Taking AI-Based Side-Channel Attacks to a New Dimension

Lucas David Meier[0009−0001−5970−0293], Felipe Valencia[0000−0001−6884−9340], Cristian-Alexandru Botocan[0009−0000−8309−808X], and Damian Vizár[0009−0006−4459−2951]

CSEM, Neuchâtel, Switzerland, {lucas.meier, andres.valencia, damian.vizAr}@csem.ch
botocan.christian@gmail.com

Abstract. This paper[1] revisits the Hamming Weight (HW) labelling function for machine learning assisted side channel attacks. Contrary to what has been suggested by previous works, our investigation shows that, when paired with modern deep learning architectures, appropriate pre-processing and normalization techniques; it can perform as well as the popular identity labelling functions and sometimes even beat it. In fact, we hereby introduce a new machine learning method, dubbed *dimension 0*, that helps solve the class imbalance problem associated to HW, while significantly improving the performance of unprofiled attacks. We additionally release our new, easy to use python package that we used in our experiments, implementing a broad variety of machine learning driven side channel attacks as open source, along with a new dataset AES_nRF, acquired on the nRF52840 SoC.

Keywords: Profiled and Unprofiled Side-Channel Attacks, Deep Learning, Softmax Function

1 Introduction

Cryptographic algorithms are designed to ensure that secret input arguments cannot be recovered given the knowledge of public data (such as ciphertexts of a blockcipher) or even given some inputs considered private (such as plaintexts of a block cipher). This is the purview of classical cryptanalysis, where the cryptographic algorithm of interest is treated as a black-box. However, in practical implementations on physical devices (software on microcontrollers or hardware accelerators), the execution of these algorithms involves physical processes. These processes can correlate with input values, making them observable through physical variables. For instance, the power consumption when loading a secret key correlates with its Hamming Weight, leaking key information. Side-Channel Attacks (SCA) exploit such physical channels to recover secret data, even from mathematically secure algorithms. Therefore, real-world security requires cryptographic robustness and secure implementation against SCA, especially in embedded systems that are often accessible to attackers.

The study of SCA began with Timing Attacks [27] and evolved to more complex methods such as Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [28], Template Attacks [7], Welch's t-test [56], and Correlation Power Analysis (CPA) [4]. SCA techniques are classified based on adversarial control (passive or active), observed variables (i.e., power, electromagnetic radiation, timing, etc.), exploitation methods (i.e., statistical analysis, machine learning, etc.), and the use of the profiling phase. This paper focuses on passive power analysis attacks using machine learning, both profiled and unprofiled.

Artificial Intelligence (AI) proved to be very effective in SCA. The use of Machine Learning (ML) in SCA began in 2011 [21], and deep learning (DL) soon became popular for overcoming SCA countermeasures that resisted traditional statistical attacks, including template attacks [31]. While other ML methods like SVMs, decision trees, random forests, KNNs, and k-means were explored, deep neural networks (DNN) have shown the best performance [19]. Recent works have gained insights into the role of various NN components to in the efficacy of DNN-based SCA [67],

[1] This article is an extended version of the manuscript submitted by the authors to the CASCADE 2025 conference and to Springer-Verlag on the 25th January 2025. The version published by Springer-Verlag will be available soon.

optimized NN architectures for faster learning and better key recovery [64], and applied various deep learning techniques to enhance attack performance [24, 66, 18, 44, 65, 49].

The power of ML-SCA (ML) attacks based on NNs depends heavily on the selection of model hyperparameters, such as network topology, activation functions and learning rate. One of these critical hyperparameters is the choice of the leakage function. The two main options from the literature are either Hamming Weight (HW) labelling function or the identity (ID) labelling function.

The experiments of existing works investigating the use of HW labelling have suggested that models with ID labelling are capable of results that surpass what can be achieved with HW. For example, Picek et al. show that HW labeling suffers from the class imbalance problem [46], which can degrade model performance. In the study by Benadjila et al. [2], it was shown that during hyperparameter searching, no parameters result in a successful attack on an MLP model using HW labeling, whereas at least one successful attack is possible with ID. Following these finding, the popularity of ID labelling has exceeded that of HW, with works such as those by Wouters et al. [64] and Zaid et al. [67] focusing on attacks using only ID labeling. Recent works that used both labeling functions seemed to confirm the ID labeling superiority. For example, Rjisdijk et al. [49] reported superior results with ID labeling in an SCA study using RL techniques. Similarly, Kerkhof et al. [23] found that a new SCA loss function performed better with ID labeling than HW across multiple datasets. Finally, a recent study on the ASCAD datasets by Egger et al. [11] considered the HW leakage model for classical CPA attacks, but relied only on the Identity labelling for ML attacks.

All in all, the results published so far show HW labelling rather unfavorably, which resonates with the more frequent use of ID labeling in the more recent works. Yet, the existing evidence may not be sufficient to designate ID labeling as a generally superior option, as no works have experimentally verified efficacy of HW labelling on the modern DNN architectures, nor have there been any new attempts to overcome the class imbalance problem since 2019 [46]. At the same time, the interest in having a more up-to-date comparison between HW and ID labelling functions is not only purely academic. HW labeling allows to mount practical attacks targeting intermediate variables of 16, 32 or more bits, while the same using ID would incur impractical computational and storage complexities [1, 32].

In this paper, we set out to close the gap between the amount of recent results ID and HW labelling, and seek to answer the question: *"Can HW labelling function match or out-perform ID with recent DNN models and suitable additional techniques?"* For that, we use state-of-the-art models and datasets from the ASCAD family [2], which have different levels of security (masked implementation and jittering). A major contribution of our work here is a new DL method, which allowed models with HW labelling to match, and sometimes outperform the same model using ID labelling. The new method consists of transposing the matrix containing a model's output logits for multiple input traces before the softmax function is evaluated.[2]

Contributions. Our contribution is threefold. First, we introduce a new DL method for SCA called *dim0*, which can outperform state-of-the-art balancing techniques on Hamming Weight labelled datasets, beat state-of-the-art using the easier identity labelling and has the potential to greatly improve unprofiled attacks as well. Second, we publish a new ML python-based package implementing a variety of the state-of-the-art ML-based SCA techniques that is easy to use and easy to extend.[3] Last, we release our home-made SCA dataset called AES_nRF acquired on the nRF52840 SoC for an unprotected implementation of AES. AES_nRF contains 47.5k profiling traces with random keys and 2.5k attack traces with fixed key.

[2] In other words, we change the *dimension*, along which the softmax is computed.
[3] The MLSCAlib is available on this link https://github.com/csem/MLSCAlib

2 Preliminaries

2.1 Notation

Throughout this document, we denote $\mathbf{v}_i$ or $\mathbf{v}[i]$ as being the i-th entry of a vector $\mathbf{v}$. Matrices are bold, capitalized and in italic, as $\boldsymbol{M}$. Let N_c denote the number of classes, N_p and N_a the number of profiling and attack traces, N_b the number of traces in a batch (batch size) and N_s the number of samples in a trace. Indexes t, c and e will refer to a trace, class and epoch number respectively.

2.2 Side-Channel Attacks

Side-Channels Attacks (SCA) are attacks that target the implementation instead of the mathematical structure of a security algorithm. They use side-channel information (timing, power, cache memories, etc.) that depends on secret values. To mount an attack the adversary 1) creates a model to estimate side-channel information as a function of secret values, 2) applies the function to multiple secret values hypothesis, then 3) measures the side-channel information from the target device, and finally 4) discriminates incorrect secret values hypothesis comparing estimations with measurements [55]. The comparison can be made using means difference, correlation, mutual information, t-test [56], etc. The estimation model can be created with measurements of a controlled device, in this case it is a profiled attack, otherwise it is an unprofiled attack. In some profiled attacks the model is constrained to use most important time samples or Points Of Interest (POI). Machine learning can be used to find the POI, create the estimation model and/or to discriminate hypothesis [35].

2.3 AES

AES (Advanced Encryption Standard) [15] is a block cipher that, given a plaintext of 128 bits and a key of 128, 192, or 256 bits, produces a ciphertext of 128 bits. The decryption algorithm of AES recovers the plaintext given a ciphertext and a key. AES is an iterated block cipher that expands the key into several 128-bit round keys and iteratively applies a round function to the plaintext and a round key. The round function is composed of four transformations: addRoundKey (AK), subBytes (SB), shiftRows (SR) and mixColumns (MC), where the state is represented as a matrix of 4x4 bytes. SubBytes transformation is an invertible non-linear byte substitution (substitution box or Sbox) applied byte-wise on the AES state. The remaining transformations are linear. The last round does not have mixColumns and is followed by an xor of the final round key. The number of rounds depends on the key length.

2.4 Leakage Function

The attacker does not directly learn the key value from the traces. Instead, it targets an intermediate value of the AES encryption (i.e., typically, the 8-bit output or input of the Sbox operation) and uses a specific leakage function for evaluation. In SCA, the leakage function is the same as the labeling function in machine learning since both functions map data to a form that can be used for analysis. A common option for labelling is the identity function (ID) which represents the actual byte value of the output, resulting in 256 different classes. Another option is the Hamming Weight (HW) of the output value. This usually works well since the power consumption of a computation is directly related to the number of bits set to one or zero. However, this gives rise to a class imbalance: the classes 0 and 8 are only reached through a single possible (Sbox) output value, whereas the class 4 is linked to 70 values. As a consequence, the rare classes are more informative than others (label 0 and 8 can only be produced by one key given a plaintext). Besides, physical observations of traces with a HW value 3,4 or 5 tend to look the same, while a trace with HW 0 and one with HW 8 are easily differentiable, as shown by Fan et al. [14] on their Figure 5 plotting the HW-two-dimensional distribution of power consumption.

2.5 Machine Learning

Classification using machine learning (ML) is defined as training a model to learn from the input data features, such that it classifies the input and divides it into discrete classes. In the SCA context, we have as a training (profiling) dataset the side-channel traces from a device with known inputs and keys, each labeled with the associated intermediate value or key byte. After training, the model helps predict secret keys by allocating probabilities to each class of traces in the testing dataset. This inference process is also called the attacking phase.

Multi-Layer Perceptron. The MLP contains multiple layers of perceptrons. A perceptron is represented as a function $f(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b$, where the trainable parameters $\mathbf{w}, b$ are called weight and bias term. The model is composed of an input layer, hidden layers in the middle of the network, and an output layer. An activation function (often RELU [37] or SELU [26]) is applied to the output of each perceptron to add non-linearity to the model.

Convolutional Neural Networks. Convolutional Neural Networks (CNN) combine data processing operations (e.g., convolutions, pooling, batch normalization, etc.) with a final Multi-Layer Perceptron layers (also called a fully Connected Layer in this context). Mathematically, the convolution is expressed as $\mathbf{x}^{i+1}[n] = \sum_{k}^{N_f} \mathbf{f}[k]\mathbf{x}^{i}[n-k]$, where $\mathbf{f}[n]$ represents a filter with N_f elements and $\mathbf{x}^i$ the output of layer i (or the input trace for $i = 0$).

Logits. Logits are the raw, unnormalized output values produced by the last layer of a neural network, such as an MLP or CNN, before applying an activation function. These values represent the model's confidence scores for each class. There is a one-to-one correspondence between the classes and the logits.

Softmax. In order to turn the logits into a probability distribution, the softmax function is used:

$$softmax_1(\boldsymbol{V}, t, c) = \frac{e^{\boldsymbol{V}_{t,c}}}{\sum_{j=0}^{N_c-1} e^{\boldsymbol{V}_{t,j}}}$$

where $\boldsymbol{V}$ represents a $N_b \times N_c$ matrix of logits, t the input number and c the class index. For each input, the sum of the probabilities over each class sums up to one.

Loss Functions. The loss function measures the discrepancy between predicted classes and real classes. We use Negative Log-Likelihood (NLL) as loss function because it is asymptotically equivalent to maximizing the Perceived Information (PI), which at the same time is the lower bound of Mutual Information (MI). Training with NLL is an efficient estimation of MI [35].

Its definition is

$$\mathcal{L}_{nll} = \frac{1}{N_p} \sum_{i=1}^{N_p} -\log_2(\tilde{\mathbf{y}}_\mathbf{i})\ \mathbf{y_i}$$

where $\mathbf{y_i}$ are the labels, $\tilde{\mathbf{y}}_\mathbf{i}$ are the model's prediction w.r.t the input vector $\boldsymbol{T}_i$ and the model parameters.

Batched Learning. In most of the cases, the training data is given in batches to the model in order to reduce the instantaneous memory consumption. The batch size is typically in the order of one hundred traces.

2.6 Confusion Matrix

The confusion matrix is a matrix whose vertical axis designates the ground truth, i.e. the real classes, and the horizontal axis is the actual prediction for each class. It shows which classes the model predicts correctly. The value $\boldsymbol{M}_{i,j}$ counts the samples from class i that were classified to class j by the model. When depicted graphically, we should see a diagonal shape (from top left to bottom right) in the confusion matrix if the model performs well. In case of imbalance, we may

see vertical lines on common classes. Formally, the i-th row of the $N_c \times N_c$ confusion matrix $\boldsymbol{M}$ is defined by:

$$\boldsymbol{M}_i = \frac{\sum_{j=0}^{N_a} \mathbb{1}\{\mathbf{y}_j = i\} \cdot model(\boldsymbol{X}_j)}{\sum_{j=0}^{N_a} \mathbb{1}\{\mathbf{y}_j = i\}}$$

where N_a is the number of traces in the attack dataset matrix $\boldsymbol{X}$, $\mathbf{y}$ is the label vector for the attack traces and *model* the trained machine learning model.

2.7 Model Sensitivity and Unprofiled Attacks

Mainly used for unprofiled attacks, what we call the model sensitivity is the analysis of which part of an input trace is being used at which intensity by the model to make a prediction. In ML-based unprofiled attacks, the correct key guess is distinguished from wrong key guesses using a metric. This metric may be the training or validation accuracy, or even the model sensitivity [34, 54, 60]. The underlying assumption of this method is that under a correct key guess, the model will be able to find the PoI of the traces and learn from that. In the other hand, a wrong key guess will lead to randomized labels, which will hinder the model from learning anything (inc. PoI) from the trace. In the current work, we'll use the model sensitivity [34, 60] computed at each epoch and accumulated. At the end, we plot the absolute value and elect the key guess leading to a sensitivity with the highest peak as being the right key guess.

Formally, the model sensitivity is defined as:

$$S_{input}[s] = \sum_{t=1}^{N_a} \frac{\partial \mathsf{L}_{\boldsymbol{T}_t}}{\partial y_s} \times \boldsymbol{T}_{t,s}$$

for $s \in \{1, \ldots, N_s\}$. The first quantity in the sum is the partial derivative of the loss with regard to the s-th sample variable (y_s) for the t-th trace of the training set ($\boldsymbol{T}_t$). $\boldsymbol{T}_{t,i}$ corresponds to the value of the t-th trace at time sample s.

3 A New Dimension

A neural network's output layer consists of neurons, one per each possible predicted class, outputting so-called *logits*. The higher a value of a logit, the more strongly is the associated class suggested by the NN for the given input sample. To normalize the output of the NN and make it consistently comparable across samples and between the individual logits, the softmax function is typically applied to the output logits of a NN for a single sample. Softmax transforms a vector of k real numbers into a distribution over k classes, i.e., a vector of k real numbers from the interval $[0, 1]$ that sum to 1, yielding pseudo-probabilities for each class given the sample. When the NN outputs are arranged into a matrix (a row holding the logits for a single sample), the softmax function is thus applied row by row. In a manner of speaking, the softmax is computed along the dimension 1 of the matrix (provided we index the dimensions from 0).[4] This section describes our newly proposed method, based on applying the softmax function to the columns of the same matrix instead, computing a distribution over traces for each NN output class.

3.1 Definition

Throughout this paper, we refer to dimension 1 (*dim1*) or dimension 0 (*dim0*) as being the dimension argument of the softmax function applied on the output layer of a deep learning ML

[4] The naming comes directly from the PyTorch library. PyTorch's softmax function has a dimension argument, which can be set to either 0 or 1 for a 2D input.

model, during the training and attacking phases. Formally, the softmax on dimension 0, denoted $softmax_0$, is defined by:

$$softmax_0(\boldsymbol{V}, t, c) = \frac{e^{\boldsymbol{V}_{t,c}}}{\sum_{j=0}^{N_b} e^{\boldsymbol{V}_{j,c}}} \tag{1}$$

where $\boldsymbol{V}$ represents the $N_b \times N_c$ logits matrix, t the trace number, N_b the number of traces in the batch and c the class index. By definition, *dim0* requires at least two input samples per batch to obtain a non-trivial output. The $softmax_0(\boldsymbol{V}, t, c)$ only depends on $\boldsymbol{V}_{i,c}$ for $i < N_b$ and a fixed class index c.

Proposition 1. *During inference, a model using $softmax_0$ will consider each class separately. During training, a model using $softmax_0$ can not increase the score of the same class for every input sample in a given batch between epochs. An increase of the normalized score for a given class input always results in the decrease of the normalized score for the same class in one or more other input samples of the same batch.*

Corollary 1. *During inference, the normalized scores for the different classes output by a model using $softmax_0$ for a given sample will not preserve the ratios between the raw class scores, mitigating the logits' inter-class bias. During training, a model using $softmax_0$ will elect best input representatives for every class, giving increased consideration to rare classes in imbalanced dataset scenarios.*

3.2 Outline

Section 3 will further explore and experimentally verify Proposition 1. Section 3.3 will show that these targeted classes have more impact in the mathematical key derivation itself. Section 3.4 will demonstrate that choosing an optimizer which allows to optimize each weight on its own maximizes the *dim0* performance. Section 3.5 will elaborate on the rationale leading to the conclusion of Proposition 1. At last, we point to general limitations and recommendations for the *dim0* approach.

3.3 Easily Classifiable Traces Have More Impact

Let's see why under *dim0*, the key ranking algorithm, combining each trace's prediction by the ML model to deduce the most probable secret key, will give more importance to traces for which the model has a greater level of certainty.

First, it is important to make the observation that an inference done with *dim0* will assign a probability distribution over the traces (in the batch) for each class. This means, for class 0, each trace will be assigned a probability of being of class 0, these probabilities summing to 1 over all traces. These probabilities could also be thought of as a measure of how well each of these traces "represent" class 0. Consequently, when we sum such probabilities (we'll call it class-scores) over all classes for a given trace, the result will not be equal to 1. For example, a trivial prediction assigning the same class-score for each class will lead to a sum of N_c/N_b for any trace, where N_c is the number of classes and N_b the batch size. As soon as the model starts to train on a particular class, the probability over the traces will diverge from $1/N_b$ for each trace, and consequently traces whose class the model choose to train on will have a higher expected sum of class scores than other traces. As we will see in Section 3.5, the model can chose to target easily classifiable traces (e.g. of label 0, 8 with HW labelling). Hence, we expect such traces to have a higher sum of class-scores. Classes which are difficult to train on will tend to keep their default $1/N_b$ probability assignment over each trace. We conducted an experiment on our AES_nRF dataset to confirm this claim. After a successful training using *dim0* and a batch size of 50, we computed the sum of each trace class-scores for each batch grouped by their true label and computed the mean value. The expected sum of class-scores is hence $N_c/N_b = 0.18$. We experimentally obtained these class-scores means during inference, for traces of label 0 to 8: $[0.2622, 0.2245, 0.2038, 0.1871, 0.1721, 0.1712, 0.1707, 0.1732, 0.1792]$. As we see, traces whose true label value is 0,1,2 or 3 tend to have

a higher class-score. The model had chosen to target on these classes. This means, traces of true label 0,1,2 and 3 will have a bigger impact during the secret key derivation, which blindly sums each of the potential key's class score (where the key is derived from the plaintext and the given class) for each trace-plaintext input.

3.4 Optimizers

In the context of machine learning, an optimizer is a mathematical algorithm used to adjust the parameters of a model in order to minimize the error in its predictions. The optimizer iteratively updates the model's parameters based on the gradients of the loss function. How the updates are performed given the gradient values depends on the type of optimizer. In fact, *dim0* allows a model to handle each class separately. But this effect is taken to its full potential if the optimizer in turn allows for a per-class (or per-weight) tuning.

We now compare five different optimizers and how their behavior adapts when confronted to a *dim0* model.

Nesterov [39] is an optimizer that works like a Stochastic Gradient Descent (SGD) with momentum but additionally regulates the momentum by approximating the values of parameters in future updates. SGD, with momentum itself, computes the gradient of the loss function by considering only one (or a subset) of the input traces to compute the gradient to reduce the computational cost. The momentum adds memory to the learning process, meaning that we add a fraction of the previously computed gradient to the current one.

Adagrad [10] Its learning rate tends to vanish because it is inversely proportional to the sum of all the past squared gradients.

Adadelta [68] It improves upon Adagrad by only considering a subset of the previous squared gradients.

RMSprop It improves upon Adagrad by dividing the learning rate by an exponentially decaying average of squared gradients, which should solve the learning rate vanishing problems.

Adam [25] It improves upon Adagrad and RMSprop by storing an exponentially decaying average of past gradients in addition to the exponentially decaying average of past squared gradients.

Except for Nesterov, these optimizers are adaptive methods, meaning they have, for each batch, an adaptive learning rate for each parameter. As a result, parameters associated with low-frequency features tend to have larger learning rates than parameters associated with high-frequency features. For example, the gradient descent formula of Adam is:

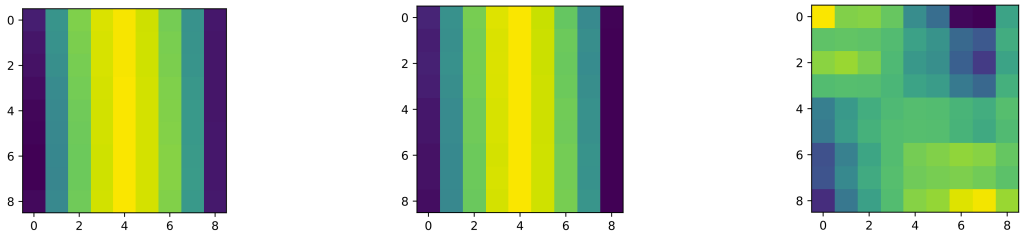
$$\theta_e = \theta_{e-1} - \alpha \frac{\hat{m}_e}{\sqrt{\hat{v}_e} + \epsilon} \tag{2}$$

with $\hat{m}_e = \frac{m_e}{1-\beta_1^e}$, $m_e = \beta_1 m_{e-1} + (1-\beta_1) g_e$, $\hat{v}_e = \frac{v_e}{1-\beta_2^e}$, $v_e = \beta_2 v_{e-1} + (1-\beta_2) g_e^2$, and where e is the current epoch, θ_e the parameters at epoch e, g_e the gradient, α the learning rate, β_1 and β_2 the exponential decay rates for the moment estimates, ϵ a small constant, m_e the first moment estimate, v_e the second moment estimate, g_e the gradient of the loss function.

The division $\frac{\hat{m}_e}{\sqrt{\hat{v}_e}+\epsilon}$ from Equation 2 ensures that each parameter has its own learning rate. Dividing by $\sqrt{\hat{v}_e} + \epsilon$ will make the optimizer adjust the learning rate for each parameter based on the historical gradient information. Parameters with smaller gradients (indicating more stable or low-frequency features) will have their updates scaled up, ensuring they receive sufficient updates (we use the term adaptive if the optmizer has this property). This is because the learning rate is scaled inversely with the square root of the sum of the squares of past gradients. Additionally, ϵ ensures that the learning rate does not vanish entirely for any parameter.

Some adaptive optimizers, such as Adam, additionally have an exponentially decay of the sum (or average) of past gradients (see β_1 and β_2), which helps in preventing the learning rate from becoming too small.

As a conclusion, we make the distinction between adaptive and non-adaptive optimizers (e.g. SGD or Nesterov). Adaptive optimizers might use exponential decay (as in Adam or RMSProp) or not (e.g. Adagrad, Adadelta).



(a) Confusion matrix during training on *dim1*.

(b) Confusion matrix on evaluating with *dim1*.

(c) Confusion matrix on evaluating with *dim0*.

Fig. 1: Confusion matrices for training/evaluating on *dim1*/*dim0*. The training for the middle and the right-hand side figures have been done with *dim1*. Y-axis: true classes. X-axis: predicted classes. Yellow/-light: high probability. Dark/blue: low probability.

Using *dim1* and HW Labelling In this paragraph only, we'll use $softmax_1$ for training and try $softmax_0$ for attacking. Independently from the optimizer we choose to use on an imbalanced dataset, the *dim1* model will be biased towards the more common classes. The learning will stall or be slowed down for rare classes. Figure 1 confirms this claim experimentally. As we see, during training on *dim1*, the model is biased towards the common class (i.e. there is no diagonal to be seen, only a straight yellow line for class 4). During evaluation, this bias persisted. However, using *dim0* at inference, we observe that the model was able to train a bit on the rare classes with *dim1* during training. In fact, *dim0* during inference will remove the inter-class bias and consider each class separately. In this scenario, the model correctly picks the traces closest to class 0.

3.5 Per Class Optimization

A ML model chooses to adapt its weights following the steepest curve of the gradient of the loss function during the optimization. In other words, the model always use the easiest way to optimize itself. In the context of imbalanced datasets, a model with *dim1* will assign higher probabilities for common classes to reduce the loss. With *dim0*, this doesn't happen as for each class the probabilities over the traces in a batch must sum to one. The loss related to common classes is nonetheless high, but it can only be minimized effectively by training on other classes. In turn, if any of the less common class is easier to train on than others, it will focus on that particular class (following the steepest curve of the loss function gradient). In the SCA context, it is known that rare classes are easier to distinguish than common classes (as mentioned in Section 2.4). The model will hence focus on these rare classes to minimize the overall loss. What is more, in the SCA context, rare classes happen to be the most informative ones: a trace of class 0 can only be linked to *one* secret key. Traces of class 4 lead to 70 potential secret keys (refer to Section 2.4). Therefore, using *dim0* allows a model to train to recognize rare classes more successfully, which in turn significantly improve the narrowing-down of the candidate key pool. We experimentally tested this claim to see if our model indeed prioritizes the rare and easily classifiable classes over the common classes. To that end, consider Figure 2. In this Figure, the y-axis represents the mean value of each logit during training. The x-axis is the epoch number. Each class (logit) is represented by a different line. The model used is the CNN_exp with AES_nRF dataset. The mean value of each logit has been calculated using the first batch (i.e., hundred traces) at each epoch. We averaged ten runs, plotting the mean and a 90% confidence interval.

In line with the expected behavior of a model trained with an imbalanced dataset, based on Figure 2, we observe that a *dim1* models' logits are ranked in order of occurrence of the underlying class. This could be confirmed for all optimizers (not shown). The most common classes lie above the others, and the rare classes are in the bottom part. Though the figure only shows Adam on *dim1*, we noticed a slight difference in behavior for the Adam and RMSprop optimizers as compared to the others, in that one of the rare classes (actually class 0) keeps dropping. This aligns with what was observed in Figure 1c: class 0 yielded a better prediction than class 8. Apart from that,

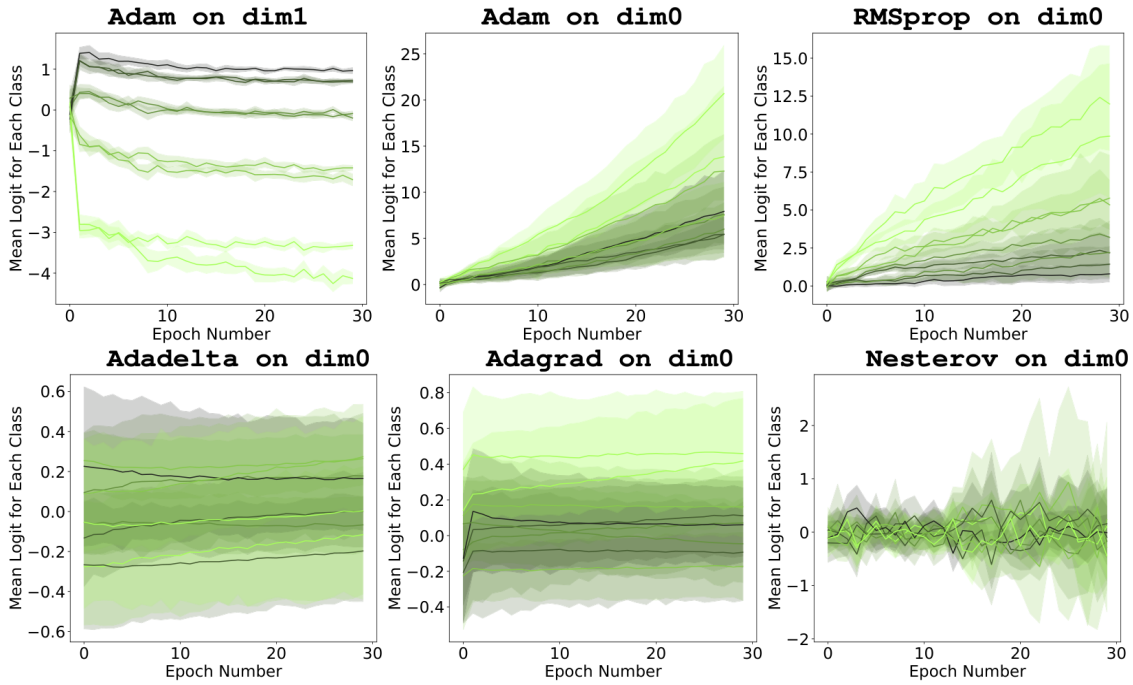


Fig. 2: Mean logit value for each class. Since we used the HW labelling, light green lines stand for rare classes. The darker the line, the more common is the underlying class (label 4 is the dark line, label 0 and 8 are the two lightest). We consider the logits of the first batch of each epoch, obtained with CNN_exp and our own AES_nRF dataset.

the logits do not seem to change much as the epoch increases: this confirms the expected behavior, the model seems to stall.

From Figure 2, we observe that with a *dim0* model, the logit's behavior strongly depends on the optimizer:

Adagrad and Adadelta They stabilize after a few epochs and change especially little afterward. But they are more or less located in the same amplitudes (the confidence intervals are large and interfere with each other). The model does not target a specific logit more than others across runs.

Adam and RMSprop Logits related to rare classes increase faster than the other. This crucial observation confirms our claim that a per-class optimization is possible under *dim0*.

Nesterov Using *dim0* does not yield interesting results. This non-adaptive optimizer was not able to take advantage of the *dim0* properties.

Mathematical Insights The generic gradient descent algorithm, based on Equation (2), stipulates that the weight are adapted at each epoch as a function of the gradient of the loss function, a possible regularization, and the learning rate (which is set by the optimizer). The observation that logits related to rare classes increase faster than the others can be caused by either of these three ingredients. However, in our case no regularization is applied. Moreover, the gradient of the loss function is expected to be higher for common classes and not rare classes. This is a consequence of the $\mathcal{L}_{nll}$ (see Section 2.5): as each sample in the input dataset contributes for the same, the gradient for the common class will be higher.

As a conclusion, only the learning rate can be the root cause of privileging a rare class during training over a common class. Here, having adaptive learning rates in the optimizer, such as in Adam, RMSprop, Adagrad and Adadelta, seems to be necessary for the learning rate of rare classes.

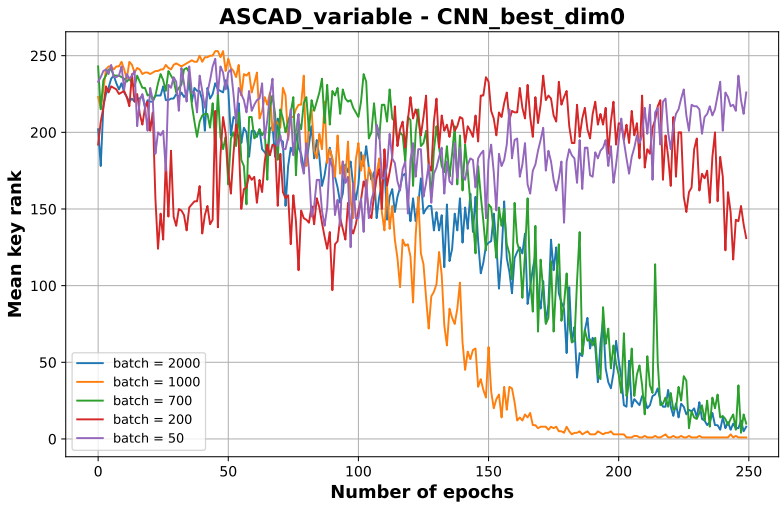


Fig. 3: Batch size influence on the dim 0 model.

3.6 Considerations

Choosing the Right Batch Size A new hyperparameter has to be taken into account when doing ML-based SCA with *dim0*: the batch size. In fact, the new softmax Equation (1) on *dim0* depends on the batch size.

If the batch size is too small, some classes may not have a trace related to it, or would be insufficiently represented in certain batches and the model will mostly have to train on unrelated labels. This is because the model looks at each class separately. As a consequence, the model will overfit on the training dataset.

Using a reasonable batch sizes (at least 50) is recommend to mitigate the effects mentioned above. Section 6 proposes some countermeasures in case the number of attack traces is too small to form a complete batch.

Based on the previous observations, we also verified the influence of the batch size on performance in *dim0* on a public dataset using 110'000 traces from ASCAD_variable with ID leakage model for profiling and 10'000 for attacking. The CNN_best model was employed for these experiments, trained for 250 epochs with the RMSprop optimizer at a learning rate of 10^{-5}. We tested the following batch sizes: $[50, 200, 700, 1000, 2000]$. Figure 3 indicates that the model performed best with a batch size of 1000. The next best performances were achieved with batch sizes of 2000 and 700, respectively. With a batch size of 1000, the model achieved a GE of 1 within the 250 epochs. No other batch size reached this level of performance within the same number of epochs.

However, we must point out that in this case *dim0* did not surpass the *dim1* state-of-the-art, which can successfully break ASCAD variable with only 1000 traces in 37 out of 60 cases [11]. This suggests that changing the batch size might not be the only variable to re-optimize when switching the dimension of the softmax function, and emphasizes the need to clarify dim0's needs and strengths in future research.

Similarities with Bayesian MAP Approach *Dim0* may be reminiscent of the Bayesian attack, as both compute overall class-scores by combining the class-scores from all leakage samples for every class. In the Bayesian attack, one does this using the Bayesian trick, necessitating to treat the distribution of samples (leakages) as statistically independent; the extended version of the paper by Standaert et al. [57, 58] and its Theorem 1 for example assumes independence of leakages. This assumption is, however, not verified in practice as they all depend on the same hardware and software implementation of the cipher.

Using *dim0*, there is an additional step where class-sample-scores of the batch are normalized. The normalization is computed one class after another, using the non-normalized scores of all

samples for the given class (i.e., logits). The normalized scores are then aggregated similarly as in the Bayesian attacks. No leakage-independence assumptions are made. Moreover, the class-wise normalization compensates the bias between the classes, which the a posteriori maximization alone does not involve. This treatment also has benefits that are specific to the ML-based approach. With conventional (i.e., *dim1*) training, the distributions of the output logits are forced to *"stay together"* (means not too far apart, with similar variances).

Discussion As we saw, an adaptive optimizer will set the learning rate differently for each parameter; proportionally to the inverse of their gradient's variance. Using *dim0* lets an adaptive optimizer take advantage of the fact that the logits' individual values can increase with more liberty than with *dim1*. What is suggested by the experimental data from Adadelta and Adam plots from Figure 2, is that an adaptive learning rate alone is not sufficient to achieve the logit's individual imbalance towards rare classes. Indeed, we additionally need them to follow an exponentially decay of the sum (or average) of past gradients (as for Adam or RMSprop). This exponential decay allows the learning rate not to vanish (or become small) on logits which have had a great change on previous steps. As a consequence, the training will not stall.

Another point we observed when using *dim0* on an imbalanced dataset is that the model may choose one particular rare class to target (in our examples it is class 0). The model targets the easiest to train classes (arguably, as pointed in Figure 5 of [14], class 0 or 8 are expected to be easier to distinguish than the other classes), and they happen to be the most informative ones. This fact is expected to also help the model perform better on balanced, ID-labeled datasets, where again some of the ID classes can be easier to distinguish than others. Last, the key-ranking algorithm itself will give more importance to easily classifiable traces, a paradigm shift from all former studies on the SCA topic.

4 Experiments

All the test cases for *dim0* presented in this Section will target a dataset using the Hamming Weight (HW) or Hamming Distance (HD) labelling function. To demonstrate its significance, *dim0* needs to outperform the current state-of-the-art class balancing technique, as well as models that use the easier Identity Labelling. According to work by Picek et al. [46], the best balancer in the SCA context is SMOTE [8].

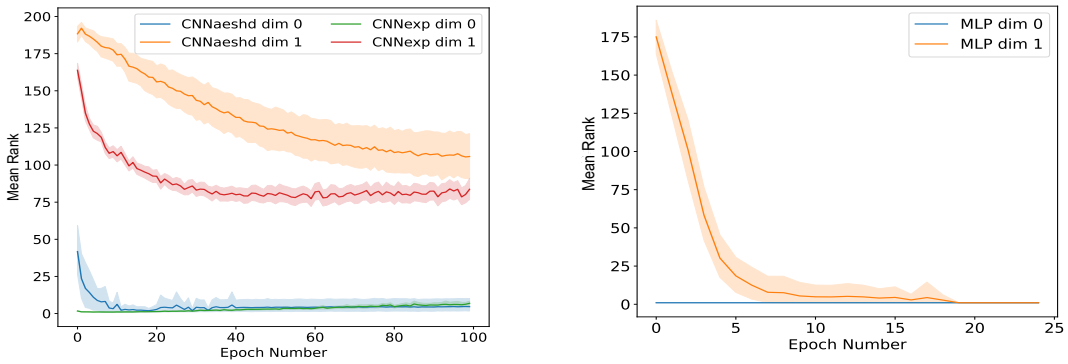
We will hence compare *dim0* with a HW labelling against SMOTE and, for research purpose, also against state-of-the-art models with the identity labelling. At last, will show that *dim0* can also be used effectively on unprofiled attacks.

A detailed overview of the datasets and the ML models used for the experiments can be found in Appendix A. The experiments on the FPGA-based AES_HD [46] and ASIC-based DPAContestv4.2 [3] datasets showcase how our *dim0* model surpassed the same model with *dim1* on profiled and unprofiled attacks. To compete against the state-of-the-art, we target the ASIC-based ASCAD dataset and its variants [2], as multiple publications compared the HW performance on ASCAD.

4.1 Profiling Attacks and the Hamming Weight Labelling

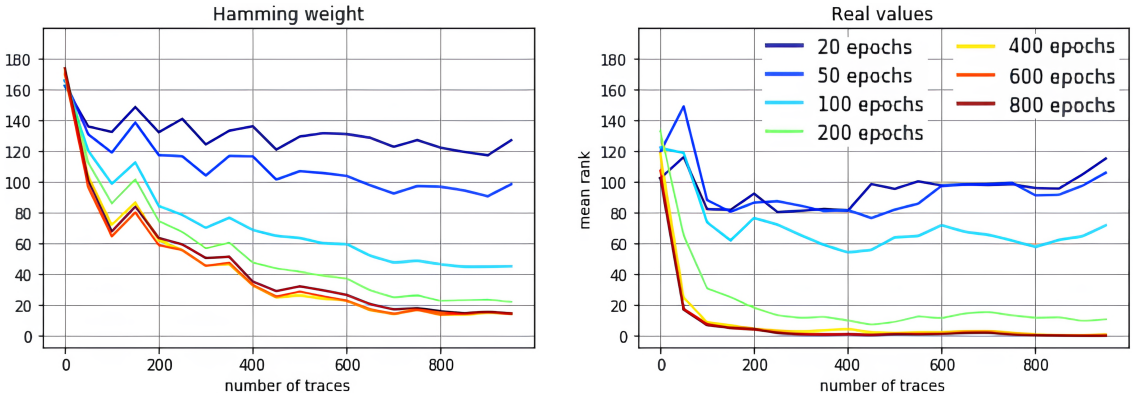
In this section, experiments on the ASCAD datasets were run 10 times and all results reported with a 95% confidence interval.

AES_HD and DPAContestv4.2 datasets. First, we show an attack of the AES_HD dataset ($N_p = 47'500, N_a = 2500$) with the HW leakage function. We used the CNNaeshd and the CNN_exp models. Figure 4a shows that *dim0* models worked much better in comparison with *dim1*, where the attacks are practically unfeasible. Additionally, we attack the DPAContestv4.2 ($N_p = 4500, N_a = 500$) dataset using the MLP_simple architecture. Figure 4b shows that the attack is easier for *dim0* than with *dim1*, as it converged within one epoch.

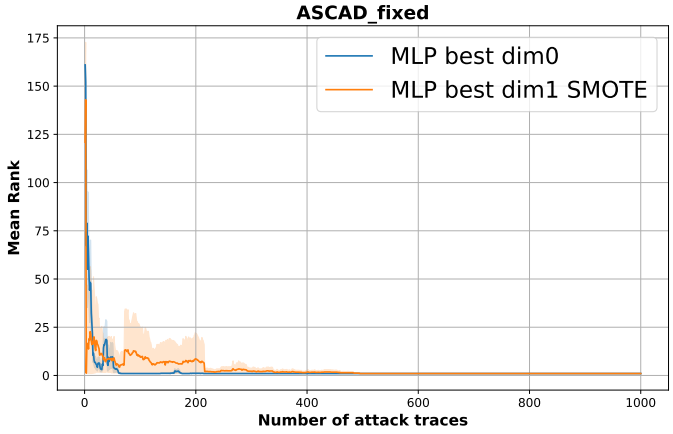


(a) AES_HD dataset. (b) DPAContestv4.2 dataset.

Fig. 4: Attacking various HW-labelled datasets 50 times, comparing *dim0* and *dim1*



(a) ASCAD_fixed, MLP_best results [2] with HW (left) and ID (right)

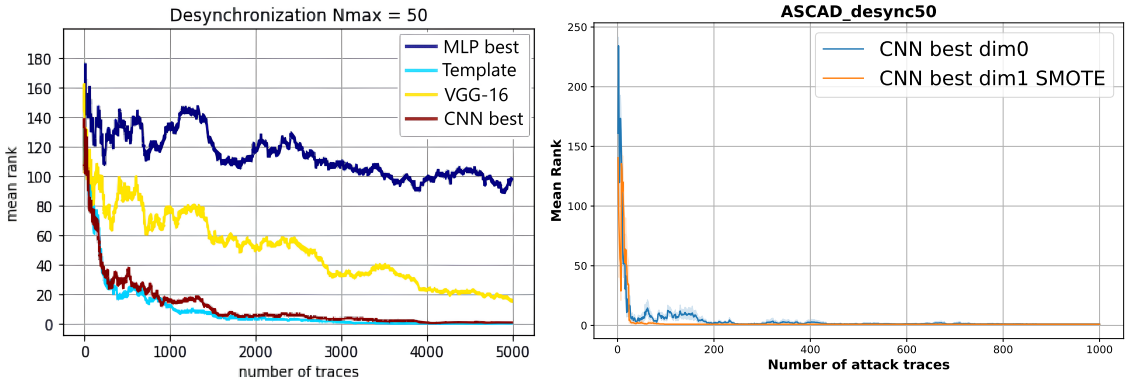


(b) ASCAD_fixed, MLP_best results *dim0* and *dim1* with SMOTE with HW labelling

Fig. 5: Attacking ASCAD_fixed with the MLP_best model and $N_b = 100$

ASCAD_fixed with MLP_best. We attack the ASCAD_fixed dataset with the MLP_best architecture, as suggested in the work by Benadjila et al. [2]. We kept the proposed experiment setup, except for the softmax activation. As such, 200 epochs will be used. Figure 5a shows the results reported in the original paper for HW labeling (left) and ID (right). The plot reports how

the number of epochs affects the attack using the MLP_best architecture. In Figure 5b, we report the performance of the MLP_best *dim0* vs the MLP_best *dim1* with SMOTE using the HW leakage model. In this scenario, the *dim0* approach with a HW labelling beats all its concurrents. We note that by flipping the dimension, the model is now able to converge within 200 traces, while the original attack (on the left hand-side of Figure 5a) could not converge after 1000 traces. For SMOTE, more than 450 traces are required. And for the conventional technique, *dim1* with ID, at least 300.

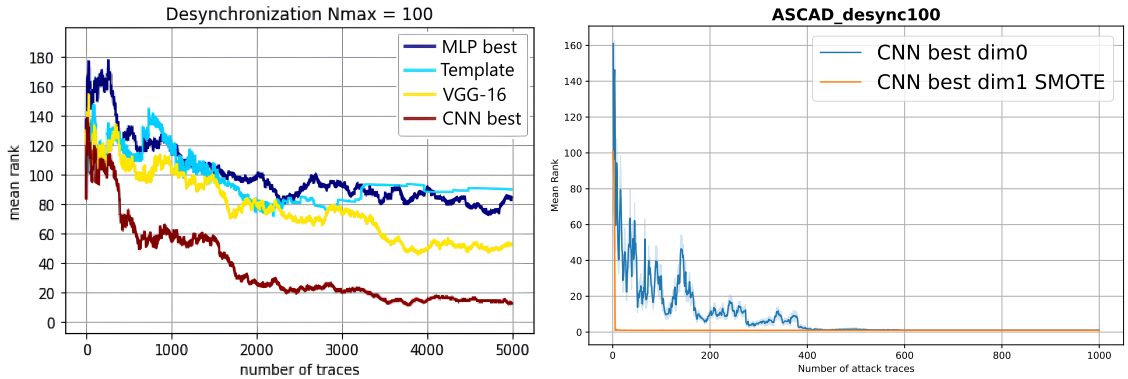


(a) ASCAD_desync50 results with ID labelling [47]

(b) ASCAD_desync50, CNN_best results with HW labelling

Fig. 6: Attacking ASCAD_desync50 with the CNN_best model and $N_b = 200$

ASCAD_desync50 with CNN_best. CNN_best is a ML model proposed in the work by Benadjila et al. [2] to attack the ASCAD_desync50 dataset. Figure 6a shows the best configurations' results on the ASCAD_desync50 dataset with the ID labeling on three ML models and a classical Template attack. In Figure 6b, we observe that CNN_best *dim0* is better than any of the ASCAD_desync50 configurations since it needs less than 800 traces to reach $GE = 1$, against more than 4000 traces for *dim1*. However, *dim0* seems slightly less powerful than *dim1* with SMOTE. For the latter configuration, a successful attack is reached by using approximately 100 traces.



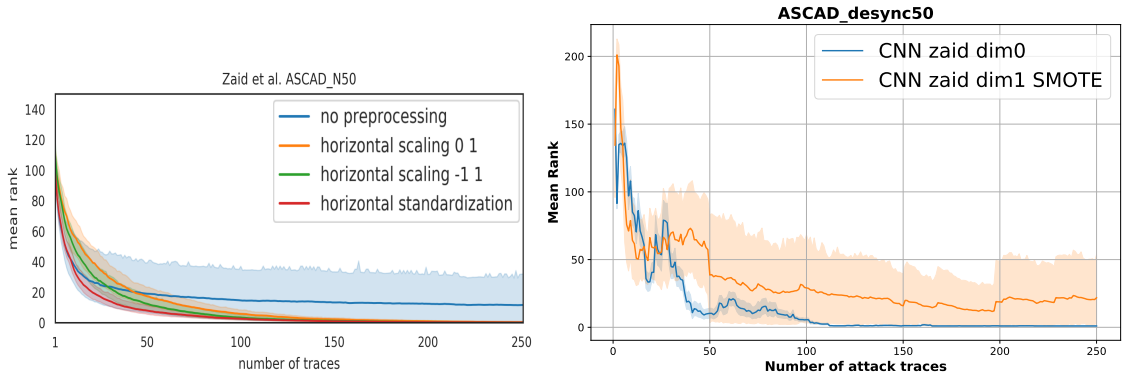
(a) ASCAD_desync100, results with ID labelling [2]

(b) ASCAD_desync100, CNN_best on *dim0* and *dim1* with SMOTE, HW labelling

Fig. 7: Attacking ASCAD_desync100 with the CNN_best model and $N_b = 200$

ASCAD_desync100 with CNN_best. Figure 7a illustrates the best configurations' results on the ASCAD_desync100 dataset using ID labeling for three ML models and one classical template attack. This plot was taken from the study by Benadjila et al. [2]. Figure 7b demonstrates that CNN_best *dim0* outperforms other ASCAD_desync100 configurations, requiring fewer than 550 traces to achieve $GE = 1$. In contrast, none of the configurations presented in the Benadjila et al. [2] is successful. Nonetheless, *dim0* is still marginally less effective than *dim1* with SMOTE—which required only a few traces (less than 30) to reach a key mean rank of 1.

In the following attacks, we report results from the work by Wouters et al. [64] for the CNN_zaid and No_conv architectures. We did not change the proposed parameters except the last layer (from *dim1* to *dim0*). We used the horizontal standardization during preprocessing, as it offers the best results for *dim0* and *dim1* with SMOTE.



(a) ASCAD_desync50, CNN_zaid results with ID labelling [64]

(b) ASCAD_desync50, CNN_zaid results with HW labelling

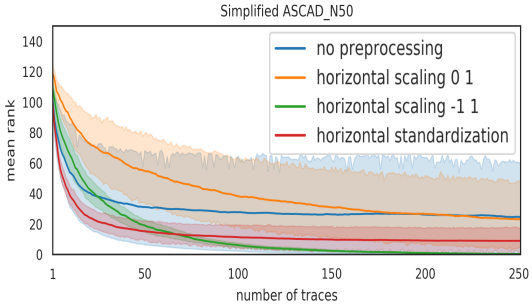
Fig. 8: Attacking ASCAD_desync50 with the CNN_zaid model and $N_b = 50$

ASCAD_desync50 with CNN_zaid. In Figure 8 we plot the results for the ASCAD_desync50 using the CNN_zaid architecture—comparing the original results of the ID leakage model with our runs in HW labelling. Figure 8b shows that *dim0* outperforms *dim1* with the SMOTE since it reached the $GE = 1$ with fewer than 175 traces. The HW-*dim0* result is not that far from with the ID-*dim1* result of the original paper, where approximately 140 traces are needed to reach a successful attack.

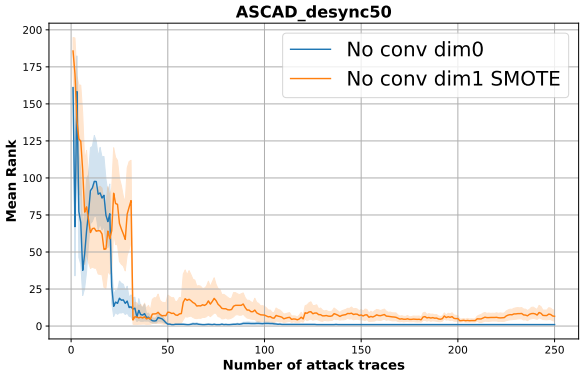
ASCAD_desync50 with No_conv. Figure 9 compares the state-of-the-art results with ID labelling against the HW labelling and *dim0* or *dim1* with SMOTE. The No_conv architecture is used to target the ASCAD_desync50 dataset. Here again, the *dim0* technique beats all its opponents, requiring approximately 110 traces for a successful attack, while in the case of SMOTE, the performance fluctuates and does not reach a $GE = 1$. Moreover, the HW-*dim0* attack is better than any ID labeling attack by a 70 trace margin.

ASCAD_desync100 with CNN_zaid. Figure 10 compares the performance of the CNN_zaid architecture on the ASCAD_desync100 dataset and our target scenarios. We see that the HW-*dim0* results are close to the Identity labelling results with *dim1*, while *dim0* definitely beats the HW-*dim1* with SMOTE which did not converge.

ASCAD_desync100 with No_conv. Figure 11 shows that for the No_conv models against the ASCAD_desync100, the ID labelling seems to perform better than any of the HW labelling attacks. We observe that in the original paper, $GE = 1$ is reached after approximately 170 traces, while in Figure 11b, none of the methods could achieve $GE = 1$ with the 250 traces.

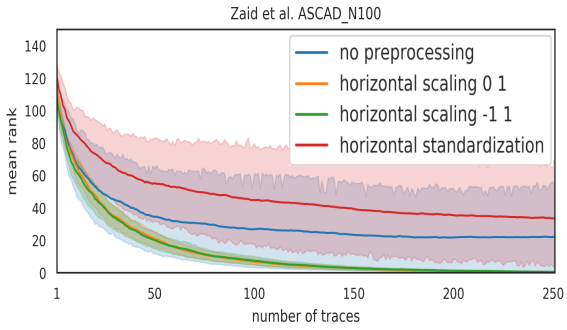


(a) ASCAD_desync50, No_conv results with ID labelling [64]

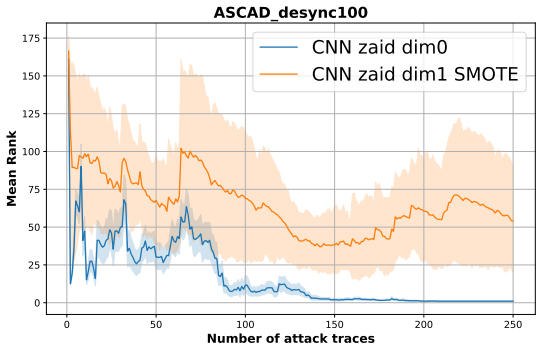


(b) ASCAD_desync50, No_conv on *dim0* and *dim1* with SMOTE, HW labelling

Fig. 9: Attacking ASCAD_desync50 with the No_conv model and $N_b = 50$



(a) ASCAD_desync100, CNN_zaid results with ID labelling [64]



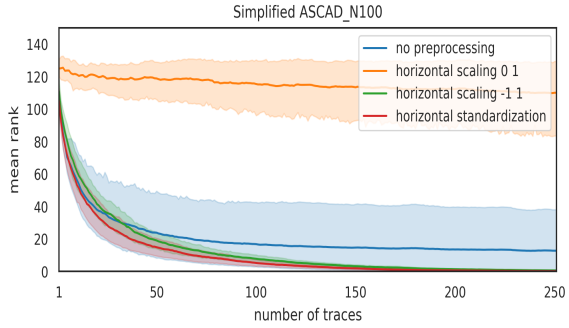
(b) ASCAD_desync100, CNN_zaid on *dim0* and *dim1* with SMOTE, HW labelling

Fig. 10: Attacking ASCAD_desync100 with the CNN_zaid model, $N_b = 50$

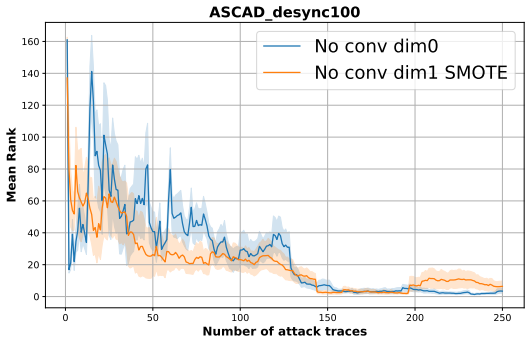
4.2 Conclusion

An interesting pattern is emerging from the experiments, regarding *dim0*'s apparent performance's instability while the number of attack traces increases. This is a consequence of *dim0*'s nature: it seems to work best when the number of attack traces in a batch is close to a multiple of the training batch size. In fact the model was trained with a static batch size, while the plots shown in this section iteratively increase the number of attack traces fed to the model. Most notably, on Figure 6b, we said that the model is successful after 800 traces (4 batches of 200 traces). However, after one batch, the model already achieved a key rank close to zero. The same at 600 traces (3 batches). Evaluating the traces for a non-batch size multiple (or close to it) seems to be the reason for the discrepancy between 600 and 800 traces as well.

Overall, we obtained good performance for profiling attacks by using *dim0* with HW labeling compared to the traditional ID leakage model in some cases. Moreover, we observed that the lightweight *dim0* method is more stable than the state-of-the-art computationaly expensive SMOTE solution for the imbalacing problems, especially for the desynchronized datasets, where we have smaller deviations from the mean over ten runs. The new method could however benefit from further specific model optimizations and is quite sensitive to the batch size. For a complete performance analysis overview, we refer the reader to Appendix B.



(a) ASCAD_desync100, No_conv results with ID labelling [64]



(b) ASCAD_desync100, No_conv on *dim0* and *dim1* with SMOTE, HW labelling

Fig. 11: Attacking ASCAD_desync100 with the No_conv model and $N_b = 50$

4.3 Unprofiled Attacks

We could successfully launch an unprofiled attack on the AES_HD dataset using the Hamming Distance labelling, 44'000 traces for training and 6000 for validation (used to compute the model sensitivity), using the CNNexp model with a classic regularization of 0.0008 applied on the Fully-Connected layer (weights & bias), and $N_b = 100$. As detailed in Section 2.7, we reuse Timon's strategy [60] exactly with *dim1* and only flip the softmax dimension for *dim0*. We launched ten attacks using *dim0* and *dim1* respectively. The results are shown in Table 1. The total duration of each attack was approximately 12 hours for the 15 epochs using an i7-13700k Intel processor and no GPU.

Table 1: Unprofiled performance of the CNNexp model on the AES_HD dataset using a HD labelling with 15 epochs and over 10 attacks for each dimension.

Attack Number	1	2	3	4	5	6	7	8	9	10
dim0 Key Rank	1	1	2	10	6	2	1	1	14	1
dim1 Key Rank	126	2	80	3	18	168	1	66	5	58

5 A New Toolbox

For our research on the dimension 0 topic, we built a comprehensive python package for SCA. Its main novelty lies in its ability to conduct non-profiled ML attacks, while combining state-of-the art breakthroughs regarding ML models and training methods. The models and most of the techniques can be applied for unprofiled as well as profiled attacks. We release it as open-source for the benefit of the research community.

5.1 Existing Tools

Table 2 lists existing side-channel attack tools along with the features they provide. In this regard, we checked for trace alignment (or synchronization) techniques, ML-based (aka autoencoders) or other data pre-processing techniques. Then, we check whether the tool supports ML-based profiled attacks or classical approaches (such as Template Attacks [48] or Stochastic Attacks [51]). We also compare the unprofiled attacks, using ML or classical techniques (such as Differential Power Analysis (DPA) [28], Correlation Power Analysis (CPA) [4] or Mutual Information Analysis [17]).

Not cited here are some other python-based SCA tools provided as a simple python script [67] or notebook [64]. These work well to reproduce a result, but do not easily facilitate further research and increments. For example, changing the labelling function would require a full re-implementation of the data generation part.

As we can see from Table 2, no other open-source library supports ML-based unprofiled Side-Channel Attacks. What is more, hidden in the (Un)Profiled ML section, the MLSCAlib hides a plethora of state-of-the-art ML techniques for SCA, which are not supported or combined in the other packages.

	Data Processing			**Profiled**		**Unprofiled**	
	Syncing	ML	Other	ML	Classic	ML	Classic
MLSCAlib		×	×	×	(×)	×	
ChipWh. [40]	×		×		×		×
Lascar [30]	×		×	×	×		×
Scared [13]	×		×		×		×
Daredevil [52]							×
Jlsca [50]	×		×				×
AisyLab [42]		×	×	×	×		×
Pysca [22]							×

Table 2: Comparing Open-Source SCA Toolboxes against our MLSCAlib.

5.2 New Toolbox Features

We combined numerous previous research on the SCA topic and were able to compile them in a single python-based state of the art machine-learning package for side-channel attacks that we named the MLSCAlib. The package yields the following advantages compared to existing open-source SCA code:

Modularity. Our object-oriented programming approach lets users easily extend the package. E.g., a new cipher can be added by implementing a new `LeakageModel` class.

Flatness. Unlike in other packages, no callbacks or other performance optimizations are used, as PyTorch already handles it well. As a consequence, a user can easily understand the different steps of the learning process and the structure of the code.

Documentation. A complete documentation is available for each public method in the package. Most of the private methods have a brief explanation as well.

Unprofiled attacks. The package allows for unprofiled attacks to be executed as proposed by Benjamin Timon [60], i.e. using the sensitivity value or the accuracy as a reference distinguisher between key guesses.

Data processing. The data pre-processing feature includes adding Gaussian noise to the training dataset [24], using autoencoders to remove noise from traces [66], balancing the data with SMOTE or other balancing techniques [9, 8, 46], ability to remove the mean of each trace [33], building a trace matrix from the input traces as did Messerges [36], to simple tools like Moving Average, decimation, Pearson or PCA.

Visualisation. Results are presented in an organized manner with meaningful graphs. Profiled attack results can be automatically merged in plots showing either the achieved Key Rank or the minimal number of attack traces needed to succeed against the epoch number. Also, by default the results include a view of the model sensitivity for each trace sample. This allows to visually see which part of the trace was used for training.

PyTorch Models The tool contains thirty PyTorch architectures following state-of-the-art research in the domain, ranging from Multi-Layer Perceptron [61], Convolutional Neural Networks (CNN) [67, 64, 60, 31], Multi-scale CNN [62, 63], ResNet [5, 6] to Visual Geometry Groups (VGG) [24]. We also propose two new models, called the MLPexp and the MeshNN respectively.

Side-Channel Techniques As mentioned earlier, the MLSCAlib implements some of the AI SCA attacks presented in literature in the last years, ready to be applied, tested, and improved by the community. These may apply to profiled attacks or Unprofiled Attacks [59, 60, 29, 12] as well. Among these techniques we find the Lottery Ticket Hypothesis [16, 43], the Learning Rate Scheduling [53, 67], the use of Domain Knowledge [18, 20], and as usual in ML we are able to chose from different Loss Functions and Optimizers. The last technique included is related to the dimension 0 trick explained in this work.

5.3 Comparing ML Features

As shown in Table 2, there exists two other toolboxes that can be used to perform ML-based SCA. We compared their specific features against our MLSCAlib in Table 3.

5.4 Workflow

The package contains four types of classes: `Attack`, `LeakageModel`, `DataManager` and PyTorch NN model classes. The loading and preprocessing of the traces is done by the `DataManager` class. Its two main methods, `prepare_un_profiled_data` & `prepare_profiled_data`, will compute the labels related to the traces, plaintext and key (and first loads the data from the file if needed) given a `LeakageModel` subclass. The `LeakageModel` class defines the relation between plaintext (or ciphertext), key and the label.

The `Attack` class contains the core-functionalities of the package. One may perform three kinds of attacks: a Profiled, UnProfiled or Classic attack. Classic attack contains a (slow) SVM and a (fast) k-NN approach.

The detailed format specification of the datasets is available in the documentation of the package.

Using the Package A user can choose to use the toolbox via a python script or the command line directly.

Using a Python script One needs to follow the steps listed in Figure 12. After having instantiated an `AESLeakageModel`, a `DataManager` and an `Attack` class, the attack is launched by calling the `Attack`'s `attack_byte` method on a given byte.

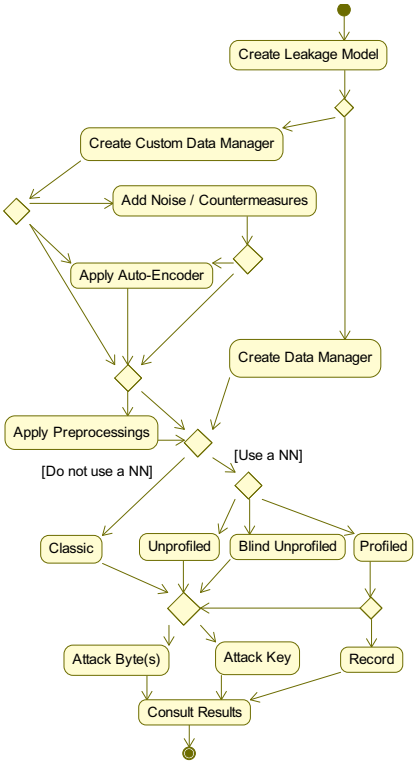


Fig. 12: The activity diagram of the MLSCAlib.

Using the command line. One needs to make calls to the `main.py` parser file which will instantiate the classes automatically given the provided command. For example, to launch using the attack dataset `/data/Set1.h5` on the first SBox location, targeting byte 3 and using 50 epochs, one runs:

```
python main.py -e 50 -b 3 --pd /data/ --fa Set1.h5 --loc SBox
```

The `--help` argument displays the available commands. The parameters that are left unspecified will be set to the classes' default values.

	MLSCAlib	**Lascar**	**AisyLab**
Documentation of Every Function	×		
Intuitive & Object Oriented	×	×	
Lottery Ticket Hypothesis [45]	×		
Domain Knowledge [20]	×		
Adding Noise [24]	×		×
ML-based Unprofiled Attacks [60]	×		
Dimension 0	×		
Imbalance Resolution [46]	×		
SCA Metrics	×	×	×
Gradient Visualization [34]	×		×
Data Augmentation	×		×
Grid Search			×
Random Search			×
Early Stopping / Fast GE [44]	×		×
Ensemble [41]			×
Profiling Analyzer			×
Custom Metrics			×
Custom Callbacks			×
Custom Loss Functions	×		×
Confusion Matrix	×		×
Easy Neural Network	×	×	×
Data Augmentation	×		×
GUI - plots, tables	×		×
Fully reproducible	×	×	×

Table 3: Comparing the Lascar [30] and AisyLab [42] ML-based SCA Toolboxes against our MLSCAlib. Not included here are features related to NN architectures themselves.

5.5 Attack Types

Leakage Models Currently, only AES-128 is supported. To attack a new scheme, one should create a `LeakageModel` instance and overwrite its abstract classes. The `LeakageModel` base class defines seven different labelling methods:

1. `ID`: Identity map, resulting in 256 balanced classes.
2. `HW`: Hamming weight, making nine imbalanced classes.
3. `HD`: Hamming distance, making nine imbalanced classes. Implicitly computes the hamming distance between the output of the target function and its input.
4. `HW2` [21]: Binary Hamming weight. Returns one if the Hamming weight is ≥ 4 and zero otherwise. Hence, it has two imbalanced classes.
5. `LSB`: The Least Significant Bit, resulting in two balanced classes.
6. `LSBS`: The two Least Significant Bits, resulting in four balanced classes.
7. `MSB`: The Most Significant Bit, resulting in two balanced classes.

The `AESLeakageModel` defines four different leakage locations:

1. `SBox`: the output of the S-Box function.
2. `key` [18]: the value of the key directly (no target function). May only be relevant if we use domain knowledge neurons with it.
3. `LastSBox`: the output of the last S-Box function. (Only available if the ciphertext is present in the dataset).
4. `AddRoundKey`: the output of the first AddRoundKey function.

The `LeakageModel` classes should overwrite the `get_snr_poi` function. Its aim is to compute the highest SNR peak location. Note that if `get_snr_poi` is called in presence of countermeasures, it will not identify the PoI peak precisely.

Profiled Attack This class performs a profiled attack. It moreover contains a `record_attack` function which allows to compare the results of different models on a single graph. Refer to its docstring for an in-depth user API. The graphs' legend are cross-graph order-consistent, meaning that two graphs containing the performance of the same models will have the legend ordered the same way. This function can be called by multiple terminals at the same time. Each result is stored in a `.npy` file and hence allows to continue/halt the plotting at any time. It also allows to re-plot the graphs without computing the results again. The graph shows the mean of the performance and a 95% confidence interval around it. It can produce four different kinds of graphs, each graph having its own y and x-axis.

Unprofiled Attack The unprofiled attacks are exclusively based on the sensitivity analysis as described in Section 2.7. However, it may be possible to disclose the correct key by looking at the training/validation accuracy between each guess (as obtained in the resulting PDF). In case you are conducting a real attack, you need to use the `BlindUnProfiled` class. This class contains two methods different from its parent class. Refer to the docstring of those functions to get the details of their functionalities.

Blind Attacks When no attack key is available, we call this situation a blind attack. In essence, blind attacks work by assigning a certainty threshold on each byte. E.g. here we consider the ratio between the highest sensitivity peak and the second one. For profiled attacks, no satisfying metric has been found yet (one could for example use the most probable key byte probability as a threshold directly or consider also the sensitivity, or compare the profiling phases themselves). The goal of this ranking is to rank the key guesses from best to worst - i.e. estimate the resulting GE of each key byte. At the end, the key guesses are combined in a 16-fold for loop. The for loop is currently optimized to consider the case where very few bytes have a large GE. In any case, if no plaintext/ciphertext pair is available, it won't be possible to know when the correct key was reached, the attack will only return a key ranking for each key byte.

5.6 Presenting Two New Models

MLPexp The first neural network model we invented is called MLPexp. It was inspired from classical template attacks, in which the attacker first discovers the PoIs and then launch an attack using only those PoIs. The MLPexp model has the structure of a classical MLP, except for the first layer, where each neuron only "sees" a portion of the trace. Each input neuron is connected to a different subset of the traces. During learning, the MLPexp model will automatically disable the input neurons having the smallest weights (hopefully discarding the useless parts of the traces). An advantage of this method is that the model will less overfit over the epochs, as the useless samples won't be accessible after some epochs. One has to be careful when choosing the different hyperparameters. If you disable the input neurons too fast, it might discard PoIs. This input layer could also be used in other models as well.

MeshNN The other neural network invented is called a MeshNN. It was built with the intention to prove that there is additional information present in the logits, as a dimension 1 trained model shows better results when tested on dimension 0 than on dimension 1. The main idea was hence to combine the logits we get from a simple model applied on every trace inside each batch and give those logits as input to a Fully-Connected layer, with the connections inspired from the softmax function. Unfortunately, it turns out to be difficult to tune and did not outperform classical approaches. We still mention it here as it may be useful in other settings. A limitation of this model is that, in the current format, it only supports same-sized input batches. Appendix C contains further details on the actual architecture used in a Mesh Neural Network.

6 Conclusion and Future Work

This paper presents a new technique for performing DL-based SCA, embedded in a new open-source toolbox for performing Side-Channel Attacks. This technique consists of transposing the logits matrix before applying the softmax function. It reduces the effect of imbalanced datasets, increases the convergence speed of some models (especially for adaptive optimizers) in the profiling phase, and can strongly improve the performance of unprofiled attacks as well. We compare our results with the state-of-the-art models used in ID labelling, and concluded that the *dim0* with the HW leakage model has the potential to match and beat the state-of-the-art in most scenarios. Additional model tuning seems to be necessary for the *dim0* to thrive and surpass the current results. While this publication focused on applying *dim0* on imbalanced datasets scenarios, Proposition 1 also applies to the Identity Labelling (ID) case. Preliminary experiments on our custom dataset showcased a faster convergence and a lowering of needed attack traces when using the ID labelling and *dim0*.

For future work, it would be interesting to see if an attack is possible with only a single attack trace using *dim0*. As a *dim0* model can only work with multiple traces (with one trace, the model outputs a 100% probability for each class), one possible avenue would be to compare the attack trace with some profiling traces and only consider the output probability of the attack trace.

Acknowledgments This research was co-funded by the European Union's Chips Joint Undertaking (JU) under grant agreement No. 10111228.

A Experimental setup

A.1 Datasets

AES_nRF. is a home-made dataset taken from a software implementation of AES-128 without protection. Sbox is implemented as a look-up table. The target hardware was an nRF52840-DK board. We used a Chipwhisperer to capture the power consumption. The dataset contains 47'500 profiling traces with random keys and random plaintexts, as well as 2500 attack traces of random plaintexts and a constant key. We standardized the profiling and attack traces such that they have 0 mean and unit variance.

DPA contest V4.2. [3] is an AES software implementation with a first-order masking technique [38].[5] Since the mask value is publicly known, we removed the masking and attack the key directly. We will use 4500 profiling traces and 500 attack traces. Each trace is 4000 samples long. We target the first S-Box output on the first byte.

AES_HD. is a hardware implementation of AES with no protection introduced by Stjepan Picek et al. [46].[6] We used only a subset of 50'000 traces among the 100'000 ones available, with 1250 samples each. We target the last S-Box of the AES on byte 0. We either used the Hamming Weight labelling function on the last S-Box or the Hamming Distance labelling function, which yields the Hamming Weight value of the input of the last S-Box XOR its output.

ASCAD_fixed. is a second-order masked implementation of the AES128, introduced by Benadjila et al. [2]. The measurements collected represent the power consumption of the first AES encryption round. Each measurement contains 700 samples, which denotes the usage of the third key byte (first masked byte) in the Sbox function. This dataset contains 50'000 traces for profiling and 10'000 for the attacking phase. All of them were created using the same key.

ASCAD_desync50, ASCAD_desync100. are the traces from the **ASCAD_fixed**, with the same dimensionality (same number of traces, number of samples, and the same dataset split), but randomly desynchronized by artificially applying jittering on a range of 50 and 100 samples, respectively. Hence, those datasets introduced by Benadjila et al. [2] denote traces created by including two side-channel counter-measurements (masking and jitter).

ASCAD_variable. is the same second-order masked implementation of the AES128 as **ASCAD_fixed**. However, the difference between those two datasets consists of having a variable key for the profiling traces and a fixed key for the attacking ones — a scenario that is more plausible in real life. We have 200'000 traces for profiling and 100'000 for attacking. The traces contain 1400 samples taken during the third Sbox transformation from the first encryption round. In our study, we will attack the third key byte (which is also masked).

A.2 Meta-architectures

Our studies used different meta-architectures from state-of-the-art DL models, which performed well in previous studies during the SCA. We use the same preprocessing techniques presented in their papers and sometimes hypertune some parameters (mainly the batch size). For all the models we use the *Negative-Log Likelihood (NLL)* loss function. The models are:

MLP_simple This model was used in the different studies [60, 29] and consists only of 3 layers. The hyperparameters reported in the previous works [60, 29] were `batch_size` = 1000, `epochs` = 100, `optimizer = Adam`, and `learning_rate` = 10^{-3}. We mainly used them, with some exceptions presented in the experiments separately.

[5] https://cloud.telecom-paris.fr/s/JM2iaRZfwrNKtSp
[6] https://github.com/AESHD/AES_HD_Dataset

CNN_exp This model is taken from the work by Timon et al. [60] with the following default parameters:`batch_size` = 1000, `epochs` = 100, `optimizer = Adam`, and `learning_rate` = 10^{-3}. We'll mention any parameter change explicitly.

MLP_best This architecture was first proposed by Benadjila et al. [2] and performed well against the ASCAD_fixed dataset. We reutilized the parameter values proposed in their study: `batch_size` = 100, `epochs` = 200, `optimizer = RMSprop`, and `learning_rate` = 10^{-5}. The model was used in attacks against ASCAD_fixed, maintaining the same profiling-attacking setup with 50,000 traces for profiling and 10,000 traces for the attacking phase.

CNN_best This architecture was also introduced by Benadjila et al. [2] as an improvement over MLP_best for the ASCAD desynchronized datasets. We reused the parameter values from their paper: `batch_size` = 200, `epochs` = 100, `optimizer = RMSprop`, and `learning_rate` = 10^{-5}. This model was tested against ASCAD_desync50 and ASCAD_desync100, using the same profiling-attacking setup with 50,000 traces for profiling and 10,000 traces for the attacking phase.

CNN_zaid This architecture, initially proposed by Zaid et al. [67], performed well against the ASCAD datasets. We used the parameter values proposed by Wouters et al. [64], based on the paper and code, as they stated these parameters were more stable: `batch_size` = 50, `epochs` = 50, `optimizer = Adam`, and `learning_rate` = 0.005. Additionally, we employed the *One Cycle Policy* learning rate strategy [53]. The models `cnn_zaid50` and `cnn_zaid100` were used to attack ASCAD_desync50 and ASCAD_desync100, respectively, with the same profiling-attacking setup, and we compared our results with those presented by Wouters et al. [64].

No_conv This architecture was proposed by Wouters et al. [64] as an improvement to the models by Zaid et al. [67]. We used the same parameter values proposed by Wouters et al. [64]: `batch_size` = 50, `epochs` = 50, `optimizer = Adam`, and `learning_rate` = 0.005. Since these models are derived from `cnn_zaid`, we also utilized the *One Cycle Policy* learning rate strategy [53]. The models `no_conv50` and `no_conv100` were used to attack ASCAD_desync50 and ASCAD_desync100, respectively, with the same profiling-attacking setup.

CNNaeshd This model was proposed in the work by Zaid et al. [67] to attack the AES_HD dataset. We use in our attacks the proposed parameters:
`batch_size` = 256, `epochs` = 20, `optimizer = Adam`, and `learning_rate` = 10^{-3}.

A.3 Preprocessing

On each of the attack mentioned in this paper, we did a standardization on the profiling and attack traces such that they have 0 mean and unit variance.

A.4 Evaluation Metric

In side-channel attacks, the effectiveness of the attack is evaluated using the average rank of the correct key candidate among all possible keys after analyzing a certain number of traces (attack traces). This metric is called the Guessing Entropy (GE). Lower GE values imply more effective attacks, with GE = 1 indicating the correct key is consistently the highest ranked. In our experiments, we measure the mean average key to compare different attacks, since the same metric is used in previous studies.

B Dimension 0 Performance Overview

Table 4 gives a succinct overview of the *dim0* performance accros all the tested scenarios.

Table 4: Comparison of performance between model configurations from the literature and from this work, targeting different ASCAD [2] datasets, namely ASCAD_fixed ("fixed"), ASCAD_desync50 ("desync50"), ASCAD_desync100 ("desync100") and , ASCAD_variable ("variable"). To have a consistent comparison of several results that fits into a table, we report two metrics (whenever available): the average guessing entropy (GE) value achieved by the model when the number of attack traces (NT) is 200 ("GE at NT=200") and the number of traces required to achieve a GE of 1 ("NT for GE=1"). In this manner, we see both how many traces a model needs to perform optimally and how it perform when a limited number of traces is available. The * values are approximated based on the figures, since in some works, precise numerical values are not reported.

Paper	**Configuration**	**fixed**		**desync50**		**desync100**		**variable**	
		GE at NT=200	**NT for GE=1**	**GE at NT=200**	**NT for GE=1**	**GE at NT=200**	**NT for GE=1**	**GE at NT=200**	**NT for GE=1**
[2]	MLP_best + ID	1	250*	-	-	-	-	-	-
	MLP_best + HW	35*	>1000	-	-	-	-	-	-
	CNN_best + ID	-	-	60*	4000	95*	>5000	-	-
[11]	CNN_best + HW	-	-	-	-	-	-	19*	>1000
[64]	CNN_zaid + ID	-	-	1	140*	1	200*	-	-
	No_conv + ID	-	-	1	180*	1	170*	-	-
Our study *-dim0-*	MLP_best + HW	1	195	-	-	-	-	-	-
	CNN_best + HW	-	-	5	780	10	570	240	>12'000
	CNN_zaid + HW	-	-	1	175	1	185	-	-
	No_conv + HW	-	-	1	110	5	>250	-	-
Our study -SMOTE-	MLP_best + HW	10	450	-	-	-	-	-	-
	CNN_best + HW	-	-	1	100	1	26	200	12'000
	CNN_zaid + HW	-	-	25	>250	60	>250	-	-
	No_conv + HW	-	-	5	>250	7	>250	-	-

C MeshNN Architecture

Figure 13 is showing a graphical representation of this model. On the graph, we see on the left the traces in the batch. At first, each trace passes though the same CNNexp model as usual. The blue dots would be the logits we obtain in a normal setting. Now, a SELU activation is applied on them, and a MLP-like model is combining the logits in the following way: for each logit of each trace, the model has two input neurons. The first one is only connected to one logit (corresponding to the target trace) while the second one is connected to every logit of the target class in the batch. The output of the MLP is going to replace the target logit we had at the beginning. The same MLP is applied on each logit of each trace. There are few parameters that can moreover be chosen by the user:

1. The first model that is used. Here, we used a CNNexp model, but a MLP might also do well.
2. The bottlenecks. Recall that the blue dots represents the logits for each trace. Hence, the information we get from a trace is only contained into one neuron per class. This is called the first bottleneck, and can be increased by having two or more neurons for each trace/class. The second bottleneck is in the red dots. Here, there is only one neuron to capture the target logit per trace and one other to combine every other logit of the same class. In the same manner, one could increase this bottleneck.
3. The number of hidden layers and neurons per layer of the output MLP.

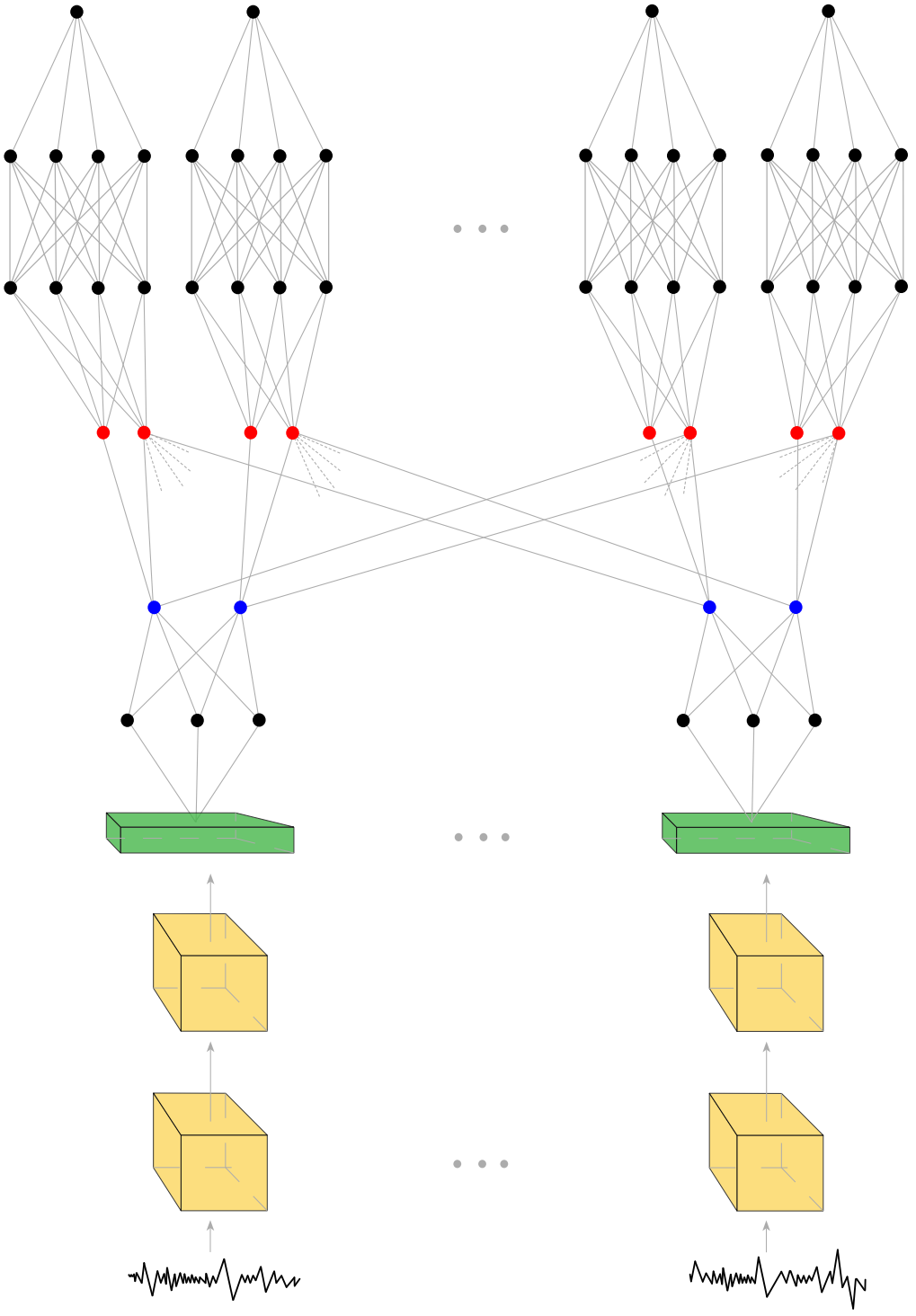


Fig. 13: MeshNN model with a two-class labelling. The yellow boxes are convolutional blocks (convolution, average pooling and batch normalization). The green box is a flattening layer. The blue dots are the neurons in the first bottleneck per class. The red neurons are in the second bottleneck per class. At the end we see four different MLPs. In fact, it is the same MLP applied on four different locations. The same goes for the convolutional blocks.

References

1. Azouaoui, M., Durvaux, F., Poussier, R., Standaert, F.X., Papagiannopoulos, K., Verneuil, V.: On the worst-case side-channel security of ecc point randomization in embedded devices. In: Bhargavan, K., Oswald, E., Prabhakaran, M. (eds.) Progress in Cryptology – INDOCRYPT 2020. pp. 205–227. Springer International Publishing, Cham (2020)
2. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ASCAD database. Journal of Cryptographic Engineering **10**(2), 163–188 (Nov 2019). https://doi.org/10.1007/s13389-019-00220-8, https://doi.org/10.1007/s13389-019-00220-8
3. Bhasin, S., Bruneau, N., Danger, J.L., Guilley, S., Najm, Z.: Analysis and improvements of the dpa contest v4 implementation. In: Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings 4. pp. 201–218. Springer (2014)
4. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: International workshop on cryptographic hardware and embedded systems. pp. 16–29. Springer (2004)
5. Bursztein, E., Picod, J.M.: A Hacker Guide To Deep Learning Based Side Channel Attacks. In: CON, D. (ed.) DEF CON 27 (2019)
6. Bursztein, E., et al.: SCAAML: Side Channel Attacks Assisted with Machine Learning. https://github.com/google/scaaml (2019)
7. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4. pp. 13–28. Springer (2003)
8. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)
9. imbalanced-learn developers, T.: API reference. https://imbalanced-learn.org/stable/references/index.html (2014-2022), accessed: 2022-07-27
10. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research **12**(7) (2011)
11. Egger, M., Schamberger, T., Tebelmann, L., Lippert, F., Sigl, G.: A second look at the ascad databases. In: International Workshop on Constructive Side-Channel Analysis and Secure Design. pp. 75–99. Springer (2022)
12. Erhan, D., Bengio, Y., Courville, A., Vincent, P.: Visualizing higher-layer features of a deep network. University of Montreal **1341**(3), 1 (2009)
13. eShard: Scared: Side-Channel Analysis Repository Database. https://gitlab.com/eshard/scared (2023), accessed on 08/01/2024
14. Fan, X., Tong, J., Li, Y., Duan, X., Ren, Y.: Power Analysis Attack Based on Hamming Weight Model without Brute Force Cracking. Security and Communication Networks **2022**, 1–11 (Jun 2022). https://doi.org/10.1155/2022/7375097, https://doi.org/10.1155/2022/7375097
15. FIPS, P.: 197: Federal information processing standards publication 197. Announcing the Advanced Encryption Standard (AES) (2001)
16. Frankle, J., Carbin, M.: The Lottery Ticket Hypothesis: Training Pruned Neural Networks. CoRR **abs/1803.03635** (2018), http://arxiv.org/abs/1803.03635
17. Gierlichs, B., Batina, L., Tuyls, P., Preneel, B.: Mutual information analysis. In: Oswald, E., Rohatgi, P. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2008. pp. 426–442. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
18. Hettwer, B., Gehrer, S., Güneysu, T.: Profiled power analysis attacks using convolutional neural networks with domain knowledge. In: International Conference on Selected Areas in Cryptography. pp. 479–498. Springer (2018)
19. Hettwer, B., Gehrer, S., Güneysu, T.: Applications of machine learning techniques in side-channel attacks: a survey. Journal of Cryptographic Engineering **10**(2), 135–162 (Apr 2019). https://doi.org/10.1007/s13389-019-00212-8, https://doi.org/10.1007/s13389-019-00212-8
20. Hoang, A.T., Hanley, N., O'Neill, M.: Plaintext: A Missing Feature for Enhancing the Power of Deep Learning in Side-Channel Analysis? IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 49–85 (Aug 2020). https://doi.org/10.46586/tches.v2020.i4.49-85, https://doi.org/10.46586/tches.v2020.i4.49-85
21. Hospodar, G., Gierlichs, B., Mulder, E., Verbauwhede, I., Vandewalle, J.: Machine learning in side-channel analysis: A first study. J. Cryptographic Engineering **1**, 293–302 (Dec 2011). https://doi.org/10.1007/s13389-011-0023-x

22. Ilya Kizhvatov: PySCA: Python Side-Channel Analysis Library. https://github.com/ikizhvatov/pysca (2018), accessed on 08/01/2024
23. Kerkhof, M., Wu, L., Perin, G., Picek, S.: Focus is key to success: A focal loss function for deep learning-based side-channel analysis. In: International Workshop on Constructive Side-Channel Analysis and Secure Design. pp. 29–48. Springer (2022)
24. Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make Some Noise. Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(3), 148–179 (May 2019). https://doi.org/10.13154/tches.v2019.i3.148-179, https://tches.iacr.org/index.php/TCHES/article/view/8292
25. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization (2014)
26. Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S.: Self-Normalizing Neural Networks. In: Advances in Neural Information Processing Systems 30 (NIPS 2017) (2017)
27. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996). https://doi.org/10.1007/3-540-68697-5_9
28. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999). https://doi.org/10.1007/3-540-48405-1_25
29. Kuroda, K., Fukuda, Y., Yoshida, K., Fujino, T.: Practical Aspects on Non-Profiled Deep-Learning Side-Channel Attacks against AES Software Implementation with Two Types of Masking Countermeasures Including RSM. In: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security. p. 29–40. ASHES '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3474376.3487285, https://doi.org/10.1145/3474376.3487285
30. Ledger Donjon: Lascar: Side-Channel Analysis Software Framework. https://github.com/Ledger-Donjon/lascar (2023), accessed on 08/01/2024
31. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking Cryptographic Implementations Using Deep Learning Techniques. In: Security, Privacy, and Applied Cryptography Engineering, pp. 3–26. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-49445-6_1, https://doi.org/10.1007/978-3-319-49445-6_1
32. Mangard, S., Oswald, E., Standaert, F.X.: One for all - all for one: Unifying standard DPA attacks. Cryptology ePrint Archive, Paper 2009/449 (2009), https://eprint.iacr.org/2009/449, https://eprint.iacr.org/2009/449
33. Martinasek, Z., Hajny, J., Malina, L.: Optimization of power analysis using neural network. In: International Conference on Smart Card Research and Advanced Applications. pp. 94–107. Springer (2013)
34. Masure, L., Dumas, C., Prouff, E.: Gradient Visualization for General Characterization in Profiling Attacks. In: Constructive Side-Channel Analysis and Secure Design, pp. 145–167. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-16350-19, https://doi.org/10.1007/978-3-030-16350-1_9
35. Masure, L., Dumas, C., Prouff, E.: A Comprehensive Study of Deep Learning for Side-Channel Analysis. Cryptology ePrint Archive, Report 2019/439 (2019), https://ia.cr/2019/439
36. Messerges, T.S.: Using Second-Order Power Analysis to Attack DPA Resistant Software. In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2000. pp. 238–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
37. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th international conference on machine learning (ICML-10). pp. 807–814 (2010)
38. Nassar, M., Souissi, Y., Guilley, S., Danger, J.: RSM: A small and fast countermeasure for AES, secure against 1st and 2nd-order zero-offset SCAs. In: Rosenstiel, W., Thiele, L. (eds.) 2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012. pp. 1173–1178. IEEE (2012). https://doi.org/10.1109/DATE.2012.6176671, https://doi.org/10.1109/DATE.2012.6176671
39. Nesterov, Y.E.: A method of solving a convex programming problem with convergence rate o\bigl(k^2\bigr). In: Doklady Akademii Nauk. vol. 269, pp. 543–547. Russian Academy of Sciences (1983)
40. NewAE Technology Inc.: ChipWhisperer: Open-Source Hardware Security. https://github.com/newaetech/chipwhisperer (2024), accessed on 08/01/2024

41. Perin, G., Chmielewski, Ł., Picek, S.: Strength in Numbers: Improving Generalization with Ensembles in Machine Learning-based Profiled Side-channel Analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 337–364 (Aug 2020). https://doi.org/10.46586/tches.v2020.i4.337-364, https://doi.org/10.46586/tches.v2020.i4.337-364
42. Perin, G., Wu, L., Picek, S.: AISY - Deep Learning-based Framework for Side-Channel Analysis. Cryptology ePrint Archive, Report 2021/357 (2021), https://eprint.iacr.org/2021/357
43. Perin, G., Wu, L., Picek, S.: Gambling for Success: The Lottery Ticket Hypothesis in Deep Learning-based SCA. Cryptology ePrint Archive, Report 2021/197 (2021), https://ia.cr/2021/197
44. Perin, G., Wu, L., Picek, S.: The Need for Speed: A Fast Guessing Entropy Calculation for Deep Learning-based SCA. Cryptology ePrint Archive, Report 2021/1592 (2021), https://ia.cr/2021/1592
45. Perin, G., Wu, L., Picek, S.: Gambling for Success: The Lottery Ticket Hypothesis in Deep Learning-Based Side-Channel Analysis. In: Artificial Intelligence for Cybersecurity, pp. 217–241. Springer (2022)
46. Picek, S., Heuser, A., Jovic, A., Bhasin, S., Regazzoni, F.: The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(1), 1–29 (Aug 2019). https://doi.org/10.13154/tches.v2019.i1.209-237, https://hal.inria.fr/hal-01935318
47. Prouff, E., Strullu, R., Benadjila, R., Cagli, E., Dumas, C.: Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. Cryptology ePrint Archive, Paper 2018/053 (2018). https://doi.org/10.1007/s13389-019-00220-8, https://eprint.iacr.org/2018/053
48. Rechberger, C., Oswald, E.: Practical Template Attacks (02 2005). https://doi.org/10.1007/978-3-540-31815-6_35
49. Rijsdijk, J., Wu, L., Perin, G., Picek, S.: Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 677–707 (2021)
50. Riscure: Jlsca: Open-Source Side-Channel Analysis Framework. https://github.com/Riscure/Jlsca (2022), accessed on 08/01/2024
51. Schindler, W., Lemke, K., Paar, C.: A stochastic model for differential side channel cryptanalysis. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 30–46. Springer (2005)
52. SideChannelMarvels: Daredevil: An Open-Source Framework for Fault Injection Attacks. https://github.com/SideChannelMarvels/Daredevil (2022), accessed on 08/01/2024
53. Smith, L.N.: Cyclical learning rates for training neural networks. In: 2017 IEEE winter conference on applications of computer vision (WACV). pp. 464–472. IEEE (2017)
54. Sobol, I.M.: Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and computers in simulation **55**(1-3), 271–280 (2001)
55. Socha, P., Miškovský, V., Novotný, M.: A Comprehensive Survey on the Non-Invasive Passive Side-Channel Analysis. Sensors **22**(21), 8096 (Jan 2022). https://doi.org/10.3390/s22218096, https://www.mdpi.com/1424-8220/22/21/8096, number: 21 Publisher: Multidisciplinary Digital Publishing Institute
56. Standaert, F.X.: How (not) to use Welch's T-test in side-channel security evaluations. In: Smart Card Research and Advanced Applications: 17th International Conference, CARDIS 2018, Montpellier, France, November 12–14, 2018, Revised Selected Papers 17. pp. 65–79. Springer (2019)
57. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks (extended version). Cryptology ePrint Archive, Paper 2006/139 (2006), https://eprint.iacr.org/2006/139
58. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Joux, A. (ed.) Advances in Cryptology - EUROCRYPT 2009. pp. 443–461. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
59. Team, T.P.: A GENTLE INTRODUCTION TO TORCH.AUTOGRAD. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html (2022), accessed: 2022-07-25
60. Timon, B.: Non-Profiled Deep Learning-based Side-Channel attacks with Sensitivity Analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(2), 107–131 (Feb 2019). https://doi.org/10.13154/tches.v2019.i2.107-131, https://tches.iacr.org/index.php/TCHES/article/view/7387
61. Weissbart, L.: Performance Analysis of Multilayer Perceptron in Profiling Side-Channel Analysis. In: Lecture Notes in Computer Science, pp. 198–216. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-61638-0\12, https://doi.org/10.1007/978-3-030-61638-0_12

62. Won, Y.S., Hou, X., Jap, D., Breier, J., Bhasin, S.: Back to the Basics: Seamless Integration of Side-Channel Pre-Processing in Deep Neural Networks. IEEE Transactions on Information Forensics and Security **16**, 3215–3227 (2021). https://doi.org/10.1109/TIFS.2021.3076928
63. Won, Y.S., Hou, X., Jap, D., Breier, J., Bhasin, S.: Multi-scale Convolutional Neural Networks (MCNN) based Side-Channel Analysis. https://github.com/mitMathe/SCA-MCNN (2021)
64. Wouters, L., Arribas, V., Gierlichs, B., Preneel, B.: Revisiting a methodology for efficient CNN architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 147–168 (2020)
65. Wu, L., Perin, G., Picek, S.: I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis. Cryptology ePrint Archive, Report 2020/1293 (2020), https://ia.cr/2020/1293
66. Wu, L., Picek, S.: Remove some noise: On pre-processing of side-channel measurements with autoencoders. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 389–415 (2020)
67. Zaid, G., Bossuet, L., Habrard, A., Venelli, A.: Methodology for Efficient CNN Architectures in Profiling Attacks (Nov 2019). https://doi.org/10.13154/tches.v2020.i1.1-36, https://tches.iacr.org/index.php/TCHES/article/view/8391
68. Zeiler, M.D.: Adadelta: An adaptive learning rate method (2012)