Methodology for Efficient CNN Architectures in Profiling Attacks – Extended Version

Gabriel Zaid[1,2], Lilian Bossuet[1], Amaury Habrard[1] and Alexandre Venelli[2]

[1] Univ Lyon, UJM-Saint-Etienne, CNRS Laboratoire Hubert Curien UMR 5516 F-42023, Saint-Etienne, France, firstname.lastname@univ-st-etienne.fr

[2] Thales ITSEF, Toulouse, France, firstname.lastname@thalesgroup.com

Abstract. The side-channel community recently investigated a new approach, based on deep learning, to significantly improve profiled attacks against embedded systems. Previous works have shown the benefit of using convolutional neural networks (CNN) to limit the effect of some countermeasures such as desynchronization. Compared with template attacks, deep learning techniques can deal with trace misalignment and the high dimensionality of the data. Pre-processing is no longer mandatory. However, the performance of attacks depends to a great extent on the choice of each hyperparameter used to configure a CNN architecture. Hence, we cannot perfectly harness the potential of deep neural networks without a clear understanding of the network's inner-workings. To reduce this gap, we propose to clearly explain the role of each hyperparameters during the feature selection phase using some specific visualization techniques including *Weight Visualization*, *Gradient Visualization* and *Heatmaps*. By highlighting which features are retained by filters, heatmaps come in handy when a security evaluator tries to interpret and understand the efficiency of CNN. We propose a methodology for building efficient CNN architectures in terms of attack efficiency and network complexity, even in the presence of desynchronization. We evaluate our methodology using public datasets with and without desynchronization. In each case, our methodology outperforms the previous state-of-the-art CNN models while significantly reducing network complexity. Our networks are up to 25 times more efficient than previous state-of-the-art while their complexity is up to 31810 times smaller. Our results show that CNN networks do not need to be very complex to perform well in the side-channel context.

Keywords: Side-Channel Attacks · Deep Learning · Architecture · Weight Visualization · Heatmap · Feature selection · Desynchronization

1 Introduction

Side-Channel Analysis (SCA) is a class of cryptographic attack in which an evaluator tries to exploit the vulnerabilities of a system by analyzing its physical properties, including power consumption [KJJ99] or electromagnetic emissions [AARR03], to reveal secret information. During its execution, a cryptographic implementation manipulates sensitive variables that depend directly on the secret. Through the attack, security evaluators try to recover this information by finding some leakage related to the secret. One of the most powerful types of SCA attacks are *profiled attacks*. In this scenario, the evaluators have access to a test device whose target intermediate values they know. Then, they can estimate the conditional distribution associated with each sensitive variable. They can predict the right sensitive value on a target device containing a secret they wish to retrieve by using multiple traces that reveal the secret. In 2002, the first profiled attack was introduced by [CRR03], but their proposal was limited by the computational complexity. Very similar to

profiled attacks, the application of machine learning algorithms was inevitably explored in the side-channel context [HGM+11, BL12, HZ12, LBM14, LPMS18].

Some recent papers have evidenced the robustness of convolutional neural networks (CNNs) to the most common countermeasures, namely *masking* [MPP16, MDP19a] and *desynchronization* [CDP17]. CNNs are at least as efficient as standard profiled attacks. One of their main advantages is that they do not require pre-processing. A CNN consists of two parts, a *convolutional part*, whose goal is to locate the features with the most impact on the classification, and a *fully-connected part*, that aggregates the most relevant features, in order to correctly classify each trace. Nonetheless, finding a suitable architecture is one of the most challenging tasks in deep learning because we have to set the parameters that configure the network properly to achieve a good level of efficiency. Two types of parameters exist: *trainable parameters* and *hyperparameters*. Trainable parameters are non-parametric variables that are internal to the model and are automatically estimated during the training process. Hyperparameters are parametric variables that need to be set before applying a learning algorithm. The hyperparameters can be divided into two categories:

- Optimizer hyperparameters that are related to the optimization and training process (learning rate, batch size, number of epochs, optimizer, activation functions, weight initialization, etc);
- Model hyperparameters, that are involved in the structure of the model (number of hidden units, number of layers, length of filters, number of convolutional blocks, etc).

Depending on the choice of these values, the performance of the network will vary considerably. The two types of hyperparameters affect each other, they both need to be correctly selected. Choosing correct model hyperparameters is the first step towards obtaining an optimal neural network. Our work focuses on building efficient and suitable architectures. Our results will of course also benefit from selecting suitable optimizer hyperparameters that have been investigated in previous literature [CDP17, KPH+19].

Contributions. In this paper, we focus on the explainability and interpretability of model hyperparameters that comprise the convolutional part of a CNN, which is the part that aims to reveal the most relevant features. In side-channel attacks, the information that helps decision making is defined by the PoIs which are revealed by leakage detection tools such as the *Signal-to-Noise Ratio* (SNR) [MOP07]. By accurately identifying these points, we can significantly reduce the complexity of the classification part because the aggregation phase will be much easier. To evaluate the impact of each model hyperparameter, we apply visualization techniques, such as *Gradient Visualization* [MDP19b] to get an overview of how the network selects the features it uses to predict the sensitive variable. However, this technique is not appropriate if we want to interpret the convolutional part of the network. Here, we propose two new visualization techniques widely used in the deep learning field i.e., *Weight Visualization* and *Heatmaps*. These tools help us to evaluate the impact of each model hyperparameter including the number of filters, the length of each filter and the number of convolutional blocks in order to identify a suitable architecture, minimizing its complexity while increasing its attack efficiency. Then, we propose a method to build efficient CNN architectures. We confirm the relevance of our methodology by applying it to the main public datasets and comparing our state-of-the-art results with those of previous studies. This new methodology also enables a better tradeoff between the complexity of the model and its performance, thereby allowing us to reduce learning time.

Paper Organization. The paper is organized as follows. Section 2 presents the convolutional neural network in particular the convolutional part. After describing standard feature selection techniques in Section 3, Section 3.2 and Section 3.3 present two new

visualization tools applied to the SCA context. In Section 4, we highlight the impact of each model hyperparameter on the convolutional part. We then propose a methodology to build an efficient architecture depending on the countermeasures implemented. The new methodology is applied to public datasets and the state-of-the-art results are detailed in Section 5. Our propostion has been recently questioned by [WAGP20]. Section 6 discusses the misinterpretations introduced by this revisit.

2 Preliminaries

2.1 Notation and terminology

Let calligraphic letters $\mathcal{X}$ denote sets, the corresponding capital letters X (resp. bold capital letters) denote random variables (resp. random vectors $\mathbf{T}$) and the lowercase x (resp. $\mathbf{t}$) denote their realizations. The i-th entry of a vector $\mathbf{t}$ is defined as $\mathbf{t}[i]$. Side-channel traces will be constructed as a random vector $\mathbf{T} \in R^{1\times D}$ where D defines the dimension of each trace. The targeted sensitive variable is $Z = f(P, K)$ where f denotes a cryptographic primitive, P ($\in \mathcal{P}$) denotes a public variable (e.g.plaintext or ciphertext) and K ($\in \mathcal{K}$) denotes a part of the key (e.g.byte) that an adversary tries to retrieve. Z takes values in $\mathcal{Z} = \{s_1, ..., s_{|\mathcal{Z}|}\}$. Let us denote k^* the secret key used by the cryptographic algorithm.

2.2 Profiled Side-Channel Attacks

When attacking a device using a profiled attack, two stages must be considered: a building phase and a matching phase. During the first phase, adversaries have access to a test device with which they can control the input and the secret key of the cryptographic algorithm. They use this knowledge to locate the relevant leakages depending on Z. To characterize the points of interest (PoIs), the adversaries generate a model $F : \mathbb{R}^D \rightarrow \mathbb{R}^{|\mathcal{Z}|}$ that estimates the probability $Pr[\mathbf{T}|Z = z]$ from a profiled set $\mathcal{T} = \{(\mathbf{t}_0, z_0), \ldots, (\mathbf{t}_{N_p-1}, z_{N_p-1})\}$ of size N_p. The high dimensionality of $\mathbf{T}$ could be an obstacle to building a leakage model. Popular techniques use dimensionality reduction, such as *Principal Components Analysis* (PCA) [APSQ06] or *Kernel Discriminant Analysis* (KDA) [CDP16], to select PoIs where most of the secret information is contained. Once the leakage model is generated, adversaries estimate which intermediate value is processed thanks to a predicting function $F(\cdot)$ they designed themselves. By predicting this sensitive variable and knowing the input used during the encryption, adversaries can compute a score vector, based on $F(\mathbf{t}_i), i \in [0, |N_a| - 1]$, for each trace included in a dataset of N_a attack traces. Then, log-likelihood scores are computed to make predictions for each key hypothesis. The candidate with the highest value will be defined as the recovered key.

Metric in Side-Channel Attacks. During the building phase, adversaries want to find a model F that optimizes recovery of the secret key by minimizing the size of N_a. In SCA, a common metric used is called guessing entropy (GE) [SMY09], which defines the average rank of k^* through all the key hypotheses. Let us denote $g(k^*)$ the guessing value associated with the secret key. We consider an attack is successful when the guessing entropy is permanently equal to 1. The rank of the correct key gives us an insight into how well our model performs.

2.3 Neural Networks

Profiled SCA can be formulated as a *classification problem*. Given an input, a neural network aims to construct a function $F : R^n \rightarrow R^{|\mathcal{Z}|}$ that computes an output called *a prediction*. To solve a classification problem, the function F must find the right prediction

$y \in \mathcal{Z}$ associated with the input $\mathbf{t}$ with high confidence. To find the optimized solution, a neural network has to be trained using a profiled set of N pairs $(\mathbf{t}_p^i, y_p^i)$ where $\mathbf{t}_p^i$ is the i-th profiled input and y_p^i is the label associated with the i-th input. In SCA, the input of a neural network is a side-channel measurement and the related label is defined by the corresponding sensitive value z. To construct a function F, a neural network is composed of several simple functions called *layers*. Three kinds of layers exist: the input layer (composed of input $\mathbf{t}$), the output layer (gives an estimation of the prediction vector y) and one or multiple hidden layers. First, the input goes through the network to estimate the corresponding score $\hat{y}_p^i = F(\mathbf{t}_p^i)$. A *loss function* is computed that quantifies the classification error of F over the profiled set. This stage is called *forward propagation*. Each trainable parameter is updated to minimize the loss function. This is called *backward propagation* [GBC16]. To accurately and efficiently find the loss extremum, an optimizer is used (e.g.Stochastic Gradient Descent [RM51, KW52, BCN18], RMSprop, Momentum [Qia99], Adam [KB15], etc). These steps are processed until the network reachs a sufficient local minimum. The choice of hyperparameters is one of the biggest challenges in the machine learning field. Some techniques exist to select them, such as Grid-Search Optimization, Random-Search Optimization [BB12, PGZ+18, BBBK11] but none are deterministic. This paper focuses on understanding model hyperparameters in order to build efficient neural network architectures for side-channel attacks.

2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is specific type of neural network. In this paper, we choose to simplify its presentation by considering that it can be decomposed into two parts: a feature extraction part and a classification part (see Figure 1-a). Features selection aims at extracting information from the input to help the decision-making. To select features, a CNN is composed of n_3 stacked convolutional blocks that correspond to n_2 convolutional layers (denoted γ), an activation function (σ) and one pooling (denoted δ) layer [ON15]. This feature recognition part is plugged into the classification part of n_1 Fully-Connected (FC) layers (denoted λ). Finally, we denote s the softmax layer (or prediction layer) composed of $|Z|$ classes. To sum up, a common convolutional network can be characterized by the following formula:

$$s \circ [\lambda]^{n_1} \circ [\delta \circ [\sigma \circ \gamma]^{n_2}]^{n_3}. \tag{1}$$

2.4.1 Convolutional layer

The convolutional layer performs a series of convolutional operations on its inputs to facilitate pattern recognition (see Figure 1-b). During forward propagation, each input is convoluted with a filter (or *kernel*). The output of the convolution reveals temporal instants that influence the classification. These samples are called **features**. To build a convolutional layer, some model hyperparameters have to be configured: the length and number of kernels, pooling stride and padding.

- Length of filters – Kernels are generated to identify features that could increase the efficiency of the classification. However, depending on their size, filters reveal local or global features. Smaller filters tend to identify local features while larger filters focus on global features. Figure 1-b gives an example in which the length of filters is set to 3.
- Stride – Stride refers to the step between two consecutive convolutional operations. Using a small stride corresponds to the generation of an overlap between different filters while a longer stride reduces the output dimension. By default, the stride is set to 1 (see Figure 1-b).

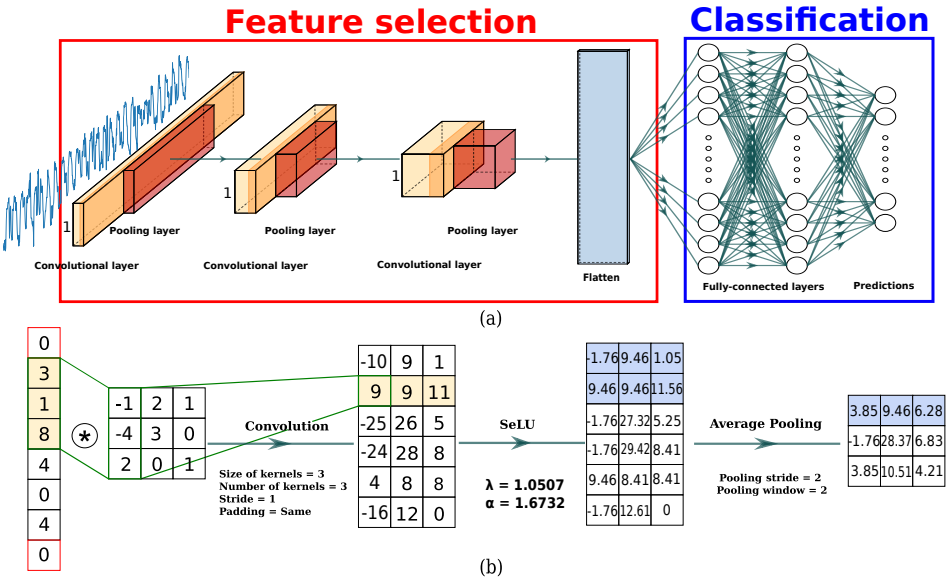

Figure 1: (a) - CNN architecture for side-channel attacks (red area: feature selection part ; blue area: classification part). (b) - Operations in convolutional blocks (convolutions, activation, average pooling)

- Padding – Let $\mathbf{a}$ and $\mathbf{b}$ be two vectors, the dimension of the convolution between these two vectors will be $dim(\mathbf{a} \circledast \mathbf{b}) = \left(\frac{dim(\mathbf{a}) - dim(\mathbf{b})}{stride} + 1\right)$ [GBC16] where $\circledast$ refers to the convolution operation. In some cases, a subsample may be generated. To avoid this phenomenon and to avoid losing information, we can use padding that adds a "border" to our input to ensure the same dimensions are retained after the convolutional operation. By default, two kinds of padding are used: *valid padding* and *same padding*. *Valid padding* means "no-padding" while *same padding* refers to a zero-padding (the output has the same dimension as the input) [GBC16]. Figure 1-b gives an example in which we select *same padding*. Indeed, two 0 values are added at the endpoints of the vector in order to obtain an output vector of dimension 6.

After each convolutional operation, an *activation function* (denoted σ) is applied to identify which features are relevant for the classification. As explained in [KUMH17], the *scaled exponential linear unit function* (SeLU) is recommended for its self-normalizing properties. The SeLU is defined as follows:

$$selu(x) = \lambda \begin{cases} x & \text{if} \quad x > 0, \\ \alpha(exp(x) - 1) & \text{if} \quad x \leqslant 0. \end{cases} \tag{2}$$

The SeLU pushes neuron activation towards zero mean and unit variance in order to prevent vanishing and exploding gradient problems. This activation function is used by default in our architectures.

2.4.2 Pooling layer

The pooling layer is a non-linear layer that divides the dimension of the input such that the most relevant information is preserved. To apply its down sampling function, a pooling window and a pooling stride have to be configured. Usually, these variables have the same value to avoid overlapping. The window slides through the input values to select a segment to be applied to the pooling function. In deep learning, two pooling functions are commonly used:

- MaxPooling – The output is defined as the maximum value contained in the pooling window.
- AveragePooling – The output is defined as the average of the values contained in the pooling window. Figure 1-b shows an example of this function applied to a 1-D input with pooling_window = pooling_stride = 2.

2.4.3 Flatten layer

The flatten layer concatenates each intermediate trace of the final convolutional block in order to reduce the 2-D space, which corresponds to the dimension at the end of the convolutional part, into a 1-D space to input into the classification part. Let us denote M the input of the flatten layer such that $M \in \mathbb{M}_{n,d}(\mathbb{R})$ where n denotes the number of outputs after the last convolutional block and d denotes the number of samples for each of these outputs such that:

$$M = \begin{bmatrix} \mathbf{t}^0[x_0] & \mathbf{t}^0[x_1] & \mathbf{t}^0[x_2] & \cdots & \mathbf{t}^0[x_{d-1}] \\ \mathbf{t}^1[x_0] & \ddots & \cdots & \cdots & \vdots \\ \mathbf{t}^2[x_0] & \cdots & \ddots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{t}^n[x_0] & \cdots & \cdots & \cdots & \mathbf{t}^{n-1}[x_{d-1}] \end{bmatrix} \tag{3}$$

where $(\mathbf{t}^i)_{0 \leqslant i \leqslant n}$ is the i-th intermediate trace and $(x_j)_{0 \leqslant j \leqslant d}$ the j-th sample of the trace.

The output of the flatten layer is a concatenated vector C that can be constructed:

- Column-wise – $C = \left[\mathbf{t}^0[x_0], \mathbf{t}^1[x_0], \mathbf{t}^2[x_0], ..., \mathbf{t}^{n-1}[x_{d-1}]\right]$,
- Row-wise – $C = \left[\mathbf{t}^0[x_0], \mathbf{t}^0[x_1], \mathbf{t}^0[x_2], ..., \mathbf{t}^{n-1}[x_{d-1}]\right]$.

2.4.4 Fully-Connected Layers

Once the convolutional part has selected features, the fully-connected layers (FC) (denoted λ) recombine each neuron to efficiently and accurately classify each input. FC layers can be compared to a MultiLayer Perceptron (MLP) where each neuron has full connections to all activations of the previous layer (see Figure 1-a).

In this paper, we propose a way to select appropriate model hyperparameters in order to identify a suitable network architecture.

3 Evaluation of feature selection

Due to the black-box nature of neural networks, it can be challenging to explain and interpret its decision-making. To overcome this problem, some visualization techniques have been developed [vdMH08, ZF14, SVZ14, ZKL+16] but their application in the SCA context is not sufficiently exploited. As shown in the following papers [MDP19b, HGG19, PEC19], these techniques could be useful in SCA to evaluate the ability of a network to extract the points of interest.

3.1 State of the art

Signal-to-noise ratio. The signal-to-noise ratio (SNR) [Man04, MOP07] is one of the most useful tools in SCA to characterize temporal instants where sensitive information leaks.

Unfortunately, in presence of countermeasures such as desynchronization, estimation of PoIs becomes much more complex. Indeed, desynchronization disturbs the detection of PoIs by randomly spreading the sensitive variable over time samples.

Visualization techniques. Gradient visualization computes the derivatives of a CNN model with respect to an input trace such that the magnitude of the derivatives indicates which features need to be modified the least to have the most effect the class score. This technique helps identify the temporal instants that influence the classification. Comparing these points to PoIs enables us to interpret the learning phase of a network [MDP19b].

Layer-wise Relevance Propagation (LRP) is introduced in [BBM+15] and applied in [HGG19] in the side-channel context. LRP propagates the prediction score through the network up to the input layer thereby indicating the relevant layers. In [HGG19], the authors use this technique to underline which temporal instants influence the learning process the most.

Limitations. The aim of these proposals is to interpret a CNN decision by identifying each time point in the trace that contributes most to a particular classification. However, these techniques are not ones required to understand how the convolutional or classification part selects its feature as they only give a general interpretation of how a network performs. If the network is unable to find the right PoIs, the faulty part (convolutional or classification) cannot be evaluated. In this case, a more precise technique should be used to interpret each part independently. To mitigate this problem, we propose to apply *Weight Visualization* and *Heatmaps* to provide information related to the convolutional part.

3.2 Weight visualization

One way to interpret the performance of a network is to visualize how it selects its features. As explained in the introduction, one common strategy is to visualize the trainable weights to interpret the recognition patterns. This method was introduced in [BPK92] as a useful framework when dealing with spatial information. During the training process, the network evaluates influencial neurons which generate an efficient classification. If the network is confident in its predictions, it will attribute large weights to these neurons. In deep learning, this technique is usually used to evaluate features extracted by the first layers of a deep architecture [HOWT06, LBLL09, OH08, HOT06]. The role of the convolutional part is to select the relevant time samples (i.e. PoIs) that constitute a trace for allowing an efficient classification. Therefore, we propose weight visualization as a new tool to evaluate the performance related to the convolutional part.

As defined in Section 2.4, the flatten operation sub-samples each intermediate trace following an axis. By concatenating the output of the flatten layer following the columns (see Equation 3), we retain timing information to be able to reveal leakages and interpret it (see Figure 2-a). If feature selection is efficient, the neurons where information leaks will be evaluated with high weights by the training process. Thus, by visualizing the weight of the flatten layer, we can understand which neurons have a positive impact on the classification and hence, thanks to the *feedforward propagation*, we can interpret which time samples influence the most our model.

Let us denote $n_u^{[\text{flatten}+1]}$ the number of neurons in the layer following the flatten, $n_f^{[\text{flatten}-1]}$ the number of filters in the last convolutional blocks and $dim_{\text{traces}}^{[\text{flatten}-1]}$ the dimension associated with each intermediate trace after the last pooling layer. Let us denote $W^{[\text{flatten}]}$ the weights corresponding to the flatten layer such that $dim(W^{[\text{flatten}]}) = (dim_{\text{traces}}^{[\text{flatten}-1]} \times n_f^{[\text{flatten}-1]})$. Let us denote $W_m^{\text{vis}} \in \mathbb{R}^{dim_{\text{traces}}^{[\text{flatten}-1]}}$ a vector that enables visualization of the weights related to the m-th neurons of the layer following the flatten:

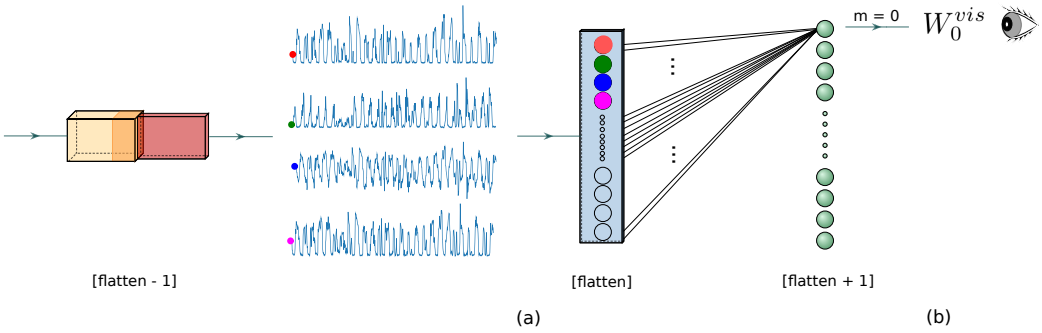

Figure 2: Methodology for weight visualization ((a) - concatenation of each intermediate trace following their temporal axis ; (b) - computation of the average of W_0^{vis} to get a temporal correspondence)

$$W_m^{\text{vis}}[i] = \frac{1}{n_f^{[\text{flatten}-1]}} \sum_{j=i\times n_f^{[\text{flatten}-1]}}^{(i+1)\times n_f^{[\text{flatten}-1]}} |W_m^{[\text{flatten}]}[j]| \tag{4}$$

where $i \in [0, dim_{\text{traces}}^{[\text{flatten}-1]}]$.

Then, let us denote $W^{\text{vis}} \in \mathbb{R}^{dim_{\text{traces}}^{[\text{flatten}-1]}}$ a vector that enables visualization of the mean weight related to each neuron of the layer [flatten +1] such that:

$$W^{\text{vis}}[i] = \frac{1}{n_u^{[\text{flatten}+1]}} \sum_{m=0}^{n_u^{[\text{flatten}+1]}} W_m^{\text{vis}}[i] \tag{5}$$

where $i \in [0, dim_{\text{traces}}^{[\text{flatten}-1]}]$.

In other words, first, we reduce the dimension of $W^{[\text{flatten}]}$ so that its dimension corresponds to the temporal space defined by $dim_{\text{traces}}^{[\text{flatten}-1]}$. This reduction is obtained by averaging the weights, associated with the same point on the intermediate traces (see Figure 2-b). Finally, to obtain a more precise evaluation, we have to compute the average of the mean weight associated with each neuron in order to evaluate the confidence of the network in revealing PoIs. Indeed, the weight reveals how important a feature is in order to accurately classify an input. The confidence of a network reflects its ability to detect the PoIs. If the weights associated with the relevant information increase, the network will be more confident in its feature detection, thus, the combination of information will be easier and the resulted training time will decrease. An example of weight visualization is given in Figure 14 in Appendix B.

3.3 Heatmap

Even if weight visualization is a useful tool, we still cannot understand which feature is selected by each filter. One solution is to analyze the activation generated by the convolution operation between an input layer and each filter. Introduced in [ZF14], heatmaps (or feature maps) help understand and interpret the role of each filter to enable the production of suitable CNN architectures. We propose to apply this technique in the side-channel context as a new tool to interpret the impact of filters so that we can adapt the network according to how the features are selected.

Let us denote $n_f^{[h]}$ the number of filters in the h-th layer of the convolutional part, $\text{input}^{[h]}$ the input(s) associated with the h-th convolutional block such that $\text{input}^{[0]}$ corresponds to the side-channel trace and $\text{output}^{[h-1]} = \text{input}^{[h]}$. Then, the convolution between $\text{input}^{[h]}$

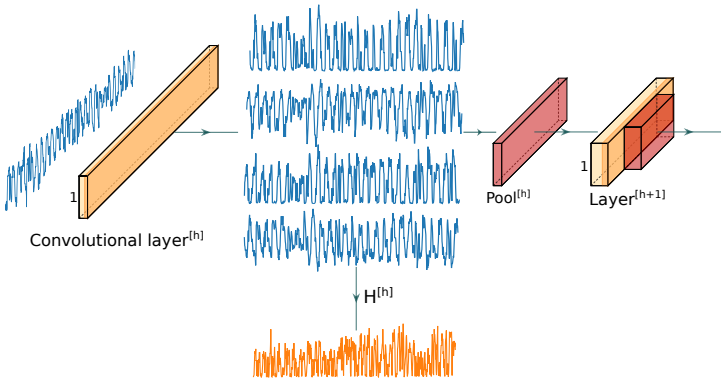

Figure 3: Heatmap

and the $n_f^{[h]}$ filters returns $n_f^{[h]}$ intermediate traces. We denote $H^{[h]}$ the heatmap (or feature map) associated with the h-th layer such that:

$$H^{[h]} = \frac{1}{n_f^{[h]}} \sum_{i=0}^{n_f^{[h]}} (\text{input}^{[h]} \circledast f_i^{[h]}) \tag{6}$$

where $f_i^{[h]}$ is the i^{th} filter of the h-th layer of the convolutional part.

Thanks to this visualization technique, we can assess which neurons are activated by the filters of each convolutional layer and understand how the features are selected (see Figure 3). Indeed, the resulted convolution operation reflects the time samples that are considered relevant during the classification. For a better understanding, we propose to explain the benefits of using heatmaps with a toy example. We simulate traces such that the SNR peaks are given in Figure 21-e in Appendix F. An architecture is generated, resulting in well-performed classification. The resulted heatmaps are given in Figure 21 in Appendix F (more details on the architecture are provided in Section 4.4.2). From the Figure 21-c in Appendix F, we are can assess which features are selected by the network. The comparison of this heatmap with the SNR peaks can help us to investigate the ability of the network to find the relevant features (i.e., PoIs).

The weight visualization and the heatmaps are two tools that help to explain and interpret feature selection more clearly. By using these techniques to understand the internal layers of a CNN, it is possible to define a method to build suitable neural network architectures in the presence of desynchronization.

4 Method to build efficient CNN architectures

In this section, we analyze the effect of each model hyperparameters in the convolutional part of a CNN and propose a method to build efficient architectures. Using the visualization techniques presented above, we can understand how each hyperparameter impacts the efficiency of the CNN and how it should be tweaked in the presence of desynchronization.

4.1 Settings

To illustrate and explain the effect of each hyperparameters as far as possible, let us consider an unmasked implementation of AES-128 on the Chipwhisperer[a] (8-bit XMEGA Target). Our experience is conducted with 45,000 power traces of 3000 points. The

[a] https://newae.com/tools/chipwhisperer/

targeted operation is $Z = Sbox[P_1 \oplus k^*]$ where P_1 denotes the first byte of the plaintext and k^* denotes the secret key. The measured SNR equals 2.28 (see Appendix A).
The CNN is implemented in Python using the *keras* library [C+15] and is run on a workstation equipped with 8GB RAM and a NVIDIA GTX1060 with 8GB memory. We use the *Categorical Cross-Entropy* as loss function because minimizing the cross entropy is equivalent to maximizing the lower bound of the mutual information between the traces and the secret variable [MDP19a]. Optimization is done using the *Adam* [GBC16] approach on *batch size* 50 and the *learning rate* is set to 10^{-3}. As explained in Section 2.4, we use the *SeLU* activation function to avoid vanishing and exploding gradient problems [KUMH17]. For improved weight initialization, we use the *He Uniform* initialization [HZRS15]. The loss function, the optimizer, the activation function and the weight initialization follow this setup in Section 5. These values are selected following a *grid search optimization* [GBC16]. For each optimizer hyperparameter, we select a finite set of values (see Table 1). Our choices are motivated by the fact that they provide good results, in terms of classification, on our datasets. Indeed, using these hyperparameters lead to a faster convergence towards a constant guessing entropy of 1.

Table 1: Grid search optimization on hyperparameters

	Values
Optimizer	$\{SGD, RMSprop, Adam\}$
Weight initialization	$\{Uniform\ distribution, Glorot\ uniform$[GB10]$, He\ uniform\}$
Activation function	$\{tanh, ReLU, eLU$[CUH16]$, SeLU\}$
Learning Rate	$One-Cycle\ policy$ (see Section 5.1)
Batch size	$\{50, 64, 128, 256\}$
Epochs	$\{10, 20, 25, 50, 75, 100, 125, 150\}$
n° of neurons (FC layers)	$\{2, 5, 10, 15, 20, 25, 100\}$
n° of layers (FC layers)	$\{1, 2, 3, 4, 5\}$

We use $40,000$ traces for the training process, $5,000$ traces for the validation and 20 epochs are used. Finally, to accelerate the learning phase, we pre-process the data such that all trace samples are standardized and normalized between 0 and 1 [GBC16]. In this section, we only focus on the pattern selection made by the convolutional part and not on the attack exploitation.

4.2 Length of filters

In this section, we demonstrate that increasing the length of the filter causes entanglement and reduces the weight related to a single information and therefore, the network confidence. The network confidence is defined as its ability to detect accurately the PoIs in the convolutional part. To this end, we need an assumption such that there exist only a small dataset $\mathbb{E} = \{x_0, x_1, ..., x_l\}$ of time samples such that $Pr[Z|\mathbf{T}] = Pr[Z|\mathbf{T}[x_0], ..., \mathbf{T}[x_l]]$ where $l << dim(\mathbf{T})$. Let us denote $W \in \mathbb{R}^n$ a filter of length n such that $W = \{\alpha_0, \alpha_1, ..., \alpha_n\}$ is optimized for maximizing the detection of l PoIs. Convolution operation between a trace $\mathbf{t}$ and a filter W follows the equation:

$$(\mathbf{t} \circledast W)[i] = \sum_{j=0}^{n} \left(\mathbf{t} \left[j + i - \frac{n}{2} \right] \times W[j] \right) \tag{7}$$

such that

$$\mathbf{t} \left[j + i - \frac{n}{2} \right] \times W[j] = \begin{cases} \alpha_j \times \mathbf{t} \left[j + i - \frac{n}{2} \right] & \text{if} \quad j \in \mathbb{E}, \\ \epsilon & \text{otherwise.} \end{cases} \tag{8}$$

where $\epsilon \approx 0$ and α_j denotes the weight related to the index j.
Denote $\mathbf{t}$ a trace such that $(\mathbf{t} \circledast W)[i]$ covers the l relevant points and $(\mathbf{t} \circledast W)[i+m]$ covers less than l relevant points. Because W maximize the detection of the l PoIs, then,

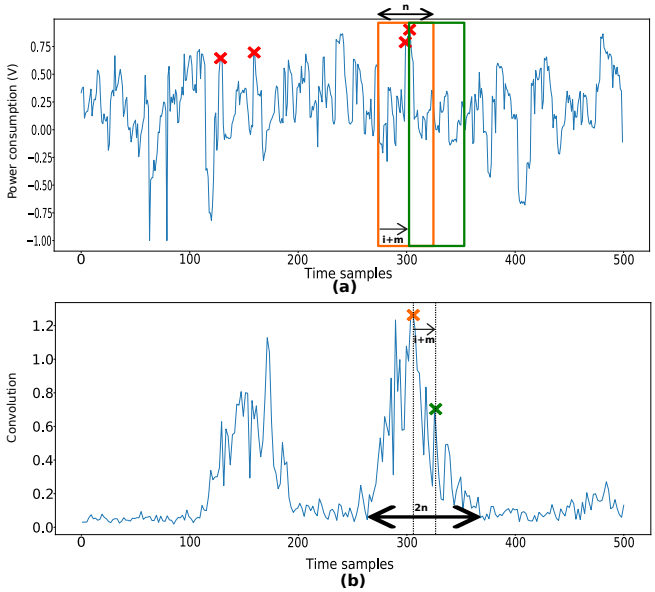

Figure 4: Convolution between a trace $\mathbf{t}$ and filter W of size n ((a) - trace; (b) - convolution between $\mathbf{t}$ and W)

$$(\mathbf{t} \circledast W)[i] > (\mathbf{t} \circledast W)[i+m] \tag{9}$$

Figure 4 provides an example of this operation. Four relevant information can be extracted from the input trace (see red crosses). In this figure, we show the result of two convolution operations $(\mathbf{t} \circledast W)[i]$ (see orange cross) and $(\mathbf{t} \circledast W)[i+m]$ (see green cross) that share the same information related to the highest relevant leakage. Consequently, when entanglement occurs the convoluted samples share the same relevant information. Therefore, in our example, the highest PoI is spread over the two convoluted samples. By increasing the size n of the filters, we increase the entanglement, consequently, more PoIs are shared between these convoluted points. Thus, the relevant information is spread over $2n$ convoluted samples. This phenomenon is illustrated in Figure 14 in Appendix B. If we want to precisely define the temporal space where leakages occur, we recommend to minimize the length of the filters in order to reduce the risk of entanglement.
Let two filters $W \in \mathbb{R}^{n_1}$ and $W^* \in \mathbb{R}^{n_2}$ such that $n_1 > n_2$. Assume that these filters share the same PoI denoted x_l. In other words, the PoI is respectively shared by n_1 samples (when W is applied) and n_2 samples (when W^* is applied). Let α_l^{opt} be the weight related to the PoI when x_l is assigned to a single convoluted sample. Then,

$$(\mathbf{t} \circledast W)[i] = (n_1 - 1) \times \epsilon + \frac{\alpha_l^{opt}}{n_1} \times \mathbf{t}[x_l] \tag{10}$$

Therefore, the weight related to $(\mathbf{t} \circledast W^*)[i]$ is higher than $\frac{\alpha_l^{opt}}{n_1}$. In conclusion, increasing the length of the filters also reduces the weight related to a single information because entanglement occurs.
We illustrate these properties by applying different architectures to our Chipwhisperer dataset to efficiently evaluate the impact of the length of the filter. We select a range of lengths (see Appendix B) and visualize the weights to experimentally verify this claim. The shorter the length, the less the relevant information is spread. Figure 14 in Appendix B shows that the weight visualization succeeds in recovering the location of the leakage. Indeed, the PoIs correspond to the relevant information detected by the SNR (see

Appendix A). The number of samples is divided by 2 because of the pooling stride value (i.e., 2) that we used in the first convolutional block. As explained above, small filters extract information better because the PoIs are not shared with a lot of convoluted samples, while using larger filters causes entanglement and provide the detection of "global features". The exploitation of this relevant information could be difficult during the classification part because the same information is spread over the trace. Thus, the resulted training time needed to reach the final performance could dramatically increase. By reducing this entanglement, we can accelerate this phase without significantly degrading the final performance.

4.3 Number of convolutional blocks

In this section, we provide an interpretation of the number of convolutional blocks. We compare the pooling functions that can be denoted as $f^{[h]}: \mathbb{R}^{n^{[h]}} \rightarrow \mathbb{R}^{\frac{n^{[h]}}{\text{pooling\_stride}^{[h]}}}$ where h corresponds to the h^{th} convolutional block. As shown in Appendix C, using average pooling has the advantage that unshared information is preserved. However, when the pooling stride is large, the relevant information is reduced. Consequently, using a lot of convolutional blocks will affect leakage detection because the information is divided by pooling stride after each convolutional block.
When MaxPooling is used, if two consecutive pooling computation share the same optimal leakage $x_{l^{[h]}_{conv,opt}}$, then this information is spread over the pooling samples (see Appendix C). As explained in Equation 10, if the same relevant information is spread over lots of samples, then the related weight decreases and the combination of relevant information could be harder in the classification part. Thus the training time necessary to reach the final solution could dramatically increase. Furthermore, because only $x_{l^{[h]}_{conv,opt}}$ is selected, all other leakages are discarded. In this way, information that seems to be less important can be lost, when in fact it is essential for the detection of relevant points. For this reason, we recommend using **Average Pooling** as much as possible because no relevant points are discarded.
Furthermore, incrementing the number of convolutional blocks reduces the trace dimension while preserving relevant information associated with the leakages. Indeed, in most cases, $dim_{\mathbf{t}}^{[h]} = \frac{dim_{\mathbf{t}}^{[h-1]}}{pooling\_stride^{[h-1]}}$. Then, we reduce by a factor $pooling\_stride^{[h-1]}$ the distance between the relevant points. Hence, the impact of desynchronization is reduced by the same factor. It will be much easier, for the network, to characterize the desynchronization. By adding more convolutional blocks, we can drastically reduce the impact of desynchronization by choosing an appropriate $pooling\_stride^{[h]}$.
These properties show that the pooling functions reduce the trace dimension while preserving the most relevant samples. To illustrate these properties, we apply an average pooling or a max pooling (see Appendix D) to the Chipwhisperer dataset. Each convolutional block is configured with one convolutional layer composed by two filters of size 1 and one pooling layer with a *pooling_stride* of 2. Surprisingly, the detection of PoIs is almost the same for both pooling operations. For the first layers, the network seems to be more confident in its detection of features when average pooling is applied. Whereas max pooling seems to be more suitable for much deeper networks. However, in both cases, we can see that, the deeper the network, the less confident it is in its feature detection. In the presence of desynchronization, a trade-off needs to be found to detect the desynchronization and to preserve the maximum amount of information related to the relevant points.

4.4 Methodology

Let us assume a device under test (DUT) in which L leakages are detected using standard leakage detection such as SNR. Let us denote the following variables:

- $N^{[h]}$: the maximum amplitude of desynchronization after the h-th convolutional block (such that $N^{[0]}$ is the desynchronization associated with the input traces and $N^{[h]} = \frac{N^{[h-1]}}{\text{pooling\_stride}^{[h-1]}}$),
- $\text{pooling\_stride}^{[h]}$: the pooling stride associated to the h-th convolutional block,
- $D^{[h]}$: the trace dimension after the h-th convolutional block (such that $D^{[0]}$ is the input dimension) and in most cases, $D^{[h]} = \frac{D^{[h-1]}}{\text{pooling\_stride}^{[h-1]}}$.

4.4.1 Synchronized traces

According to the observation made in Section 4.3, adding convolutional blocks reduces the distance between the features to facilitate the detection of desynchronization. When we want to build an architecture such that traces in the training, validation and test sets are synchronized (*i.e.*$N^{[0]} = 0$), we do not need to configure more than one convolutional block. Indeed, using more than one convolutional block has two major drawbacks. First, in the case when PoIs are close to each other temporally, adding convolutional blocks increases the risk of entanglement. As we showed in Section 4.2, entanglement can generate a spreading of relevant information through the convoluted time samples. Secondly, because of the pooling (see Section 4.3), some information can be lost: the same most relevant feature can be spread over the samples (MaxPooling) or each relevant information can be reduced following the pooling stride value (Average Pooling). When no desynchronization occurs, we recommend setting the number of convolutional blocks to 1.
In [MPP16] and [PSB+18], the authors show that MLP can be a good alternative when an adversary wants to build a suitable architecture when traces are synchronized. However, MLPs are only composed of fully-connected layers. A CNN is viewed as a MLP where only each neuron of the layer l is linked with a set of neurons of the layer $l-1$ [Kle17]. In the side-channel context, only a few samples are needed for decision making. Using a CNN with short filter helps to focus its interest on local perturbations and dramatically reduces the complexity of the networks. It is thus recommended to use CNNs with the shortest filters possible (*i.e.*, 1). Moreover, thanks to the validation phase, an evaluator can estimate the best length in each context. Finally, the number of filters depends on the "imbalanced" sampling representation. For a set of synchronized traces, the distribution is uniform, so using a small number of filters is recommended (*i.e.*, $2, 4, 8$).

4.4.2 Desynchronization: Random delay effect

Let us denote $N^{[0]}$ the maximum amplitude of the random delay effect. We define $Nt_{GE}(model_{N^{[0]}})$ as the number of traces that a model, trained on desynchronized traces needs to converge towards a constant guessing entropy of 1 on test traces. Let us denote T_{test} this set of traces such that:

$$Nt_{GE}(model_{N^{[0]}}) := min\{t_{test} \mid \forall t \geq t_{test},\ g(k^*)(model_{N^{[0]}}(t)) = 1\}. \tag{11}$$

Adding desynchronization does not reduce the information contained in a trace set. However, retrieval of the secret is more challenging. In this section, we aim to find a methodology such that we optimize a $model_{N^{[0]}}$ where the associated $Nt_{GE}(model_{N^{[0]}})$ is as close as possible to $Nt_{GE}(model_0)$. In other words, we want a model to be able to handle desynchronisation with a minimal deterioration of its performances on synchronized

traces. To build a suitable architecture, we suggest using a new methodology that helps detect desynchronization and the reduction of trace samples so as to focus the network on the leakages (see Figure 5). We divide our convolutional part into three blocks. For a better understanding, we propose to apply the following methodology on toy example. We simulate traces such that the SNR peaks are given in Figure 21-e in Appendix F:

- As shown in Section 4.2, the first layer aims at minimizing the length of the filters in order to mimic the entanglement between each PoIs and extract the relevant information. Reducing the length of the filters helps the network to maximize the extremum of a trace to be able to easily extract secret information. Next, we suggest setting the length of the filters in the first convolutional block, to 1 in order to optimize the entanglement minimization. The first pooling layer is set to 2 in order to reduce the dimension of the trace and help detect desynchronization. In order to visualize the results, we compute the heatmaps between the input of the network and the filters that composed the first layer (see Figure 21-b in Appendix F). Therefore, we are able to evaluate which neurons are activated through this convolution.

- The second block tries to detect the value of the desynchronization. By applying filter length of size $\frac{N^{[0]}}{2}$, we focus the interest of the network on the detection of the desynchronization of each trace. The network obtains a global evaluation of the leakages by concentrating its detection on the leakage desynchronization and not on the leakages themselves. Then, we set the $pooling\_layer^{[1]}$ to $\frac{N^{[0]}}{2}$ to maximize the reduction of the trace dimension while preserving information related to the desynchronization. This process is illustrated in Figure 21-c in Appendix F. Thanks to the heatmaps, we evaluate that the SNR peaks are perfectly identified by the filters. Indeed, by comparing the activity detected by the convolution operation with the SNRs peaks, we can argue that the features are correctly selected by the filters.

- The third block aims at reducing the dimensionality of each trace in order to focus the network on the relevant points and to remove any irrelevant ones. In the case where our traces have L PoIs, then, only the L time samples help the network to make a decision. By dividing our trace in L different parts, we force the network to focus its interest on this information. Indeed, each part contains information related to a single spread leakage. This process reduces the dimensionality of each trace by L such that each point of the output of the convolutional block defines a leakage point. Furthermore, applying this technique limits the desynchronization effect because we force the network to concentrate the initial desynchronized PoIs into a single point. This property can be visualized in Figure 21-d in Appendix F. In our simulation, we can identify $L = 5$ SNR peaks. Therefore, we want to reduce the trace dimension into 5 samples such that each sample contains the information related to a single leakage. For example, the yellow point defines the relevant information containing in the first part of the trace (see Figure 21-e in Appendix F). Because we concentrate the network on the leakages, the yellow point contains the information related to the first peak. The other points define the information related to the other relevant features.

Discussion This methodology can be applied to unprotected AES, implementations where desynchronization occurs and boolean masking scheme (see experiments on Section 5.2.3 and Section 5.3.2). Indeed, if the network is able to find the PoIs related to the mask and the masked values, therefore, the network will be able to recombine them by following the *difference absolute combining function* [PRB09] (see proof on Appendix G). Consequently, the methodology is not impacted by boolean masking schemes. However the fully-connected

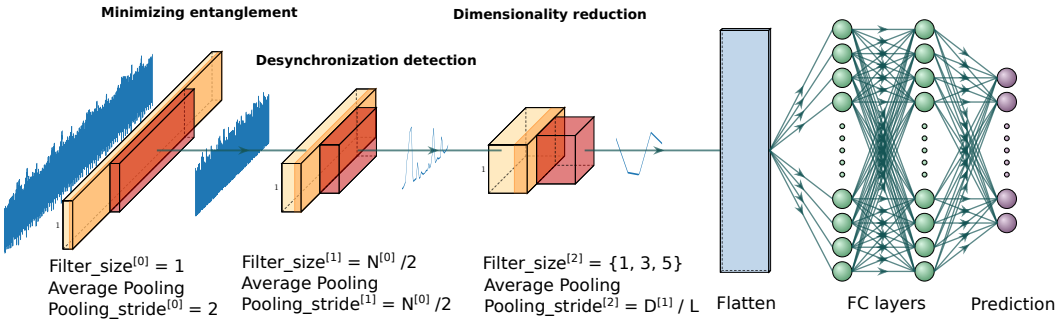

Figure 5: Architecture for desynchronized traces

layers have to be adapted following the masking implementation. This aspect should be a part of a future work.

5 Experimental results

In the following section, we apply our methodology on different publicly available datasets. The methodology is applied on both unprotected and protected implementations. We compare the performance of our architectures[b] with state-of-the-art results. For a good estimation of Nt_{GE}, the attack traces are randomly shuffled and 100 Nt_{GE} are computed to give the average value for Nt_{GE}. We denote $\bar{N}t_{GE}$ this average over the 100 tests. Because the attack efficiency is not the only metric, we also compare the complexity of our architectures with the state-of-the-art. Furthermore, when considering profiled SCA, the template attack (TA) is the best technique from a theoretical point of view. Therefore, we will compare our result with the TA that are performed on the same datasets. In the following sections, the hyperparameters related to the fully-connected layers (e.g. number of neurons and layers), the batch size and the number of epochs follow the grid search optimization defined in Section 4.1.

5.1 Datasets

We used four different datasets for our experiments. All the datasets correspond to implementations of *Advanced Encryption Standard* (AES) [DR02]. The datasets offer a wide range of use cases: high-SNR unprotected implementation on a smart card, low-SNR unprotected implementation on a FPGA, low-SNR protected with a random delay desynchronization, low-SNR protected implementation with first-order masking [SPQ05] and desynchronization with a random delay.

- **DPA contest v-4** is an AES software implementation with a first-order masking [BBD+14][c]. Knowing the mask value, we can consider this implementation as unprotected and recover the secret directly. In this experiment, we attack the first round S-box operation. We identify each trace with the sensitive variable $Y^{(i)}(k^*) = Sbox[P_0^{(i)} \oplus k^*] \oplus M$ where M denotes the known mask and $P_0^{(i)}$ the first byte of the i-th plaintext. The measured SNR equals 4.33 (see Appendix E Figure 17).
- **AES_HD** is an unprotected AES-128 implemented on FPGA. Introduced in [PHJ+19][d], the authors decided to attack the register writing in the last round

[b] https://github.com/gabzai/Methodology-for-efficient-CNN-architectures-in-SCA
[c] http://www.dpacontest.org/v4/42_traces.php
[d] https://github.com/AESHD/AES_HD_Dataset

such that the label of the i-th trace is $Y^{(i)}(k^*) = Sbox^{-1}[C_j^{(i)} \oplus k^*] \oplus C_{j'}^{(i)}$ where $C_j^{(i)}$ and $C_{j'}^{(i)}$ are two ciphertext bytes associated with the i-th trace, and the relation between j and j' is given by the ShiftRows operation of AES. The authors use $j = 12$ and $j' = 8$. The measured SNR equals 0.01554 (see Appendix E Figure 18).

- **AES_RD** is obtained from an 8-bit AVR microcontroller where a random delay desynchronization is implemented [CK09][e]. This countermeasure shifts each trace following a random variable of 0 to $N^{[0]}$. This renders the attack more difficult because of the misalignment. Like DPA-contest v4, the sensitive variable is the first round S-box operation where each trace is labeled as such $Y^{(i)}(k^*) = Sbox[P_0^{(i)} \oplus k^*]$. The measured SNR equals 0.0073 (see Appendix E Figure 19).
- **ASCAD** is introduced in [PSB+18] and is the first open database [f] that has been specified to serve as a common basis for further works on the application of deep learning techniques in the side-channel context. The target platform is an 8-bit AVR microcontroller (ATmega8515) where a masked AES-128 is implemented. In addition, a random delay is applied to the set of traces in order to make it more robust against side-channel attacks. The leakage model is the first round S-box operation such that $Y^{(i)}(k^*) = Sbox[P_3^{(i)} \oplus k^*]$. As explained in [PSB+18], the third byte is exploited. The measured SNR equals 0.007 (see Appendix E Figure 20).

Remark 1: the learning rate. During the training phase, for each architecture, we use a technique called *One Cycle Policy* [ST17, Smi17, Smi18] that helps choose the right *learning rate*. The learning rate (LR) is one of the most challenging hyperparameters to tune because it defines the learning time and the robustness of the training. If it is too low, the network will take a long time to learn. If it is too high, each learning step will go beyond the loss minimum. The *One Cycle Policy* provides very rapid results to train complex models. Surprisingly, using this policy means we can choose much higher learning rates and significantly reduce learning time while preventing overfitting.

Remark 2. Our methodology is applied only on very small traces (i.e., less than $10,000$ time samples) where few sensitive variables leak (i.e., less than 10). It could be very interesting to pursue our study on larger traces, where more leakages occur, to investigate if our methodology can detect all of the relevant features.

5.2 Results on synchronized traces

As explained in Section 5.2, only 1 convolutional block is set for each network. Using our methodology, we want to locate local perturbations that generate confusion during the classification of the network. Thus, the length of the filters is set to 1. Finally, the number of filters and the dense layers that compose the classification part should be managed depending on the device under test. To accelerate the learning phase, we also pre-processed each dataset such that all trace samples are standardized and normalized between 0 and 1 [LBOM12].

5.2.1 DPA-contest v4

DPA-contest v4 is the easiest dataset because we consider it without countermeasures, consequently the relevant information leak a lot. Thus, it is straightforward for the network to extract sensitive variables. To generate our CNN, we divide the dataset into three subsets such that $4,000$ traces are used for the training, 500 for the validation set and 500

[e] https://github.com/ikizhvatov/randomdelays-traces
[f] https://github.com/ANSSI-FR/ASCAD

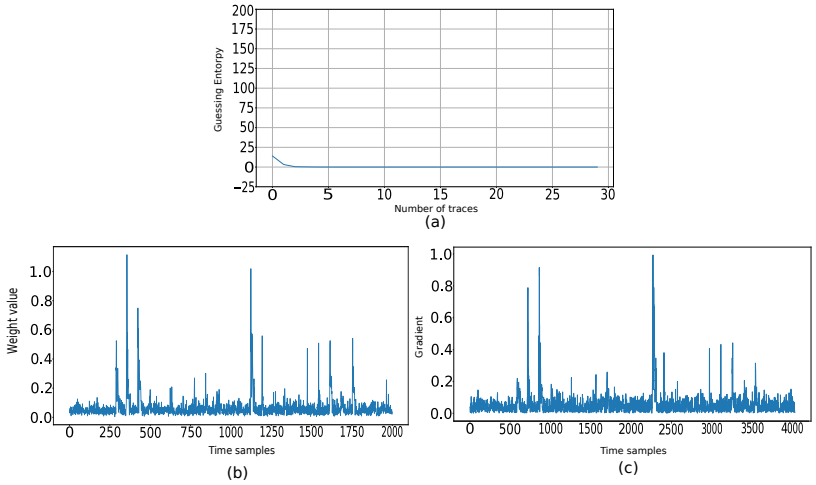

Figure 6: (a) Guessing entropy result ; (b) Weight visualization ; (c) Gradient visualization for the DPA-contest v4 dataset

for attacking the device. Only two filters are used and the classification part is composed of one dense layer of 2 nodes. Finally, we set our learning rate to 10^{-3}. The network is trained for 50 epochs with a batch size of 50.
Visualizations help evaluate the training process (see Figure 6). As explained in section Section 3.2, visualization of the weights shows that the convolutional part is accurately set because the PoIs are extracted. Furthermore, by visualizing the gradient, we can consider that the classification part is also working accurately because the PoIs detected at the end of the network are the same as those recognized by weight visualization. Thus, no information is lost between the convolutional part and the end of the network.
We compare our result with [PHJ+19] in which the authors published the best performance on DPA-contest v4 using deep learning techniques. The results related to $\bar{N}t_{GE}$ seem to be similar. However, when comparing the complexity of the networks, we verify that the complexity associated with our network is 6 times lower than the previous best architecture. By reducing the complexity of the network, we achieve a remarkable reduction in learning time (see Table 2). In conclusion, our network is more appropriate than the state-of-the-art. Finally, our performance is equivalent to the template attacks performed by Kim et al. [KPH+19].

Table 2: Comparison of performance on DPA-contest v4

	Template attacks ([KPH+19])	**State-of-the-art** ([PHJ+19])	**Our methodology** (Section 4.4)
Complexity (trainable parameters)	/	52,112	8,782
$\bar{N}t_{GE}$	3	4	3
Learning time (seconds)	/	1,000	23

5.2.2 AES_HD

We use 75,000 measurements such that 50,000 are randomly selected for the training process (45,000 for the training and 5,000 for the validation) and we use 25,000 traces for the attack phase. We set the number of filters to 2. Because the PoIs have to be accurately revealed, we minimize the classification part with 1 dense layer of 2 neurons. Finally, we set our learning rate to 10^{-3}. The training runs for 20 epochs with a batch size of 256.
Looking at the Figure 7, we can conclude that our model is not optimized. Indeed, the weight visualization shows us that the selection of features is effective because our network

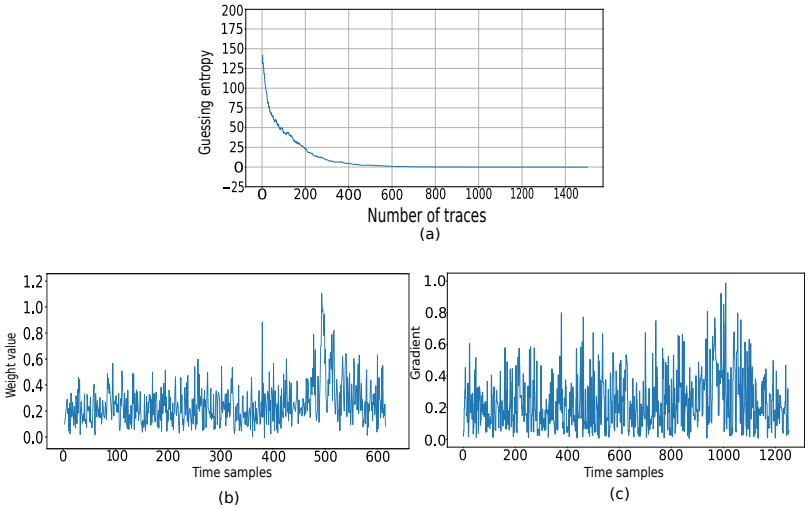

Figure 7: (a) Guessing entropy result ; (b) Weight visualization ; (c) Gradient visualization for the AES_HD dataset

can detect the relevant leakage points. When we look at the SNR value (see Figure 18 in Appendix E), the sensitive information leaks between samples 950 and 1,050 which corresponds to the features detected by the convolutional layer. However, by visualizing the gradient, the global learning achieved by our network does not enable it to significantly recognize the relevant points. Thanks to these visualization techniques, we can identify which part of the network needs optimizing. Currently, no tool is available that allows interpretation of the classification part to improve its efficiency. However, even if the classification part is not powerful, our network still performs much better than the state-of-the-art.

We compare our results with the architecture proposed in [KPH+19] where they achieved the best performance on this dataset. On average, $\bar{N}t_{GE}$ reaches 25,000 traces. By applying our methodology, we dramatically improve this result. First, the new architecture has 31,810 times fewer parameters. Now, only 31 seconds are necessary to train the network. Finally, our network outperforms the ASCAD network by getting $\bar{N}t_{GE}$ around 1,050 (see Table 3). By comparing our $\bar{N}t_{GE}$ result with the template attacks performed by Kim et al. [KPH+19], we can note that the performance related to the deep learning techniques is more efficient. Indeed, Kim et al. selected 50 features with highest absolute correlation coefficient between the measurement traces and the corresponding leakage. Through this preprocessing, the authors could discard relevant information relative to the target variable while the deep learning approach analyze the entire trace in order to extract the most important features.

Table 3: Comparison of performance on AES_HD

	Template attacks ([KPH+19])	**State-of-the-art** ([KPH+19])	**Our methodology** (Section 4.4)
Complexity (trainable parameters)	/	104,401,280	3,282
$\bar{N}t_{GE}$	25,000	25,000	1,050
Learning time (seconds)	/	6,075	31

5.2.3 ASCAD with $N^{[0]} = 0$

Following the previous studies, we can find a suitable architecture for the ASCAD database. To generate our network, we divide the dataset of ASCAD into three subsets: 45,000

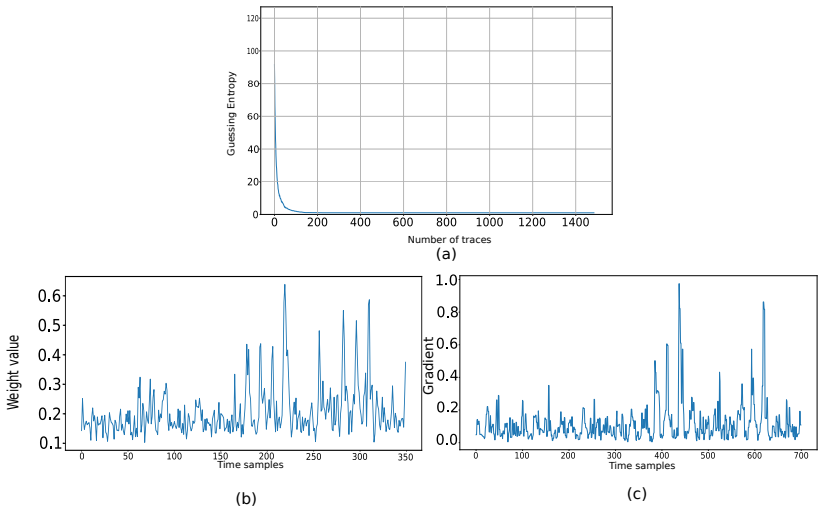

Figure 8: (a) Guessing entropy result ; (b) Weight visualization ; (c) Gradient visualization for the ASCAD (traces synchronized) dataset

traces for the training set, 5,000 for the validation set and 10,000 for the attack phase. We initialize the number of filters to 4. Then, two dense layers composed of 10 neurons complete the network. Thanks to the one cycle policy, we are able to configure our learning rate in the range $\left[5 \times 10^{-4}; 5 \times 10^{-3}\right]$. The network is trained for 50 epochs with a batch size of 50.

Next, we compare our result with the original paper [PSB+18]. Surprisingly, we can notice that the network detects more PoIs than the mask and the masked value. By visualizing the weights, we can evaluate that the network recognizes relevant information related to the mask and the masked values, although a sensitive area (between samples 380 and 450) is detected by the network whereas nothing appears on the SNR. Like the DPA-contest v4 experiment, the weight visualization and the gradient visualization are similar. The classification part appears to be optimized because no feature information is lost between the end of the convolutional part and the prediction layer.

When we compare the new performances, we notice that our new network is 3,930 times less complex than the original paper. Finally, in terms of performance, our simpler network achieves $\bar{N}t_{GE}$ in 191 traces while the original network reaches the same performance only after 1,146 traces (see Table 4). As well as the previous section, the deep learning approach is more efficient than the standard profiling attacks (e.g. template attacks). Indeed, Prouff et al. [PSB+18] performed template attacks with a PCA to reduce the trace dimension from 700 to 50. Therefore, some relevant information could be discarded because of this processing. Consequently, our methodology is twice more efficient than the state-of-the-art template attacks.

Table 4: Comparison of performance on ASCAD with $N_0 = 0$

	Template attacks ([PSB+18])	**State-of-the-art** ([PSB+18])	**Our methodology** (Section 4.4)
Complexity (trainable parameters)	/	66,652,444	16,960
$\bar{N}t_{GE}$	450	1,146	191
Learning time (seconds)	/	5,475	253

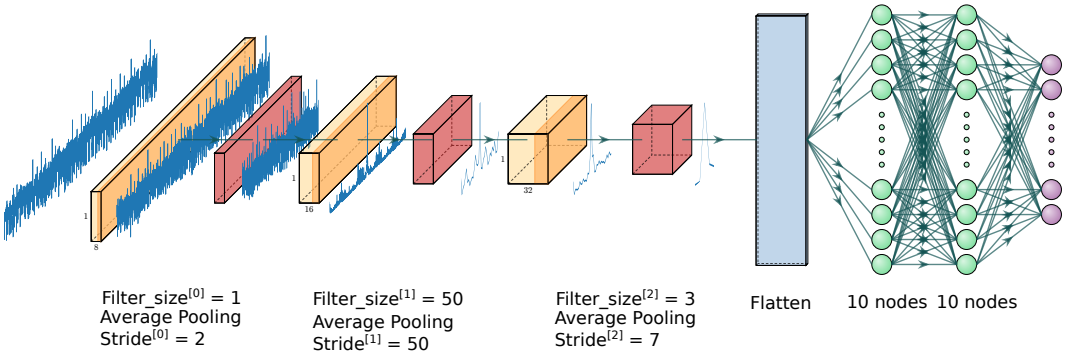

Figure 9: Architecture for AES_RD dataset

5.3 Results on desynchronized traces

After demonstrating that our methodology is efficient on synchronized traces for unprotected AES and 1^{st}-order masked AES, we decide to test our methodology on systems that implement random delay desynchronization.

5.3.1 AES_RD

The dataset consists of $50,000$ traces of $3,500$ features each. For this experiment, we use $20,000$ traces for the training set, $5,000$ for the validation set and $25,000$ to attack the device. Because desynchronization is generated, we have to use more than one convolutional block. According to Section 4.4.2, we recommend using three convolutional blocks. Concentrating the attention of the network on the PoIs is helpful in order to reduce the desynchronization effect. Our new architecture can be set as shown in Figure 5. Each hyperparameter, defining the convolutional part, is set as mentioned in Section 4.4.2. In the second convolutional block, the length of the filters and the pooling stride are configured following the value of the desynchronization effect (i.e., $N^{[1]} = \frac{N^{[0]}}{pooling\_stride^{[0]}}$). Through the SNR computation (see Figure 19), we are not able to define the number of leakages L. Therefore, we assume L large (e.g. $L = 5$) to make sure that the relevant leakage is extracted from this final convolutional block. Therefore, $pooling\_stride^{[2]}$ is set to 7 in order to minimize the number of relevant samples (i.e., $\frac{D^{[1]}}{L} = \frac{dim(input\_trace)}{pooling\_stride^{[0]} \times pooling\_stride^{[1]} \times L} = \frac{3,500}{2 \times 50} \times \frac{1}{5} = 7$). More detail about these hyperparameters are provided in Table 8.

Then, two fully-connected layers composed of 10 neurons define the classification part (see Figure 9). Thanks to the one cycle policy, we are able to set our learning rate in the range $\left[10^{-4}; 10^{-3}\right]$. The network is trained for 50 epochs with a batch size of 50.

To evaluate the feature selection, we visualize the heatmaps associated with each layer in the convolutional part. We can evaluate filter selection by analyzing the heatmaps in Figure 10, . As explained in Section 4.4.2, our methodology reduces the dimension of each trace in order to focus feature selection on the relevant points. Other points are discarded. Information aggregation for the classification part is then easier.

When our methodology is applied, the impact of the desynchronization is dramatically reduced and the performance related to the network will also be less affected by this countermeasure. Compared with the state-of-the-art [KPH+19], the performance of $\bar{N}t_{GE}$ is similar but the complexity related to our network proposal is greatly reduced (40 times smaller).

One of the most powerful countermeasure against template attacks is desynchronization. While a deep learning approach can prevent the desynchronization effect [CDP17], the template attacks are drastically impacted by this countermeasure. Consequently, we cannot

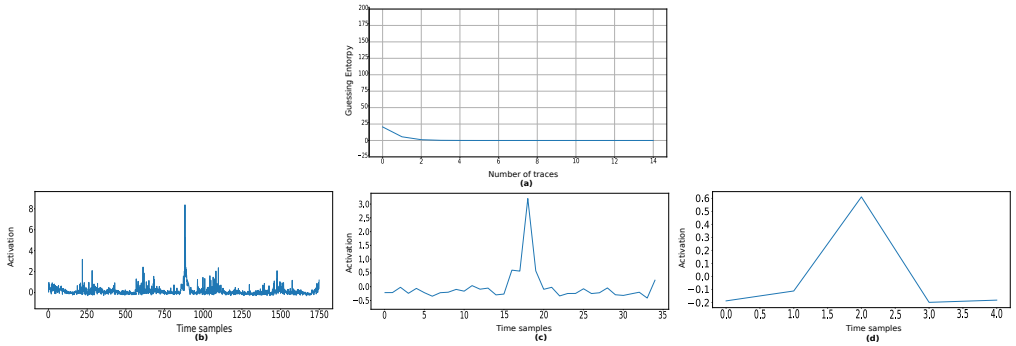

Figure 10: (a) Guessing entropy ; (b) Output 2^{nd} convolutional layer ; (c) Output 3^{rd} convolutional layer ; (d) Output 3^{rd} convolutional block for AES_RD dataset

perform template attacks on this dataset while only 5 traces are needed to recover the right key value when our methodology is performed.

Table 5: Comparison of performance on AES_RD

	Template attacks ([KPH+19])	**State-of-the-art** ([KPH+19])	**Our methodology** (Section 4.4)
Complexity (trainable parameters)	/	512,711	12,760
$\bar{N}t_{GE}$	> 500	10	5
Learning time (seconds)	/	4,500	380

5.3.2 ASCAD

Finally, we test our methodology on a system that implements random delay and 1^{st} order-masked AES. We evaluate our methodology when $N^{[0]} = 50$ and $N^{[0]} = 100$. Like in Section 5.2.3, we split our dataset into three subsets: 45,000 traces for the training process, 5,000 for the validation set and 10,000 traces for the attack.

Random delay: $N^{[0]} = 50$. By applying our new methodology, we want to generate a suitable architecture according to Section 4.4.2. Thanks to Figure 8, we set $L = 3$ because three global leakage areas appear. As the previous section, the hyperparameters has configured following Section 4.4.2. In this dataset, $N^{[0]} = 50$, the length of the filters and the pooling stride defining in the second convolutional block are configured as $N^{[1]} = \frac{N^{[0]}}{pooling\_stride^{[0]}}$. Thus, the network focuses its interest on the detection of the desynchronization effect. In order to preserve the information related to the leakages, we set $pooling\_stride^{[2]}$ to 4. This results in an output with 3 samples. More detail about these hyperparameters are provided in Table 8. The best performance is obtained when we configure three fully-connected layers with 15 neurons. We apply a range of learning rate between $\left[5 \times 10^{-4}; 5 \times 10^{-3}\right]$ and we set the number of epochs to 50 with a batch size 256.

To check that the convolutional part is correctly customized, we visualize the heatmaps to evaluate feature selection. The heatmaps (see Figure 11) show that the network recognizes some influential patterns for classification. Most of these patterns appear to correspond to the different leakages revealed by Figure 8.

Applied on this dataset, our methodology outperforms the state of the art remarkably. In the original paper, Prouff et al. did not reach a constant guessing entropy of 1 with 5,000 traces. Applying our methodology, we converge towards a constant guessing entropy of 1 with 244 traces while the complexity of our networks is divided by 763. As a reminder, the performance related to the network trained with synchronized traces converges towards a guessing entropy of 1 with around 200 traces. Thus, we succeed in dramatically reducing

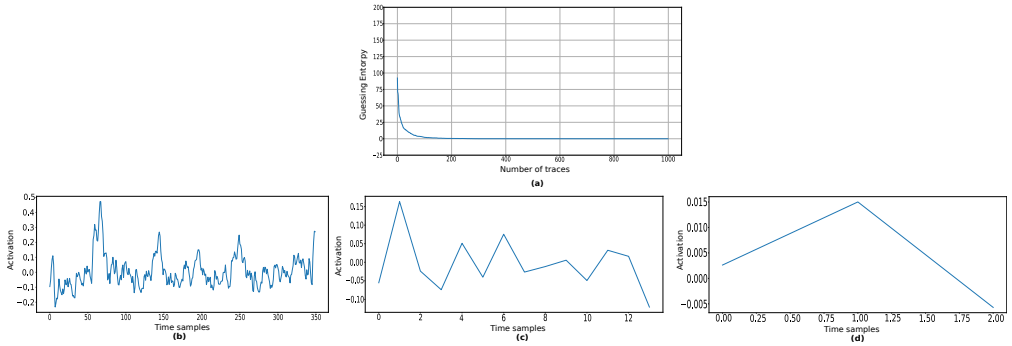

Figure 11: (a) Guessing entropy ; (b) Output 2^{nd} convolutional layer ; (c) Output 3^{rd} convolutional layer ; (d) Output 3^{rd} convolutional block for ASCAD dataset with $N^{[0]} = 50$

the impact of the random delay effect. Obviously, the network can be optimized with random search optimization, data augmentation or by adding noise.

Table 6: Comparison of performance on ASCAD with $N^{[0]} = 50$

	Template attacks ([PSB+18])	**State-of-the-art** ([PSB+18])	**Our methodology** (Section 4.4)
Complexity (trainable parameters)	/	66,652,444	87,279
$\bar{N}t_{GE}$	4,900	> 5,000	244
Learning time (seconds)	/	5,475	380

Random delay: $N^{[0]} = 100$. Finally, we apply our methodology on an even more complex system implementing a random delay with $N^{[0]} = 100$. As the previous sections, we set our network with the following hyperparameters. In the second convolutional block, we set the length of the filters and the pooling stride to $N^{[1]} = 50$ to focus the interest of the network on the desynchronization effect. In order to preserve the information related to the leakages, we set $pooling\_stride^{[2]}$ to 2. This results in an output with 3 samples. More detail about these hyperparameters are provided in Table 8.

Thanks to the one cycle policy, we define our learning rate in the range $\left[10^{-3}; 10^{-2}\right]$ to allow a robust training and to reach a good local minimum. As before, our network is trained during 50 epochs with a batch size 256.

Thanks to the visualization tools (see Figure 12), we can interpret our network and conclude that it is suitable for our evaluation because the gradient visualization and heatmaps are similar and show the same relevant points as the SNR.
Our network is far less complex than the architecture introduced in the original ASCAD paper and $\bar{N}t_{GE}$ is outperformed. Indeed, we converge towards a constant guessing entropy of 1 with 270 traces while the original paper couldn't converge with 5,000 traces. Furthermore, by reducing the complexity, we can train our network much faster. Comparing this performance with Section 5.2.3, we note that our methodology dramatically reduce the impact of the desynchronization. Indeed, the performance of the two models is similar, thus, our methodology eliminates the effect of the random delay countermeasure.

In the both cases ($N^{[0]} = 50$ and $N^{[0]} = 100$), we are not able to perform efficient template attacks because of the desynchronization effect. As mentioned Cagli et al. [CDP17], the deep learning approach is more suitable when desynchronization occurs.

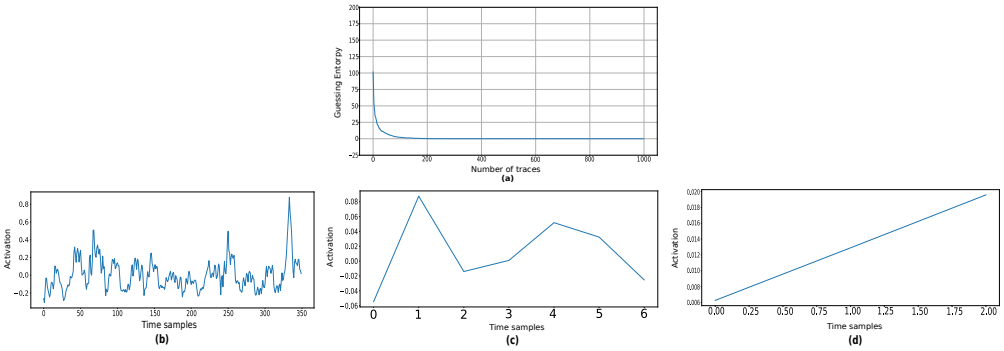

Figure 12: (a) Guessing entropy ; (b) Output 2^{nd} convolutional layer ; (c) Output 3^{rd} convolutional layer ; (d) Output 3^{rd} convolutional block for ASCAD dataset with $N^{[0]} = 100$

Table 7: Comparison of performance on ASCAD with $N^{[0]} = 100$

	Template attacks ([PSB+18])	**State-of-the-art** ([PSB+18])	**Our methodology** (Section 4.4)
Complexity (trainable parameters)	/	$66,652,444$	$142,044$
$\tilde{N}t_{GE}$	$> 5,000$	$> 5,000$	270
Learning time (seconds)	/	$5,475$	512

Table 8: Convolutional hyperparameters for each dataset

	layers	*filter_size*	*pooling_stride*	*number_filter*
AES_RD	1^{st} convolutional block	1	2	8
	2^{nd} convolutional block	$\frac{N^{[0]}}{pooling\_stride^{[0]}} = \frac{100}{2} = 50$	$\frac{N^{[0]}}{pooling\_stride^{[0]}} = \frac{100}{2} = 50$	16
	3^{rd} convolutional block	3	$\lfloor \frac{D^{[1]}}{L} \rfloor = 7$	8
ASCAD ($N^{[0]} = 50$)	1^{st} convolutional block	1	2	32
	2^{nd} convolutional block	25	25	64
	3^{rd} convolutional block	3	4	128
ASCAD ($N^{[0]} = 100$)	1^{st} convolutional block	1	2	32
	2^{nd} convolutional block	50	50	64
	3^{rd} convolutional block	3	2	128

6 Discussion

The methodology proposed in this work has been recently questioned by [WAGP20]. However this revisit is based on wrong assumptions and misinterpretations. Hence, many of the claims of [WAGP20] are unfounded regarding our propositions. In [ZBHV20], we clear out the potential misunderstandings and explain more thoroughly our contributions. We discuss the difference between the gradient visualization and the weight visualization tools. We show how [WAGP20] misinterprets the weight visualization by introducing the concepts of *Leakage Detection*, *Leakage Exploitation* and *Network Confidence*. Then, we use these tools to link the concepts of *Entanglement* and *Receptive Field*. [WAGP20] wrongly claims that we correlate the network performance with the number of convolutional blocks. Then, we correct the methodology applied to the AES_RD dataset.

7 Conclusion

In this paper, we analyze the interpretability of convolutional neural networks. We interpret the impact of different hyperparameters that compose the feature selection part in order to generate more suitable CNNs. To evaluate our methodology, we introduce two visualization tools called weight visualization and heatmaps that help analyze which patterns are the

most influential during the training process. These patterns are similar to PoIs which are well known in the side-channel context.
We demonstrate the theoretical effect of the length of filters and the number of convolutional blocks and using these visualization techniques in order to check the validity of our demonstrations. The visualization techniques help us find a method of generating an appropriate CNN architecture according to the countermeasures implemented. If an evaluator wants to generate a network for synchronized traces, we recommend using only one convolutional block and minimizing the length of the filters to enable accurate identifying the relevant information. However, when desynchronization is applied, we propose an architecture that enables the network to focus its feature selection on the PoIs themselves while the dimensionality of the traces is reduced. This dramatically reduce the complexity of the CNN.
We test our methodology on four datasets with a wide range of use-cases. In all cases, our methodology outperforms the state-of-the-art. Indeed, our attacks are performed much faster and each new architecture is less complex than the state of-the-art. This paper thus shows that the networks needed to perform side channel attacks do not have to be complex. This methodology is not focused on optimizer hyperparameters and could benefit from techniques such as data augmentation [CDP17] or adding noise [KPH+19].
In some cases, we show that the aggregation of information through the classification part could be difficult. Future works could study this issue to generate suitable fully-connected layers. More complex systems with dynamic desynchronization or in presence of higher order masking scheme could be analyzed. Finally, other masking scheme should be investigated in order to validate our methodology on other masking implementation.

Acknowledgement

The authors would like to thank François Dassance for fruitful discussions about this work. We would also like to thank Benoît Gérard and the anonymous reviewers for their valuable comments which helped to improve this work.

References

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side—channel(s). In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[APSQ06] Cédric Archambeau, Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Template attacks in principal subspaces. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2006.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pages 2546–2554, 2011.

[BBD+14] Shivam Bhasin, Nicolas Bruneau, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. Analysis and improvements of the dpa contest v4 implementation. In Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 201–218, Cham, 2014. Springer International Publishing.

[BBM+15] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10(7):1–46, 07 2015.

[BCN18] L. Bottou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[BL12] Timo Bartkewitz and Kerstin Lemke-Rust. Efficient template attacks based on probabilistic multi-class support vector machines. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2012.

[BPK92] H. Bischof, A. Pinz, and W. G. Kropatsch. Visualization methods for neural networks. In *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*, pages 581–585, Aug 1992.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[CDP16] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Kernel discriminant analysis for information extraction in the presence of masking. In Kerstin Lemke-Rust and Michael Tunstall, editors, *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*, volume 10146 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2016.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 45–68. Springer, 2017.

[CK09] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 156–170, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 13–28, London, UK, UK, 2003. Springer-Verlag.

[CUH16] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[HGG19] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Deep neural network attribution methods for leakage analysis and symmetric key recovery. *IACR Cryptology ePrint Archive*, 2019:143, 2019.

[HGM+11] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *J. Cryptographic Engineering*, 1(4):293–302, 2011.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[HOWT06] Geoffrey Hinton, Simon Osindero, Max Welling, and Yee-Whye Teh. Unsupervised discovery of nonlinear structure using contrastive backpropagation. *Cognitive science*, 30:725–31, 07 2006.

[HZ12] Annelie Heuser and Michael Zohner. Intelligent machine homicide - breaking cryptographic devices using support vector machines. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2012.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034, Washington, DC, USA, 2015. IEEE Computer Society.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[Kle17] Matthew Kleinsmith. CNNs from different viewpoints - prerequisite : Basic neural networks, 2017.

[KPH+19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):148–179, 2019.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 971–980. Curran Associates, Inc., 2017.

[KW52] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[LBLL09] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *J. Mach. Learn. Res.*, 10:1–40, June 2009.

[LBM14] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Power analysis attack: an approach based on machine learning. *IJACT*, 3(2):97–115, 2014.

[LBOM12] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[LPMS18] Liran Lerman, Romain Poussier, Olivier Markowitch, and François-Xavier Standaert. Template attacks versus machine learning revisited and the curse of dimensionality in side-channel analysis: extended version. *J. Cryptographic Engineering*, 8(4):301–313, 2018.

[Man04] Stefan Mangard. Hardware countermeasures against DPA ? A statistical analysis of their effectiveness. In Tatsuaki Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.

[MDP19a] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. A comprehensive study of deep learning for side-channel analysis. *IACR Cryptology ePrint Archive*, 2019:439, 2019.

[MDP19b] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. Gradient visualization for general characterization in profiling attacks. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, volume 11421 of *Lecture Notes in Computer Science*, pages 145–167. Springer, 2019.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2016.

[OH08] Simon Osindero and Geoffrey E Hinton. Modeling image patches with a directed hierarchy of markov random fields. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1121–1128. Curran Associates, Inc., 2008.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[PEC19] Guilherme Perin, Baris Ege, and Lukasz Chmielewski. Neural network model assessment for side-channel analysis. Cryptology ePrint Archive, Report 2019/722, 2019. https://eprint.iacr.org/2019/722.

[PGZ+18] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *JMLR Workshop and Conference Proceedings*, pages 4092–4101. JMLR.org, 2018.

[PHJ+19] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):209–237, 2019.

[PRB09] E. Prouff, M. Rivain, and R. Bevan. Statistical analysis of second order differential power analysis. *IEEE Transactions on Computers*, 58(6):799–811, June 2009.

[PSB+18] Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, and Cécile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. *IACR Cryptology ePrint Archive*, 2018:53, 2018.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[Smi17] Leslie N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017, Santa Rosa, CA, USA, March 24-31, 2017*, pages 464–472. IEEE Computer Society, 2017.

[Smi18] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. *CoRR*, abs/1803.09820, 2018.

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009.

[SPQ05] François-Xavier Standaert, Eric Peeters, and Jean-Jacques Quisquater. On the masking countermeasure and higher-order power analysis attacks. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pages 562–567. IEEE Computer Society, 2005.

[ST17] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017.

[SVZ14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*, 2014.

[vdMH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[WAGP20] Lennert Wouters, Victor Arribas, Benedikt Gierlichs, and Bart Preneel. Revisiting a methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):147–168, Jun. 2020.

[ZBHV20] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Understanding methodology for efficient cnn architectures in profiling attacks. Cryptology ePrint Archive, Report 2020/757, 2020. https://eprint.iacr.org/2020/757.

[ZF14] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.

[ZKL+16] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2921–2929. IEEE Computer Society, 2016.

A Signal to Noise Ratio (Chipwhisperer dataset)

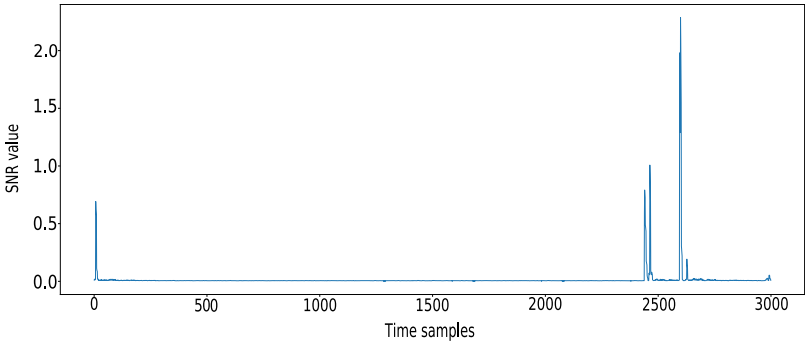

Figure 13: Signal to Noise Ratio related to the Chipwhisperer dataset

B Hyperparameter Architectures

Table 9: Networks (length of filters)

Hyperparameters	value
n° of blocks	1
n° of filters	2
length_filter	**[1,3,5,7,9,11,13,15,17,19,21,23,25,50,75,100]**
n° of FC_layers	1
n° of neurons_FC	4

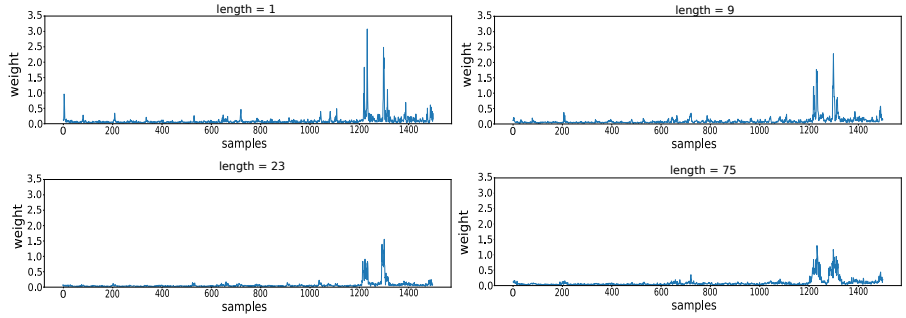

Figure 14: Impact of the filters length in the PoIs detection using weight visualization

C A study of pooling operations

Average pooling Let $\mathbf{t}^{[h]}$ be the input of the h^{th} convolutional block and $W^{[h]} \in \mathbb{R}^n$ the filter related to the h^{th} convolutional block. Let $ps^{[h]}$ be the pooling stride related to the h^{th} convolutional block. Assume that the j^{th} convoluted sample $(\mathbf{t}^{[h]} \circledast W^{[h]})[j]$ covers $l^{[h]}$ relevant information from the h^{th} convolutional block. The resulted $l^{[h]}_{conv}$ relevant convoluted samples are covered by the i^{th} pooling sample $AvgPool(\mathbf{t}^{[h]} \circledast W^{[h]})[i]$. Finally, the $(i+m)^{th}$ pooling sample $AvgPool(\mathbf{t}^{[h]} \circledast W^{[h]})[i+m]$ covers $(l+m)^{[h]}_{conv}$ PoIs such that the first $l^{[h]}_{conv}$ leakages are shared with $AvgPool(\mathbf{t}^{[h]} \circledast W^{[h]})[i]$. Assume that the most relevant leakage, denoted $x_{l^{[h]}_{conv,opt}}$, is included in both situation. Then,

$$
\begin{aligned}
AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i] &= \sum_{j=i}^{i+ps^{[h]}} \frac{\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j]}{ps^{[h]}} \\
&= \frac{1}{ps^{[h]}} \sum_{j=i}^{i+ps^{[h]}} \sigma\left(\sum_{k=0}^{n} \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right] \times W^{[h]}[k]\right) \\
&\approx \frac{1}{ps^{[h]}} \sum_{j=0}^{l_{conv}^{[h]}} \sigma\left(\sum_{\substack{k=0 \\ k\in\mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right]\right)\right).
\end{aligned}
$$

Because $k \in \mathbb{E}$, $\alpha_k^{[h]} \times \mathbf{t}^{[h]}[k]$ represent the most relevant features in the trace $\mathbf{t}$ then, $\alpha_k^{[h]} \times \mathbf{t}^{[h]}[k] > 0$. Following the SeLU function (see Equation 2), we can simplify our solution such that,

$$
\frac{1}{ps^{[h]}} \sum_{j=0}^{l_{conv}^{[h]}} \sigma\left(\sum_{\substack{k=0 \\ k\in\mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right]\right)\right) = \frac{\lambda}{ps^{[h]}} \sum_{j=0}^{l_{conv}^{[h]}} \sum_{\substack{k=0 \\ k\in\mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right]\right).
$$

Moreover,

$$
\begin{aligned}
AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i+m] &= \sum_{j=i+m}^{i+m+ps^{[h]}} \frac{\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j]}{ps^{[h]}} \\
&= \frac{1}{ps^{[h]}} \sum_{j=i+m}^{i+m+ps^{[h]}} \sigma\left(\sum_{k=0}^{n} \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right] \times W^{[h]}[k]\right) \\
&\approx \frac{\lambda}{ps^{[h]}} \sum_{j=0}^{(l+m)_{conv}^{[h]}} \sum_{\substack{k=0 \\ k\in\mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right]\right).
\end{aligned}
$$

then,

$$
\begin{aligned}
&AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i+1] - AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i] = \\
&\qquad \frac{\lambda}{ps^{[h]}} \sum_{j=(l+1)_{conv}^{[h]}}^{(l+m)_{conv}^{[h]}} \sum_{\substack{k=0 \\ k\in\mathbb{E}}}^{l_j^{[h]}} (\alpha_k^{[h]} \times \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right]).
\end{aligned}
$$

MaxPooling In the case where the MaxPooling is used, then,

$$
\begin{aligned}
MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i] &= max(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j] \mid j \in \{i, ..., i+ps^{[h]}\}) \\
&= x_{l_{conv,opt}^{[h]}}
\end{aligned}
$$

and,

$$
\begin{aligned}
MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i+m] &= max(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j] \mid j \in \{i+m, ..., i+m+ps^{[h]}\}) \\
&= x_{l_{conv,opt}^{[h]}}
\end{aligned}
$$

then,

$$
MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i+m] - MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})))[i] = 0.
$$

When MaxPooling is used, if two consecutive pooling computation share the same optimal leakage $x_{l_{conv,opt}^{[h]}}$, then this information is spread over the pooling samples.

D Number of convolutional blocks : Pooling

Table 10: Networks (number of convolutional blocks)

Hyperparameters	value
n° of blocks	**[1,2,3,4,5]**
n° of filters	2
length_filter	1
n° of FC_layers	1
n° of neurons_FC	4

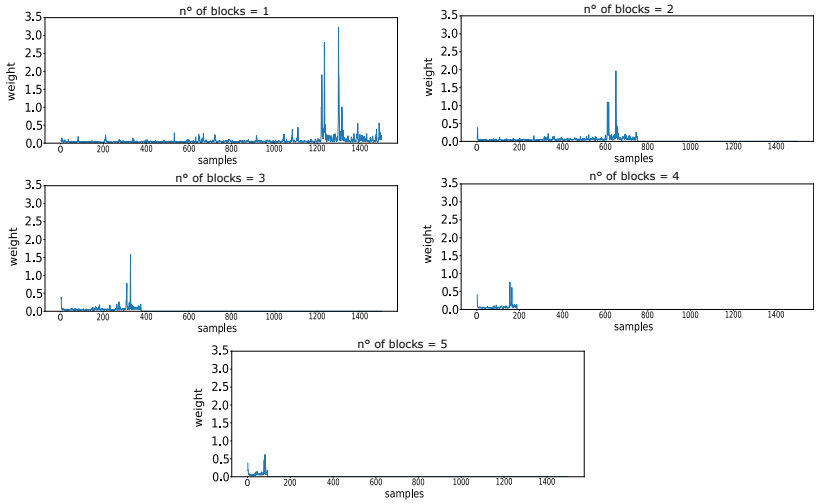

Figure 15: Impact of the number of convolutional blocks on the detection of PoIs (Average Pooling) using weight visualization

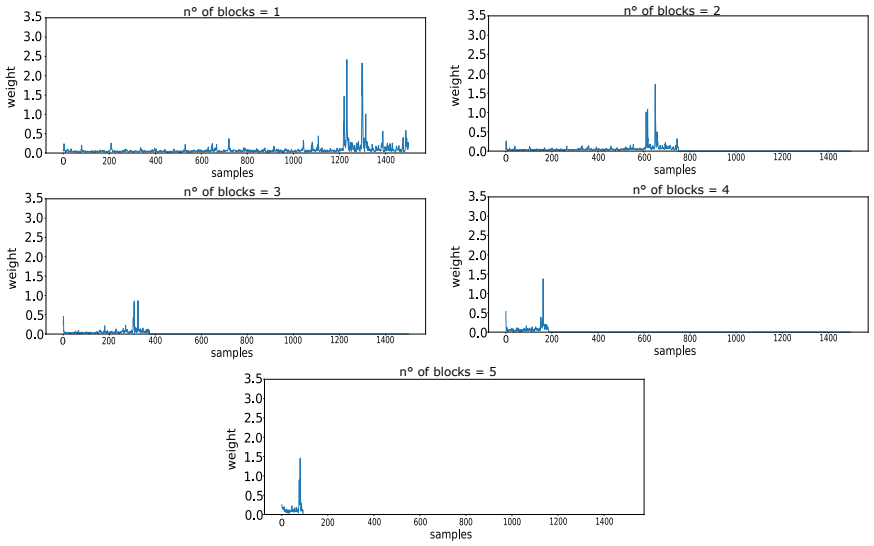

Figure 16: Impact of the number of convolutional blocks on the detection of PoIs (Max Pooling) using weight visualization

E Signal to Noise Ratio of the experimental datasets

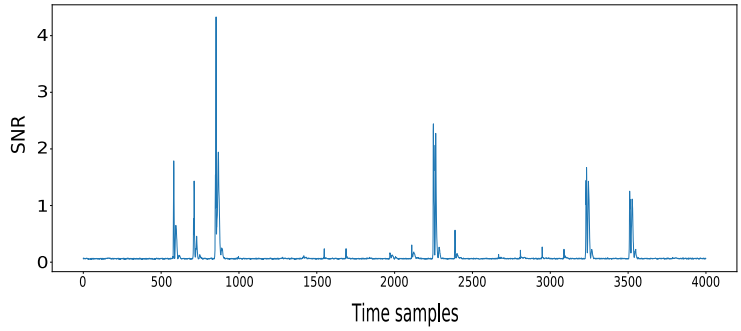

Figure 17: SNR of the **DPA-contest v-4** dataset

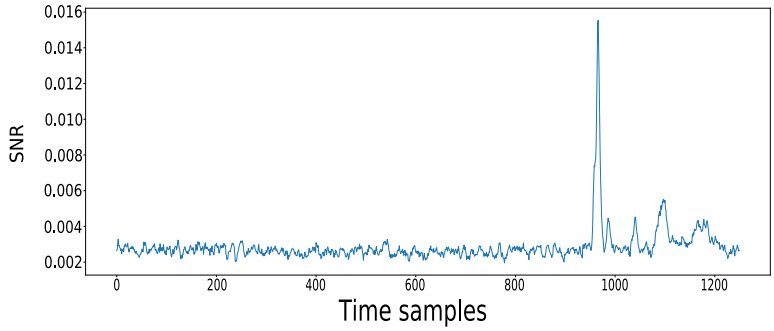

Figure 18: SNR of the **AES_HD** dataset

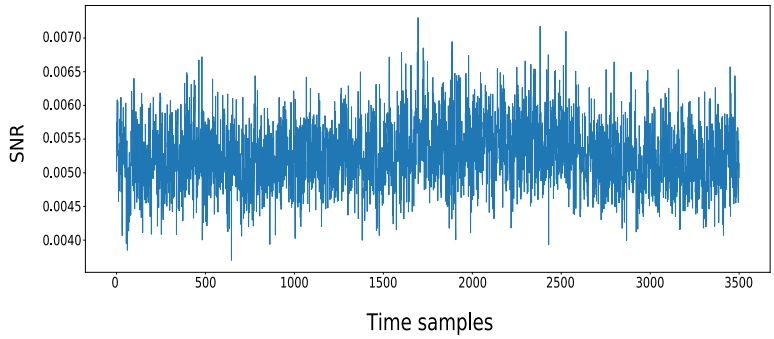

Figure 19: SNR of the **AES_RD** dataset

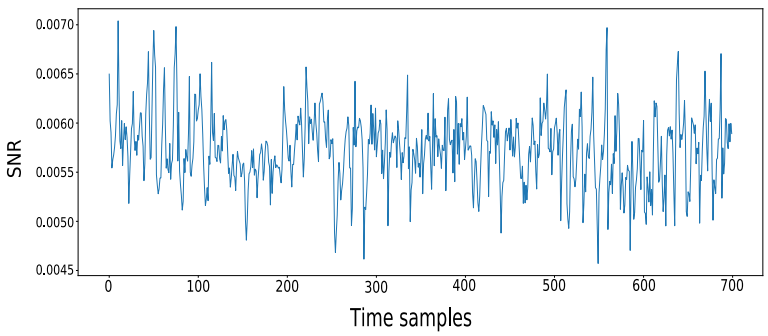

Figure 20: SNR of the **ASCAD** dataset

F Methodology for generating architecture (desynchronized traces)

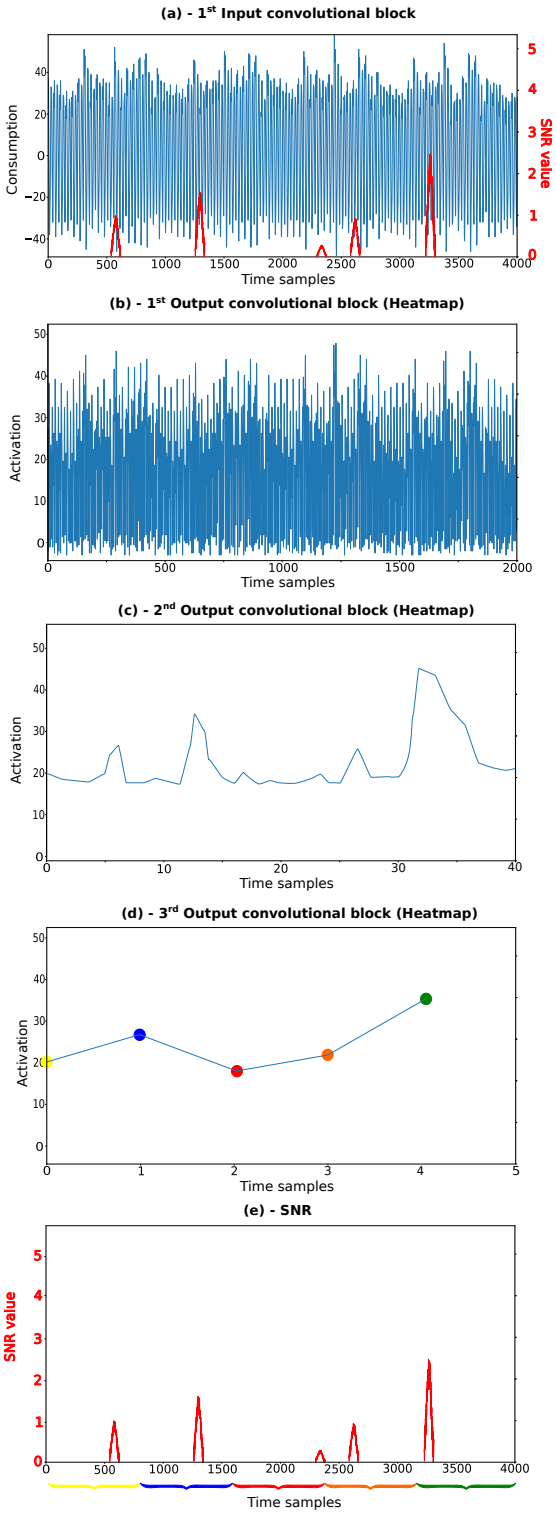

Figure 21: Output of each convolutional layer

G Combining function in DL-SCA

Let assume two leakages $L(t_1)$ and $L(t_2)$ such that:

$$L(t_1) = \delta_1 + HW(Z \oplus M) + B_1.$$

$$L(t_2) = \delta_2 + HW(M) + B_2.$$

where δ_1 and δ_2 denote the constant part of the leakages, $HW(.)$ is the Hamming Weight function, B_1 and B_2 are two gaussian random variables (centered in 0 with a standard deviation σ) and Z, M, B_1 and B_2 are mutually independent.

Let assume a model F that fully extracted the leakages $L(t_1)$ and $L(t_2)$ during the convolutional process such that l_1 (resp. l_2) neurons contain information related to $L(t_1)$ (resp. $L(t_2)$) (see Figure 22). Therefore,

$$\begin{aligned} n_i^{[flatten+1]} = \sum_{j=0}^{N}(w_j \times n_j^{[flatten]}) &= \sum_{j=u}^{u+l_1}(w_j \times n_j^{[flatten]}) + \sum_{j=v}^{v+l_2}(w_j \times n_j^{[flatten]}) \\ &= \sum_{j=u}^{u+l_1}(w_j \times (\delta_j + HW(Z \oplus M) + B_j)) + \sum_{j=v}^{v+l_2}(w_j \times (\delta_j + HW(M) + B_j)) \\ &= L(t_{l_1}) + L(t_{l_2}). \end{aligned}$$

where $L(t_{l_1}) = \sum_{j=u}^{u+l_1}(w_j \times (\delta_j + HW(Z \oplus M) + B_j))$ and $L(t_{l_2}) = \sum_{j=v}^{v+l_2}(w_j \times (\delta_j + HW(M) + B_j))$.

Because $(w_j)_{0 \leqslant j \leqslant N} \in \mathbb{R}$, $L(t_{l_1})$ and $L(t_{l_2})$ can be positive or negative. Therefore, if the network is perfectly trained, it can recombined the mask and the masked values following the *absolute difference combining function*. As mentioned by Prouff et al. [PRB09], this combination can be used in order to perform a high order attack.

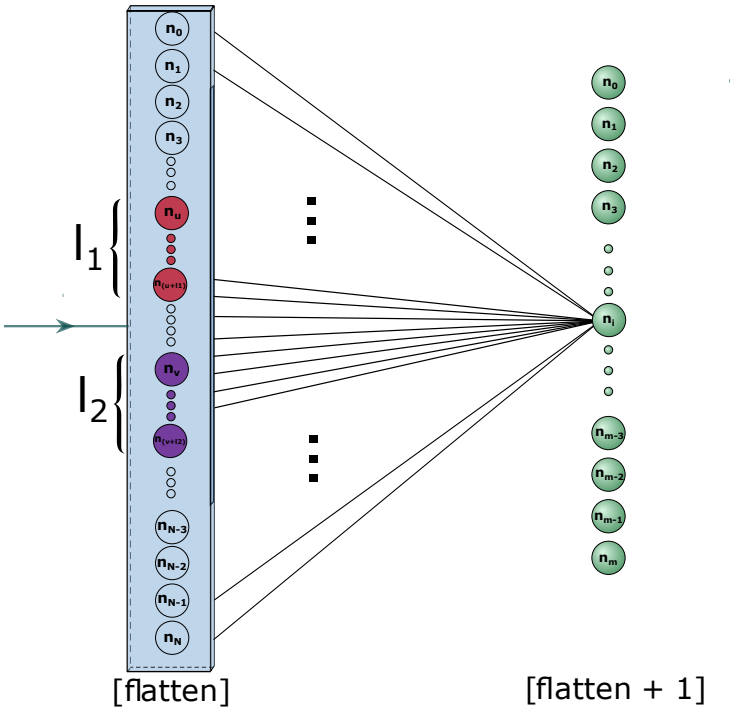

Figure 22: Recombination of the mask and the masked values through the classification part