \cdot, 0] \oplus \cdots \oplus \tilde{x}[\cdot, \cdot, d-1]$

The masked state $\tilde{x} \in \mathbb{F}_2^{s\times l\times d}$ is initialized with the shared key $\tilde{k} \in \mathbb{F}_2^{s\times l\times d}$ in line 3, before the tweak $T \in \mathbb{F}_2^{s\times l}$ and the plaintext $p \in \mathbb{F}_2^{s\times l}$ are added to the first share slice $\tilde{x}[\cdot, \cdot, 0] \in \mathbb{F}_2^{s\times l}$ in lines 4 and 5 respectively. The masked $\tilde{S}$-box is implemented based on (1). For the multiplications (AND-gates), the ISW-multiplication algorithm [12] given in Algorithm 3 is used, while the additions (XOR-gates) are computed sharewise. During the first computation of the $\tilde{S}$-box in each of the six rounds, a mask refreshing of $y[1, \cdot]$ is performed according to Algorithm 4 as an additional countermeasure (line 7). For $d \in \{1, 2, 3\}$, the mask refreshing algorithm is used with $m = 1$, while for $d \in \{4, \ldots, 8\}$, it is run twice, first with $m = 1$ and then with $m = 3$. The refreshing method has been proposed and further analyzed in [2, 1].
The linear $\tilde{L}$-box in line 8 is computed based on (2) in parallel for the different shares and the constant $W(r) \in \mathbb{F}_2^s$ is added to the first share slice $x[\cdot, 0, 0] \in \mathbb{F}_2^s$ in line 12. Then, in line 13 the key is added to the current state and next, in line 14 the tweak to the first slice of the shares. Finally, the unmasked output is obtained by XORing the different shares.
The recorded traces capture the first half of the first round of Algorithm 2. An exemplary trace is given for $d = 4$ shares in Figure 1, where the ISW-multiplication of the $\tilde{S}$-box, the mask refreshing as well as the $\tilde{L}'$-box computation are clearly visible.[3].

[3]Image taken from https://git-crypto.elen.ucl.ac.be/spook/masked_spook_sw

Algorithm 3 ISW-Multiplication Algorithm.

Input: d-shares $(a_0, \ldots, a_{d-1})$ of $a \in \mathbb{F}_2^l$ and $(b_0, \ldots, b_{d-1})$ of $b \in \mathbb{F}_2^l$
Output: d-share $(c_0, \ldots, c_{d-1})$ with $\bigoplus_{i=0}^{d-1} c_i = \left(\bigoplus_{i=0}^{d-1} a_i\right) \odot \left(\bigoplus_{i=0}^{d-1} b_i\right)$
for $i = 0, \ldots, d-1$ **do**
  $c_i \leftarrow a_i \odot b_i$
end for
for $i = 0, \ldots, d-1$ **do**
  for $j = i+1, \ldots, d-1$ **do**
    $r_{ij} \xleftarrow{R} \mathbb{F}_2^l$
    $r_{ji} \leftarrow (a_i \odot b_j) \oplus r_{ij}$
    $r_{ji} \leftarrow r_{ji} \oplus (a_j \odot b_i)$
    $c_i \leftarrow c_i \oplus r_{ij}$
    $c_j \leftarrow c_j \oplus r_{ji}$
  end for
end for

Algorithm 4 Mask Refreshing.

Input: d-share $(a_0, \ldots, a_{d-1})$ of $a \in \mathbb{F}_2^l$ satisfying $a = a_0 \oplus \cdots \oplus a_{d-1}$, parameter m
Output: refreshed d-share $(a'_0, \ldots, a'_{d-1})$ of $a \in \mathbb{F}_2^l$ satisfying $a = a'_0 \oplus \cdots \oplus a'_{d-1}$
for $i = 0, \ldots, d-1$ **do**
  $r_i \xleftarrow{R} \mathbb{F}_2^l$
  $a_i \leftarrow a_i \oplus r_i$
  $a_{(i+m) \bmod d} \leftarrow a_{(i+m) \bmod d} \oplus r_i$
end for

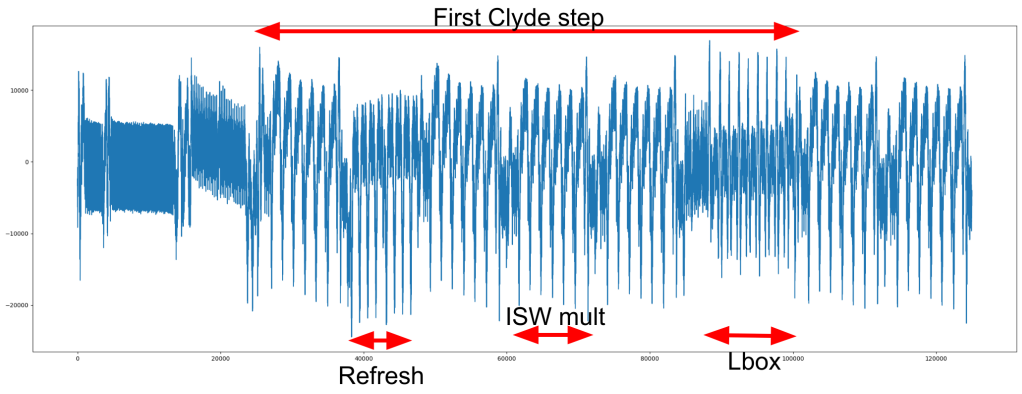


Figure 1: Exemplary trace for $d = 4$.

4 Deep Learning Approach

4.1 Overview

Our deep learning based solution to the CHES 2020 side-channel challenge relies on the following main ideas:

- We *basically* predict Hamming weights of some sensitive variables derived from the cipher state shares. To this aim, we use a deep residual neural network following the architecture first introduced in [8].
- In the Hamming weight model, we actually derive many different but related sensitive variables from each state word via a technique that we call the *scattershot encoding*. In a nutshell, the intuition behind the scattershot encoding is to provide the neural network with a large number of interrelated tasks of varying difficulty that are close to the assumed leakage model (here: Hamming weight leakage of the shared cipher state) and which together solve the desired prediction problem. With their help we can then obtain bit-level predictions of the targeted share values with high precision by post-processing the predicted Hamming weights by some linear algebra. Our findings indicate that these bit-level predictions are difficult or impossible to learn with the required precision for our network without the scattershot encoding.
- Finally, we combine bit-level predictions over many traces by predicting the unshared values of the sensitive variables. Comparing predictions from traces and key hypotheses, we get a ranking of the unshared keys.

In combination, these ideas allowed us to break all software challenges of the contest with a data complexity that improved the state-of-the-art. For instance, the 3-share challenge can be broken with ≈ 25 traces.
In the sequel, we first describe our leakage target and the high-level strategy of our key recovery method. Our strategy presupposes that the shared cipher state at a particular point of execution leaks to the power trace essentially in its entirety, with only a modest amount of noise. This capability is achieved by combining a deep neural network with a simple post-processing step. Then we describe the structure of our deep neural network, the scattershot encoding and how the latter is used to obtain bit-level prediction on the shared cipher state.
Finally, we provide some experimental evidence that our methods solve the challenge and that in particular the scattershot encoding is helpful for solving the challenge problem.

4.2 Overall Strategy

Targeted State We target the cipher state after the first S-box. The unshared cipher state $x \in \mathbb{F}_2^{4\times 32}$ is given by $x = S(p \oplus T \oplus K)$, where $p \in \mathbb{F}_2^{4\times 32}$ denotes the plaintext, T denotes the tweak value, and S denotes application of the S-box layer. $S(p \oplus T \oplus K)$ depends in a simple way on plaintext, tweak and key and should therefore be well suited to constrain K given partial knowledge of $S(\cdot)$ for several p_i, T_i. The fact that the S-box is involved means that errors in the key get spread out to 4 bits of the observed state, which should amplify the ability of the adversary to distinguish between almost-correct and fully correct key hypotheses. The dependence of the observed value on p, T, K holds for each S-box separately, so that given successful recovery of the cipher state, a key ranking can be derived one S-box at a time, implying the ability to recover the whole key efficiently.

Dealing With the Masking Countermeasure Because of the presence of the masking countermeasure the cipher state is only available during execution in a secret-shared form. We will therefore try to read out the secret-shared cipher state $\tilde{x} \in \mathbb{F}_2^{4\times 32\times d}$ instead of

the unshared state. If we had the full shared state, the unshared state x could be simply obtained by setting $x[a,b] = \oplus_{b=0}^{d-1} \tilde{x}[a,b,c]$.
Let X be a random variable that is realized by a power trace reading and let Y a random variable whose distribution coincides with that of the target states. Then, the image of X is a subset of $\mathbb{R}^n$, where n is the number of samples in each power trace, Y is uniformly distributed on $\mathbb{F}_2^{4\times d\times 32}$, and X and Y are dependent variables.
Assume that we possess an oracle O which given a trace t outputs values $p \in \mathbb{R}^{4\times d\times 32}$, where $p[a,b,c] \approx \mathbb{P}(Y[a,b,c] = 1 | X = t)$, i.e. $p[a,b,c]$ approximates the probability that $\tilde{x}[a,b,c]$ is set given the information contained in the trace t. Then

$$\mathbb{P}(x[a,b] = 1 | X = t) \approx \sum_{v \in M} \left(\prod_{0\le i\le d-1, v_i = 1} p[a,b,i] \prod_{0\le i\le d-1, v_i = 0} (1 - p[a,b,i]) \right),$$

where $M = \{v \in \mathbb{F}_2^d : v^t v = 1\}$.

Remark 1. Of course, $\{X = t\}$ is a zero set. In the above formula we tacitly assume that the conditional densities converge if we replace t on the right-hand side by ϵ-balls around t.

Combining Leakage Across Traces Let $\{t_0, t_1, t_2, \ldots, t_k\}$ be a set of traces with associated plaintexts p_i and tweaks T_i. Given our oracle O, we can calculate for each i and each $(a,c) \in \{0,1,2,3\} \times \{0,1,\ldots,30,31\}$ a probability $p_i[a,b] \coloneqq O(t_i)$ for the corresponding bit value of the unshared target state to be set. We partition the key K into 32 nibbles aligned with the S-boxes. Now, if $K[b] \in \mathbb{F}_2^4$ is a hypothesis for the b-th partial key, we calculate the true values $x[\cdot, b](K[b])$ given p_i, T_i and $K[b]$. We consider the score $s(K[b])$

$$s(K[b]) \coloneqq \left\| x[\cdot,b](K[b]) - p_i[\cdot,b] \right\|_2$$

and rank the key hypotheses in lowest-is-best order. Since $\|u_1 + u_2\|_2 = \|u_1\|_2 + \|u_2\|_2$ for orthogonal vectors u_1, u_2, adding up these partial key rankings for all indices b is the same as using an analogous ranking method on the full key. We thus get a ranking of the full key space, where it is efficiently possible to determine the rank of any given key hypothesis up to a small error.

4.3 Deep Neural Network

In the sequel, we implement an oracle of the desired kind with the help of a deep neural network and a slight post-processing of its outputs. To this aim, we explain in this section how our network is designed and trained as well as the required post-processing step.

Preliminary Investigations Based on the public description and source code available for the implementation under attack, we regarded it likely that the word-wise Hamming weights of the key shares would leak. We were rapidly able to confirm this hyptothesis using a straightforward application of a deep residual neural network following the architecture used in [8]. However, although we could see the Hamming weights of the words fairly well, Hamming weight leakage was not enough to break the challenge and various attempts to directly find sufficient single-bit leakage were not successful, even in case of the 3-share challenge.

Basic Ideas The deep residual neural network follows the architecture introduced in [8]. In particular, this entails the following:

- We treat Hamming weight prediction as a *regression problem and not as one of categorical prediction.* Simply put, our predictions are real-valued approximations of

the integer values that represent the ground truth Hamming weights to be extracted. This helps to overcome the problem of poorly represented (i.e. rare) output classes by exploiting the natural order on the output domain.

- We treat the prediction problem as approximately *subsampling invariant*. This means that reducing the number of points in a trace might harm prediction accuracy, but should not introduce *systematic* errors in the predictions.

- A prediction of the sensitive variable produced by a relatively simple, shallow model may be improved upon by further post-processing.

On top of these ideas, the most important ingredient to the success of our deep learning model is that we create a very large number of mutually correlated prediction targets for the network. Together, these prediction targets solve the relatively difficult problem of predicting the value of the shared cipher state at a certain point of its operation with bit-level precision. For our model, we create a total of $400d$ prediction targets, where d is the number of secret shares in the implementation under attack. These prediction targets can be broken down into $4d$ groups of targets, where each group contains hundred targets that work together to solve one word of the shared state. Basically, the scattershot encoding transforms a bit-string into a sequence of Hamming weights of a large number of random sub-strings.
Since the resolution of the trace data and the number of state shares differs between the different software challenges, network parameters differ accordingly for the different challenges. Our neural networks are therefore parameterized by the following variables:

- *Input size* I: number of data points (equivalently, the size of the input layer) in each trace to be processed by the network.

- *Subsampling factor* q_1 and *subsample size* q_2: number of subsampled versions of the input trace that are analyzed in parallel by the network as in [8] and the size of each subsampled trace. By definition, we have $I = q_1 \cdot q_2$.

- *Network depth*: number of residual blocks used by the network.

- *Word size* n: number of bits in each sensitive variable that we will try to extract from the traces. Generally, there will be $4d$ variables to extract for a challenge with d secret shares.

- *Scattershot size* N and a set of *scattershot masks* $m_1, m_2, \ldots, m_N \in \mathbb{F}_2^n$: number of auxiliary sensitive variables that we generate in the scattershot encoding for each secret share and a randomly chosen generating set of $\mathbb{F}_2^n$.

For all four challenges we used a network depth of 10 residual blocks, a word size of 32, a scattershot size of 100 and a set of scattershot masks chosen at random once for all experiments. In contrast, q_1, q_2 and some training parameters needed to be chosen differently for different challenges; Table 1 gives details on all the challenges. Further details can be found in the supplementary data to this paper.

Preprocessing Given an input trace T of size I, we first decompose the trace into subsampled traces $T_1, T_2, \ldots, T_{q_1}$, where T_i consists of the points of T with index i mod q_1. Each subsampled trace T_i therefore contains q_2 points. The T_i are subsequently called *slices* of the original trace.

Number of Outputs, Output Activation In total, our network generates $4Nd$ outputs for a d-share target. The ground truth associated to each output is the Hamming weight of some bit vector and therefore a priori a non-negative integer, making rectified linear units a natural choice for final layer activations, since they calculate a linear function of their input that is truncated to non-negative output values.

The First Layer After the preprocessing, each of the q_1 subsampled partial traces is processed first by a batch normalization layer. Each batch-normalized partial trace is then processed separately by a shared fully connected layer with $4Nd$ rectified linear outputs. The entire neural network state at this stage is thus comprised of q_1 vectors v_i of $4Nd$ real values each, where v_i is the vector associated to subsampled slice i. Each slice v_i of the state is further subdivided into $4d$ *columns* of N values. Each column of N values is subsequently processed separately from the others (with communication only to the corresponding column in neighbouring slices of the overall network state) and intended to produce a prediction for the Hamming weights of the N prediction targets corresponding to one 32-bit word of the shared state.

Residual Blocks Subsequently, each column is updated by repeating the following steps in a number of *residual blocks*, where each residual block computes the input of the next residual block:

1.) Batch normalize the current output state of each column.

2.) Apply a one-dimensional convolution of kernel size 3 with N output channels to the batch normalized state of each column, running the convolution across all the slices after padding with zeros on both ends ("same" padding in keras). Call the output vector of the convolution $(w_0, w_1, \ldots, w_{q_1-1})$.

3.) Add v_i and w_i together to obtain the input for the next residual block.

This structure is repeated for t blocks, where $t = 10$ in all our experiments.

Predicting the Target by Averaging Finally, we are left with q_1 vectors $v_i \in \mathbb{R}^{4Nd}$ that we hope will approximate our prediction targets. These are subjected to a final averaging step that simply calculates the mean of the q_1 vectors and outputs it as our prediction.

Intuitions Behind this Architecture This neural network architecture was first introduced for an AES side-channel task in [8], where it achieved highly precise byte-level Hamming weight prediction for all bytes of the expanded keys of a protected AES implementation. Here, splitting up the processing of the full trace into a number of subtraces helped to achieve dimensionality reduction, to reduce overfitting, and to leverage subsampling invariance of the side-channel prediction problem. The use of convolutional windows of width 3 allows some communication between the subnetworks processing different subsampled slices of the trace under consideration, which should improve slice-wise prediction quality. The choice of a deep residual neural network was motivated mainly by its good scaling with network depth.

4.4 The Scattershot Encoding: Achieving Bit-Level Precision

Motivating Problem In side-channel analysis, the adversary obtains usually only partial information about the target key by using the side-channel, but desires bit-level information, typically of some cryptographic key. They therefore will likely need to combine information from a large number of traces and/or use additional cryptanalytic techniques to determine the secret. For instance, in [8] first a neural network is used to approximately determine

the Hamming weights of all bytes of the expanded AES key of the implementation under attack, and then a SAT solver subsequently computes the exact key from this partial information. Effectively, this strategy combines leakage from various parts of a single power trace with prior knowledge about the cipher under attack to exactly recover the key. For the masked implementation of Clyde-128 under study in the CHES Challenge 2020, we were easily able to show significant leakage of the Hamming weights of the 32-bit words of secret-shared subkeys and cipher state at various points during the cipher's execution. However, each execution of the cipher uses fresh randomness for the computation of the initial secret sharing and the shares are refreshed with new randomness during the execution as well. This severely limits the ability of an adversary to combine leakage either across traces or across different leakage points within a single trace. Essentially, to find the secret key a significant amount of information about the *unshared* key must be exploitable from a single trace. This information can then be combined across traces to extract enough information on the unshared secret in order to enable an exhaustive attack on the remaining possibilities.
We tried to directly learn to predict single bits of the secret shares of the keys and cipher states. Our attempts, however, did not yield useful results even for the 3-share challenge, which is why we came up with the *scattershot encoding* that we detail next.

Scattershot Encoding Assume that we are given the task of guessing a bit string $b = b_1, b_2, \ldots, b_k$ and the only available information are the Hamming weights h_i of $b \cdot m_i$, where $m_0, m_1, \ldots, m_l$ is a collection of known random bit strings of length k and where $\cdot$ denotes component-wise multiplication. Once the m_i form a generating subset of $\mathbb{R}^k$, this task is easy: assume without loss of generality that the m_i form a basis, then

$$\sum_{j=1}^{k} m_i(j)\lambda_j = h_i$$

is a linear equation system with a coefficient matrix M of full rank and $\lambda_i = b_i$ is its only solution. We call the weights h_i the *scattershot encoding* of b with respect to the masks m_i.

Recovering Target Variables from Noisy Scattershot Encodings In the presence of noise, recovering a bit string from its scattershot encoding with respect to a known set of masks becomes a problem of linear approximation in a natural way. Indeed, we can model the problem in an idealized setting as follows: assume that in the above situation we are given $\tilde{h}_i \coloneqq h_i + X_i$ instead of the true encoding by the h_i values, where the $X_i \sim N(0, \sigma^2)$ are i.i.d. normally-distributed random variables centered at zero. For a given combination of $\lambda_i \in \{0, 1\}$ to explain the observed $\tilde{h}_i$, we get $X = \tilde{h} - M\lambda$, where $X = (X_i)_{i=1,\ldots,k}$, $\tilde{h} = (\tilde{h}_i)_{i=1,\ldots,k}$. The probability density $p(\lambda)$ of X at this value is (up to multiplicative, constant factor) given by

$$p(\lambda) \propto \mathrm{e}^{-\frac{\|\tilde{h} - M\lambda\|_2^2}{2\sigma^2}}.$$

If there is no prior knowledge about the true values of the b_i, sorting the candidates by descending value of $p(\lambda)$ yields an optimal guessing strategy in this situation. Since the exponential is strictly monotonously increasing irrespective of base (as long as the base is > 1), this ranking is the same as that obtained by ordering the values by $\|\tilde{h} - M\lambda\|_2$ in ascending order.
While it is possible to efficiently compute the ranking so induced (see e.g. [7]), it is in our context more useful to simply relax to the real numbers and find a minimal real solution for the λ_i by linear regression. This is due to the fact that solutions to the relaxed problem contain some information on the amount of uncertainty about the value of specific bits b_i of the sequence to be guessed, which is crucial for the implementation of our secret state

Table 1: Challenge-specific parameters for our deep neural network.

Challenge	Number of Epochs N_e	Batch Size	Steps per Epoch	Validation Steps	q_1	q_2
Sw3	800	128	500	50	100	625
Sw4	800	128	500	50	167	499
Sw6	1000	32	2000	200	250	625
Sw8	1000	32	2000	200	250	875

oracle from the previous section. In addition, with a scattershot size of N, solving the linear regression can simply be done by multiplying the output array of the neural network for each share word from the right with a fixed $N \times 32$-matrix that only depends on the choice of scattershot masks.

Intuitions behind the Scattershot Encoding We assumed that some kind of Hamming weight model would describe the leakage well, which was quickly confirmed by experiments. However, the question is which parts of the cipher state to calculate Hamming weights over: we were able to see word-level Hamming weights very well, but bit-level state recovery seemed significantly harder. In particular, achieving uniform convergence for all target bits of the shared state proved difficult in our experiments to recover the state bits directly. The scattershot encoding gives the neural network a number of different Hamming weight models to work with. In addition, all of these problems are closely correlated to each other, so progress on any of the scattershot targets should help the next layer in the deep residual network achieve better precision on all of the other targets. We hoped that with these assumptions, the scattershot encoding would help the network achieve faster and more uniform convergence across all target bits, which turned out to be true.

4.5 Training the Network

Preprocessing the Data To train our deep neural network, we first merged all of the released 10,000-trace random-key files from the competition website. This gave us a dataset of 200,000 traces with associated keys and tweaks, the plaintext being uniformly the zero message. We then computed a random shuffling of the training data and divided the shuffled data into a training and a validation set, holding out 20,000 traces for validation. We determined the cipher state based on the tweak, the secret-shared key and the plaintext data given in the original dataset using the Python implementation of the masked Clyde implementation provided by the organizers and subsequently brought it into scattershot encoded form by applying the relevant linear transformation. A set of scattershot masks was generated at random and can be found in the supplementary data set to this paper.

Training We generated minibatches for training and validation by random selection of samples from the training respectively validation set at training time using a generator, where we used different settings for the minibatch size for the different challenges. Likewise, the length of training (number of epochs and training steps per epoch, i.e. the number of minibatches used in each epoch) varied between the challenges.
For all training runs we used the Adam optimizer with a cyclic variable learning rate and mean square error loss against the ground truth targets as loss function. At the end of each epoch, we calculated the mean square error loss for the current network also on the validation set and stored the best network weights (according to validation loss) to disk. We set the learning rate lr_i for epoch i so that

$$\log_2(lr_i) = \log_2(\mathsf{low}) + \big(\log_2(\mathsf{high}) - \log_2(\mathsf{low})\big) \cdot \big((N_e - 1) - i \bmod N_e\big)/(N_e - 1),$$

where N_e is the number of epochs in each cycle and low and high denote upper and lower bounds for the learning rates used in each cycle. In other words, each cycle used an exponential learning rate decay followed by resetting the learning rate to the *high* value at the beginning of the next learning rate cycle. We hoped that this setup would allow sufficient time for fine-tuning of weights at the lower learning rate levels as well as escape from local minima at the higher learning rates. We used low $= 0.001$, high $= 0.00002$ and $N_e = 10$ throughout our experiments. All relevant parameters are summarized in Table 1.

Computational Cost of Training With network and training parameters as well as the sizes of input and output vectors of our networks being quite different between the different challenges, it is no surprise that the computational cost of training runs differs accordingly. For Sw3, a single training epoch using the parameters given in Table 1 takes about 100 seconds on a computer equipped with a single GTX 1080 Ti graphics card and a few gigabytes of not otherwise occupied RAM. A full training run can therefore be completed in about a day. For Sw8, on the other hand, a single epoch takes about 1050 seconds on the same machine, resulting in roughly a million seconds for a complete training run, or a bit less than two weeks.

4.6 Results

Evaluation Data Along with the random-key training data for each challenge, the organizers of the competition released fixed-key datasets to allow participants to evaluate the performance of their solutions. The fixed-key datasets were never used in either training or model selection, so results on these datasets should be representative of the performance reached for the key extraction task at hand.
In the sequel, we will have a closer look the behavior of our solution on the simplest software challenge (Sw3, $d = 3$ shares). For the other challenges, we only report some statistics on whole-key ranking achieved with the appropriate number of traces.

Single-Trace State Extraction Our attack essentially first tries to match the observed traces to a probability distribution on the internal secret-shared cipher state. Then it computes the induced probability distribution on the corresponding unshared cipher state by calculating the convolution product of the d share-distributions, compare equation (4.2), and finally matches the inferred distribution to all possible key hypotheses. For this to work well, we need fairly strong biases on the shared cipher state from a single trace, since the convolution product leads to a decay of any biases that is exponential in the number of shares.
Fortunately, we do get strong biases on the secret-shared state. Indeed, as Figure 2 shows, even some nibbles of the unshared key can be read out with high reliability already from a single trace in the Sw3 case. Interestingly, however, the results strongly depend on the considered S-box: some of them (e.g. S-box 17, but also S-boxes 23 and 26) seem to be rather easy to predict, while most S-boxes are harder (e.g. S-boxes 0 or), if not impossible to predict (e.g. S-boxes 16 or 21, whose key rank distribution is close to uniform).
This is even more apparent when we look at extracting the target secret-shared state after the first S-box layer. Table 2 shows some fairly representative examples of bitwise differences between predicted state and ground truth for a version of our Sw6 model trained on a slightly smaller data set than the model used in our attack on Sw6.

Global Key Rank Our attack needs 20-30 traces to break the 3-share challenge; a few hundred to break the 4-share challenge; a few thousand for the six-share challenge, and tens of thousands for the 8-share challenge. For details, we refer to the independent evaluation given by the organizers of the CHES 2020 competition in [5]. Compared to running our

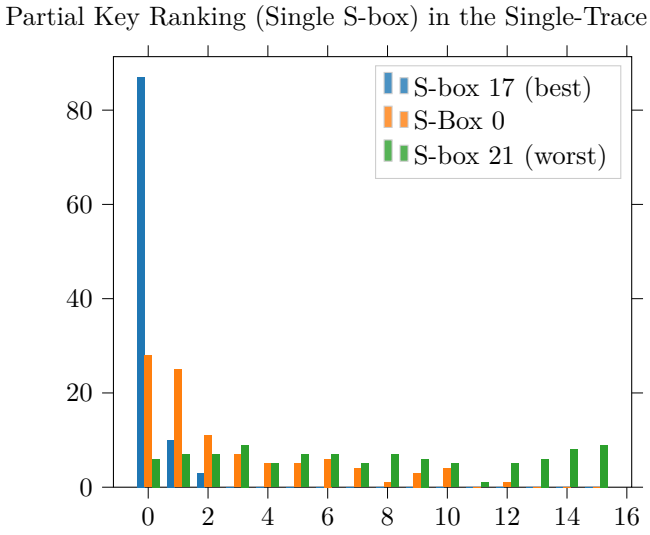


Figure 2: Key rank distribution in a single-trace attack for three selected S-boxes. For this test, 100 traces were selected at random from the FKEY_SW3_K1_1000_0 data set and the key extraction algorithm was run separately on each trace. The statistic here shown gives the number of cases where the true partial key for the nibble in question was found at the rank given on the x-axis.

Table 2: Bitwise differences between the predicted and correct first word of secret-shared cipher state on Sw6. The first five traces of RKEY_SW6_10000_19 served as test dataset. The model used in this experiment has been trained only on RKEY_SW6_10000_0-14, so these examples were not seen by this model during training.

Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
0x81050d0	0x802e1810	0x5490	0x2015010	0x600142

own evaluation using the test data available in the competition, this has the advantage that their evaluation included their own solution, against which our solution can be compared. In general, the number of traces needed to achieve a drop of the key rank under the 2^{32} bound required follows a distribution with a fairly heavy tail, making it difficult to obtain reliable measurements of the expected data cost for the 8-share challenge.

4.7 Dissecting the Scattershot Encoding

Problem Statement Our networks failed to learn the extraction of all output bits when we tried to learn to predict these bits directly. Instead, we predict a scattershot encoding of our target data. In the attack phase, we have to reverse the encoding in order to retrieve predictions for the bit-values of the shared target variables from the output of the network. This is surprising, since the scattershot encoding is nothing but a linear transformation of the output values. We therefore did some further investigations understand the success of this trick a bit better.

Difficulties in Studying the Scattershot Encoding The task under discussion in this paper makes it difficult to study problems in network convergence in a systematic way: even for the 3-share problem, the time to train an instance of our network is about a day on our hardware. Furthermore, it is difficult to say, in absolute terms, how well the obtained network exploits the available signal and to find out exactly which properties of the task cause the convergence problems.

A Synthetic Problem We therefore aimed at figuring out a simpler learning problem that would exhibit similar convergence problems as our side-channel task. To this end, we tried to learn the $\mathbb{F}_2$-linear function

$$f\colon \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}, \quad f(x) = Ax,$$

where A is the lower triangular matrix with all entries below or on the main diagonal equal to one. We tried to learn f by observing random input-output pairs using a fully connected deep neural network.

Remark 2. The i-th coordinate function of f is $f_i(x) = \bigoplus_{j=1}^{i} x_j$. The problem of learning the last of these coordinate functions in isolation (i.e. given only random inputs and $\{0,1\}$-valued outputs of the coordinate function) using feed-forward neural networks is expected to be difficult, which is confirmed by experiments: training feed-forward networks to predict the parity of 32-bit bitstrings fails unless the networks are trained with structured input. However, learning f is expected to be easier, since the network is now forced to learn a number of XOR-sums. Some of these are of low Hamming weight and therefore easy to learn, and knowing how to compute these might be helpful for learning the components that are harder to predict.

Neural Network Designs In order to test the difficulty of learning to compute f with and without the scattershot encoding, we considered two simple network architectures. Both of them are deep residual networks with ten residual blocks, where each block consists of a batch normalization layer followed by a fully connected layer with rectifier nonlinearities. They take a vector of 32 bits as input, have 100 nodes in each of the internal layers and use a single dense layer with ReLu activations after the input to expand the input to a vector of dimension 100. However, they differ in the output layers:

- The final layer of Model 1 uses a sigmoid activation funtion to obtain output values between 0 and 1 and outputs 32 such values, thereby directly predicting the target bits.

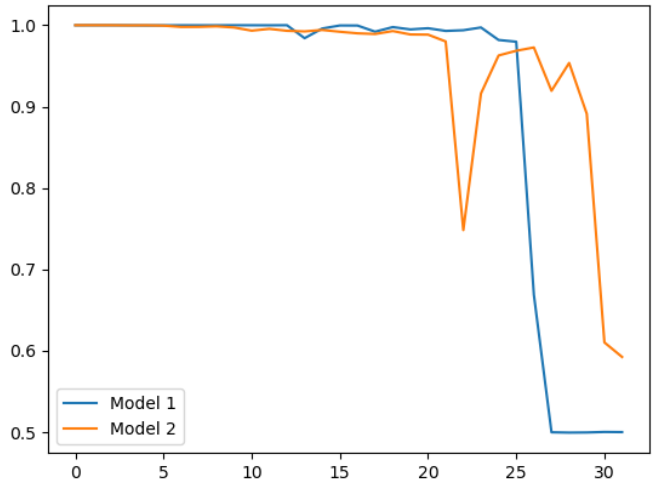


Figure 3: Accuracy of target prediction for Model 1 and Model 2 on learning a simple $\mathbb{F}_2$-linear function. The x-axis gives the targeted bit position inside the target and the y-axis shows the obtained accuracy. Model 2 uses a scattershot encoding of the targets while Model 1 does not.

- Model 2 predicts a scattershot encoding of the target values and therefore outputs 100 values using a final linear activation.

Training and Validation Both models were trained for 100 epochs on the same training data set. Training was performed with the Adam optimizer using default parameters in Keras and a constant batch size of 5,000. Ten percent of the training set was withheld for validation in order to track the evolution of the loss value for both models.

Testing Finally, both models were tested using a freshly generated test set consisting of another 10^6 example vectors. For each bit of the target output, we measured how often either prediction output matched ground truth.

Results Within the training budget of the experiment, Model 1 failed to converge on the last five bits of the target function; it predicted the other bits very well. Model 2, on the other hand, showed some degree of convergence for all bits, although the last two bits had accuracies of only 59 and 61 percent respectively when training was stopped. Figure 3 gives additional details.
Additionally, a qualitative comparison of the loss evolution for both models shows that Model 1 converges through punctuated equilibria, effectively learning to predict one bit of the target at a time and then getting stuck. The loss of Model 2, on the other hand, is smoothly decreasing throughout the training run. It is also worth noting that many of the equilibria require a temporary significant increase of validation loss to leave, suggesting some stability of the corresponding local minima of the loss function. This is illustrated in Figure 4 which validation loss progress for Model 1 (left) and Model 2 (right) over 100 epochs of training. Note that the scale of the loss is not comparable between both approaches, since the range of the scattershot targets (which are Hamming weights of parts of the target vector) is much larger than the binary values of the target values themselves.

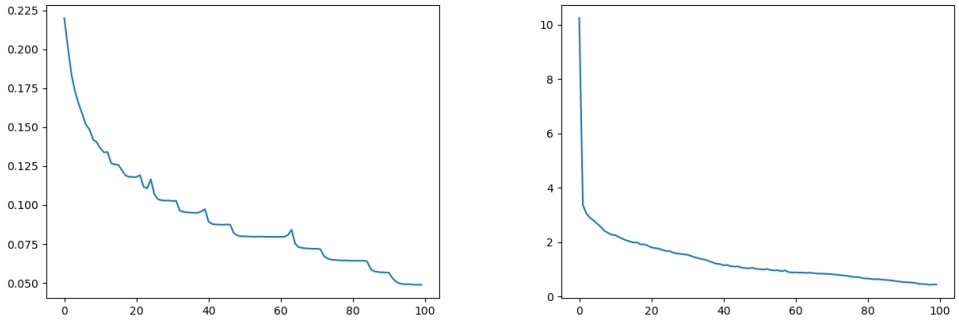


Figure 4: Validation loss evolution for Model 1 (left) and Model 2 (right) over 100 epochs of training.

Remark 3. It is worth mentioning that our experiment described here using synthetic data contains no noise. In principle, there is no uncertainty about the signal and there is a simple algorithm that recovers all the target bits. Still, at least the results of the simple ML-based analysis we conducted suggest that some bits of the target are much less readily exploitable than others; the direct attack has difficulties to see any bias at all for some of them. It is tempting to speculate that the very strong differences we see in the apparent exploitability of different S-boxes in the CHES 2020 challenge task are likewise due partly or wholly to logical difficulties of extracting the signal instead of being due to the absence of an exploitable signal for the problematic S-boxes. This suspicion is (weakly) supported by the observation that the implementation provides no clear reason for the existence of these differences in leakage and the fact that the use of the scattershot encoding helps to solve some S-boxes that otherwise show no significant bias. The learning history of our direct attack attempt also showed some support for a pattern of temporary stalling over the first 400 epochs of training that is similar to that observed in the synthetic task, in spite of our use of a cylic learning rate schedule to prevent the model from getting stuck in local minima (see Figure 5).

5 The Stochastic Approach

In this section we present further analyses and insights that we gained with the help of the stochastic approach [16]. To this aim, we first extend its concept in Subsection 5.1 to a tweakable block cipher with shared keys whose implementation is protected by Boolean masking, before we show the results of some numerical experiments (Subsection 5.3). Finally, Subsection 5.4 contains a brief comparison between the attack efficiencies of the stochastic approach and of the neural network. As in the "classical" stochastic approach, we assume that the random variable

$$I_t(x, z, \tilde{k}) \coloneqq h_t(x, z, \tilde{k}) + R_t \tag{3}$$

quantifies the random electrical current (or the power consumption) at time t, where its distribution depends on the triple $(x, z, \tilde{k})$. Here, x and z denote a part of the plaintext (possibly including a tweak) and the masking bits, while $\tilde{k}$ denotes the part of the *shared key* which is targeted by the attack. The leakage function $h_t(x, z, \tilde{k})$ quantifies the deterministic part of the leakage while the random variable R_t models the noise. The noise is caused by other operations which are computed in parallel to the targeted one (independent of $(x, z, \tilde{k})$) and maybe to some degree on the effect of the power measurement. We assume

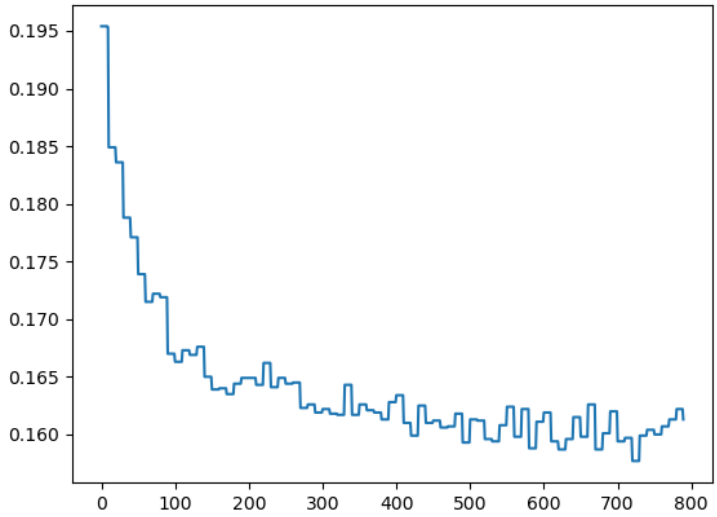


Figure 5: Development of the loss rate of our model in an attempt to directly learn predicting the target bits of the 3-share challenge. The loss given for each epoch is the lowest validation loss attained in the next ten epochs of training; this removes the effect of the cyclic learning rate schedule on loss development, which would otherwise clearly dominate. Some signs of capture by local minima are evident for epochs 60-87, 90-139 and 150-269.

that the noise vector R_t is independent of $h_z(x, z, \tilde{k})$ and is normally distributed with expectation 0. The profiling essentially works as usual (with shared subkeys in place of subkeys), while the key extraction is more challenging because of the shared keys.

5.1 Different Steps of the Stochastic Approach

In the following we adjust the stochastic approach to shared keys, and we explain and justify the decision rule.

5.2 Vector space bases for different substeps

At first the attacker/evaluator needs to select appropriate vector space bases together with corresponding points of interest. To this aim it is necessary to identify and model the operations which are relevant for the current at a given timepoint. This requires a good understanding of the implementation and usually some trial and error, that means trying different points of interest and different vector space bases and observing which one fits best at the given time instance. In this paragraph we describe different operations of Algorithm 2 that can be used in the stochastic approach and provide the corresponding bases. To this aim, we fix an S-box s and consider the different S-boxes separately. We always include $g_{0,t;\widetilde{k}} = 1$ in our bases to model the (sub-)key-independent signal. In order to guarantee that the coefficients of the different basis vectors are of the same magnitude, we subtracted a global mean trace from all traces before computing the coefficients. (This significantly reduces the absolute values of the key-independent coefficients $\beta_{0,t;\widetilde{k}}$, which would dominate otherwise, possibly spoiling the parameter estimation.) The remaining

estimation of the basis coefficients can be performed as usual by solving a least-squares problem (with shared key bits in place of key bits).

(1) Key Loading (Line 3 of Algorithm 2), $4 \cdot d$ POIs
Assuming a Hamming weight leakage (see also Subsection 4.3), we can take the single bits of $\tilde{x}[\cdot, s, \cdot]$ as basis vectors, which are in total $4 \cdot d$ ones. This can be achieved if we set

$$\bar{h}_t^*(\tilde{x}, z, \tilde{k}) := \beta_{0,t;\tilde{k}} + \sum_{i=0}^{3} \sum_{j=1}^{d} \beta_{i,j,t;\tilde{k}} \bar{g}_{i,j,t;\tilde{k}}(\tilde{x}, z, \tilde{k}),$$
$$\bar{g}_{i,j,t;\tilde{k}}(\tilde{x}, z, \tilde{k}) = \tilde{k}[i, s, j-1].$$

(2) Tweak Addition (Line 4 of Algorithm 2), 4 POIs

$$\bar{h}_t^*(\tilde{x}, z, \tilde{k}) = \beta_{0,t;\tilde{k}} + \sum_{i=0}^{3} \beta_{i,t,\tilde{k}} \bar{g}_{i,t;\tilde{k}}(\tilde{x}, z, \tilde{k}),$$
$$\bar{g}_{i,t;\tilde{k}}(\tilde{x}, z, \tilde{k}) = \tilde{k}[i, s, 0] \oplus T[i, s].$$

(3) Computation of the S-Box (Line 7 of Algorithm 2), d POIs
During the computation of the S-box there are according to (1) ISW-multiplications and XOR-operations that can be exploited. The evaluation of the S-box involves several masking bits in the thereby used ISW-multiplication, we first provide some further details on its evaluation, before we propose basis functions.

The S-box is applied to all shares of the j^{th} column $\tilde{x}[\cdot, j, \cdot]$ of the working state, $j \in \{0, 1, \ldots, 31\}$. In the notation of (3)) we have

- known plaintext $x = T(0)[\cdot, j] \oplus p[\cdot, j] \in \{0,1\}^4$, inclusive the tweak value; in the challenges $p[i, j] = 0$ for all $(i, j) \in \{0, 1, 2, 3\} \times \{0, \ldots, 31\}$. Formally, we may extend x to the shares $1, \ldots, d-1$ by setting $x \equiv 0$ there, denoting this consistently by $\tilde{x}$.
- shared key $\tilde{K}[\cdot, j, \cdot] \in \{0,1\}^{4d}$ that is relevant for the j^{th} S-box.
- masking bits: The evaluation of an S-box (1)) requires four times Algorithm 3 (ISW-multiplication), four times an XOR-operation and once Algorithm 4 (mask refreshing). All together, these subalgorithms require $4\binom{d}{2} + 0 + d = 2d(d-1) + d = 2d(d - \frac{1}{2})$ masking bits.

In particular, $(x, z, \tilde{k}) \in \{0,1\}^4 \times \{0,1\}^{2d(d-\frac{1}{2})} \times \{0,1\}^{4d}$, and all the intermediate values that occur within the evaluation of the S-box can be expressed in terms of x, z and $\tilde{k}$.

The amount of masking bits might become critical as it grows quadratically in the number of shares. Altogether, there are $2d(d - \frac{1}{2}) + 4d = 2d(d + \frac{3}{2})$ unknown bits (masking bits and shared key bits). This does not cause a problem in the profiling phase, but the key extraction (see Section 4) becomes numerically infeasible as the number of shares d increases. For $d = 3$ the number of unknown bits equals 27, which should be feasible, but already for $d = 4$ it is 44.

Hence, at least for $d > 3$ strategies are needed that reduce the amount of unknown bits. All the shared key bits occur (implicitly) in the first two ISW-multiplications and in the subsequent XOR-additions (computing $y[1]$ and $y[0]$ for all shares). Mask refreshing is not relevant for these operations because it only updates $y[1]$ after it has been calculated. Restricting our attack on the S-box to these four substeps "wastes"

the side-channel information which stems from the computation of $y[3]$ and $y[2]$. On the positive side this limits the number of unknown bits to $2\binom{d}{2} = d(d-1)$.

After these considerations, we exemplary show the proposed basis functions for the masked stated $\tilde{y}[1,:] = (\underbrace{\tilde{x}[0,:] \odot \tilde{x}[1,:]}_{=\tilde{q}}) \oplus \tilde{x}[2,:]$:

(a) (Single) ISW-Multiplication (Line 4 of Algorithm 3), d POIs, masking bits $c_i = a_i \odot b_i$

$$\bar{h}_t^*(\tilde{x}, z, \tilde{k}) = \beta_{0,t;\tilde{k}} + \sum_{j=1}^{d} \beta_{j,t,\tilde{k}} \bar{g}_{j,t;\tilde{k}}(\tilde{x}, z, \tilde{k}),$$
$$\bar{g}_{j,t;\tilde{k}}(\tilde{x}, z, \tilde{k}) = \tilde{x}[0, s, j-1]\tilde{x}[1, s, j-1].$$

(b) (Single) ISW-Multiplication (Lines 9 and 10 of Algorithm 3), $d(d-1)$ POIs $a_i \odot b_j$ and $a_j \odot b_i$, similarly as above

(c) (Single) XOR (according to (1)), d POIs

$$\bar{h}_t^*(\tilde{x}, z, \tilde{k}) = \beta_{0,t;\tilde{k}} + \sum_{j=1}^{d} \beta_{j,t;\tilde{k}} \bar{g}_{j,t;\tilde{k}}(\tilde{x}, z, \tilde{k}),$$
$$\bar{g}_{j,t;\tilde{k}}(\tilde{x}, z, \tilde{k}) = \tilde{q}[j-1] \oplus \tilde{x}[2, s, j-1].$$

Remark 4. (i) The real amount of masking bits depends on how far one evaluates computation of the S-box, e.g. whether one only considers $y[1]$, or $y[1]$ and $y[0]$, or the complete S-box. As mentioned earlier, using only $y[1]$ and $y[0]$ the masking bits needed for the mask refreshing of $y[1]$ can be neglected. On the negative side, some information gets lost.

(ii) Furthermore, the multiplications in the ISW-Algorithm 3 (line 4: $c_i = a_i \odot b_i$, line 9: $a_i \odot b_j$, line 10: $a_j \odot b_i$) can be observed *without* masking bits. As a consequence, the operations in (1), (2), (3)(a) and (3)(b) for $y[1]$ and $y[0]$ can be exploited completely without masking bits. Only the subsequent XOR in step (3)(c) that requires the execution of the full ISW-multiplication needs the masking bits r_{ij} of the ISW-algorithm.

5.2.1 Estimation of the Covariance Matrix

Let $t_1 < \ldots < t_m$ be the chosen points of interest. As mentioned earlier, we assume $R_t = I_t(x, z, \tilde{k}) - h_t(x, z, \tilde{k})$ to be centered normally distributed, so that the m-dimensional random vector $(I_{t_1}(x, z, \tilde{k}) - h_{t_1}(x, z, \tilde{k}), \ldots, I_{t_m}(x, z, \tilde{k}) - h_{t_m}(x, z, \tilde{k}))$ is centered normally distributed as well for some covariance matrix C. Then

$$\vec{I}_{\vec{t}}(x, z, \tilde{k}) \coloneqq (I_{t_1}(x, z, \tilde{k}), \ldots, I_{t_m}(x, z, \tilde{k}))$$

has m-dimensional density

$$f_{(x,z,\tilde{k})} \colon \mathbb{R}^m \to \mathbb{R}, \quad f_{(x,z,\tilde{k})}(\vec{i}_{\vec{t}}) = \frac{1}{(2\pi)^{m/2}\sqrt{\det C}} \mathrm{e}^{-0.5(\vec{i}_{\vec{t}} - \vec{h}_{\vec{t}}(x,z,\tilde{k}))^T C^{-1} (\vec{i}_{\vec{t}} - \vec{h}_{\vec{t}}(x,z,\tilde{k}))}. \quad (4)$$

Here, $\vec{t} = (t_1, \ldots, t_m)^T$ denotes the collection of points of interest and $\vec{i}_{\vec{t}}$ refers to the measured current vector at the time instances $t_1, \ldots, t_m$. In the key extraction the attacker

does not know the masking values and treats them as realizations of random variables. The random vector $\vec{I}_{\vec{t}}(x, Z, \tilde{k})$ has density

$$\overline{f}_{x,\tilde{k}}(\cdot) \coloneqq |M|^{-1} \sum_{z \in M} f_{x,z,\tilde{k}}(\cdot),$$

where $M \coloneqq \{0,1\}^u$ denotes the set of admissible masking values, and $u \leq 2d(d-0.5)$ denotes the number of considered masking bits.
Since the exact leakage function $\vec{h}_{\vec{t}}$ (m-dimensional vector) is unknown in the key extraction phase, we substitute in (4) the estimated approximate leakage function $\vec{\tilde{h}}^*_{\vec{t}}$ and the estimated covariance matrix $\tilde{C}$, yielding an approximate density $\tilde{f}_{(x,z,\tilde{k})} \colon \mathbb{R}^m \to \mathbb{R}$.

5.2.2 Key extraction

Let $\tilde{k} = (\tilde{k}_0, \ldots, \tilde{k}_{d-1}) \in \{0,1\}^{4d}$ be the targeted part of the shared key, while its d components denote the 4-bit subvectors of the particular shares. Since for each power trace the key shares are selected randomly they are in general different for all power traces. However, it is not necessary to determine all these key shares. Instead, we have to guess the (unshared) subkey $k \in \{0,1\}^4$ on the basis of N_3 power traces.
We introduce the subkey classes

$$\mathcal{C}(k') \coloneqq \{\tilde{k}' \coloneqq (\tilde{k}'_0, \ldots, \tilde{k}'_{d-1}) \mid \tilde{k}'_0 \oplus \cdots \oplus \tilde{k}'_{d-1} = k'\} \quad \text{for } k' \in \{0,1\}^4,$$

that means $\mathcal{C}(k')$ denotes the set of all shared key candidates for the slice $\tilde{K}[\cdot, i, \cdot]$ whose XOR-sum equals k'. Each class $\mathcal{C}(k')$ contains $2^{4(d-1)}$ elements because the shares $\tilde{k}'_0, \ldots, \tilde{k}'_{d-2}$ may attain any value while $\tilde{k}'_{d-1}$ is determined via $\tilde{k}'_{d-1} = k' \oplus \tilde{k}'_0 \oplus \cdots \oplus \tilde{k}'_{d-2}$.
We define the likelihood function $\alpha(\cdot)$ by

$$\alpha_{N_3}(k') \coloneqq \prod_{j=1}^{N_3} \sum_{\tilde{k}' \in \mathcal{C}(k')} \sum_{z' \in M} \tilde{f}_{x,z',\tilde{k}'}(\vec{i}_{j;\vec{t}}) \quad \text{for } k' \in \{0,1\}^4, \tag{5}$$

where $\vec{i}_{j;\vec{t}}$ denotes the current vector of the j^{th} power trace. We decide for that key candidate $k^* \in \{0,1\}^4$, for which $\alpha_{N_3}(k^*)$ is maximal (maximum likelihood estimator). If the power consumption at the considered time instants $t_1, \ldots, t_m$ does not depend on all masking bits (e.g. because only one or two ISW-multiplications are considered), the inner sums in (5) have less than $2^{2d(d-\frac{1}{2})}$ terms, speeding up the evaluation of (5).

Remark 5. Depending on the number m of points of interest it may be profitable to apply e.g. a PCA or LDA. This does not affect formula (5) in general, but only the densities $f(x, z, \tilde{k})$.

Background and justification of decision rule (5) While for masked implementations with "usual" (i.e. unshared) keys the optimal decision rule in terms of the densities is obvious, which is, however, not the case for implementations with shared keys. Although decision rule (5) appears natural, it deserves some further explanation.
We assume that the relevant (targeted) parts of the plaintexts $x_1, \ldots, x_{N_3}$ and masking values $z_1, \ldots, z_{N_3}$ of the power traces $j = 1, 2, \ldots, N_3$ are realizations of independent and identically distributed random vectors $(X_1, Z_1), \ldots, (X_{N_3}, Z_{N_3})$. Since the tweak values are uniformly distributed, the X_j can be assumed to be uniformly distributed on their domain as well. The random vector $\vec{I}_{\vec{t}}(X_j, Z_j, \tilde{k}_j)$ describes the j^{th} random m-dimensional current vector at the time instants $\vec{t} = (t_1, \ldots, t_m)$, while $\tilde{k}_j \in \{0,1\}^{4d}$ denotes the correct shared subkey of trace j.

Assume that the random variables X and Z are independent and uniformly distributed on their respective domains. Then the random variable

$$f_{x,z';\tilde{k}'}(\vec{I}_{\vec{t}}(x,z,\tilde{k}))$$

quantifies the random m-dimensional density of a power trace for the shared key candidate $\tilde{k}'$ when $\tilde{k}$ is the correct shared subkey. At this point, z denotes the correct masking value, while $z' \in M$ is an admissible masking value. We define

$$\begin{aligned} A(k' \mid k) &\coloneqq \sum_{\tilde{k}' \in \mathcal{C}_{k'}} \sum_{\tilde{k} \in \mathcal{C}_k} \sum_{x \in \{0,1\}^p} \sum_{z \in M} \sum_{z' \in M} E_{\vec{R}_{\vec{t}}}\left(f_{x,z';\tilde{k}'}\left(\vec{I}_{\vec{t}}(x,z,\tilde{k})\right)\right) = \\ &= 2^p |M| \sum_{\tilde{k}' \in \mathcal{C}_{k'}} \sum_{\tilde{k} \in \mathcal{C}_k} \sum_{z' \in M} E_{X,Z,\vec{R}_{\vec{t}}}\left(f_{X,z';\tilde{k}'}\left(\vec{I}_{\vec{t}}(X,Z,\tilde{k})\right)\right) \quad \text{for } k,k' \in \{0,1\}^4. \end{aligned} \tag{6}$$

Since the shared subkeys are selected uniformly within the particular subkey classes, the term (6) equals (up to a constant) the expected value of $f_{x,z';\tilde{k}'}\left(\vec{I}_{\vec{t}}(x,z,\tilde{k})\right)$, averaged over randomly selected shared subkeys $\widetilde{k}' \in \mathcal{C}(k')$, $\widetilde{k} \in \mathcal{C}(k)$, random plaintexts X, random masking value Z and admissible masking values Z'.
As noted above, the random vectors $(X_1, Z_1), \ldots, (X_{N_3}, Z_{N_3})$ and the selection of the shared subkeys are independent for all traces. Thus, to decide whether decision rule (5) can be successful it suffices to focus on the expectation of a single trace. Thus, if

$$A(k \mid k) > A(k' \mid k) \quad \text{for all } k,k' \in \{0,1\}^4, \tag{7}$$

the decision rule (5) finds the correct key (with large probability), provided that the sample size N_3 is sufficiently large. In other words, (7) provides a sufficient condition for the success of decision rule (5). At this point, note that if Y denotes a $\mathbb{R}^n$-valued random variable with Lebesgue density $f(\cdot)$ and $g\colon \mathbb{R}^n \to \mathbb{R}$, then $E(g(Y)) = \int_{\mathbb{R}^n} g(y) f(y)\,\mathrm{d}y$ (provided that the integral exists).
Depending on the leakage model, other decision rules than (5) may be reasonable as well. The correct shared subkey $\tilde{k}$ should always provide a large contribution to the outer sum of (5) for the correct subkey class $\mathcal{C}(k)$. However, it might happen that within some wrong subkey class $\mathcal{C}(k'')$ several wrong shared key candidates make a significant contribution to the corresponding outer sum, in total exceeding the sum for the correct subkey class $\mathcal{C}(k)$. In this case, a reasonable option would be to define the likelihood function $\alpha^*(\cdot)$ by

$$\alpha^*_{N_3}(k') \coloneqq \prod_{j=1}^{N_3} \max_{\tilde{k}' \in \mathcal{C}(k')} \left\{ \sum_{z \in M} \tilde{f}_{x,z,\tilde{k}'}(\vec{i}_{j;\vec{t}}) \right\} \quad \text{for } k' \in \{0,1\}^4, \tag{8}$$

and to decide for that key candidate $k^* \in \{0,1\}^4$ for which $\alpha^*_{N_3}(k^*)$ is maximal (maximum likelihood estimator). For decision rule (8) the equivalent to (6) reads as

$$A^*(k' \mid k) \coloneqq \sum_{\tilde{k} \in \mathcal{C}_k} \sum_{x \in \{0,1\}^p} \sum_{z \in M} \max_{\tilde{k}' \in \mathcal{C}(k')} \left\{ \sum_{z' \in M} E_{\vec{R}_{\vec{t}}}\left(f_{x,z';\tilde{k}'}\left(\vec{I}_{\vec{t}}(x,z,\tilde{k})\right)\right) \right\} \text{for } k,k' \in \{0,1\}^4.$$

The equivalent of the sufficient condition (7) is given by

$$A^*(k \mid k) > A^*(k' \mid k) \quad \text{for all } k,k' \in \{0,1\}^4.$$

Example 1. This example illustrates definition (6) and the sufficient condition (7). To keep it simple we consider 1-bit subkeys instead of 4-bit subkeys as above. We assume a non-masked implementation with leakage model $h_t(\tilde{x},\tilde{k}) = a + b \cdot \mathrm{ham}(\tilde{x} \oplus \tilde{k})$. Here,

$\tilde{x}, \tilde{k} \in \{0,1\}^d$, $a \geq 0$ and $b > 0$ are constants, while $\mathrm{ham}(\cdot)$ denotes the Hamming weight. In this scenario, only two subkey classes $\mathcal{C}(0)$ and $\mathcal{C}(1)$ exist, where

$$\mathcal{C}(i) = \left\{ \tilde{k}' = (\tilde{k}'_0, \ldots, \tilde{k}'_{d-1}) \mid \bigoplus_{j=0}^{d-1} \tilde{k}'_j = i \right\}.$$

We focus on a single time point t (i.e., $m = 1$) and assume that the leakage R_t is $N(0, \sigma^2)$-distributed. Since the implementation is not masked, the two inner sums (over z and z') in the first line of (6) vanish. We compute the expectation

$$\begin{aligned}
&E_{R_t}\left(f_{\tilde{x};\tilde{k}'}\left(\vec{I}_t(\tilde{x},\tilde{k})\right)\right) = \int_{-\infty}^{\infty} f_{\tilde{x};\tilde{k}'}(y) f_{\tilde{x};\tilde{k}}(y)\,\mathrm{d}y \\
&= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{\left(y-(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k}')b)\right)^2}{2\sigma^2}} \mathrm{e}^{-\frac{(y-(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k})b))^2}{2\sigma^2}} \frac{}{\sqrt{2\pi}\sigma}\,\mathrm{d}y \\
&= \frac{1}{\sqrt{2\pi}\sigma\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{2y^2-2y(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k}')b)-2y(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k})b)+(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k}')b)^2+(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k})b)^2}{2\sigma^2}}\,\mathrm{d}y \\
&= \frac{1}{2\sqrt{\pi}\sigma} \frac{1}{\sqrt{2\pi}\frac{\sigma}{\sqrt{2}}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{\left(y-\frac{(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k}')b)+(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k})b)}{2}\right)^2}{2(\sigma/\sqrt{2})^2}} \mathrm{e}^{-\frac{\left((a+\mathrm{ham}(\tilde{x}\oplus\tilde{k}')b)-(a+\mathrm{ham}(\tilde{x}\oplus\tilde{k})b)\right)^2}{4\sigma^2}}\,\mathrm{d}y \\
&= \frac{1}{2\sqrt{\pi}\sigma} \mathrm{e}^{-\frac{\left(\mathrm{ham}(\tilde{x}\oplus\tilde{k}')-\mathrm{ham}(\tilde{x}\oplus\tilde{k})\right)^2}{4(\sigma/b)^2}}. \qquad (9)
\end{aligned}$$

Equation (9) confirms the intuition that the expectation is maximal if $\tilde{k}' = \tilde{k}$, and that it is the smaller the more the assumed Hamming weight $\mathrm{ham}(\tilde{x} \oplus \tilde{k}')$ differs from the correct Hamming weight $\mathrm{ham}(\tilde{x} \oplus \tilde{k}')$. We note that $G = \{0,1\}^d$, equipped with $\oplus$ (bitwise XOR-operation), is an Abelian group. In particular, $\mathcal{C}(0)$ is a subgroup of G with index 2. If $\tilde{x} \in \mathcal{C}(0)$, then $\mathcal{C}(i) \oplus \tilde{x} = \mathcal{C}(i)$ for $i = 0, 1$, whereas $\mathcal{C}(i) \oplus \tilde{x} = \mathcal{C}(1-i)$ if $\tilde{x} \in \mathcal{C}(1)$. If we assume that the shared plaintexts are uniformly distributed on $\{0,1\}^d$, then (9), Fubini's Theorem and the above considerations simplify (6) (with $\tilde{x}$ in place of x) to

$$\begin{aligned}
A(k' \mid k) &= \sum_{\tilde{x}\in\{0,1\}^d} \sum_{\tilde{k}'\in\mathcal{C}(k')} \sum_{\tilde{k}\in\mathcal{C}(k)} \frac{1}{2\sqrt{\pi}\sigma} \mathrm{e}^{-\frac{\left(\mathrm{ham}(\tilde{x}\oplus\tilde{k}')-\mathrm{ham}(\tilde{x}\oplus\tilde{k})\right)^2}{4(\sigma/b)^2}} \\
&= 2^{d-1} \left(\sum_{\tilde{k}'\in\mathcal{C}(k')} \sum_{\tilde{k}\in\mathcal{C}(k)} \frac{1}{2\sqrt{\pi}\sigma} \mathrm{e}^{-\frac{\left(\mathrm{ham}(\tilde{k}')-\mathrm{ham}(\tilde{k})\right)^2}{4(\sigma/b)^2}} \right. \\
&\qquad \left. + \sum_{\tilde{k}'\in\mathcal{C}(1-k')} \sum_{\tilde{k}\in\mathcal{C}(1-k)} \frac{1}{2\sqrt{\pi}\sigma} \mathrm{e}^{-\frac{\left(\mathrm{ham}(\tilde{k}')-\mathrm{ham}(\tilde{k})\right)^2}{4(\sigma/b)^2}} \right). \qquad (10)
\end{aligned}$$

Likewise, (10) applies when x is uniformly distributed on $\{0,1\}$ and $\tilde{x} = (x, 0 \ldots, 0)$, the case which corresponds to the Clyde cipher. In this case, the factor 2^{d-1} is replaced by 1, and $\tilde{x} \in \{0,1\}^d$ in the outer sum reads $x \in \{0,1\}$ (as in (6)). Note that (10) does not depend on $\tilde{x}$. Furthermore, also (11) remains valid with smaller constant c.

By (10) it suffices to verify the sufficient condition (7) for $(k, k') \in \{(0,0), (0,1)\}$. Obviously, in $\mathcal{C}(0)$ there exist $\binom{d}{2u}$ elements with Hamming weight $2u$ (for $0 \leq 2u \leq d$), while $\mathcal{C}(1)$ contains $\binom{d}{2v+1}$ elements with Hamming weight $2v+1$ (for $0 \leq 2v+1 \leq d$). This observation

allows to further simplify (10) to

$$A(0 \mid 0) = c\left(\sum_{u=0}^{\lfloor \frac{d}{2} \rfloor}\sum_{v=0}^{\lfloor \frac{d}{2} \rfloor}\binom{d}{2u}\binom{d}{2v}z_0^{(2u-2v)^2} + \sum_{u=0}^{\lfloor \frac{d-1}{2} \rfloor}\sum_{v=0}^{\lfloor \lfloor \frac{d-1}{2} \rfloor}\binom{d}{2u+1}\binom{d}{2v+1}z_0^{(2u-2v)^2}\right)$$

$$A(1 \mid 0) = c\left(\sum_{u=0}^{\lfloor \frac{d-1}{2} \rfloor}\sum_{v=0}^{\lfloor \frac{d}{2} \rfloor}\binom{d}{2u+1}\binom{d}{2v}z_0^{(2u+1-2v)^2} + \sum_{u=0}^{\lfloor \frac{d}{2} \rfloor}\sum_{v=0}^{\lfloor \lfloor \frac{d-1}{2} \rfloor}\binom{d}{2u}\binom{d}{2v+1}z_0^{(2u-2v-1)^2}\right), \quad (11)$$

where $c = \frac{2^{d-1}}{2\sqrt{\pi}\sigma} > 0$ and $z_0 = \mathrm{e}^{-\frac{1}{4(\sigma/b)^2}} \in (0,1)$.
For the special case $d = 3$, careful computations yield

$$A(0 \mid 0) = 2c\left(10 + 6z_0^4\right) \qquad \text{and} \qquad A(1 \mid 0) = 2c\left(15z_0 + z_0^9\right).$$

Let

$$p(z) \coloneqq 10 - 15z + 6z^4 - z^9,$$

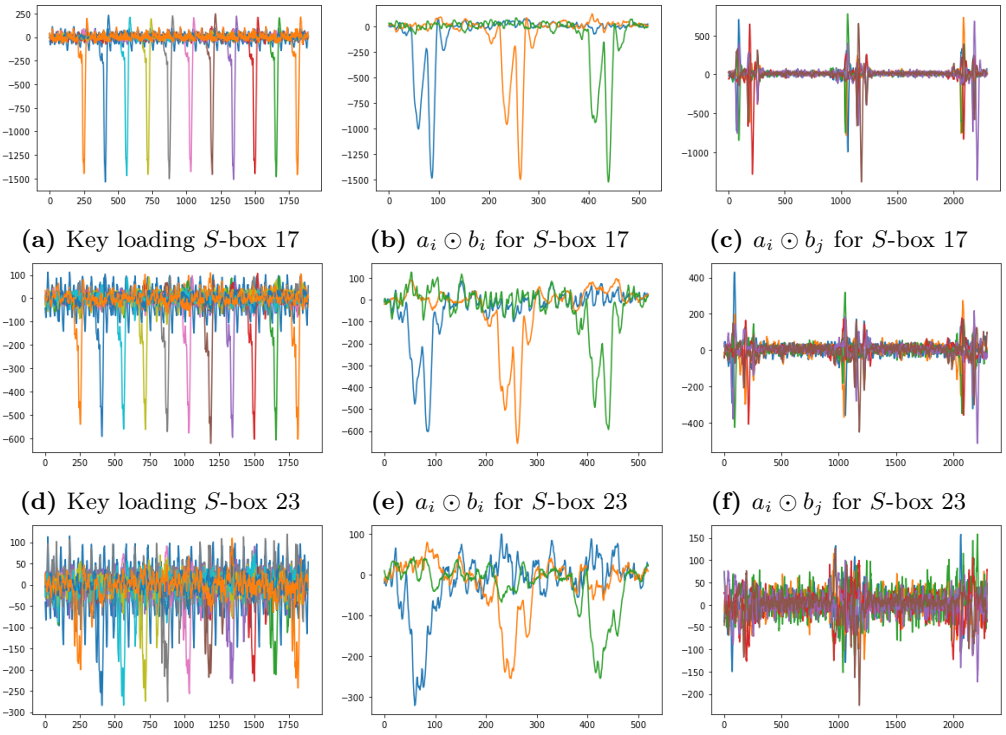
so that the sufficient condition (7) is equivalent to $p(z_0) > 0$. It holds that $p(z) > 0$ for $z \in (0,1)$: We have $p'(z) = -15 + 24z^3 - 9z^8$ and $p''(z) = 72z^2 - 72z^7$, so in particular $p(0) = 10$, $p(1) = 0$, $p'(0) = -15$, $p'(1) = 0$ and $p''(z) > 0$ for $z \in (0,1)$. Thus $p'(z) < 0$ for $z \in (0,1)$, which proves the claim. Hence for $d = 3$, condition (7) is fulfilled. Similarly,

$$A^*(0 \mid 0) = c^* \cdot 1 \qquad \text{and} \qquad A^*(1 \mid 0) = c^* z_0,$$

where $c^* = \frac{1}{2\sqrt{\pi}\sigma} > 0$ and $z_0 = \mathrm{e}^{-\frac{1}{4(\sigma/b)^2}} \in (0,1)$, and thus $A^*(0 \mid 0) > A^*(1 \mid 0)$.

5.3 Numerical Results

Next, we show some numerical results of the stochastic approach and discuss its advantages and shortcomings. In particular, we have a closer look at the estimated basis coefficients and the information they contain. We focus on $d = 3$ shares here, substract a global mean trace before the estimation and use only those operations that can be exploited without any masking bits, namely the key loading, the plaintext and tweak addition and the ISW-multiplication in the first two steps of the S-box computation, see also Remark 4. This yields in total 50 POIs that are chosen manually by inspecting the evolution of the estimated coefficients for S-box 17 in the interval in which the respective operation takes place. Indeed, for the selected S-box there is always a sharp peak in the signal that allows to take the corresponding time point as POI. As we know that all $s = 32$ S-boxes are processed in parallel, the same POIs can be taken for all other S-boxes as well. Figure 6 exemplarily shows the evolution of the $4 \cdot d = 12$ estimated coefficients for the key loading (left column) and the ISW-multiplications $a_i \odot b_i$ (middle column) respective $a_i \odot b_j$ (right column) in case of the "easy" S-box 17 (top row), an "medium" S-box 23 (middle row) and a "difficult" S-box 0 (bottom row). Each subfigure depict the evolution of the estimated values of the coefficients in front of the different basis functions, plotted over the range in which the respective operations take place. Already here the coefficients indicate that the different S-boxes are differently difficult to attack, as the coefficients of the "easy" S-box 17 are very clear, while the ones of the harder S-boxes are much noisier. In particular for S-box 17 one sees a sharp peak for each basis function, which allows to select appropriate POIs simply by visual inspection. Also, depending on the exploited operation, the order of magnitude of the coefficients for the different S-boxes varies considerably, the coefficients for S-box 17 are up to five times as large as those for e.g. S-box 0.
To further illustrate the different difficulty levels, we display in Figure 7 the evolution of the coefficient corresponding to a single key bit if all 32 S-boxes are estimated simultaneously. In accordance with the observations in the neural network approach, see also Figure 2,



(a) Key loading S-box 17 **(b)** $a_i \odot b_i$ for S-box 17 **(c)** $a_i \odot b_j$ for S-box 17

(d) Key loading S-box 23 **(e)** $a_i \odot b_i$ for S-box 23 **(f)** $a_i \odot b_j$ for S-box 23

(g) Key loading S-box 0 **(h)** $a_i \odot b_i$ for S-box 0 **(i)** $a_i \odot b_j$ for S-box 0

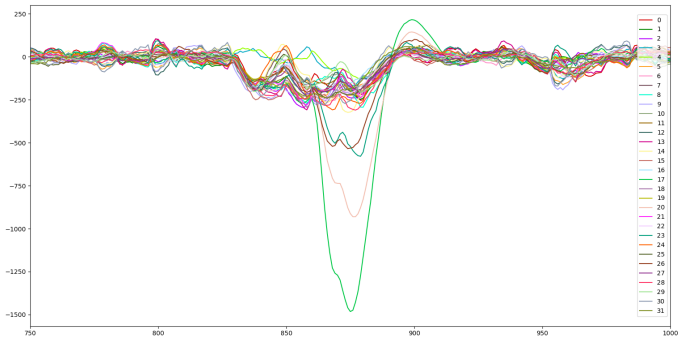
Figure 6: Estimation of basis coefficients for different operations and S-boxes.

the easiest S-box 17 shows the highest peak, while the medium S-boxes 20, 23 or 26 already yield significantly lower peaks (approximately one half of the maximal peak), which are however still clearly distinguishable from the other, rather noisy signals of the remaining S-boxes whose height is only one sixth compared to the maximal height. These observations are qualitatively similar in case of the ISW-multiplication, see Figures 7(b) and (c), where we exemplarily show the first coefficients for the computation of the first $c_i = a_i \odot b_i$ respective $a_i \odot b_j$ in the ISW-multiplication for $x[0] \odot x[1]$.

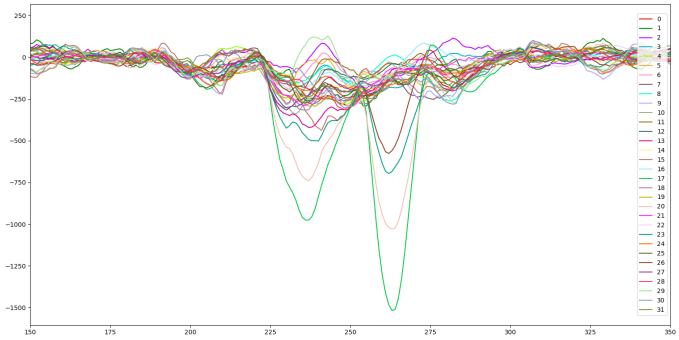
Next, we examined how many traces are needed to identify the correct (unshared) key. In case of S-box 17, these are approximately 50 traces when avoiding operations that involve masking bits. It turned out that including masking bits does not yield significantly better results, but leads to a much higher numerical effort in the attack phase. This might be explained by the observation that already the key loading and the plaintext addition involve all unknown key bits and that their estimated coefficients are higher and clearer as in case of the operations in the context of the ISW-multiplication.

Contrary to the deep learning based approach, the attack phase does not succeed for the difficult S-boxes, even when taking a high number of traces. This is most likely due to the fact that already the estimation of the coefficients yield rather poor results.

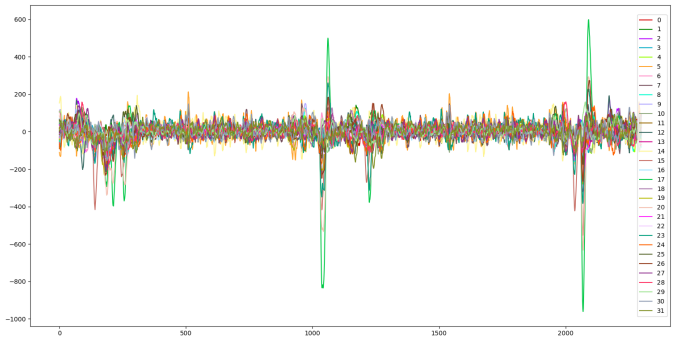
The leakage of the "easy" S-box 17 is very large, allowing a successful attack on subkey k_{17}. Unfortunately, its large leakage spoils the sigma-to-noise ratio of the other S-boxes. The signal-to-noise ratio quantifies the impact of small and large β-coefficients. We define $\vec{x} = (x_0, \ldots, x_{31})$, $\vec{z} = (z_0, \ldots, z_{31})$, and $\vec{\tilde{k}} = (\tilde{k}_0, \ldots, \tilde{k}_{31})$ and assume that the power consumption of all S-boxes are independent. This justifies to extend equation (3) from a



(a) Key loading



(b) $a_i \odot b_i$



(c) $a_i \odot b_j$

Figure 7: Estimation of basis coefficients for different operations and all S-boxes simultaneously.

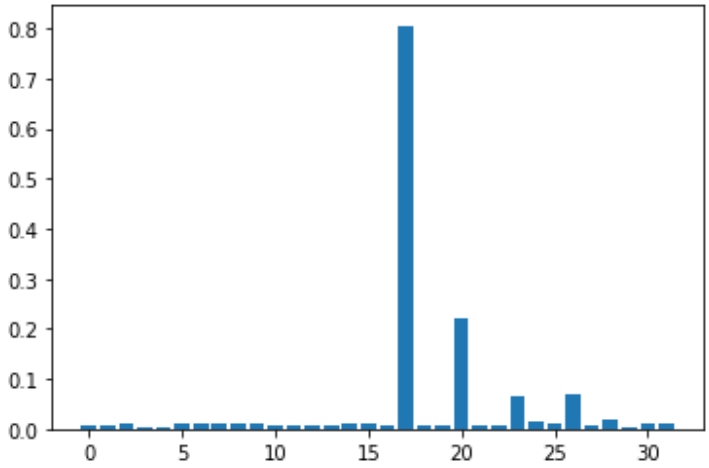


Figure 8: Estimation of the signal-to-noise ratio of the different S-boxes for the key loading operation.

single to all S-boxes by

$$I_t(\vec{x}, \vec{z}, \vec{\tilde{k}}) = \sum_{j=0}^{31} h_{t(j)}(x_j, z_j, \tilde{k}_j) + R_t^*.$$

The random variable R_t^* summarizes the (centered) noise at time t that does not originate from any of the 32 S-box operations but from further operations that take place at the same time. We interpret the values $x_0, \ldots, x_{31}$ and $z_0, \ldots, z_{31}$ as realizations of random variables $X_0, \ldots, X_{31}$, $Z_0, \ldots, Z_{31}$, so that

$$\mathrm{Var}\left(I_t(\vec{X}, \vec{Z}, \vec{\tilde{k}})\right) = \sum_{j=0}^{31} \mathrm{Var}_{X_j, Z_j}\left(h_{t(j)}(X_j, Z_j, \tilde{k}_j)\right) + \mathrm{Var}\left(R_t^*\right).$$

The term $\mathrm{Var}_{X_j,Z_j}(\cdot,\cdot,\cdot)$ quantifies the "algorithmic noise", which depends on X_j, Z_j and $\tilde{k}_j$. Given a fixed shared key $\vec{\tilde{k}}$, it depends on the signal-to-noise ratio

$$\mathrm{SNR}(r) \coloneqq \frac{\mathrm{Var}_{X_r, Z_r}\left(h_{t(r)}(X_r, Z_r, \tilde{k}_r)\right)}{\sum\limits_{j=0,\ldots,31; j\neq r} \mathrm{Var}_{X_j, Z_j}\left(h_{t(j)}(X_j, Z_j, \tilde{k}_j)\right) + \mathrm{Var}\left(R_t^*\right)}$$

whether an S-Box r is "easy" or "difficult".
If $g_{(j),0,t,\tilde{k}_j} = 1, g_{(j),1,t,\tilde{k}_j}, \ldots, g_{(j),u-1,t,\tilde{k}_j}$ denotes an orthonormal basis for S-box j, the computation of the algorithmic variance of S-box j simplifies to

$$\mathrm{Var}_{X_j, Z_j}\left(h_{t(j)}(X_j, Z_j, \tilde{k}_j)\right) \approx \sum_{\tau=1}^{u-1} \beta^2_{(j),\tau,t;\tilde{k}_j} \approx \sum_{\tau=1}^{u-1} \tilde{\beta}^2_{(j),\tau,t;\tilde{k}_j}.$$

Figure 8 shows the estimated signal-to-noise ratio of the different S-boxes for the key loading operation. The corresponding figures of the other operations look qualitatively similar.

5.4 Leakage Assessment

The implementation at hand is especially protected against side-channel attacks. Nevertheless, with our neural network-based approach it is possible to extract enough information

to reveal the secret key, and in the aftermath of the contest we had a closer look at the parts of the implementation that are actually leaking sensitive information.
By its design it is evident which operations of the algorithm the stochastic approach exploits, as these are directly reflected in the choice of the basis functions. For the neural network approach, however, it is less clear which information the network uses, since it is fed with the whole (subsampled) traces and performs the selection of relevant points on its own. At the same time, this illustrates one of the major advantages of neural network-based approaches, as they do not require the preselection of points of interest, but find them of their own.
We did different experiments to find out which points in the trace are important for our neural network based leakage extractor. All of them were run on the 3-share data set, as we did not expect substantially different phenomena to appear for the higher targets. We did, however, not test this assumption, but leave it for future research. We tried the following techniques:

- First, we subdivided our 62,500 point into 625 windows, each of which consisting of 100 points, and compared the predicted bit probabilities for the shared state before and after zeroizing each window, averaging results over a small set of traces.
- We calculated the gradient of the predictor as a function of the input trace for a subset of input traces in our data set.

In both experiments, we assumed that points that are of interest to the neural network would show up as points in which the influence of small changes to the trace on the prediction output is high. Both experiments yielded qualitatively similar results, so we only describe the results of the window-wise zeroization experiment in more details.

Methods We selected 200 traces from the data set FKEY_SW3_K1_1000_0, sent them through our predictor, and converted the scattershot predictions into predictions of single key bits. We then zeroized each of our 625 windows in turn, ran the same prediction using the altered trace, and finally calculated the mean square error of the new prediction with respect to the prediction obtained from the unaltered trace.

Results The main active region for all traces was found to be between data points 35,000 and 60,000, with smaller and less significant active regions also at the very beginning of the trace for some S-boxes. Within that very broad active region, large differences in signal attribution were observed between different S-boxes. Figure 9 shows the relevant profile for the S-boxes 17, 0 and 22. It is worth noting here that the prediction of the particularly difficult S-box 22 draws on some regions of the trace that are not used at all for the other two S-boxes. Matching the operations of the algorithm with the shape of the trace and the time points the first two active regions around time point 38,000 and 43,000 are during the S-box computation, more precisely they correspond to the ISW-multiplications $\big(y[1,\cdot] \odot x[3,\cdot]\big) \oplus x[0,\cdot]$ and $y[2,\cdot] = \big(y[0,\cdot] \odot y[1,\cdot]\big)$ (see equation (1)) after the mask refreshing. The very active region after point 48,000 is during the computation of the L-box, which is rather surprising and seems a bit odd since the network actually aims at predicting the S-box output. Furthermore, it might serve as an explanation why the AI-based approach considerably outperforms the stochastic approach (and any other classical approach we tried such as templates), since these approaches cannot exploit the L-box computation acting on rows of the Clyde-state instead of columns. This is mainly due to numerical reasons, in particular the huge amount of masking bits and the fresh randomness caused by the mask refreshing. The stochastic approach yet helped us to understand why attacks on different S-boxes are so differently efficient and allowed to quantify this knowledge.

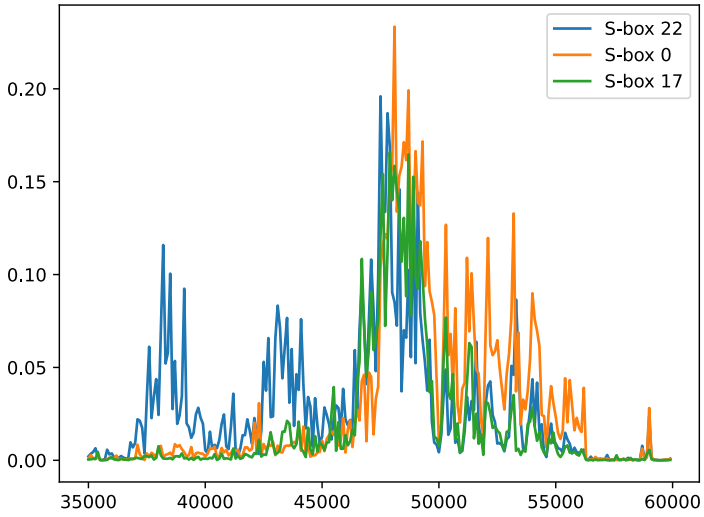


Figure 9: Results of our main leakage attribution experiment for our deep neural network. Input traces for the 3-share challenge were divided into 625 windows 100 data points in length and the perturbation of the bit-wise prediction vector induced by zeroizing each window was recorded. Results here shown are for three selected S-boxes.

6 Conclusion and Future Work

In this paper we presented our deep learning-based solution to the CHES 2020 challenge and gave further insights into certain aspects of our approach, in particular the scattershot encoding.

Furthermore, we provided and analyzed an alternative, "classical" approach, namely the stochastic approach. Compared to previous works, both the AI-method (in form of the scattershot encoding) as well as the stochastic approach (in form of the key extraction phase) had to be adapted to the masked setting encountered in the contest, where the randomly chosen shared states make the combination of leakage information across different traces much more difficult compared to an unshared setting.

Our results show that fairly strong logical countermeasures can be broken by a deep learning-based attack that uses only a limited amount of information about the implementation. This might be a hint that complete new strategies and countermeasures are needed to protect implementations against AI-based attacks.

However, our results also show that explaining the leakage found by such methods is not always easy: we do, for instance, not have a good explanation from first principles yet for the massive difference in the magnitude of leakage observed for different S-boxes. In addition, it can be difficult to distinguish logical problems of leakage extraction from the lack of leakage: this point is driven home especially by our experiments to better understand the effect scattershot encodings using a completely synthetic problem.

Our research also shows that much remains to be discovered in a wide variety of areas around ML-based side-channel attacks: pre-processing, data representation, network architectures and post-processing model outputs are all less well understood for side-channel analysis than in more standard application domains of ML-methods such as signals processing tasks.

References

[1] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, F.-X. Standaert, and P.-Y. Strub. Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations. *Journal of Cryptographic Engineering*, pages 1–10, 2019.

[2] G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F.-X. Standaert, and P.-Y. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. Cryptology ePrint Archive, Report 2016/912, 2016. https://eprint.iacr.org/2016/912.pdf.

[3] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Annual International Cryptology Conference*, pages 123–153. Springer, 2016.

[4] D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, et al. Spook: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher. *IACR Transactions on Symmetric Cryptology*, 2020.

[5] O. Bronchain and F. Standaert. Breaking masked implementations with many shares on 32-bit software platforms or when the security order does not matter. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):202–234, 2021.

[6] G. Cassiers, B. Grégoire, I. Levi, and F.-X. Standaert. Hardware private circuits: From trivial composition to full verification. Cryptology ePrint Archive, Report 2020/185, 2020. https://eprint.iacr.org/2020/185.

[7] C. Glowacz, V. Grosso, R. Poussier, J. Schüth, and F.-X. Standaert. Simpler and more efficient rank estimation for side-channel security assessment. In *International Workshop on Fast Software Encryption*, pages 117–129. Springer, 2015.

[8] A. Gohr, S. Jacob, and W. Schindler. Subsampling and knowledge distillation on adversarial examples: New techniques for deep learning based side channel evaluations. In *Selected Areas in Cryptography - SAC 2020*, volume 12804 of *Lecture Notes in Computer Science*, pages 567–592. Springer, 2020.

[9] D. Goudarzi and M. Rivain. How fast can higher-order masking be in software? In J.-S. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 567–597, Cham, 2017. Springer International Publishing.

[10] V. Grosso, G. Leurent, F.-X. Standaert, and K. Varıcı. LS-designs: Bitslice encryption for efficient masked software implementations. In *International Workshop on Fast Software Encryption*, pages 18–37. Springer, 2014.

[11] V. Grosso, G. Leurent, F.-X. Standaert, K. Varici, F. Durvaux, L. Gaspar, and S. Kerckhof. SCREAM & iSCREAM side-channel resistant authenticated encryption with masking. *Submission to CAESAR*, 2014. https://competitions.cr.yp.to/round2/screamv3.pdf.

[12] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[13] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 2017.

[14] National Institute of Standards and Technology (NIST). Lightweight cryptography standardization process, August 2018. https://csrc.nist.gov/Projects/lightweight-cryptography.

[15] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina. Sok: Deep learning-based physical side-channel analysis. Cryptology ePrint Archive, Report 2021/1092, 2021. https://ia.cr/2021/1092.

[16] W. Schindler, K. Lemke, and C. Paar. A stochastic model for differential side channel cryptanalysis. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 30–46, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] L. Wu, G. Perin, and S. Picek. The best of two worlds: Deep learning-assisted template attack. Cryptology ePrint Archive, Report 2021/959, 2021. https://ia.cr/2021/959.

[18] G. Zaid, L. Bossuet, M. Carbone, A. Habrard, and A. Venelli. Conditional variational autoencoder based on stochastic attack. Cryptology ePrint Archive, Report 2022/232, 2022. https://ia.cr/2022/232.