

# Spread: a new layer for profiled deep-learning side-channel attacks

Christophe Pfeifer<sup>1,2</sup> and Patrick Haddad<sup>3</sup>

<sup>1</sup> Karlsruhe Institute of Technology, Karlsruhe, Germany

<sup>2</sup> Grenoble INP - Ensimag, Grenoble, France

`christophe.pfeifer@grenoble-inp.org`

<sup>3</sup> STMicroelectronics SA, Rousset, France

`patrick.haddad@st.com`

**Abstract.** Recent publications, such as [10] and [13], exploit the advantages of deep-learning techniques in performing Side-Channel Attacks. One example of the Side-Channel community interest for such techniques is the release of the public ASCAD database, which provides power consumption traces of a masked 128-bit AES implementation, and is meant to be a common benchmark to compare deep-learning techniques performances. In this paper, we propose two ways of improving the effectiveness of such attacks. The first one is as new kind of layer for neural networks, called “Spread” layer, which is efficient at tackling side-channel attacks issues, since it reduces the number of layers required and speeds up the learning phase. Our second proposal is an efficient way to correct the neural network predictions, based on its confusion matrix. We have validated both methods on ASCAD database, and conclude that they reduce the number of traces required to succeed attacks. In this article, we show their effectiveness for first-order and second-order attacks.

**Keywords:** Deep-learning · Side-Channel Attack · Spread layer · ASCAD · Confusion matrix · Bayesian correction.

## 1 Introduction

### 1.1 Side-channel attacks

Side-channel attacks are a set of attack methods used to target cryptographic devices and retrieve sensitive information. These methods may leverage power consumption traces gathered when executing an algorithm.

*Order of a side-channel attack* During the execution of the algorithm, many variables are processed. If one of these depends on secret data (e.g. a secret key), then this variable is called *sensitive* since its knowledge may lead an attacker to retrieve the concealed data. Moreover, the minimal amount of sensitive variables required to retrieve the key defines the order of an attack. For instance, if we only need to know one sensitive variable, and that this variable is correlated at some points in the consumption traces, then this leakage is called a *first-order leakage* and we need to perform a *first-order side-channel attack* to retrieve the secret key.

*Classes of side-channel attacks* These attacks may work according to two scenarios: profiled or non-profiled.

- In the profiled case, the attacker possesses a device similar to the one being targeted and acquires beforehand many execution traces according to different parameters. Based on these observations, he elaborates and tunes a model able to retrieve the sensitive information targeted. This category of attacks includes Templates Attacks [4] and Stochastic models (a.k.a Linear Regression Analysis) [5, 12, 11].
- In the non-profiled case, the attacker does not have such a device at his disposal. This set of non-profiled attacks includes, among others, Differential Power Analysis [14], Correlation Power Analysis [8] and Mutual Information Analysis [6, 3].

In both scenarios, the adversary obtains afterwards attack traces and shall retrieve some sensitive information used. In addition, he may know some public information such as the the plaintext, the ciphertext, etc.

## 1.2 Deep-learning

Some models used to predict sensitive data rely on deep-learning, which is a branch of machine learning, namely methods learning how to extract relevant information out of a usually large amount of data, without being explicitly told how to. Deep-learning is specifically based on the use of neural networks to make predictions. The basis component of a neural network is the neuron, which performs a single operation on its input(s). Neurons are assembled into layers, which are then stacked.

**Architecture** Some common layers used in deep-learning are the following:

- *Dense layers* are composed of many neurons, whose inputs are connected to all outputs of the previous layer. Each input is then multiplied by some trainable weights, and summed up before being passed to the next layer.
- *Activation layers* are used to increase the computational capacity of the network. They implement an activation function, which is applied to each output of the previous layer, and whose results make the output of the layer. Such functions may be non-linear or almost everywhere differentiable and are usually not trainable. For instance, the softmax function (a special case of activation functions since it is applied on the whole previous layer) is used to obtain a probability distribution out of its input.  
A network exclusively made of the two aforementioned types of layers is called a Multi-Layer Perceptron (MLP).
- *Flatten layers* are used to reshape the output of the previous layer into a one-dimensional output.

**Learning phases** The learning process of a neural network implies many elements:

- A *training set* contains observations, for which the correct output labels (e.g. the value of the targeted data) are known. it is used before the attack to train and tune the neural network.
- A *validation set* contains observations, whose correct labels are also known, but which are not used for the training. They serve as a token to prevent lack of generality (called “overfitting”). The accuracy of the neural network predictions on these data is regularly verified and the learning process may be stopped if a decrease occurs.
- A *loss function* is applied to measure the inconsistency between the predictions and the expected labels.
- An optimization algorithm is then applied on the network to tune the trainable weights in order to reduce the loss function.

The neural network loops through the training set many times (many “epochs”) to optimize its weights. Since this set may be very large, it is chunked into small batches of observations, which are then processed simultaneously (e.g. taking advantage of GPU). After each epoch, we monitor the accuracy of the predictions on the training and validation set. At the end of the learning phase, a confusion matrix, showing the distribution of the predicted classes for a given true label, can be computed so as to visualize which classes are correctly predicted and which ones are not.

**Application to side-channel attacks** Deep-learning models may be used to retrieve sensitive data in profiled or non-profiled contexts. In the former, one way to adapt the problem to their usage is to give the power consumption traces directly as input for the neural network to learn. The targeted labels are not necessarily the raw sensitive data, but may be the result of a leakage model function applied to them. A common choice is the use of the Hamming weight function.

### 1.3 Related works

Side-channel attacks based on deep-learning is still a recent research field but some works have already been published.

**ASCAD public database** In order to have common benchmarks for researchers to evaluate their neural networks, ANSSI and CEA have published a public database of power consumption traces for side-channel attacks, called ASCAD [10]. Their traces were gathered during the execution of a masked 128-bit AES implementation on the ATMega8515 MCU device.

*Advanced Encryption Standard* The AES is a encryption algorithm using the same key for both encryption and decryption (i.e. symmetric). This algorithm manipulates the following data:

- A plaintext, which is the message being encrypted. It has a length of 128 bits.
- A ciphertext, which is the resulting encrypted message. It has also a length of 128 bits.
- A secret key, which is used to encrypt the plaintext. This key is a sensitive data and has also a length of 128 bits in the case of the ASCAD database.

During the AES execution, some internal variables are correlated to the power consumption. One of the operations executed is a non-linear byte substitution, called a *Sbox*. In the case of a 128-bit AES, the first time this operation is done, it is applied to all bytes of the plaintext, performing the following operation:

$$S = Sbox(plaintext\_byte \oplus key\_byte) \quad (1)$$

where  $(S, plaintext\_byte, key\_byte) \in \llbracket 0, 255 \rrbracket^3$

The substitution table being known and reversible,  $S$  is a sensitive data because it depends on the secret key. Since the plaintext may be known by the attacker, he may retrieve the secret key with a first-order attack.

*Masking* In order to prevent such attacks, countermeasures are implemented. In ASCAD database, a 8-bit random mask is applied to the output of the Sbox:

$$S = Sbox(plaintext\_byte \oplus key\_byte) \oplus M \quad (2)$$

where  $M \in \llbracket 0, 255 \rrbracket$

This increases by one the order of the attack required to retrieve the key. One shall know the mask  $M$  and the masked Sbox output  $S$  so as to retrieve the key byte.

In this article, we compare the performances of the best MLP proposed in [10] to our solution. The Signal-to-Noise ratio of the sensitive data in these traces has been discussed in [10].

**Non-profiled deep-learning-based side-channel attacks** In [13], a new side-channel attack method in non-profiled context has been introduced. It relies on the difference of learning rates of a neural network, when the output labels are based on the correct key hypothesis or not. This has been shown to be efficient against masked attack traces, but requires high computational power due to the fact that all key hypotheses must be tested.

## 1.4 Our contribution

First of all, we introduce in Section 2 a new layer called “Spread”, which improves greatly the performance of first-order side-channel attacks based on neural networks, compared to the ASCAD best MLP. Then, in Section 3, we extend this kind of attack to second-order attacks. Eventually, we propose in Section 4 a Bayesian correction of the neural network predictions, which reduces even more the number of attack traces required.

## 2 First-order attack

### 2.1 Adversary strength

In this model, as summarized in Figure 1, the adversary has at his disposal a similar device to the one being attacked, and can run it a given amount of times during a training phase. At each run, he varies the plaintext and acquires the corresponding consumption trace. In this scenario, the adversary can not change the key, but knows its value and is also able to read the ciphertexts.

After that, during a so called *attack phase*, the targeted device may be run a given amount of times. At each run, the adversary may change the plaintext and acquires the attack trace with its respective ciphertext. The goal is to retrieve the key used by the device under attack. This is the same attack model as in ASCAD publication [10].

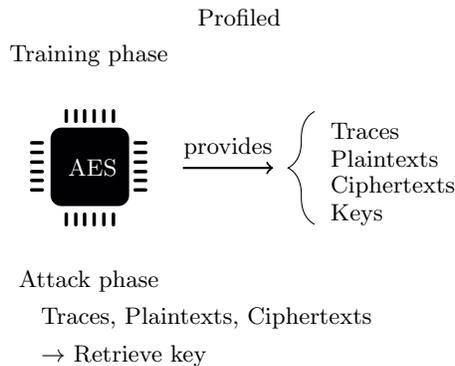


Fig. 1: Attack context.

### 2.2 Protocol

In the first phase of the protocol, the neural network is trained to predict the Hamming weight of the Sbox output value, using the acquired traces from the

available device. After that, the second phase of our protocol relies on the Maximum Likelihood Estimator (MLE). First (step A), we apply the trained neural network to each attack trace, so as to obtain a probability vector of the Hamming weight of the Sbox output value. Then (step B), knowing the Hamming weight distribution and assuming that the Sbox outputs are uniformly distributed over all possible byte values, we compute the probability for each Sbox output byte to be the correct one, using Equation 3.

$$Pr(S = s) = \frac{Pr(HW(S) = HW(s))}{C_8^{HW(s)}} \quad (3)$$

where  $Pr(HW(S) = HW(s))$  is given by our neural network (step A). Afterwards (step C), these probabilities can be used to derive probabilities over the keys, since one Sbox output and one plaintext match exactly one key byte, as recalled in Equation 4.

$$key\_byte = Sbox^{-1}(S) \oplus plaintext\_byte \quad (4)$$

Eventually (step D), the sum of the logarithm of each key hypothesis probability over all attack traces gives us the likelihood of each key. The higher the likelihood, the more likely is one key hypothesis. The entire recombination process is summarized in Figure 2 and the algorithm is described, step by step, in Algorithm 1.

**Design choice** We decided to implement the Maximum Likelihood Estimator (MLE) to combine together the probability vectors of each attack trace, since it has two interesting properties, according to [7]:

- The MLE is the best estimator asymptotically. This means that it has the best convergence rate when the number of traces increases.
- The MLE is consistent under appropriate conditions. Therefore, it should converge to the true value when the number of traces increases.

More details about these conditions may be found in [7].

Given this protocol, the neural network predictions play a determining role in the key retrieval process. In the following two subsections, we give more details about its implementation and specifically about the new layer, called *Spread layer*, which we devised.

### 2.3 Spread layer

**Core concept** The design of the Spread layer is based on the observation that the leakages in our traces are located at some time samples, which are always matching extreme values (e.g. consumption peaks). At these points, the power consumption is linearly correlated to the Hamming weight of the sensitive data targeted. Therefore, we need a neural network able to efficiently classify traces based on different values at some points. The only layer able to perform such

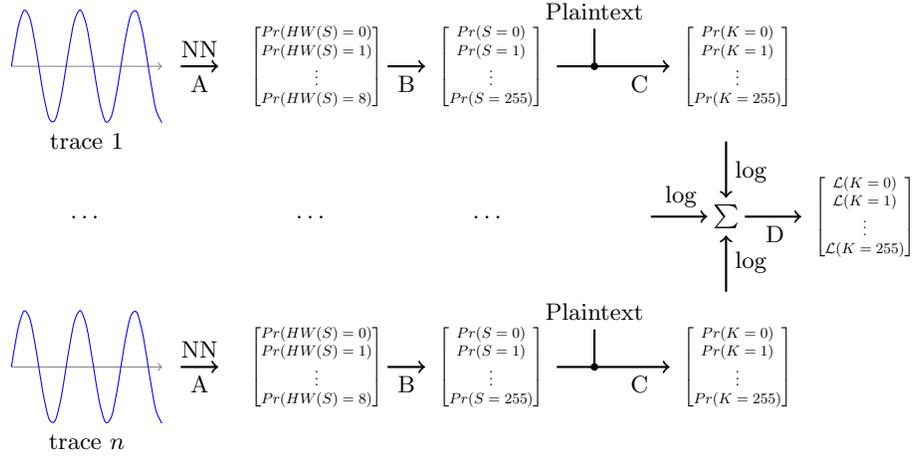


Fig. 2: Recombination of attack traces for the first-order attack.

**Algorithm 1:** Secret key byte retrieval for the first-order attack.

---

**Input:** *traces*: Attack traces  
**Input:** *NN*: Neural network giving prob. vect. for HW(Sbox.byte)  
**Output:** Secret key byte

**for** *trace*  $\in$  *traces* **do**

**A**     $probSboxHW[] \leftarrow NN(trace);$   
       **for**  $s \in All(Sbox.bytes)$  **do**

**B**        $probMatrix[s] \leftarrow \frac{probSboxHW[HW(s)]}{C_8^{HW(s)}};$   
       **end**

**for**  $s \in All(Sbox.bytes)$  **do**

**C**        $v_n[Sbox^{-1}(s) \oplus ptxt] \leftarrow probMatrix[s];$   
       **end**

**D**        $v \leftarrow v + \log(v_n);$

**end**  
**return**  $argmax(v);$

---

classification is the activation layer, because its non-regularity can be used to have a very different response according to the input value. For instance, a binary step, having 0 or 1 as output if its input is respectively negative or positive, will efficiently discriminate 2 values, which have been centered around zero. But in our case, we want to discriminate between all possible Hamming weights. The Spread layer is intended to perform this operation more efficiently. The basic idea is to transform the output of a neuron into a spatially-encoded information at the expense of a dimensionality increase, since the latter is known to be efficiently handled by neural networks. We implemented it directly using Tensorflow (and not Keras backend, which lacks some functions at the time of this writing).

**Implementation details** As described in Figure 3, the Spread layer extends the dimension of the previous layer by a *spread factor*, its only hyperparameter. Each new neuron created is related to a neuron of the previous layer and is assigned a centroid from 0.5 to *spread factor* - 0.5. When a value  $x$  is passed as input by the previous neuron, this value is remapped linearly in the interval  $[0, \textit{spread factor}]$  to a new value  $x'$ . Then, the 1 or 2 neurons, whose centroids are the closest to  $x'$ , output a value proportional to their distance to  $x'$ , more precisely:

$$\forall x' \in [0, \textit{spread factor}], \forall \textit{neuron } n_c \text{ with centroid } c, \quad n_c(x') = \begin{cases} 1 & \text{if } c = 0.5 \wedge x' < c \\ 1 & \text{if } c = \textit{spread factor} - 0.5 \wedge x' > c \\ \max(0, 1 - \textit{abs}(c - x')) & \text{otherwise} \end{cases} \quad (5)$$

**Behavior** The Spread layer outputs values belonging to  $[0,1]$ , whose sum per input neuron always equals one. Most of the time, 2 neurons have a non-zero output per input neuron, except when one has a value equal to 1, which occurs either when the input exactly matches a centroid, or when the output saturates at the borders.

The training phase consists in learning the minimum and maximum output values of each input neuron, in order to map  $x$  without saturation. Due to implementation constraints, the initial interval is  $[0,1]$  and the previous layer shall output at least over this interval for better fitting. But this is the case most of the time with the usual initializers.

In the next subsections, we describe how we implemented the spread layer in a neural network.

## 2.4 Neural network structure

Our neural network has the structure described in Figure 4. The couples indicate the shape of the data, the first dimension being the not-fixed batch size, hence

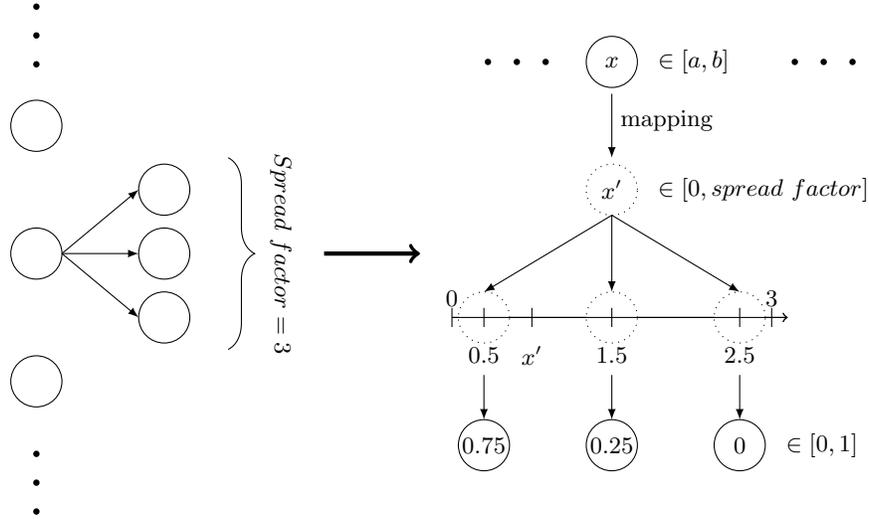


Fig. 3: Spread layer principle with example spread factor of 3.

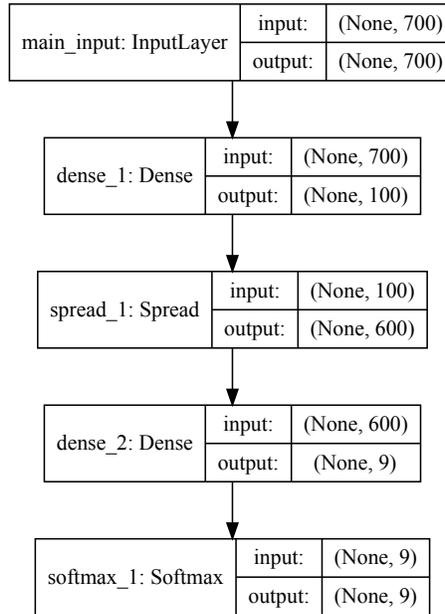
“None”, but equals to 100 for all our tests. It has been trained to predict the Hamming weight of the Sbox output, that is:

$$y = \text{HammingWeight}(\text{Sbox}(\text{plaintext} \oplus \text{key})) \tag{6}$$

It has been compiled using the Nadam optimizer and the categorical crossentropy loss function. At the beginning of this section, we presented the protocol used to retrieve the key and the way we implemented the neural network. In the next two sections, we evaluate and compare our proposed improvement to state-of-the-art results.

### 2.5 Validation

To compare our model (referred to as *Spread* in the following parts) to others, we implemented the ASCAD best MLP (referred to as *MLP* in the following parts). Since this model uses the identity leakage model instead of the Hamming weight, we decided to test all combinations of models and leakage models; that is: *MLP & ID*, *MLP & HW*, *Spread & ID* and *Spread & HW*. We also adapted the whole attack process in order to handle the identity leakage model, and tested all these combinations with various amounts of training traces. We stopped the training after 80 epochs so as to lower the overfitting. Therefore, we deem the learning speed as an important factor in our tests, since the attack process shall be repeated 16 times to recover all key bytes.

Fig. 4: Architecture of the *Spread & HW* neural network.

We then performed the attack on subsets of remaining ASCAD traces. Each subset contains 5,000 attack traces. We computed the mean rank of the correct key byte over all attacks, and considered the attack as a success (PASS) if all the attacks on subsets succeeded within less than 5,000 attack traces; that is, that the correct key byte is the most likely (i.e. has rank 0) at some points and keeps its rank until the end of our subset. We indicated in brackets the minimal amount of traces required such that all attacks succeed.

On the other hand, we considered the attack a failure (FAIL) if any of our attacks did not lead to the correct key byte in the end, in which case we indicate the amount of succeeded attacks divided by the amount of total attacks.

In practice, the labels do not match those of Equation 6, but also take into account the mask values. As we know them during the attack phase, this is equivalent to a first-order AES implementation, and is what has been done in [10].

## 2.6 Results and comparisons

Figure 5 compares the mean rank of the correct key byte according to the amount of attack traces for the four combinations of neural networks and leakage models, given 1,000 training traces. We observed that the only neural network able to pass the test is the *Spread & HW*. If we use the full ASCAD training database, that is 50,000 training traces, all combinations succeed. As the results are very

clear for low amounts of training traces, we wanted clear results on the whole training set too, and decided to lower the attack set size to 100 traces, in order to compute our mean over 100 attack sets, as shown in Figure 6. In this case, *MLP & ID* and *Spread & ID* have similar results. As already stated in [10], we can verify that *MLP & HW* is much worse than *MLP & ID*. But *Spread & HW* shows impressive results by requiring for its worst-case attack less than half of the traces needed by the worst-case *MLP & ID* attack.

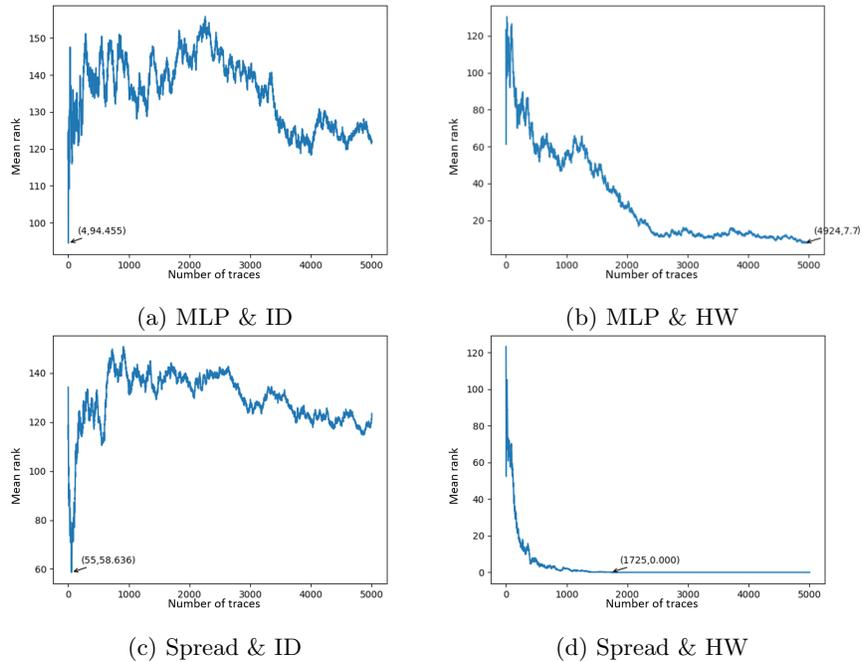


Fig. 5: Mean rank of the correct key byte as a function of number of attack traces, given 1,000 training traces.

Table 1 compares all four combinations for various amounts of training traces.

## 2.7 Conclusion

The use of our Spread layer-based network shows better results regarding the number of attack traces required, regardless of the number of training traces. It allows successful attacks with a small number of available training traces and improves the convergence speed towards the correct key byte.

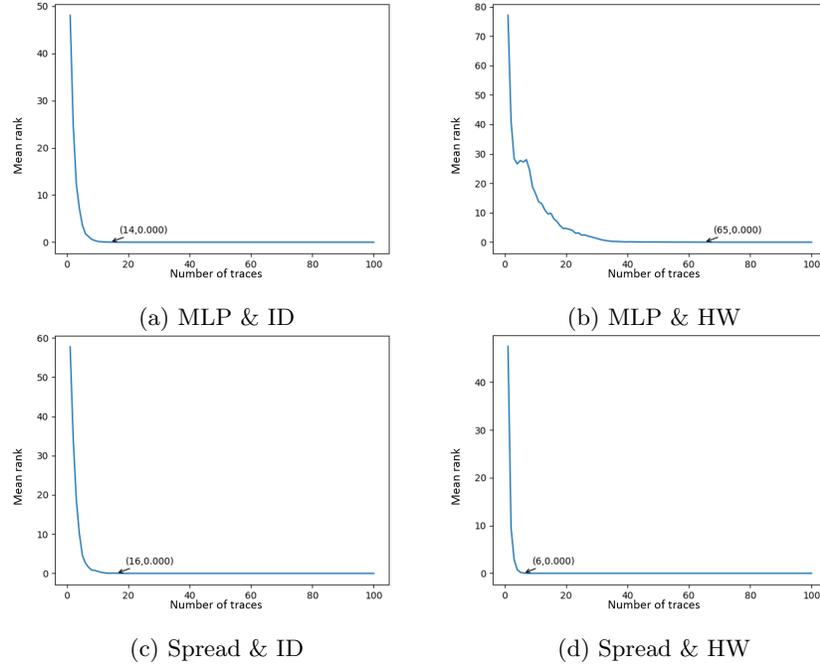


Fig. 6: Mean rank of the correct key byte as a function of number of attack traces, given 50,000 training traces.

Table 1: Comparison of all four networks according to the amount of training traces.

Nb of training traces	1,000	2,000	4,000	8,000
MLP & ID (ASCAD)	FAIL (0/11)	FAIL (0/11)	FAIL (0/11)	FAIL (1/10)
MLP & HW	FAIL (2/11)	FAIL (9/11)	FAIL (1/11)	PASS (230)
Spread & ID	FAIL (0/11)	FAIL (0/11)	FAIL (7/11)	PASS (100)
Spread & HW	PASS (1302)	PASS (116)	PASS (35)	PASS (13)
Nb of training traces	16,000	32,000	50,000	
MLP & ID (ASCAD)	PASS (868)	PASS (12)	PASS (7)	
MLP & HW	PASS (353)	PASS (9)	PASS (5)	
Spread & ID	PASS (15)	PASS (11)	PASS (6)	
Spread & HW	PASS (7)	PASS (7)	PASS (5)	

This behavior is all the more interesting since real-life devices, like the one used for the profiling phase, usually implement counters, limiting the number of consumption traces, which can be acquired. Encouraged by the promising results obtained in Section 2, we decided to adapt this first-order attack protocol to a second-order attack.

### 3 Second-order attack

#### 3.1 Adversary strength

In this model, the adversary has at his disposal a similar device as the one being attacked. During the training phase, he can vary the plaintexts and the masks used a given amount of times, and acquires the consumption traces as well as the ciphertexts. The key is fixed but known to him.

During the attack phase, the adversary can run the attacked device a given amount of times, may change the plaintexts, but knows neither the masks, nor the key. The adversary’s goal is to retrieve this latter key.

#### 3.2 Protocol

Here, we adapted the previously used protocol to the case, where we know neither the Sbox output, nor the mask. First (step A), we apply a trained neural network on each of our attack traces, so as to obtain a probability vector of the Hamming weights of the masked Sbox output value and of the mask. Then (step B), knowing the Hamming weight distribution and assuming that the keys and the masks are uniformly distributed over all possible byte values, we compute the probability for each couple (*Mask\_byte*, *Masked\_Sbox\_byte*) to be the correct one, using Equation 7. This is justified by the fact that the Sbox output and the mask are independent random variables when the key is unknown.

$$Pr(\text{Mask\_byte} = m \cap \text{Sbox\_byte} = s) = \frac{\text{probMask}[m] \cdot \text{probSbox}[s]}{C_8^{HW(m)} \cdot C_8^{HW(s)}} \quad (7)$$

Afterwards (step C), these probabilities can be used to derive probabilities over the keys. Since one couple and one plaintext match exactly one key byte, as recalled in Equation 8, we keep the maximum probability associated to all couples implying the same given key.

$$\text{key\_byte} = \text{Sbox}^{-1}(\text{sbox\_byte} \oplus \text{mask\_byte}) \oplus \text{plaintext\_byte} \quad (8)$$

Eventually (step D), the remaining recombination is the same as for the previous attack scenario. The whole recombination process is summarized in Figure 7 and the algorithm is described, step by step, in Algorithm 2.

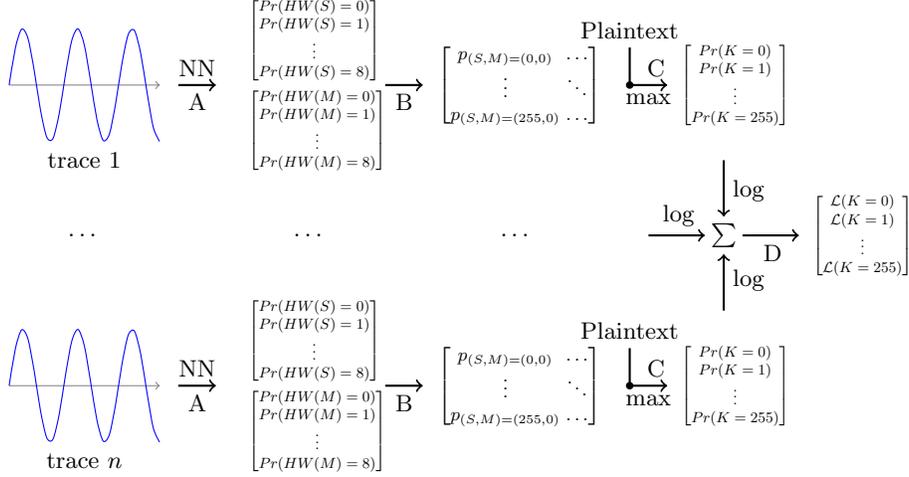


Fig. 7: Recombination of attack traces for the second-order attack.

---

**Algorithm 2:** Secret key byte retrieval for the second-order attack.
 

---

**Input:** *traces*: Attack traces**Input:** *NN*: Neural network giving prob. vect. for HW(Mask\_byte) and HW(Sbox\_byte)**Output:** Secret key byte**for** *trace*  $\in$  *traces* **do**

A  $\left| \begin{array}{l} \textit{probMask}[], \textit{probSbox}[] \leftarrow \textit{NN}(\textit{trace}); \\ \textbf{for } (m, s) \in \textit{AllCouples}(\textit{Mask\_bytes}, \textit{Sbox\_bytes}) \textbf{ do} \\ \textbf{B } \quad \left| \textit{probMatrix}[m, s] \leftarrow \frac{\textit{probMask}[m] \cdot \textit{probSbox}[s]}{C_8^{\textit{HW}(m)} \cdot C_8^{\textit{HW}(s)}}; \\ \textbf{end} \\ \textbf{for } \textit{key\_byte} \in \textit{All}(\textit{Key\_bytes}) \textbf{ do} \\ \textbf{C } \quad \left| v_n[\textit{key\_byte}] \leftarrow \max_{(m,s), \textit{key\_byte}=\textit{Sbox}^{-1}(s \oplus m) \oplus \textit{ptxt}}(\textit{probMatrix}[m, s]); \\ \textbf{end} \\ \textbf{D } \quad v \leftarrow v + \log(v_n); \\ \textbf{end} \\ \textbf{return } \textit{argmax}(v); \end{array} \right.$

---

**Design choice** We decided to keep the maximal probability over all possible couples of Mask and Sbox for the following reason. We may model the choice of the correct key byte  $k$  as  $2^8$  statistical hypothesis tests like this: the null hypothesis is “ $k$  is the correct key” and the other alternative is “ $k$  is not the correct key”. Then, we want to minimize the risk to reject the null hypothesis when this one appears to be true (also called minimizing the Type-I error) at the expense of requiring more attack traces. Keeping the maximal probability minimizes this kind of error.

### 3.3 Architecture of our neural network

Our neural network has been designed with two branches. The first one is trained to predict the Hamming weight of the masked Sbox output. The other one is intended to predict the Hamming weight of the mask applied, which is also to be found in the consumption traces. This architecture is depicted in Figure 8. We altered the MLP best from ASCAD the same way, adding a branch for each sensitive data we want to retrieve. The compilation parameters have not been changed.

### 3.4 Results

**Behavior comparison** In Figure 9, we depicted the confusion matrices of the two branches for the *MLP & HW* and the *Spread & HW* trained with 2,000 training traces. Two different behaviors can be noticed. On the one hand, the *MLP & HW* tends to predict the most represented label first, and then to grasp the more distant labels. On the other hand, the *Spread & HW* best predicts labels around the most represented one, which leads to a slightly lesser accuracy. Nevertheless, this is of no use for the secret key retrieval algorithm to have only the most represented label predicted. Rather than that, we would like to have at least two distinct labels with accurate enough predictions. A way to represent that is to compute the Pearson correlation coefficient of the confusion matrix. If it has a high absolute value, then the predictions are linearly correlated to the true labels, and we can use these for our algorithm. But if the predictions do not depend on the true label, the coefficient will be near to zero. And indeed, we observe in Figure 10 that this coefficient is higher for the *Spread & HW* neural networks, which leads to the success of all attacks, compared to the other one, having no successful attack at all.

**Comparison according to number of training traces** Table 2 compares all four combinations for various amounts of training traces. The identity leakage model does not succeed for both neural networks. The *Spread & HW* succeeds from 2,000 training traces, and requires on the whole training set two times less attack traces than *MLP & HW*.

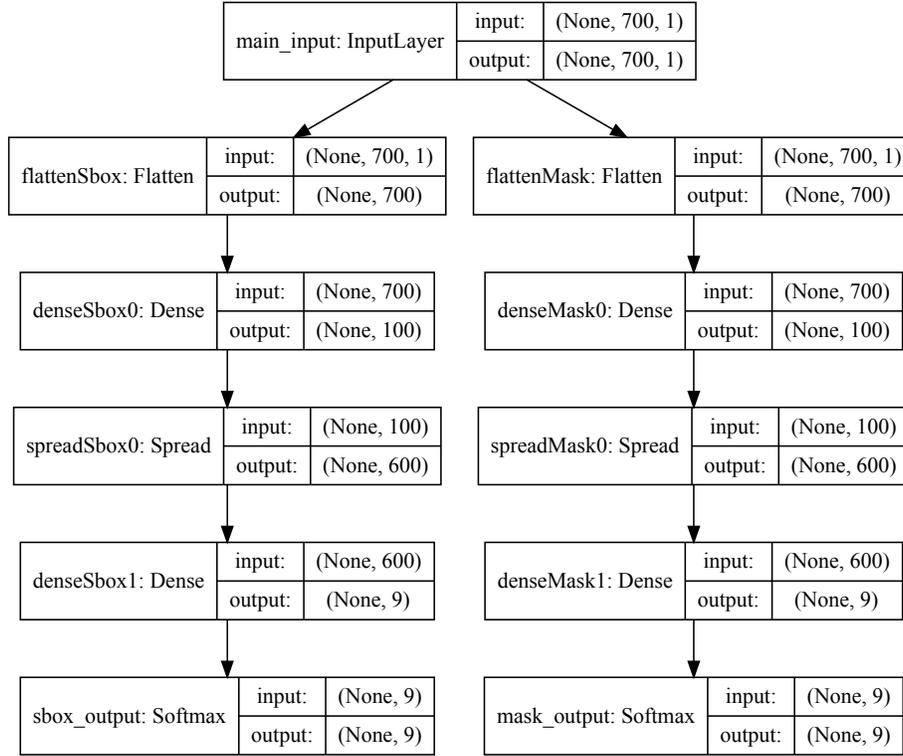


Fig. 8: Architecture of the *Spread & HW* neural network with two branches.

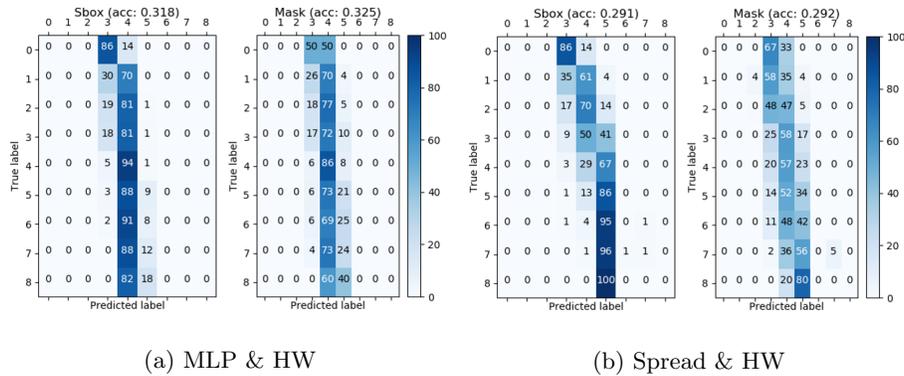


Fig. 9: Comparison of confusion matrices in the second-order attack case with 2,000 training traces.

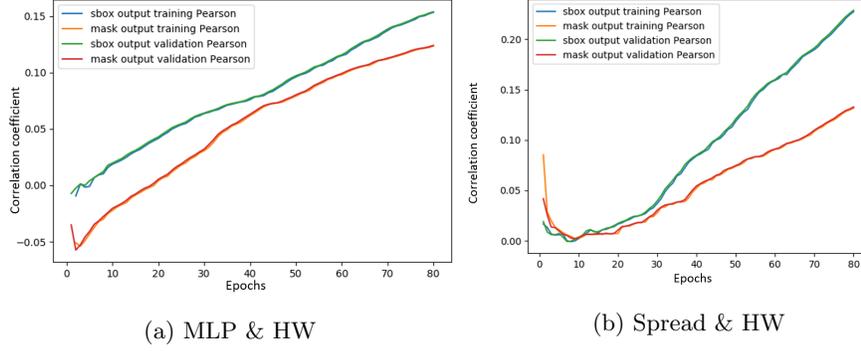


Fig. 10: Comparison of Pearson correlation coefficient according to the number of epochs in the second-order attack case with 2,000 training traces.

Table 2: Comparison of all four networks according to the amount of training traces.

Nb of training traces	1,000	2,000	4,000	8,000
MLP & ID (ASCAD)	FAIL (0/11)	FAIL (0/11)	FAIL (0/11)	FAIL (1/10)
MLP & HW	FAIL (0/11)	FAIL (0/11)	FAIL (0/11)	FAIL (7/10)
Spread & ID	FAIL (0/11)	FAIL (0/11)	FAIL (0/11)	FAIL (0/10)
Spread & HW	FAIL (0/11)	PASS (2308)	PASS (2465)	PASS (1113)
Nb of training traces	16,000	32,000	50,000	
MLP & ID (ASCAD)	FAIL (0/8)	FAIL (0/5)	FAIL (0/2)	
MLP & HW	PASS (1809)	PASS (1700)	PASS (786)	
Spread & ID	FAIL (0/8)	FAIL (0/5)	FAIL (0/2)	
Spread & HW	PASS (878)	PASS (699)	PASS (344)	

### 3.5 Conclusion

Similarly to what we have seen in Section 2 for first-order attacks, we observed here that the use of our Spread layer-based network also shows better results regarding the number of attack traces required for second-order attacks. Moreover, our improvement allows successful attacks with a small number of available training traces and improves the convergence speed towards the correct key byte. In the next section, we introduce the last improvement of this article. It reduces the number of attack traces required. As it relies on the confusion matrix and Bayes' theorem, we call it the *Bayesian correction*.

## 4 Bayesian correction

The use of confusion matrices to provide information on the errors and improve the accuracy has already been studied, among others, in [2, 1, 9, 15]. Here, we adapted these approaches to our case.

### 4.1 Principle

The confusion matrix gives us valuable information about the bias and variance of the neural network for each input label. We decided to leverage it in order to correct the predictions. Indeed, its output may be considered as a random variable  $Y$ , whose value depends on the expected label, a random variable  $X$ . The confusion matrix let us know the distribution of  $Y$  knowing  $X$ , that is  $Pr(Y|X)$ . By means of Bayes' theorem, which is recalled in Equation 9, we can retrieve the probability of the true label knowing the output of the neural network, that is  $Pr(X|Y)$  and thus correct somewhat its predictions.

$$Pr(X|Y) = \frac{Pr(Y|X) \cdot Pr(X)}{Pr(Y)} \quad (9)$$

In the case of the Hamming weight leakage model,  $X$  follows a binomial distribution. More precisely,  $X \sim \mathcal{B}(8, \frac{1}{2})$  and the distribution of  $Y$  is known thanks to the law of total probability. The final formula is summarized in Equation 10.

$$Pr(X|Y = y) = \frac{Pr(Y = y|X) \cdot Pr(X)}{\sum_{x=0}^8 Pr(Y = y|X = x) \cdot Pr(X = x)} \quad (10)$$

*Limitation* This correction is only valid if the confusion matrix over the training set is the same as over the attack set. This should approximately be the case if the neural network did not overfit during the training phase, as shown in Figure 11.

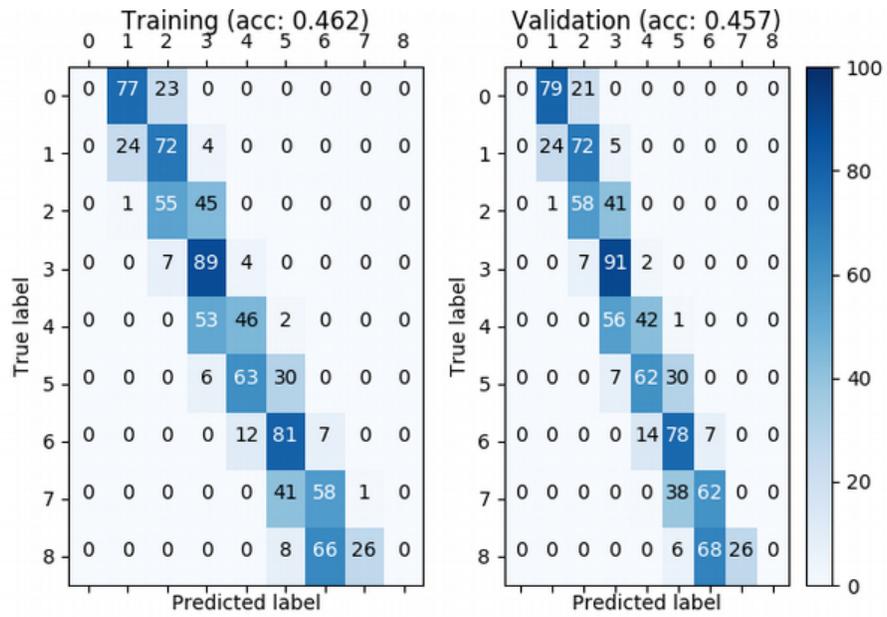


Fig. 11: Comparison of confusion matrix over training and validation set for *Spread & HW* in the first-order attack trained over 8,000 traces.

## 4.2 Results

We compared the mean ranks of the correct key byte with and without correction for the same trained *Spread & HW* neural network. Figure 12 shows the confusion matrices with and without this correction in the second-order attack. We notice that the correction enables better predictions of some Hamming weights with a few samples (e.g. labels 0 and 8). On the other hand, the mask predictions, which were strongly biased, become very fuzzy. Their respective correct key byte mean ranks are depicted in Figure 13. The results are worse than expected. It seems that a biased estimator is better than a corrected one in the case of a fuzzy confusion matrix, due to the presence of “columns” in the original confusion matrix, that is subgroups of  $X$  values for which the distribution of  $Y$  is almost independent. The correction is not working great in this case, and even though our neural network is biased, the amount of required traces will be lower without correction, since it is more accurate over the most represented classes. Therefore, we decided to correct only the Sbox predictions.

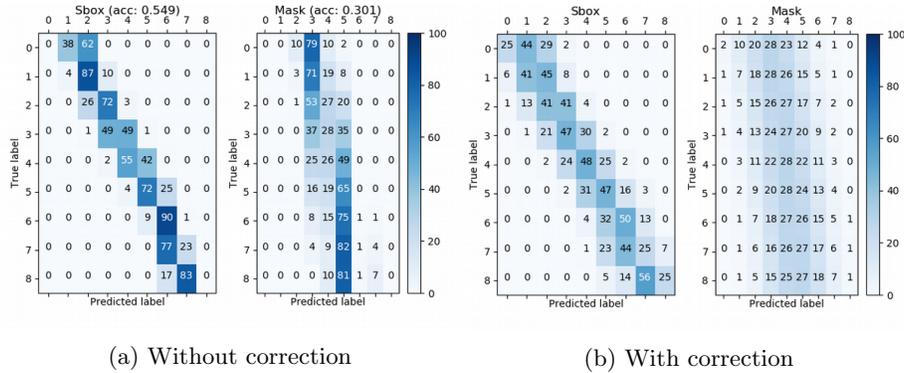


Fig. 12: Confusion matrices of *Spread & HW* trained on 16,000 training traces.

**First-order attack scenario** Table 3 compares the effect of the correction in the first-order attack case. The failure of the improvement on 1,000 training traces is due to the neural network inclination to predict the most represented labels and thus having a low Pearson correlation coefficient, as depicted in Figure 14.

**Second-order attack scenario** Table 4 compares the effect of the correction in the second-order attack case. We do not apply any correction to the mask predictions because of their high tendency to be less correlated to their inputs.

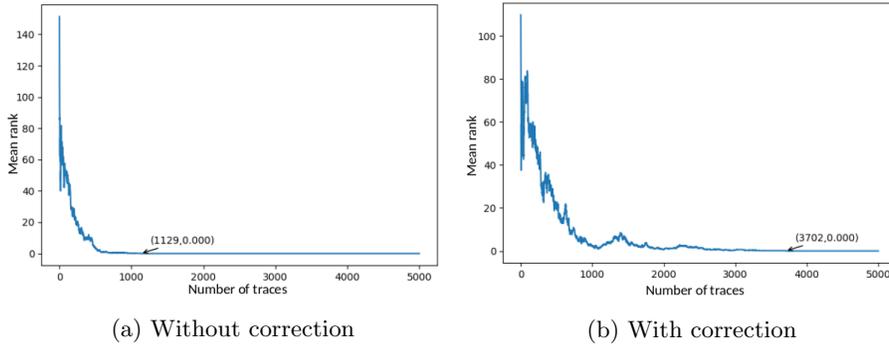


Fig. 13: Mean rank of correct key byte over 5 attacks.

Table 3: Comparison of *Spread* & *HW* without and with correction.

Nb of training traces	1,000	2,000	4,000	8,000
Spread & HW	PASS (1302)	PASS (138)	PASS (44)	PASS (18)
Spread & HW & Correction	FAIL (2/11)	PASS (57)	PASS (23)	PASS (9)
Nb of training traces	16,000	32,000	50,000	
Spread & HW	PASS (8)	PASS (8)	PASS (6)	
Spread & HW & Correction	PASS (8)	PASS (7)	PASS (5)	

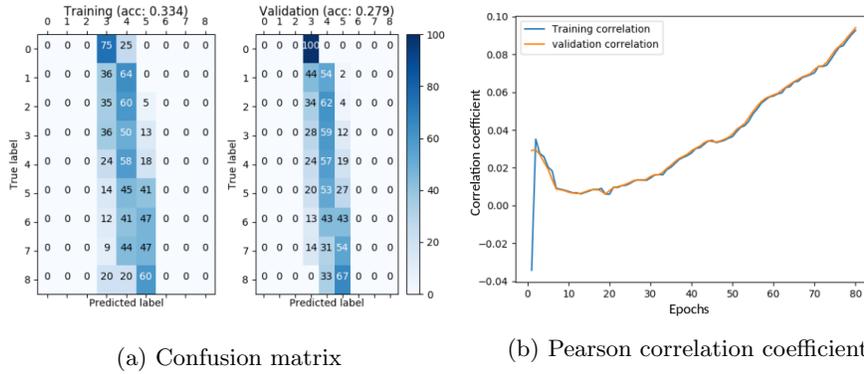


Fig. 14: Metrics of *Spread* & *HW* trained with 1,000 traces in the first-order attack scenario.

Table 4: Comparison of *Spread* & *HW* without and with correction.

Nb of training traces	4,000	8,000	16,000	32,000
Spread & HW	PASS (2465)	PASS (1237)	PASS (1305)	PASS (1205)
Spread & HW & Correction	PASS (2272)	PASS (947)	PASS (1042)	PASS (556)
Nb of training traces	50,000			
Spread & HW	PASS (466)			
Spread & HW & Correction	PASS (338)			

### 4.3 Conclusion

The Bayesian correction of the *Spread & HW* predictions enables all our attacks to succeed with even less attack traces, by reducing this amount of up to more than 50%. Unfortunately, this method does not succeed when the output of the neural network is almost independent of its input. In such cases, it seems better to have few variances on the most represented labels, rather than to make the confusion matrix fuzzy. This is why we decided to correct only the Sbox predictions for the second-order attack.

## 5 Conclusion and perspectives

In this paper, we propose two different improvements, which increase the effectiveness of profiled side-channel attacks based on deep-learning techniques.

The first improvement is a new neural network layer, called Spread layer. Its goal is to reduce the amount of both training and attack traces required to retrieve the key. In order to evaluate the level of this reduction with the state-of-the-art neural network used for side-channel attacks, we followed the benchmark strategy promoted in [10]. The result of this evaluation confirms that the goal of the Spread layer has been achieved for first-order attacks as well as for second-order attacks.

The second improvement is a method of correcting the neural network predictions. It reduces most of the time even more the number of attack traces required to succeed the key retrieval.

Our new layer might be useful for other applications in the field of deep-learning, when one needs to perform a classification relying on the values of some input features. Indeed, the resulting neural network might be shallower than these based on the common activation layers.

Future works may be to assess our method with higher-order masking. In this case, the neural network would have one branch per share and the post-processing algorithm would comprise a matrix with more dimensions. Another possibility is to lower the hypothesis so as to target traces without mask knowledge.

## Acknowledgment

The authors also wish to thank Mr. Michael Miller, Mr. Guillaume Reymond, Mr. Michel Agoyan and Mr. Thomas Sarno for helpful discussions.

## References

1. 5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016, Kumamoto, Japan, July 10-14, 2016. IEEE (2016), <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7557346>
2. 20th International Conference on Information Fusion, FUSSION 2017, Xi'an, China, July 10-13, 2017. IEEE (2017), <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8002854>
3. Batina, L., Gierlichs, B., Prouff, E., Rivain, M., Standaert, F., Veyrat-Charvillon, N.: Mutual information analysis: a comprehensive study. *J. Cryptology* **24**(2), 269–291 (2011). <https://doi.org/10.1007/s00145-010-9084-8>, <https://doi.org/10.1007/s00145-010-9084-8>
4. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: Jr., B.S.K., Koç, Ç.K., Paar, C. (eds.) *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers. Lecture Notes in Computer Science*, vol. 2523, pp. 13–28. Springer (2002). [https://doi.org/10.1007/3-540-36400-5\\_3](https://doi.org/10.1007/3-540-36400-5_3), [https://doi.org/10.1007/3-540-36400-5\\_3](https://doi.org/10.1007/3-540-36400-5_3)
5. Doget, J., Prouff, E., Rivain, M., Standaert, F.: Univariate side channel attacks and leakage modeling. *IACR Cryptology ePrint Archive* **2011**, 302 (2011), <http://eprint.iacr.org/2011/302>
6. Gierlichs, B., Batina, L., Preneel, B., Verbauwhede, I.: Revisiting higher-order DPA attacks: Multivariate mutual information analysis. *IACR Cryptology ePrint Archive* **2009**, 228 (2009), <http://eprint.iacr.org/2009/228>
7. Goodfellow, I., Bengio, Y., Courville, A.: *Deep Learning*. MIT Press (2016), <http://www.deeplearningbook.org>
8. Joye, M., Quisquater, J. (eds.): *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings, Lecture Notes in Computer Science*, vol. 3156. Springer (2004). <https://doi.org/10.1007/b99451>, <https://doi.org/10.1007/b99451>
9. Leijon, L., Henter, G.E., Dahlquist, M.: Bayesian analysis of phoneme confusion matrices. *IEEE/ACM Trans. Audio, Speech & Language Processing* **24**(3), 469–482 (2016). <https://doi.org/10.1109/TASLP.2015.2512039>, <https://doi.org/10.1109/TASLP.2015.2512039>
10. Prouff, E., Strullu, R., Benadjila, R., Cagli, E., Dumas, C.: Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. *IACR Cryptology ePrint Archive* **2018**, 53 (2018), <http://eprint.iacr.org/2018/053>
11. Rao, J.R., Sunar, B. (eds.): *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings, Lecture Notes in Computer Science*, vol. 3659. Springer (2005). <https://doi.org/10.1007/11545262>, <https://doi.org/10.1007/11545262>
12. Schindler, W.: Advanced stochastic methods in side channel analysis on block ciphers in the presence of masking. *J. Mathematical Cryptology* **2**(3), 291–310 (2008). <https://doi.org/10.1515/JMC.2008.013>, <https://doi.org/10.1515/JMC.2008.013>
13. Timon, B.: Non-profiled deep learning-based side-channel attacks. *IACR Cryptology ePrint Archive* **2018**, 196 (2018), <http://eprint.iacr.org/2018/196>
14. Wiener, M.J. (ed.): *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, Lecture Notes in Computer Science*, vol. 1666. Springer (1999). <https://doi.org/10.1007/3-540-48405-1>, <https://doi.org/10.1007/3-540-48405-1>

15. Yu, J., Xiang, K., Cao, S., Song, T., Wang, X.: Ensemble simplified fuzzy artmap with modified plurality voting. In: 2014 7th International Conference on Intelligent Computation Technology and Automation. pp. 236–239 (Oct 2014). <https://doi.org/10.1109/ICICTA.2014.64>