# Non-Profiled Deep Learning-based Side-Channel attacks with Sensitivity Analysis

Benjamin Timon

eShard, Singapore
benjamin.timon@eshard.com

**Abstract.** Deep Learning has recently been introduced as a new alternative to perform Side-Channel analysis [MPP16]. Until now, studies have been focused on applying Deep Learning techniques to perform Profiled Side-Channel attacks where an attacker has a full control of a profiling device and is able to collect a large amount of traces for different key values in order to characterize the device leakage prior to the attack. In this paper we introduce a new method to apply Deep Learning techniques in a Non-Profiled context, where an attacker can only collect a limited number of side-channel traces for a fixed unknown key value from a closed device. We show that by combining key guesses with observations of Deep Learning metrics, it is possible to recover information about the secret key. The main interest of this method is that it is possible to use the power of Deep Learning and Neural Networks in a Non-Profiled scenario. We show that it is possible to exploit the translation-invariance property of Convolutional Neural Networks [CDP17] against de-synchronized traces also during Non-Profiled side-channel attacks. In this case, we show that this method can outperform classic Non-Profiled attacks such as Correlation Power Analysis. We also highlight that it is possible to break masked implementations in black-box, without leakages combination pre-preprocessing and with no assumptions nor knowledge about the masking implementation. To carry the attack, we introduce metrics based on Sensitivity Analysis that can reveal both the secret key value as well as points of interest, such as leakages and masks locations in the traces. The results of our experiments demonstrate the interests of this new method and show that this attack can be performed in practice.

**Keywords:** side-channel attacks · deep learning · machine learning · non-profiled attacks · profiled attacks · sensitivity analysis

## 1 Introduction

Side-Channel attacks, introduced in 1996 by P. Kocher [Koc96], exploit side-channel leakages such as power consumption from a device to extract secret information. Side-Channel attacks can be classified into two classes:

- *Profiled* Attacks such as Template Attacks [CRR03], Stochastic attacks [SLP05] or Machine-Learning-based attacks [HGDM+11, LPB+15, LBM15].
- *Non-Profiled* Attacks such as Differential Power Analysis (DPA) [KJJ99], Correlation Power Analysis (CPA) [BCO04], or Mutual Information Analysis (MIA) [GBTP08].

To mount a Profiled Side-Channel attack, an attacker needs to have access to a pair of identical devices called the *target* device and the *profiling* device. The attacker has a limited control over the target device which is running a cryptographic operation with a


IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2019, No. 2, pp. 107–131
DOI:10.13154/tches.v2019.i2.107-131

fixed unknown key value $\boldsymbol{k}^* \in \mathcal{K}$, where $\mathcal{K}$ is the set of possible key values. On the other hand, the attacker has a full control and knowledge of the inputs and keys of the profiling device. In such a context, a Profiled Attack is performed in two steps:

1. A profiling phase, while the leakage of the targeted cryptographic operation is profiled for all possible key values $k \in \mathcal{K}$ using side-channel traces collected from the profiling device.

2. An attack phase, where traces collected from the target device are classified based on the leakage profiling in order to recover the secret key value $\boldsymbol{k}^*$.

Profiled attacks are considered as the most powerful form of side-channel attacks as the attacker is able to characterize the side-channel leakage of the device prior to the attack. However, the profiling phase requires to have access to a profiling device, which is a strong assumption that cannot be always met in practice. Indeed, for *closed* products (for example smart cards running banking applications) an attacker does not have control of the keys and is usually limited by a transaction counter which caps the number of side-channel traces that can be collected. In such a context, Profiled attacks cannot be performed. However, Non-Profiled attacks such as DPA, CPA, or MIA can still threaten the device. The only assumption for Non-Profiled attacks is that the attacker is able to collect several side-channel traces of a cryptographic operation with a fixed unknown key value $\boldsymbol{k}^* \in \mathcal{K}$ and known random inputs (or outputs) from the targeted device. The attacker then combines key hypotheses with the use of statistical distinguishers such as Pearson's Correlation or Mutual Information to infer information about the secret $\boldsymbol{k}^*$ from the side-channel traces.

## 1.1 Motivation

Recently, Deep Learning has been introduced as an interesting alternative to perform Side-Channel attacks [MPP16, CDP17]. However, so far, the studies have only focused on applying Deep Learning to perform Profiled Side-Channel attacks. As mentioned previously, mounting a Profiled attack requires to have access to a profiling device, which is a strong assumption and limits the usage of Deep Learning techniques. The motivation of this research is to study how Deep Learning and deep neural networks can be used to perform Non-Profiled attacks.

## 1.2 Our contribution

In this paper we introduce a new side-channel attack method to apply Deep Learning techniques in Non-Profiled scenarios. The method that we present is a type of partition-based side-channel attack [SGV09] which uses Deep Learning trainings to reveal the correct key value. We show that using this method it is possible to use the power of Deep Learning for Non-Profiled attacks. We show that as in the Profiled context [CDP17] it is possible to use the translation-invariance property of Convolutional Neural Networks against de-synchronized traces also in a Non-Profiled attack setting. This leads to results showing that in some cases, this attack method can outperform other Non-Profiled attacks as CPA. Additionally, we show that this attack method can be used to break masked implementations with a reasonable number of traces, without leakages combination pre-processing and without knowledge nor assumptions about the implemented protections. To perform the attack, we propose to exploit a set of techniques from the literature called Sensitivity Analysis to reveal the secret key as well as points of interest such as leakage and masks locations in the traces. In this paper, we focus on the application of Sensitivity Analysis in a Non-Profiled context, even though the same technique can be used in a Profiled context to reveal points of interest as well. All these points are supported by

experiments performed on simulated data and traces from the ASCAD database and collected from the ChipWhisperer-Lite board [CW].

### 1.3 Related work

The attack presented in this paper can be related to previous works on Non-Profiled partition-based DPA attacks [SGV09]. Partition-based DPA attacks follow a strategy in two steps. First, for each key guess, the set of traces is partitioned according to guessed intermediate values. Then, a statistical distinguisher is used to measure the consistency of each partition and reveal the correct key. Many statistical metrics for partition-based DPA were proposed in the literature. Some examples are the Difference of Means [KJJ99], the Mutual Information [GBTP08], the Variance-Ratio [SGV09] and clustering techniques [BGLR09]. Our work can be seen as a partition based DPA attack which uses Deep Learning trainings to evaluate the consistency of the partitions and reveal the correct key value. Additionally, we advise readers to refer to [DPRS11] which presents how Profiled Linear Regression Analysis presented in [SLP05] can be turned into a Non-Profiled attack. [DDP13] presents a similar process but in the context of High-Order attacks. Both papers can provide interesting perspectives on the topic as the present work follows a similar approach in the context of Deep Learning instead of Linear Regression.

### 1.4 Outline

The paper is organized as follows: In Section 2, Deep Learning and Deep Learning-based Side-Channel attacks are described. In Section 3, we present our method to apply Deep Learning techniques in a Non-Profiled scenario with illustrations and examples. In Section 4, we give more detailed results from experiments performed on simulated data and traces collected from the ChipWhisperer-Lite board and from the ASCAD database. Finally, in Section 5 we conclude and summarize the interests of this new attack.

## 2 Preliminaries

### 2.1 Deep Learning

Deep Learning (DL) is a branch of Machine Learning which uses deep neural networks and which has been successfully applied to many fields such as image classification, speech recognition or genomics [Bis06, LBH15, DLw]. In this section, we give a brief description of DL for data classification. In such a case, the objective is to classify some data $x \in \mathbb{R}^D$ based on their labels $z(x) \in \mathcal{Z}$, where $D$ is the dimension of the data to classify and $\mathcal{Z}$ is the set of classification labels. For simplicity's sake, we can consider $\mathcal{Z} = \{0, 1, \ldots, U-1\}$ with $U$ is the number of classification labels. We define the so-called *one-hot encoding* of the labels as $C : \mathbb{R}^D \longrightarrow \mathbb{R}^{|\mathcal{Z}|}$ with:

$$C(x)[i] = \begin{cases} 1 & \text{if } i = z(x) \\ 0 & \text{otherwise} \end{cases}$$

which can be seen as a vector representation of the label $z(x)$.

A Neural Network is a function $\texttt{Net} : \mathbb{R}^D \longrightarrow \mathbb{R}^{|\mathcal{Z}|}$ which takes as input a data to classify $x \in \mathbb{R}^D$, and outputs a score vector $y = \texttt{Net}(x) \in \mathbb{R}^{|\mathcal{Z}|}$. A neural network is internally composed of a set of trainable parameters $\theta$ which can be tuned during a training phase in order to improve the efficiency of the network. At the beginning of the training, the trainable parameters $\theta$ are usually initialized as small random values chosen in a given

interval. To quantify the efficiency of the network for a given input $x$, one can define an error function $E : \mathbb{R}^D \longrightarrow \mathbb{R}$ for instance as the Euclidean distance[1] between the output of the Neural Network and the one-hot encoding of the label:

$$E(x) = \left( \sum_{i=1}^{|\mathcal{Z}|} (C(x)[i] - \texttt{Net}(x)[i])^2 \right)^{\frac{1}{2}}.$$

The error function quantifies how far the network output is from the expected output. To quantify the error of the network over a whole set of training data $X = (x_i)_{1 \leq i \leq M}$, one can define a so-called loss function as the average error over all training inputs:

$$\mathcal{L}_X = \frac{1}{M} \sum_{i=1}^{M} E(x_i).$$

This loss function can be seen as a function $\mathcal{L}_X(\theta)$ which depends on the trainable parameters $\theta$. Formalized like this, a DL training can be seen as a classic numerical optimization problem, where the goal is to find the optimal parameters $\theta_{best}$ minimizing the loss function $\mathcal{L}_X$. The preferred approach in DL is to use the Gradient Descent technique to optimize the loss function and train the network. During a series of iterations, the gradient $\nabla \mathcal{L}_X(\theta)$ of the loss with regards to the trainable parameters $\theta$ is computed and the trainable paramaters are updated by following the invert direction of the gradient:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla \mathcal{L}_X(\theta^{(t)})$$

where $\alpha$ is called the learning rate, and is a parameter controlling the amplitude of the parameters update. This is repeated until the minimum of the loss function is found. Deep neural networks are usually composed of different layers. In order to compute the gradients of the trainable parameters for the different layers, one usually uses the *backpropagation* technique which is based on the derivative chain rule. The gradients are computed backward, layer by layer, starting from the last layer of the network. Once the gradients are computed, one can update the trainable parameters of each layer with the corresponding gradients. In practice, computing the gradient of the loss over the whole training set $X$ is too expensive and the Stochastic Gradient Descent [GBC16] (SGD) technique is used: instead of computing the gradient over the whole training set $X$, the gradient is computed over small subsets of $X$. When all the training samples have been used, the training samples are shuffled and the process is repeated. One iteration over all the training samples is called an epoch. SGD is repeated multiple epochs until the loss converges and reaches its minimum. For the rest of the paper we denote by $\mathbf{DL}(\texttt{Net}, X, Y, n_e)$, a deep learning training of the network $\texttt{Net}$ over $n_e$ epochs with $X$ the training data and $Y$ the corresponding training labels.

Once the network parameters are optimized, the network can be used to classify data. To classify a data $x$ whose corresponding label is unknown, one computes $\ell = \underset{j \in \mathcal{Z}}{\text{argmax}}\ \texttt{Net}(x)[j]$. The classification of $x$ is successful if $\ell = z(x)$.

### 2.1.1 Multi Layer Perceptron

A Multi Layer Perceptron (MLP) is a type of Neural Network composed of several *perceptron* units [Bis95]. A perceptron $\texttt{P} : \mathbb{R}^n \longrightarrow \mathbb{R}$ takes as input a vector $x \in \mathbb{R}^n$ and outputs a weighted sum evaluated through an *activation function* denoted $A$ as follows:

$$\texttt{P}(x) = A\big(b + \sum_{i=1}^{n} w_i x_i\big).$$

[1] The error and loss functions presented here are given only as examples. There exist actually many different error/loss functions which can be used in Deep Learning.

$(w_i)_i$ are called the *weights* and $b$ the *bias* of the perceptron unit. Common activation functions are for instance the Rectified Linear function (relu) or the Hyperbolic Tangent function (tanh). A Multi Layer Perceptron is a Neural Network which is a combination of many perceptron units organized in layers as shown in Fig. 1. Each perceptron output of one layer is connected to each perceptron of the next layer. A MLP is composed of an input layer, and output layer and a series of intermediate layers called *hidden layers*. Each layer is composed of one or several perceptron units. The weights and biases of the MLP are the trainable parameters which are updated during SGD.

### 2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a family of deep neural networks composed of two types of layers called *Convolutional* layers and *Pooling* layers and which has shown good results specially in the field of image recognition [LB95, ON15]. Convolutional layers apply convolution operations to the input by sliding a set of *filters* along the traces. The pooling layers are non-linear layers which slide a window over the input data and output a local summary such as the mean or maximum of the input in the window. Fig. 2 shows an example of a convolution operation with 3 filters of size 3 and an example of maximum pooling operation performed using a window of size $2 \times 2$. The CNN architecture has a natural translation-invariance property due to the use of pooling operations and shared weights applied across space during the convolution operations. Therefore, CNN is particularly interesting when dealing with de-synchronized side-channel traces as it is able to learn and detect features even if the traces are not perfectly aligned [CDP17].

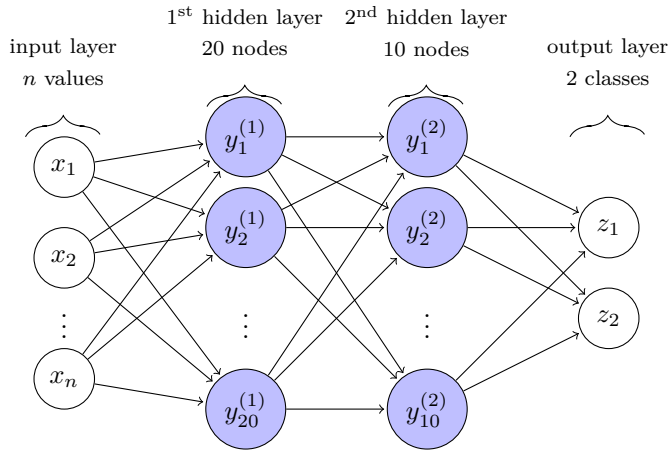


**Figure 1:** Multi Layer Perceptron with 2 hidden layers.

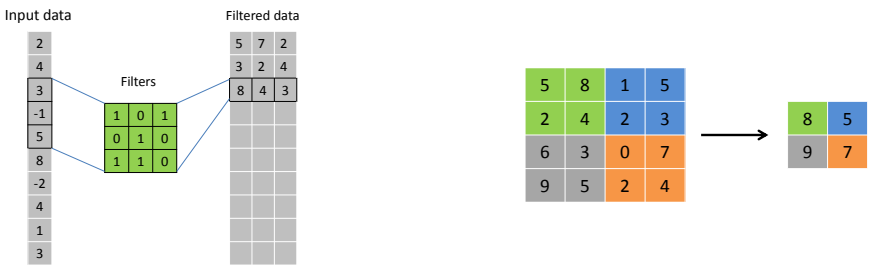


**Figure 2:** Example of convolution (left) and max pooling (right) operations used in CNN

### 2.1.3 Loss and Accuracy

During a Deep Learning training, it is possible to monitor the *loss* and *accuracy* of the training. As introduced previously, the loss quantifies the average classification error of the network. The accuracy at a given epoch can be defined for instance as the proportion of training samples that are correctly classified by the Neural Network. Both metrics give information about the evolution of the training. A decreasing loss and increasing accuracy usually indicate that the network is properly learning, except in case of *overfitting* where the Neural Network actually *memorizes* the data instead of learning the targeted features.

### 2.1.4 Sensitivity analysis

The Sensitivity Analysis (SA) of a mathematical model is the analysis of the model output sensitivity with regards to some of the model parameters [SRA+08]. SA can, for instance, provide a better understanding about the relationship between input and output parameters of a model. Many methods are known to study the sensitivity of a model, such as variance-based methods [Sob01] or methods based on partial derivatives. In this paper we will focus on SA based on partial derivatives. In Deep Learning, SA can be used for example to determine which pixels of a picture contributed the most to an image classification [SVZ13]. It can also be used to observe which neurons of a neural network contribute the most to the classification. To analyze the sensitivity of a network with regards to a given parameter $x$, a classic approach is to observe the partial derivative of the network output with regards to the parameter $x$. In Section 3 we show how Sensitivity Analysis for Deep Learning can be used as a metric to reveal secrets such as the key value and leakage locations during Non-Profiled attacks.

## 2.2 Profiled Deep Learning Side-Channel attacks

In this section, we remind how Deep Learning can be applied to perform Profiled Side-Channel attacks [MPP16, CDP17, PSB+18]. We consider that the attacker has access to a pair of identical devices: a target device running a cryptographic operation with a fixed unknown key $\boldsymbol{k^*} \in \mathcal{K}$ and a profiling device with knowledge and control of the keys and inputs. We consider that a divide-and-conquer strategy is applied and that $\mathcal{K} = \{0, 1, \ldots, 255\}$. The goal of the attack is to recover the secret key byte $\boldsymbol{k^*}$. The method proposed in [MPP16] is to perform a Profiled attack similar to a Template Attack [CRR03], but using Deep Learning training as a profiling method instead of using Multivariate gaussian profiling as in Template Attacks.

**Profiling phase** For the profiling phase, a set of $N$ traces $\mathcal{P}_k = \{T_i^{(k)} \mid i = 1, \ldots, N\}$ is collected from the profiling device for each key $k \in \mathcal{K}$ leading to a set $X$ of $(N \times 256)$ training traces:

$$X = \bigcup_{k=0}^{255} \mathcal{P}_k \ .$$

The set of training labels $Y$ is defined as the set of keys $z(T_i^{(k)}) = k$ corresponding to the training traces. To profile the leakage, a Deep Learning training $\mathbf{DL}(\texttt{Net}, X, Y, n_e)$ is performed using the Side-Channel traces as training data in order to build a Neural Network `Net` able to classify the side-channel traces based on their corresponding key values.

**Attack phase** To recover the secret key value $\boldsymbol{k^*} \in \mathcal{K}$ using $M$ side-channel traces $(T_i)_{1 \leq i \leq M}$ collected from the target device, one first evaluates each trace $T_i$ using the trained Neural Network to get $M$ score vectors $y_i = \texttt{Net}(T_i) \in \mathbb{R}^{|\mathcal{K}|}$. One can then select

the key $k$ leading to the highest summed score: $k = \text{argmax}_{j \in \mathcal{K}} \left( \sum_{i=1}^{M} y_i \right)[j]$ . The attack is successful if $k = \boldsymbol{k}^*$.

**Interests** Previous publications studied the interests of using Deep Learning to perform Profiled Side-Channel attacks. In [MPP16], Maghrebi *et al.* showed that Deep Learning can outperform other Profiled attacks such as Template Attacks in some cases. In [CDP17], the authors showed that the translation-invariance property of CNNs can be used against de-synchronized traces to improve the attacks results. However, all these studies focused only on applying Deep Learning to perform Profiled attacks.

# 3 Non-Profiled Deep Learning Side-Channel attacks

In this section we present a new attack method to apply Deep Learning techniques in a Non-Profiled context. In Section 3.1, we describe the principle of the attack. In the next subsections, we further discuss about some specific points of the attack and provide illustrations. More advanced experiments of the attack are presented in Section 4.

## 3.1 Differential Deep Learning Analysis

For the rest of the paper, we consider a Non-Profiled Side-Channel attack scenario. In such a context, an attacker collects $N$ side-channel traces $(T_i)_{1 \leq i \leq N}$ corresponding to the manipulation of a sensitive value $F(d_i, \boldsymbol{k}^*)$ where $(d_i)_{1 \leq i \leq N}$ are known random values and $\boldsymbol{k}^* \in \mathcal{K}$ is the fixed secret value. Usually such an attack is performed following a divide-an-conquer strategy, and one has for instance $|\mathcal{K}| = 256$ with $d_i$ and $\boldsymbol{k}^*$ 8-bit values. For the rest of the paper we focus on the AES algorithm even though the attack method is not tied to this algorithm. In this case, the target function $F$ can be chosen as the AES Sbox function, meaning that $F(d_i, \boldsymbol{k}^*) = Sbox(d_i \oplus \boldsymbol{k}^*)$.

To perform a partition-based DPA attack, one first needs to define a partition function $h$. For example, for a classic DPA attack, $h$ can be defined as the Most Significant (MSB) or Least Significant Bit (LSB) of $F(d_i, \boldsymbol{k}^*)$. Then, for each key hypothesis $k \in \mathcal{K}$ the attacker computes a series of hypothetical intermediate values $(V_{i,k})_{1 \leq i \leq N}$ with $V_{i,k} = F(d_i, k)$ and then partitions the traces based on the values $(H_{i,k})_{1 \leq i \leq N}$ with $H_{i,k} = h(V_{i,k})$. The attacker then uses a statistical distinguisher to evaluate the consistency of each partition and reveal the secret key. For DPA one for example uses the Difference of Means. For the correct key value $\boldsymbol{k}^*$, the partition of the traces will be consistent, and one should observe a high difference of means. For all the other key candidates, the partition is basically a random partition of the traces, leading to a difference of means close to 0.

To apply DL in a Non-Profiled context, our idea is to partition the traces as for a partition-based attack and use DL trainings to evaluate the consistency of the partitions. For each key hypothesis $k \in \mathcal{K}$ the attacker computes the series $(V_{i,k})_{1 \leq i \leq N}$ and partition the traces based on the values $H_{i,k} = h(V_{i,k})$. He then performs a DL training using the traces $(T_i)_{1 \leq i \leq N}$ as training data, and the series $(H_{i,k})_{1 \leq i \leq N}$ as the corresponding classification labels. When the correct key guess $\boldsymbol{k}^*$ is used, the series of intermediate values will be correctly guessed, and therefore the partition and the labels used for the DL training will be consistent with the corresponding traces. On the other hand, for all the other key guesses, the labels used for the trainings will be inconsistent with the traces. Therefore, if the network architecture is well-suited to target the set of traces, one should be able to observe a more efficient training for the correct key value than for the other guesses. The attacker can then discriminate the correct key value from the other candidates by selecting the key leading to the best training metrics. A description of

different Deep Learning metrics which can be used is given in Section 3.2. To ensure each guess is treated independently, it is important to re-initialize the trainable parameters of the network after each training. We use the name *Differential Deep Learning Analysis* (DDLA) for this new attack method. Algorithm 1 summarizes the DDLA procedure to perform a Non-Profiled attack using Deep Learning:

**Algorithm 1** Differential Deep Learning Analysis (DDLA)

**Inputs**: $N$ traces $(T_i)_{1\leq i\leq N}$ and corresponding plaintexts $(d_i)_{1\leq i\leq N}$. A network `Net` and number of epochs $n_e$.

1: Set training data as $X = (T_i)_{1\leq i\leq N}$.
2: **for** $k \in \mathcal{K}$ **do**
3: Re-initialize trainable parameters of `Net`.
4: Compute the series of hypothetical values $(H_{i,k})_{1\leq i\leq N}$.
5: Set training labels as $Y_k = (H_{i,k})_{1\leq i\leq N}$.
6: Perform Deep Learning training: $\mathbf{DL}(\texttt{Net}, X, Y_k, n_e)$.
7: **end for**
8: **return** key $k$ which leads to the best DL training metrics

**Network architecture** It is important to note that the DDLA attack method is not limited to a specific type of Neural Network. In the next section, we introduce metrics which can be used to perform DDLA with any type of Neural Networks. This provides many possibilities when performing the attacks as the attacker can adapt the architecture based on the targeted implementation and device. In this paper, we focus on two variants of DDLA, using MLP and CNN architectures. In this paper, we usually used MLP when traces were synchronized as this architecture was sufficient to obtain good results in this case. We used CNN mainly when targeting de-synchronized traces. For the rest of the paper, MLP-DDLA will refer to a DDLA attack using a MLP architecture and CNN-DDLA will refer to a DDLA attack using CNN. For the results presented in Section 3.2 we used two architectures $MLP_{sim}$ and $CNN_{sim}$ where $MLP_{sim}$ is composed of two hidden layers of 70 and 50 neurons and $CNN_{sim}$ is composed of two convolution layers of respectively 8 filters of size 8 and 4 filters of size 4. For each result presented in this paper, the details of the networks architectures and other training parameters (learning rate, batch size, loss function etc) are always given in Appendix A.

## 3.2 Metrics

In this section we introduce different metrics that can be used to reveal the correct key value during a DDLA attack. The two first metrics are based on sensitivity analysis and can also reveal points of interest such as the leaking samples in the trace. For masked implementation, it can also reveal masks locations, as we show in Section 4. To illustrate how the metrics can reveal the key and points of interest, we present some results obtained from a simulation data set. We generated $N = 5,000$ simulated traces as follows:

- $n = 50$ samples per trace.
- Sbox leakage set at time sample $t = 25$ and defined as $Sbox(d_i \oplus \boldsymbol{k}^*) + \mathcal{N}(0, 1)$ with $d_i$ a known randomized byte and $\boldsymbol{k}^*$ a fixed key byte. $\mathcal{N}(0, 1)$ corresponds to a Gaussian noise of mean $\mu = 0$ and standard deviation $\sigma = 1$.
- All other points on the traces are chosen as random values in $[0; 255]$.

The purpose of this simulation is only to illustrate how some Deep Learning metrics can be used to discriminate the correct key from the other candidates. Results obtained with

non-simulated traces are presented in Section 4. Using this simulation data, we performed the attack as defined in Algorithm 1 and observed the following metrics.

### 3.2.1 Sensitivity analysis based on MLP first layer weights

In this section we introduce a metric which can be used to reveal the correct key candidate when performing a DDLA attack with a Multi Layer Perceptron architecture. A noteworthy advantage of this metric is that it can also be used to reveal points of interest, such as the leakages or masks locations in the trace. The technique is based on the sensitivity analysis of the network with regards to the first layer weights during the DDLA trainings. For a trace of size $n$, the neural network takes as input the $n$ samples of the trace. When using a MLP architecture, each time sample $t$ of the trace is paired with $R$ trainable weights $(W_{t,j})_{1\leq j\leq R}$ where $R$ is the number of neurons in the first hidden layer. Therefore, the first hidden layer weights can be seen as a $(n \times R)$ matrix $W$ where $W_{i,j}$ is the weight between the $i^{\text{th}}$ sample of the trace and the $j^{\text{th}}$ neuron of the first hidden layer. During backpropagation, the gradient of the first layer weights is computed and can also be seen as a matrix $\nabla W$ of size $(n \times R)$ where

$$\nabla W_{i,j} = \frac{\partial \mathcal{L}}{\partial W_{i,j}} \ ,$$

corresponds to the derivative of the loss with regards to the weight $W_{i,j}$. The absolute value of the derivative $|\nabla W_{i,j}|$ measures the sensitivity of the loss with regards to the corresponding weight. The higher the absolute value of the derivative is, the more the corresponding weight contributes to the loss minimization. To measure the sensitivity related to each time sample $t$, one can sum the absolute values of the derivatives for the weights linked to this time sample as follows:

$$S_{weights}[t] = \sum_{j=1}^{R} |\nabla W_{t,j}| \ . \tag{1}$$

With our simulated dataset and the $MLP_{sim}$ network, we compared the sensitivity values obtained with equation (1) for the good key guess and for a wrong key guess over 250 SGD iterations. The results are presented in Fig. 3

For the good key guess: We can observe that the derivatives linked to the leakage sample $(t = 25)$ are in average much higher than the derivatives linked to the other time samples, especially during the first epochs of the training while the loss converges towards its minimum. As we mentioned, the absolute value of the derivative indicates how much the corresponding parameter contributes to the loss minimization. On one hand, the weights of the leakage sample has a direct impact on the loss, as it is the sample which carries the information useful for the classification. On the other hand, updating the weights of the non-leakage samples has usually a much smaller impact on the loss minimization, as these samples basically only carry noise and no information for the classification. Therefore, it is normal to observe that the derivatives of the weights linked to the leakage sample are significantly bigger than the derivatives corresponding to the non-leakage samples. As we observe, this is especially true during the first epochs of the training, while the loss converges towards its minimum. During this phase the derivatives related to the leakage sample are high and the corresponding weights are updated and converge towards their optimal values. When the loss reaches almost its minimum, the derivatives values decrease as only small adjustments are needed as the loss is already almost optimal.

For the wrong key guess: when using a wrong key candidate, the guessed intermediate values are wrong, and therefore the labels used for the DL training are not correct. We

can observe that in this case, all the derivatives are in average small, and that none of the sample leads to bigger derivatives values. This is normal as it is not possible to find some weights that have a significant impact on the loss minimization due to the inconsistent partition of the traces. Indeed, we can observe that in this case, the loss decreases significantly less over the epochs than for the good key value.

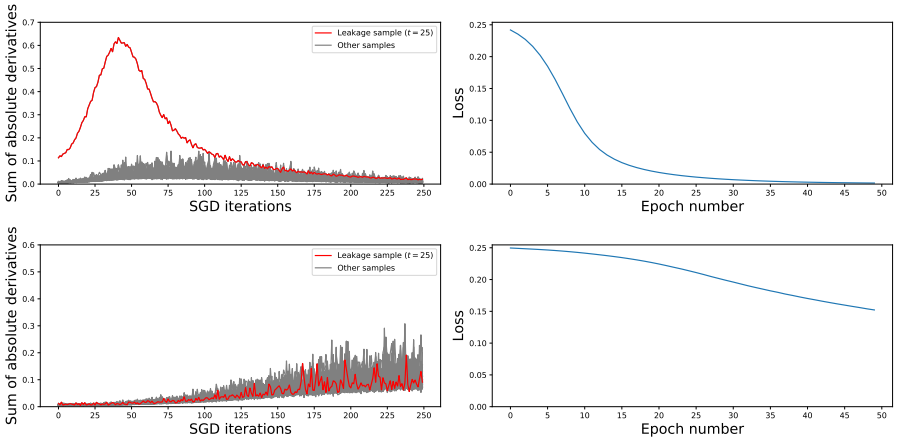


**Figure 3:** Sum of absolute derivatives and loss over SGD iterations. Top Left: sum of absolute derivatives for good key. Top right: loss for good key. Bottom left: sum of absolute derivatives for bad key. Bottom right: loss for bad key

One can sum the values $S_{weights}[t]$ over the SGD iterations, and compare the accumulated sums of derivatives at the end of the training for every key guess as presented in Fig. 4. We can observe that as expected, the correct key guess clearly leads to a higher value at precisely $t = 25$ which corresponds to the location of the Sbox leakage. On the other hand, all the wrong key guesses leads to low sensitivity values.

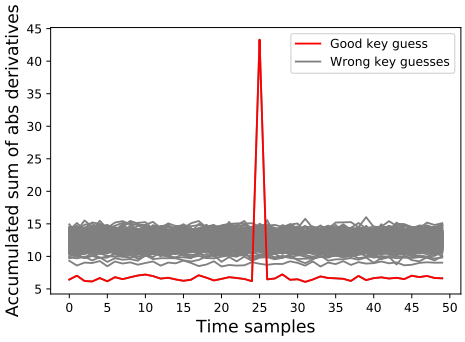


**Figure 4:** Sum of absolute derivatives accumulated over 250 iterations of SGD (50 epochs)

Therefore, using such a metric allows to reveal both the correct key guess and the leakage location at the same time. However, observing the first layer derivatives only makes sense for architectures like MLP. It is not directly applicable to other architectures like CNN. In the next section we introduce a second metric based on sensitivity analysis which can be used with any architecture.

### 3.2.2 Sensitivity analysis based on network inputs

A generic approach to measure the sensitivity of a network with regards to its inputs is to directly study the partial derivatives of the loss with regards to the network inputs [SVZ13]. The interest of this method is that it is applicable with any network architecture. For a set of training traces $T = (T_i)_{1 \leq i \leq N}$ composed of $n$ time samples, let's denote as

$$\nabla T_{i,j} = \frac{\partial \mathcal{L}_{T_i}}{\partial x_j}, \quad \text{for } i \in \{1, ..., N\} \text{ and } j \in \{1, ..., n\},$$

the partial derivative of the loss with regards to the $j^{\text{th}}$ sample variable, for the $i^{\text{th}}$ trace of the training set. To measure the sensitivity of the network with regards to its inputs, we start to compute the derivatives $\nabla T_{i,j}$ for each trace of the training set and each sample of the trace. Then, for each time sample $t$ we can add up the absolute value of the derivatives over the $N$ training traces as follows:

$$S_{input}[t] = \sum_{i=1}^{N} |\nabla T_{i,t}|, \quad \text{for } t \in \{1, ..., n\}.$$

This gives a measure of the sensitivity of the loss with regards to each time sample $t$. We can perform this operation at the end of each epoch, and accumulate the sensitivity values over the epochs. Similar arguments as in the previous section can be applied here. For the good key guess, the derivative(s) of the loss with regards to the leakage sample(s) will in average be higher than for the other samples. For the wrong key guess, all derivatives should be in average small due to wrong predictions and labels. Therefore, observing this metric should also allow to reveal both the leakage position and the correct key value. In [SGSK16], authors show that instead of considering the absolute value of the derivatives, another approach is to multiply the raw derivatives with the corresponding inputs. Therefore, one can also consider the following sensitivity measure:

$$S_{input}[t] = \sum_{i=1}^{N} (\nabla T_{i,t} \times T_{i,t}), \tag{2}$$

where $T_{i,t}$ corresponds to the value of the $i^{\text{th}}$ traces at the time sample $t$. During our tests we observed that this approach usually leads to better results. Therefore, it is the sensitivity measure that we will use for the rest of the paper. We applied this method to our simulated data set. To illustrate that this method is applicable to any architecture, we applied it using both $MLP_{sim}$ and $CNN_{sim}$ architectures. In Fig. 5 we present the results obtained after accumulating the sensitivity values computed with equation (2) over 50 epochs (the results of accumulation are presented in absolute value). We observe similar results as previously. The good key guess leads to a much higher sensitivity value at $t = 25$, which means this metric can also be used to reveal both the key and the points of interest. As the inputs-based sensitivity metric can be used with any architecture, we chose to focus on this one rather than on the weights-based sensitivity for the rest of the paper.

### 3.2.3 Loss and accuracy metrics

As we can observe on Fig. 3, when the correct key guess is used, the SGD algorithm is more efficient at decreasing the loss. Therefore, it is possible to observe the impact of the key guess directly on the training loss. The same phenomenon can be observed on the training accuracy. In Fig. 6 we present the losses and accuracies obtained for all key guesses when performing a DDLA attack using our simulation data set with $MLP_{sim}$ and with $n_e = 50$ epochs per guess. The figure clearly shows that the training using the correct key value leads to a higher accuracy and lower loss compared to the trainings for the other

candidates. These metrics can therefore be used to reveal the correct key by selecting the guess leading to the highest accuracy or lowest loss values.

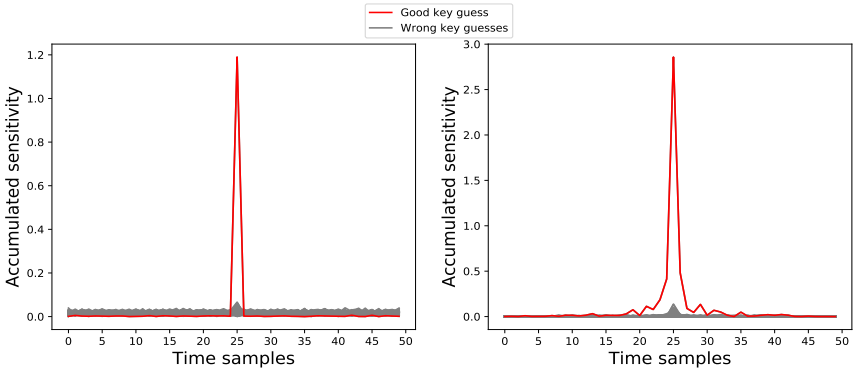


**Figure 5:** Inputs-based sensitivity accumulated over 50 epochs for all key guesses. Left: with $MLP_{sim}$ network. Right: with $CNN_{sim}$ network.

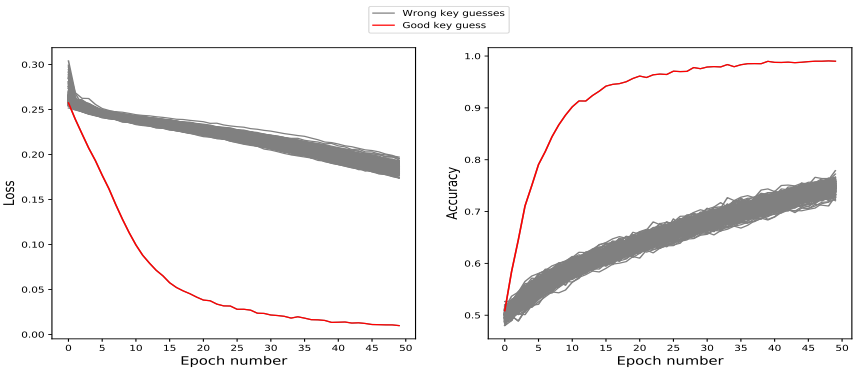


**Figure 6:** Loss (left) and accuracy (right) over the training epochs for all the key guesses when applying MLP-DDLA.

### 3.2.4 Summary

In this section we presented different metrics which can be used to reveal the correct key value when performing DDLA. The two metrics based on Sensitivity Analysis can also reveal points of interest such as the leakage location in the traces. We show in Section 4 that it can also reveal masks locations when attacking masked implementations. Moreover, it is important to note that using Sensitivity Analysis to reveal points of interest is not limited to the Non-Profiled context. All the arguments developed in this section related to derivatives are applicable to Profiled trainings as well. For the rest of the paper we will use the accuracy metric and the inputs-based sensitivity metric to evaluate the attacks. In the rest of the paper, each time we refer to the *inputs-based sensitivity*, it will correspond to the sensitivity measured with equation (2) accumulated over the epochs of the trainings. Each time we will present the absolute values obtained after accumulation over the epochs.

## 3.3 Labels

As mentioned for example in [SGV09][WOS14], for injective target functions like the AES Sbox, using the trivial partitioning where each intermediate value is a distinct class is not possible. Such partitioning will always fail to reveal the correct key value for any partition-based DPA attack. Indeed, if one uses the identity labeling $H_{i,k} = Sbox(d_i \oplus k)$, the partition of the attack traces derived from this labeling method will be equivalent for all the key guesses. In other words, the partition of the traces $P_k = \{E_u^{(k)} \mid u \in \{0, \dots, 255\}\}$ defined by the sets $E_u^{(k)} = \{T_i \in (T_i)_{1 \leq i \leq N} \mid Sbox(d_i \oplus k) = u\}$ is the same for all key guesses $k$. This means that from one key guess to another, there is no difference in the partition of the attack traces, and that only the labels are permuted which does not impact the training metrics. It means that using the identity labeling $H_{i,k} = Sbox(d_i \oplus k)$ will naturally lead to similar Deep Learning metrics for all the key candidates, making it impossible to discriminate the correct key value. For this reason, it is necessary to apply a non-injective function to the Sbox output to compute the labels so that the partition of the attack traces is different from one guess to another. We propose hereafter two methods:

**Hamming Weight labeling** One solution is to use labels based on the Hamming Weight of the guessed value as follows: $H_{i,k} = HW(V_{i,k})$.

**Binary labeling** The MSB or LSB of the guessed values $V_{i,k}$ can also be used to partition the traces.

To illustrate the importance of the labeling method, we performed the same attack as in Section 3.2 but using the identity labeling ($H_{i,k} = Sbox(d_i \oplus k)$). The comparison of accuracies obtained when using the identity labeling and binary labeling is presented in Fig. 7.

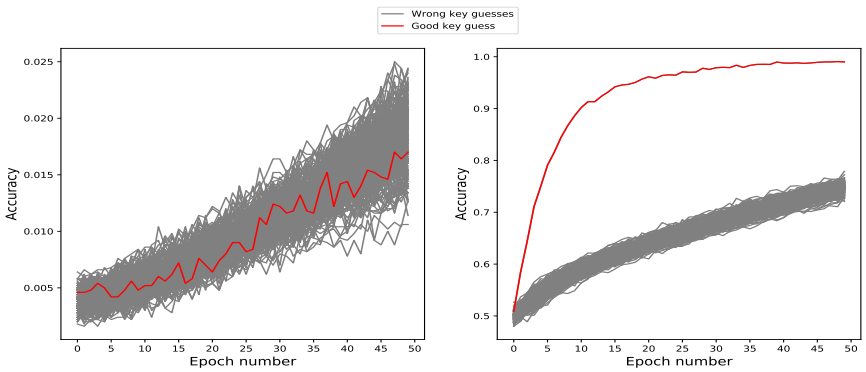


**Figure 7:** MLP-DDLA accuracies using two different labeling methods. Left: Identity labeling. Right: Binary labeling (MSB).

As expected, the left graph shows that all key guesses lead to similar accuracies when using the identity labeling. All accuracies are not perfectly identical even when using the identity labeling as the Deep Learning training is not a deterministic process. Indeed, the training always depends on the weights initialization as well as the shuffling of the input data during the different epochs, which explains the slight differences between the accuracies even though the identity labeling is used. However, using the identity labeling will always lead to similar accuracies making it impossible to distinguish the correct key

value. That is why it is necessary to use other labeling methods, such as the Hamming Weight or binary labeling methods. During our experiments, the binary labeling usually provided better results than using Hamming Weight labels. For the rest of the paper, all the results presented were obtained using the MSB and LSB labeling methods.

## 3.4 CNN-DDLA and de-synchronized traces

In [CDP17], Cagli *et al.* highlighted that due to its translation-invariance property, the CNN architecture is naturally efficient to extract information even from de-synchronized traces. We show in Section 4 that this property of CNNs can also be exploited when performing DDLA attacks in a Non-Profiled context. Using this property, we show that DDLA can outperform classic Non-Profiled attacks like CPA when attacking de-synchronized traces and is therefore an interesting alternative when the traces cannot be perfectly re-synchronized.

## 3.5 High-Order DDLA

A common countermeasure to protect cryptographic implementations against Profiled and Non-Profiled attacks is to conceal the sensitive intermediate values with *masks*. In the following, we focus on Boolean masking, which is commonly used to protect symmetric algorithms like AES [AG01]. In the case of Boolean masking, a sensitive intermediate value, for instance the AES Sbox output, is never manipulated in plain, but instead, is represented as a XOR of $s+1$ *shares*: $S = Sbox(d \oplus k) \oplus m_1 \oplus \cdots \oplus m_s$. The values $m_1, \ldots, m_s$ are called the *masks* and $S$ is called the *masked value*. Each mask $m_i$ is generated as a random value for each execution of the algorithm, making the leakages uncorrelated to the sensitive values. However, *High-Order* attacks such as High-Order CPA have been developed to target such implementations [Mes00, JPS05, WW04, PRB09]. A High-Order attack is usually composed of two steps:

- A pre-processing phase: the leakages of the masks are combined with the leakage of the masked value using combination functions such as the absolute difference or centered product [PRB09].
- The attack phase: a statistical distinguisher, for instance the Pearson's Correlation is used to extract information from the combined leakages traces.

A high-order attack targeting a value protected with one mask is called a *second order* attack, and a *third order* attack corresponds to a high-order attack targeting a value protected with 2 masks. For a second order attack, one needs to combine the leakage of the mask $m_1$ with the leakage of the masked Sbox value $Sbox(d \oplus k) \oplus m_1$. If the locations of the mask and masked value are known, one only needs to combine these two leakage locations together. If the locations of the mask and masked value leakages are unknown, a solution is to combine all the possible couples of points in the trace together. If the traces are of size $n$, such processing will lead to combined traces of size $\frac{n \times (n-1)}{2}$. Therefore, for large traces, such processing can become too complex and not practical.

In [MPP16] and [PSB+18], the authors successfully attacked first order protected AES implementations, showing that it is possible to break 1-mask protected implementations using CNN and MLP networks in a Profiled attack context. We show in Section 4.2 that it is possible to break implementations protected with 1 and 2 masks using Deep Learning in a Non-Profiled context with a reasonable number of traces. In comparison with High-Order CPA, DDLA does not require to combine the leakages prior to the attack. Moreover, it is not even required to know or guess the details of the implementation, such as the masking technique or the number of masks. Finally, combined with the sensitivity metric, it actually can reveal masks positions in the traces.

# 4 Experiments

In this section we perform experiments to study some interests of DDLA. In a first section we study how CNNs can be used in a Non-Profiled context against de-synchronized traces and compare it with CPA. In a second section we show how DDLA can break masked implementations in black-box and reveal masks locations in the trace. We perform these experiments using simulated traces and traces collected with the ChipWhisperer-Lite (CW) platform [CW] and from the public database ASCAD [PSB+18]. With the CW, we collected power traces of implementations running on an Atmel XMEGA128 chip. The traces of ASCAD were collected from an 8-bit ATMega8515 board. To attack traces from CW and ASCAD, we used the architectures $MLP_{exp}$ and $CNN_{exp}$. $CNN_{exp}$ is composed of two convolution layers of respectively 4 filters of size 32 and 4 filters of size 16. $MLP_{exp}$ is composed of two hidden layers of 20 and 10 neurons. Again, a complete description of the networks architectures and trainings parameters (learning rate, batch size, loss function, labeling method etc) is given in Appendix A.

## 4.1 CNN-DDLA against de-synchronized traces: comparison with CPA

In this section we show how CNNs can be used in a Non-Profiled context against de-synchronized traces and we compare its efficiency with CPA. We implemented an unprotected AES Sbox operation and loaded it in the ChipWhisperer-Lite board. We collected $N = 3,000$ traces of $n = 500$ samples containing the copy of $Sbox(d \oplus \boldsymbol{k^*})$ in memory.

### 4.1.1 Reference attack against synchronized traces

By default, the traces collected from the CW are well synchronized. First, we attacked the synchronized traces in order to get reference results. We performed a first order CPA and a DDLA attack using the network $MLP_{exp}$. The results are presented in Fig. 8. As expected, the CPA attack is successful as the targeted implementation is unprotected and the traces are synchronized. We can observe that the MLP-DDLA attack is also successful with only $3,000$ traces and the sensitivity metric reveals the same leakage location as the CPA. In this example we can notice that the CPA reveals two leakage areas, while the sensitivity analysis of the MLP only reveals one main leakage area. This is due to the univariate nature of CPA where each sample is attacked independently, which explains why we can observe two leakage areas. On the other hand, the inputs-based sensitivity analysis of the network reveals points of interest based on how the network uses the input samples to classify the data. In this case, it seems that the network only uses the first leakage area to classify the data, which explains why only the first area is highlighted by the sensitivity analysis.

### 4.1.2 CNN-DDLA against de-synchronized traces

To study the efficiency of CNN-DDLA against de-synchronized traces we applied a software de-synchronization to the set of traces by shifting each trace left or right by a random number chosen in $[-25; 25]$. We then applied a DDLA attack against the $N = 3,000$ de-synchronized traces using the $CNN_{exp}$ network. For comparison, we also performed the attack with the same network $MLP_{exp}$ as previously and also performed a CPA attack. The results presented in Fig. 9 show that both the CPA and the MLP-DDLA fail to recover the key due to the de-synchronization of traces. On the other hand, the CNN-DDLA is successful and reveals the key. This confirms that the translation-invariance property of CNNs can be used against de-synchronized traces during Non-Profiled attacks. Moreover, the sensitivity metric also reveals the leakage area which corresponds to the same area as before but spread along multiple samples due to the de-synchronization of the leakage.

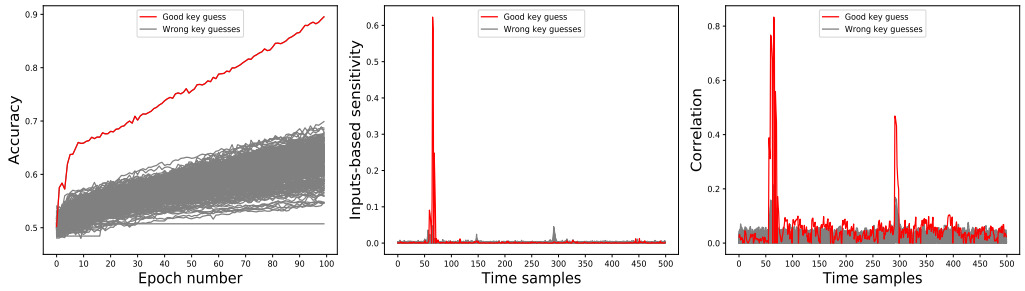


**Figure 8:** Attack on CW unprotected implementation without de-synchronization. Left: MLP-DDLA accuracies. Center: MLP-DDLA inputs-based sensitivity Right: CPA.

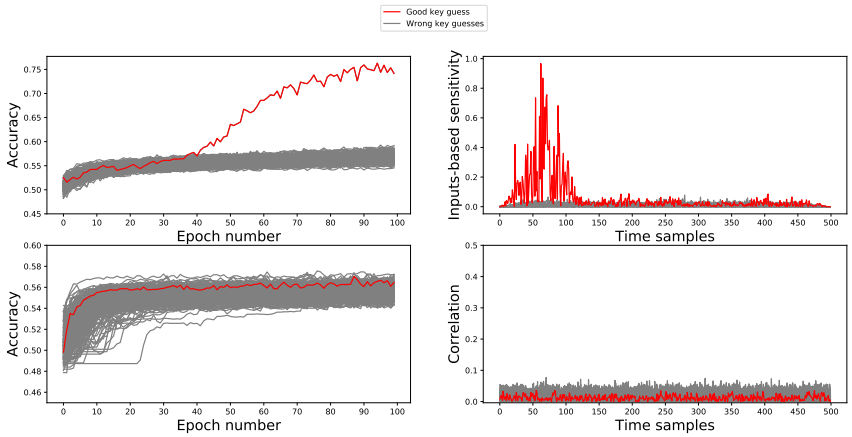


**Figure 9:** Attack on CW unprotected implementation with de-synchronization. Top-left: CNN-DDLA accuracies. Top-right: CNN-DDLA inputs-based sensitivity. Bottom-left: MLP-DDLA accuracies. Bottom-right: CPA.

### 4.1.3 Conclusions on CNN-DDLA

In this section we showed that the translation invariance property of CNNs can be succesfully used during Non-Profiled attacks against de-synchronized traces. In these conditions, DDLA clearly outperform CPA. We can conclude that CNN-DDLA could be an interesting alternative to other Non-Profiled attacks, specially when traces cannot be perfectly re-synchronized before the attack.

## 4.2 High-Order DDLA

In this section we study how DDLA can be used to break masked implementations in black-box and reveal masks and leakages locations.

### 4.2.1 High-Order DDLA simulations

In this section we study the efficiency of MLP-DDLA when targeting a simulated AES Sbox operation protected with 1 and 2 random masks. A similar procedure as in Section

3.2 was used to generate simulated traces. The only difference is that for this experiment, random masks values are added to the simulation traces. To simulate a 1-mask protected Sbox, we generated traces as follows:

- $n = 50$ samples per trace.
- Masked Sbox leakage set at $t = 25$ and defined as $Sbox(d_i \oplus \boldsymbol{k^*}) \oplus m_1 + \mathcal{N}(0, 1)$ with $d_i$ and $m_1$ randomized bytes and $\boldsymbol{k^*}$ a fixed key byte. Mask leakage set at $t = 5$ and defined as $m_1 + \mathcal{N}(0, 1)$.
- All other points on the traces are chosen as random values in $[0; 255]$.

To simulate the protection with 2 masks, we followed the same procedure except that a second mask $m_2$ was used and the corresponding leakage set at $t = 45$. In this case the Sbox leakage was defined as $Sbox(d_i \oplus \boldsymbol{k^*}) \oplus m_1 \oplus m_2 + \mathcal{N}(0, 1)$. We applied DDLA as in Algorithm 1 with $N = 5,000$ traces for 1 mask and $N = 10,000$ traces for 2 masks. For this experiment we used the same architecture $MLP_{sim}$ as in Section 3.2. In Fig. 10 we present the accuracy and sensitivity metrics values obtained for all the key guesses.

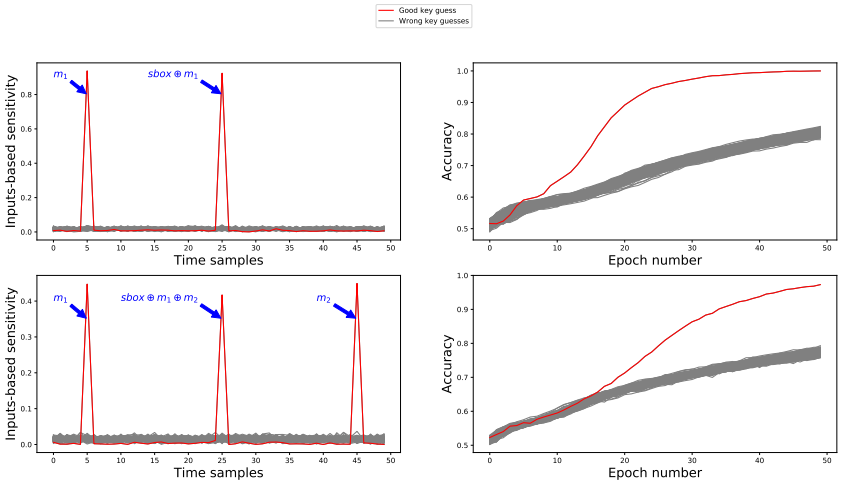


**Figure 10:** MLP-DDLA applied to 1 and 2 masks protected Sboxes. Top-left: sensitivity for 1 mask. Top-right: accuracies for 1 mask. Bottom left: sensitivity for 2 masks. Bottom right: accuracies for 2 masks

For 1 and 2 masks, the attacks are successful with both the sensitivity and the accuracy metrics. We can observe in both cases that the sensitivity metric reveals the exact locations of the masks and masked Sbox.

It is important to note that these results were obtained without any leakages combination pre-processing nor any assumptions about the masking method. Compared with CPA, one does not need to adapt the DDLA attack to the masking scheme. The DDLA attack procedure presented in Algorithm 1 can be applied to both unprotected and masked implementations similarly. It is the neural network which adapts itself to each situation. DDLA is therefore particularly interesting when targeting implementations in black-box, as it does not require to make assumptions about the implementation or masking scheme. Combined with sensitivity analysis, it can even reveal information about the

implementation, such as the number of masks and their locations in the traces. In the next sections we validate these observations with traces from the ASCAD database and with traces collected from the ChipWhisperer.

### 4.2.2 Second order DDLA on ASCAD

To experiment a second order DDLA attack on non-simulated traces, we decided to use the ASCAD database. ASCAD is a public database introduced by Prouff *et al.* in [PSB+18] to provide a common set of side-channel traces for research on Deep Learning-based Side-Channel attacks. The targeted implementation is a first order protected Software AES implementation running on an 8-bit ATMega8515 board. The main database `ASCAD.h5` is composed of two sets of traces: a profiling set of $50,000$ traces to train Deep Learning architectures and an attack set of $10,000$ traces to test the efficiency of the trained Neural Networks. Each trace of the database is composed of 700 samples focusing on the processing of the third byte of the masked state $Sbox(p[3] \oplus k[3]) \oplus r[3]$ where $p$, $k$ and $r$ are respectively the plaintext, the key and the mask values. In [PSB+18] Prouff *et al.* focused on providing reference results for Profiled Deep Learning attacks using the profiling and attack sets of the ASCAD database.

For both the profiling set and the attack set of `ASCAD.h5`, the same 16-byte fixed key is used while the plaintexts and masks are randomized. Therefore, as the key is always fixed, both the attack set and profiling set can be considered as traces obtained from a closed device to perform a Non-Profiled attack. We decided to use the profiling set to perform our experiment as it contains more traces than the attack set. We applied a DDLA attack with $MLP_{exp}$ on the first $20,000$ traces of the profiling set of `ASCAD.h5` with $n_e = 50$ epochs per guess. We observed both the accuracy and inputs-based sensitivity metrics. Moreover, to validate our results, we used the knowledge of the key and of the masks to perform a reverse engineering CPA to highlight the locations of the masks and masked Sbox. The results presented in Fig. 11 show a clear success of the DDLA attack after only a few epochs. Moreover, we can observe that the sensitivity analysis values reveal two main areas, which match with the areas of the mask and masked Sbox obtained through reverse engineering CPA. It is important to note that the locations highlighted by the DDLA are obtained without any knowledge of the mask or key values. These results validate our observations made on simulation traces in the previous section.

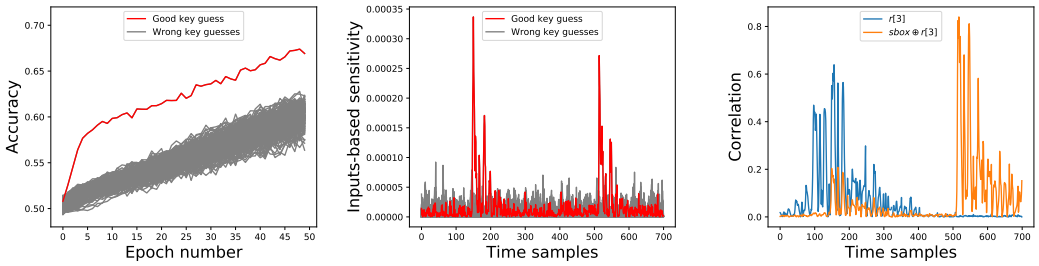


**Figure 11:** MLP-DDLA attack on ASCAD. Left: Accuracy. Center: Inputs-based sensitivity. Right: CPA reverse engineering.

### 4.2.3 Third order DDLA on ChipWhisperer

We implemented an Sbox operation protected by 2 masks using the re-computed Sbox method described in [AG01]. We collected $N = 50,000$ traces from the ChipWhisperer-Lite

and selected $n = 150$ samples containing the copies in memory of the first mask $m_1$, the second mask $m_2$ and the masked Sbox value $Sbox(d \oplus \boldsymbol{k}^*) \oplus m_1 \oplus m_2$. We performed a first order CPA and a second order CPA attack on the traces to confirm that the implementation did not have first or second order leakages. We performed the DDLA attack using the $MLP_{exp}$ network with $n_e = 100$ epochs per guess. As previously, we also used the knowledge of the key and of the masks to perform a CPA-based reverse engineering in order to reveal the locations of the masks and masked Sbox in the traces for comparison with the sensitivity metric. The results of this experiment are presented in Fig. 12.

We can observe that the MLP-DDLA attack reveals the correct key value after around 20 epochs per guess. As the implementation does not have first or second order leakages, this shows that the MLP-DDLA method is able to combine the leakages of 3 different shares to reveal the secret key, even on non-simulated data, with a reasonable number of traces and without traces pre-processing. Moreover, we can observe that the sensitivity metric clearly reveals the locations of the masks and masked Sbox which match with the locations revealed by the CPA reverse engineering. Again, it is important to highlight that the DDLA reveals these locations without knowledge of the key or masks values.

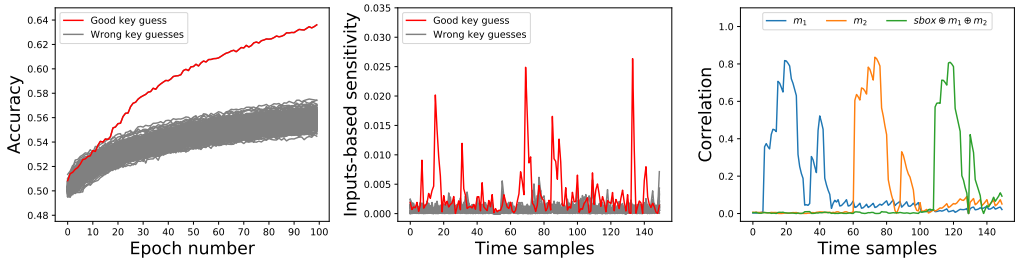


**Figure 12:** DDLA on CW 2-masks protected implementation. Left: Accuracy. Center: Inputs-based sensitivity. Right: CPA-based reverse engineering.

### 4.2.4 Conclusions on High-Order DDLA attacks

In this section we showed that it is possible to use DDLA to break masked AES implementations without any leakages combination pre-processing nor any assumptions about the masking method. We showed that the attack procedure introduced in Algorithm 1 can be applied to both unprotected and masked implementations without distinction as it is the neural network which adapts itself to the context. We showed that using the sensitivity metric it can even reveal masks locations in the traces. Therefore, it makes DDLA an interesting alternative to perform High-Order side-channel attacks specially in black-box when the masking technique and masks locations are unknown.

## 4.3 Complexity

One drawback of DDLA is that it is necessary to perform a Deep Learning training for each key guess. When using 8-bit key guesses, it means that 256 trainings are necessary. The execution times of different DDLA attacks to recover 1 key byte are summarized in Table. 1. We recorded these values when running the experiments in Python using the PyTorch framework [PyT], on our personal computer with 64 GB of RAM, a GeForce GTX 1080Ti GPU and two Intel Xeon E5-2620 v4 @2.1GHz CPUs.

**Table 1:** Execution times comparison for 1 key byte attacks.

| Target | Architecture | Nb traces | Nb samples | Nb epochs | Time |
|---|---|---|---|---|---|
| CW no mask | $MLP_{exp}$ | 3,000 | 500 | 100 | 4min20s |
| CW no mask | $CNN_{exp}$ | 3,000 | 500 | 100 | 33min17s |
| CW 2-masks | $MLP_{exp}$ | 50,000 | 150 | 100 | 1h03min16s |
| ASCAD | $MLP_{exp}$ | 20,000 | 700 | 50 | 14min12s |
| ASCAD | $MLP_{exp}$ | 20,000 | 700 | 5 | 1min54s |

The table shows that even though multiple trainings are needed, DDLA attacks can be performed in reasonable time and are therefore practical. All experiments were performed using many epochs per guess, but it can be observed that most of the time, only a few epochs were needed to reveal the correct key value. It means that these attacks can be performed faster by reducing the number of epochs per guess. For example, if we limit our attack on ASCAD to only 5 epochs per guess, the attack only requires less than 2 minutes on our setup and is still successful. As the number of epochs needed to recover the key is usually unknown, it may also be interesting to optimize the DDLA execution by using an incremental procedure. Indeed, instead of performing a full Deep Learning training for each key guess and checking the metrics at the end, one can perform a series of partial trainings, and check the metrics every few epochs. If the key is not recovered at one step, one continues the Deep Learning trainings and check the metrics again after a few epochs. This approach allows to control the total number of epochs used for the attack and therefore could be used to reduce the complexity.

As a remark, the neural networks used for the experiments in this paper were purposely kept small to limit the complexity of the attacks. The architectures used are surely not optimal and more complex networks might lead to better results. As in the case of Profiled Deep Learning attacks, attacking more difficult targets may require the usage of more complex neural networks to succeed a Non-Profiled Deep Learning attack. In such a case, the main consequence will be a higher time complexity due to longer trainings.

# 5 Conclusion

In this paper we introduced Differential Deep Learning Analysis (DDLA) a new side-channel attack method to apply Deep Learning techniques in a Non-Profiled context. The attack presented is a type of partition-based side-channel attack which uses Deep Learning trainings to reveal the secret key value. The main interest of this method is that it is possible to use the power of Deep Learning and deep neural networks in a Non-Profiled context. We showed that even in a Non-Profiled context, the translation-invariance property of Convolutional Neural Networks can be exploited against de-synchronized traces. Using this property, we showed that DDLA can outperform CPA and could be an interesting alternative to other Non-Profiled attacks when the traces cannot be perfectly re-synchronized. This new attack method can also be used to break masked implementations in black-box, without any leakages combination pre-processing nor assumptions about the implemented protections. We showed that the same attack procedure can be applied to both unprotected and masked implementations as neural networks have the ability to adapt to the different situations. To perform the attack, we introduced metrics based on Sensitivity Analysis which can reveal both the secret key and points of interest such as leakages and masks locations in the traces. Finally, the complexity snapshot that we provide shows that although this method requires multiple Deep Learning trainings, the attack can still be performed in practice.

## Acknowledgements

I would like to thank Houssem Maghrebi for reviewing the first version of the paper. I would also like to thank Lejla Batina and the anonymous reviewers for their valuable comments which helped to improve this work. Finally, I would like to thank Hugues Thiebeauld for the fruitful and stimulating discussions and for reviewing the paper.

## References

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BGLR09] Lejla Batina, Benedikt Gierlichs, and Kerstin Lemke-Rust. Differential Cluster Analysis. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 112–127, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 45–68, Cham, 2017. Springer International Publishing.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CW] ChipWhisperer Website. https://newae.com/tools/chipwhisperer/.

[DDP13] G. Dabosville, J. Doget, and E. Prouff. A new second-order side channel attack based on linear regression. *IEEE Transactions on Computers*, 62(8):1629–1640, Aug 2013.

[DLw] Deep learning website. https://deeplearning.net/tutorial/tutorial.

[DPRS11] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate side channel attacks and leakage modeling. *Journal of Cryptographic Engineering*, 1(2):123, Aug 2011.

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

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, pages 426–442, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

[JPS05] M. Joye, P. Paillier, and B. Schoenmakers. On Second-Order Differential Power Analysis. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[KJJ99] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[LB95] Yann Lecun and Yoshua Bengio. *Convolutional networks for images, speech, and time-series*. MIT Press, 1995.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LBM15] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. A machine learning approach against a masked AES. *Journal of Cryptographic Engineering*, 5(2):123–139, Jun 2015.

[LPB+15] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template Attacks vs. Machine Learning Revisited (and the Curse of Dimensionality in Side-Channel Analysis). In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 20–33, Cham, 2015. Springer International Publishing.

[Mes00] Thomas S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 238–251, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[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.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. 11 2015.

[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 Cecile Dumas. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database. Cryptology ePrint Archive, Report 2018/053, 2018. https://eprint.iacr.org/2018/053.

[PyT] PyTorch framework. https://pytorch.org.

[SGSK16] Avanti Shrikumar, Peyton Greenside, Anna Shcherbina, and Anshul Kundaje. Not Just a Black Box: Learning Important Features Through Propagating Activation Differences. *CoRR*, abs/1605.01713, 2016.

[SGV09] François-Xavier Standaert, Benedikt Gierlichs, and Ingrid Verbauwhede. Partition vs. Comparison Side-Channel Distinguishers: An Empirical Evaluation of Statistical Tests for Univariate Side-Channel Attacks against Two Unprotected CMOS Devices. In Pil Joong Lee and Jung Hee Cheon, editors, *Information Security and Cryptology – ICISC 2008*, pages 253–267, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 30–46, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Sob01] I.M Sobol. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and Computers in Simulation*, 55(1):271 – 280, 2001. The Second IMACS Seminar on Monte Carlo Methods.

[SRA+08] Andrea Saltelli, M Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global Sensitivity Analysis. The Primer*, volume 304. 01 2008.

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[WOS14] Carolyn Whitnall, Elisabeth Oswald, and François-Xavier Standaert. The Myth of Generic DPA...and the Magic of Learning. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, pages 183–205, Cham, 2014. Springer International Publishing.

[WW04] Jason Waddle and David Wagner. Towards Efficient Second-Order Power Analysis. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 1–15, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

# A Networks and training parameters

In this appendix we provide all the details needed for reproducibility of the attacks such as the networks architectures as well as general training parameters.

## A.1 Trainings parameters and details

### A.1.1 Loss function

We used the Mean Squared Error (MSE) loss function for all experiments.

### A.1.2 Accuracy

The accuracy was computed as the proportion of samples correctly classified.

### A.1.3 Batch size

A batch size of 1000 was used for all experiments.

### A.1.4 Learning rate

For all experiements we used a learning rate of 0.001.

### A.1.5 Optimizer

We used the Adam optimizer with default configuration ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-08$, no learning rate decay).

### A.1.6 Input normalization

We normalize the input traces by removing the mean of the traces and scaling the traces between $-1$ and 1.

### A.1.7 Labeling

- For all simulations (first and high-order), we used the MSB labeling.
- For attacks on the unprotected CW and on ASCAD, we used the LSB labeling.
- For the attack on the CW with 2 masks, we used the MSB labeling.

### A.1.8 Deep Learning Framework

We used PyTorch 0.4.1.

## A.2 Networks architectures

### A.2.1 $MLP_{sim}$

- Dense hidden layer of 70 neurons with relu activation
- Dense hidden layer of 50 neurons with relu activation
- Dense output layer of 2 neurons with softmax activation

### A.2.2 $CNN_{sim}$

- Convolution layer with 8 filters of size 8 (stride of 1, no padding) with relu activation.
- Max pooling layer with pooling size of 2.
- Convolution layer with 4 filters of size 4 (stride of 1, no padding) with relu activation.
- Max pooling layer with pooling size of 2.
- Dense output layer of 2 neurons with softmax activation

### A.2.3 $MLP_{exp}$

- Dense hidden layer of 20 neurons with relu activation
- Dense hidden layer of 10 neurons with relu activation
- Dense output layer of 2 neurons with softmax activation

### A.2.4 $CNN_{exp}$

- Convolution layer with 4 filters of size 32 (stride of 1, no padding) with relu activation.
- Average pooling layer with pooling size of 2.
- Batch normalization layer
- Convolution layer with 4 filters of size 16 (stride of 1, no padding) with relu activation.
- Average pooling layer with pooling size of 4.
- Batch normalization layer
- Dense output layer of 2 neurons with softmax activation