Being Patient and Persistent: Optimizing An Early Stopping Strategy for Deep Learning in Profiled Attacks

Servio Paguada, Lejla Batina, Senior Member, IEEE, Ileana Buhan, and Igor Armendariz

Abstract—The absence of an algorithm that effectively monitors the deep learning models used in side-channel attacks increases the difficulty of a security evaluation. If an attack is unsuccessful, that could be due to multiple reasons. It can be that we are indeed dealing with a resistant implementation, but it is possible that the deep learning model used is faulty. In this contribution, we formalize two conditions, persistence and patience, for a deep learning model to be optimal and we propose an early stopping algorithm that reliably recognizes the model’s optimal state during training. The novelty of our solution is in an efficient implementation of guessing entropy estimation as a success metric used to measure the strength of a side-channel adversary. As a result, the model which uses our strategy for learning converges with fewer traces than other known methods.

Index Terms—Side-channel analysis, deep learning, early stopping

1 INTRODUCTION

In recent years, deep learning (DL) techniques [1] have been widely adopted to evaluate the resilience of cryptographic implementations against side-channel attacks. Several milestones have been reached, and deep learning became a mainstream side-channel analysis (SCA) evaluation technique [2], [3], [4], [5], [6], [7], [8], [9], [10]. Nevertheless, several aspects still have no satisfactory solution. For instance, preventing the model from underfitting/overfitting [11], [12], [13] is essential because these two phenomena make any deep learning model perform poorly. In the context of side-channel analysis, this represents an important issue because it is uncertain if we are dealing with a resistant cryptographic implementation or a faulty deep learning model.

A faulty model could be explained by several causes such as a sub-optimal architecture, poor choice of hyper-parameters, insufficient, or on the contrary, excessive training. In this paper, we address the problem of undert raining (insufficient) and overtraining (excessive). While we know which architectures work well in the area of side-channel attacks, training a deep network for an attack needs to be repeated for every side-channel evaluation. A failure corresponds to the case when the network is unable to work with a set of data different than the one used for its training, which is an essential feature for the success of a profiled attack [14], [15], [16]. A sub-optimal network is particularly inconvenient in the framework of evaluating the side-channel resilience of an implementation, as it leads to inconclusive evaluation results.

Solving the overtraining problem is not trivial, particularly in the case of side-channel evaluations as deep learning metrics do not match the metrics used for side-channel analysis. For example, accuracy the classical metric used to monitor the network state through the training process is used to stop the training process once the network reaches the desired accuracy. Unfortunately, it was shown that accuracy (and indeed all other machine learning metrics), which work well in other scenarios, is not suitable for monitoring the training of a deep network used to attack a cryptographic implementation [17], [18].

Only a few studies have addressed early stopping frameworks in the context of side-channel analysis. The closest work related to ours is from Robissout et al. [19], as they developed an early stopping strategy that computes the rank of a key using guessing entropy (GE) algorithm [20]. Their results suggested that a deep learning model reaches its optimal state when the rank of the correct key is close to the lower value of the averaged guessing entropy. However, this strategy is not reliable, because the window within their strategy claims the model reached its optimal state is too short; consequently, it has a high probability to outcome a false positive since there is not way to evaluate the repeatability.

Let us observe the example explained by Robissout et al. [19] and depicted in Fig. 1. Their strategy suggests that the deep deep learning network has reached one of its optimal states, once the guessing entropy of the correct key reaches the lower values in average. However, observe that we have two possible moments when the value of the guessing
proposed an early stopping strategy using a technique compatible with our early stopping strategy. We designed a new algorithm for guessing entropy in time, which significantly improves the computation time, and allows us to efficiently integrate guessing entropy in the training process.

Finally, we developed a customized version of a grid search algorithm compatible with our early stopping strategy. This grid search is interrupted when our early strategy declares the deep learning model optimal (according to the evaluator’s expectations), preventing the deep network from performing additional and unnecessary training.

entropy is “locally” the lowest. We would not know immediately if after epoch 6 (1st window of an optimal state) we will have a low value of the average guessing entropy or later (2nd window of an optimal state). Further, notice that both windows elapsed only 1 epoch, as the windows are too short we cannot ensure the network is able to repeat the result or to stay in the state indicated by the strategy.

Perin et al. [21] proposed an early stopping strategy using mutual information. However the overhead in computing mutual information during training iterations significantly increase the training time. Moreover, their approach requires to treat the neural network model as a Markov chain, which suggests that the amount of information related to the sensitive values that remains after passing through the layers of the neural network points to its optimal state. However, this resemblance of a Markov chain cannot be achieve with any activation functions. As showed [22], tanh is one of the few activation functions to apply this approach, limiting the application of the Perin et al. early stopping strategy.

While overfitting/underfitting represents the state when the network cannot generalize beyond its training set, there is another challenge related to the training process of a neural network. For a deep learning side-channel evaluation, it is not straightforward to find a model that effectively addresses the evaluation. In the deep learning field of general applications, the body of work in hyper-parameter tuning has motivated several works [23], [24]. However, regarding deep learning in side-channel analysis, just a few works have addressed this issue [25]. Arguably, we claim this is due to the lack of a metric to evaluate the state of the network, as we said before. Furthermore, a searching algorithm compatible with an early stopping strategy is also a factor.

To contribute in overcoming these issues, this paper introduces an early stopping strategy that relies on the guessing entropy metric. Our approach also includes a grid search algorithm [26] compatible with tuning a deep learning model when performing a side-channel evaluation. Our proposal uses an optimized version of the guessing entropy algorithm. Taking advantage of the faster computation time, it is feasible to evaluate the guessing entropy after every iteration during the training process. With this approach, guessing entropy plays the role of the metric that monitors the network stopping its training when reaching an optimal state. For this purpose, we define: (i) persistence the attribute which monitors the guessing entropy convergence in terms of true positive outcomes and (ii) patience the attribute that controls the confidence in the optimality of the model. By using the persistence and patience criteria, we avoid the limitations that affect the approach in [19]. Basically, our framework defines an area (we called it the area of hit) to evaluate the optimality of a deep learning model through the persistence and the patience axes. The persistence criterion (or axis) evaluates the learning of the deep learning model by measuring the amount of traces the guessing entropy remains inside the area. On the other hand, the patience measures the number of training iterations the persistence criterion repeats. In other words, the evaluator sets how many training iterations the guessing entropy should pass in meeting the persistence criterion to achieve the stop conditions.

Since the persistence attribute gives us the flexibility to specify the number of traces we require the guessing entropy to converge in. We define two possible scenarios or use cases where our strategy is used (i) the soft case and the (ii) greedy case. The soft case is convenient when conducting a side-channel evaluation without a requested number of traces. For instance, if the evaluation goal is to assess that it is feasible to exploit the device no matter the number of leakage traces. In contrast, if the evaluation requests to assess that the attack exploit the leakage within a number of traces. We need a way to specify it. The greedy case becomes the right choice. The latter use case could be applied when we look for a deep learning model that outperforms a previous attack applied to the same side-channel data.

Contrary to the mentioned works, our framework computes the monitor metric —guessing entropy in this case— in a feasible amount of time. Further, the architecture details of the neural network do not restrict the application of our early stopping strategy. Given that our algorithm, over which the frameworks lies, considers the times the neural network keeps the guessing entropy within the optimal state; then, our strategy’s outcomes are likely true positive all the time.

### 1.1 Contribution

The contributions of this work are threefold:

- We introduce an effective early stopping mechanism to monitor the deep learning models used in side-channel evaluation. Our early stopping strategy reliably recognizes the model’s optimal state during training; consequently, we increase the chance to assess the leakage correctly. Our results demonstrate that state-of-the-art models sub-optimally evaluate the leakage traces due to overfitting; our algorithm stops the training process at the model optimal state, resulting in a guessing entropy converging with fewer traces.
- We designed a new algorithm for guessing entropy which significantly improves the computation time, and allows us to efficiently integrate guessing entropy in the training process.
- Finally, we developed a customized version of a grid search technique compatible with our early stopping strategy. This grid search is interrupted when our early strategy declares the deep learning model optimal (according to the evaluator’s expectations), preventing the deep network from performing additional and unnecessary training.

![Fig. 1. Example of the early stopping strategy from [19] when it finds the optimal state of a deep learning model during training.](image-url)
steps. We used our grid search to test the greedy case of our experiments (Sect. 5).

1.2 Paper Organization
Section 2 briefly introduces the background on SCA and profiled side-channel attacks. Section 3 introduces the dataset used in the experiments. Section 4 discusses the main contribution of this paper and Section 5 gives results of our experiments. Section 6 concludes the paper.

1.3 Implementation Code
The repository containing all the implementations' code is available at: https://github.com/altter/being-patient-and-persistent.

2 BACKGROUND
In this section, we introduce the necessary background and notation for the rest of the paper.

2.1 Side-Channel Evaluation
Side-channel information (power, EM, or timing) obtained during the execution of a cryptographic implementation leaks information about sensitive data. A side-channel attack often leads to the cryptographic key's full compromise, making it an important topic in the hardware evaluation industry [27]. A side-channel evaluation is typically executed in three stages: (i) the acquisition of side-channel information. In this stage, the evaluator collects leakage traces\(^1\), (ii) pre-processing, where the goal is to remove noise from the traces and (iii) finally, leakage evaluation [28] or key extraction [29] is performed.

The third stage typically consists of statistical analysis, leakage assessment, or conducting side-channel analysis using classical approaches such as Differential Power Analysis (DPA) [29]. Recently introduced deep learning-based attacks belong to the class of profiled side-channel attacks. The appeal of a deep learning-based evaluation [30], [31], [32] is that it removes the necessity of performing the second step of pre-processing. Hence, the second stage mentioned above is not always mandatory but can still be helpful for the deep-learning attack.

2.1.1 Deep Learning Profiled Side-Channel Attack
A classical side channel attack [29] is performed in a black-box approach where the evaluator has no information about the target she is attacking. In a profiled side-channel attack, the evaluator has a clone of the target device, which she controls. This evaluation approach typically means that she can program the target device with a key of her choice and also choose the messages to be encrypted (i.e., the plain text). The goal of this so-called profiling phase is to learn how the device leaks information. In particular, during the evaluation of profiling attacks, several leakage traces of side-channel information are collected when the device under test performs the encryption using the plain text the evaluator sends. Repeating the process \( N \) times, the evaluator composes the set of traces \( \mathcal{P} \).

\[
\mathcal{P} = \left[ \begin{array}{c}
\{w_{0,1}, \ldots, w_{0,M}\}, \{p_0, k_0\}
\vdots \\
\{w_{N,1}, \ldots, w_{N,M}\}, \{p_N, k_N\}
\end{array} \right]
\]

Where \( t_i = \{w_{i,1}, \ldots, w_{i,M}\} \) is the \( i \)th trace(\( w_{i,j} \) its \( j \)th sample point). The \( p_i \) is its corresponding plain text vector while \( k_i \) is its previously configured secret key vector (note that \( p_i \neq p_j \) and it could also happen that \( k_i \neq k_j \)). Hence each trace has an associated plain text and key. The length of these two vectors are based on the keyspace \( K \) as \( k_i \in K \) where \( K \) maps to a \( GF(n) \). For instance, if the cryptographic primitive is the AES S-box (which maps to a \( GF(2) \)). The elements of the key sub-space take values from the set \( \{0, \ldots, 255\} \). From the plain text and key vectors, the evaluator chooses the index to attack, that is, the corresponding byte of the secret key. Normally, the exploit of a single byte is enough to assess if the device’s cryptographic implementation is compromised. By choosing a byte to attack, the evaluator defines an attack model in the form of an operation in the cryptographic algorithm. Particularly, this paper uses an attack model based on the AES substitution box:

\[
\delta = S-box(p_i[3] \oplus k_i[3])
\]

We chose the third byte as the traces dataset; ASCAD uses the same attack model (see Section 3).

Profiled attacks require a so-called classifier as the objective is to classify the chosen byte of the key out of all key candidates. Deep learning-based classifiers demonstrate being robust distinguishers in assessing the profiled attacks evaluation [2], [32]. Supervised training is the strategy we use to train the classifier [1]. Hence, we require to label the traces. The output of the attack model \( \delta \) is, in this case, the label for each trace in \( \mathcal{P} \). As a consequence, the dataset \( \mathcal{P} \) becomes:

\[
\mathcal{P} = \left[ \begin{array}{c}
\{w_{0,1}, \ldots, w_{0,M}\}, \delta_0, p_0
\vdots \\
\{w_{N,1}, \ldots, w_{N,M}\}, \delta_N, p_N
\end{array} \right]
\]

Given a deep learning model, we use an iterative training process that fits its hyper-parameters by updating them during the back-propagation stage. We feed the model with batches of profiling traces from \( \mathcal{P} \). After feed-forwarding the traces through the model, a loss function computes the error between the models’ label predictions and the true labels. This stage is repeated during several iterations known as epochs. As said, \( \mathcal{P} \) is the set of traces collected from the clone device to train the classifier. Denoted as \( A \) is the set of attack traces collected from the target device to evaluate if the deep learning model is able to exploit the secret key. The attack starts by randomly choosing a batch of \( N_a \) attack traces from \( A \) and use the classifier to generate a prediction vector \( \hat{p}_a \). The classifier predicts to which attack model’s output an attack trace belongs.

Until here we have conducted the first part of the attack, because the objective is to recover the actual key; the second operand in the XOR operation of the attack model argument. The second part consists in computing the guessing entropy

---

\(^1\) Leakage traces are time series with the measured power consumption of the target device while executing a cryptographic algorithm.
value using $P$. An average guessing vector $g$, for different sets of $N \in A$ is built as follows: $g = \text{sort}(E[\log(P)])$ where $E$ is the expectation for the different values of $P$. Given $g$, the guessing entropy (GE) function is defined as: $\text{GE}(g) = \text{rank}_{k^*}(g)$ where $\text{rank}_{k^*}$ is the position of the correct key $k^*$ in the average guessing vector.

The guessing entropy is the side-channel metric that connects a classifier (in this case, a deep learning model) with the chosen attack model. As said in the introduction, this contribution aims to incorporate guessing entropy into the training process. The guessing entropy is reliable for monitoring if the classifier has achieved an optimal state. Since it connects the attack model, we can compute it during each epoch and decide if we have trained the model enough. Otherwise, the over-training will make the model over-fit the profiling data, and it will not be able to classify the attack data correctly. In Section 4.2 we explain our optimization strategy to do the incorporation.

3 ASCAD Fixed Key Dataset

ASCAD fixed key (ASCAD F) was introduced by Prouff et al. in [2]. The target device is an Atmega8515 8-bit microcontroller. The target algorithm is a masked AES-128 [33], [34]. Conveniently, the leakage traces are pre-processed to include the relevant part of the cryptographic algorithm execution, specifically the third masked S-box in the first round. The dataset’s structure allocates leakage traces in two sets; (i) profiling traces ($P$) comprise 50,000 traces, and (ii) attack traces ($A$) comprise 10,000 traces.

4 Optimized Early Stopping Strategy

The first step in designing our early stopping strategy is to describe how we optimize the computation of the guessing entropy function. The rest of the section explains our proposed strategy and the use cases in practical applications for our early stopping strategy.

4.1 Guessing Entropy Optimization

Guessing entropy is expensive in computation time, representing a drawback for integrating it into the training process. We designed an optimized version of its algorithm to overcome the disadvantage created by the computation time. Starting from the algorithm suggested in [2], we reduced the computation time based on: (i) removing nested loops and (ii) applying per-block operation using linear algebra.

Let us analyze the original version of the guessing entropy algorithm by taking the fragment of the pseudo code at the Algorithm 1, where $t_i \subset P \subseteq P$ such as $|t| = \text{steps}, |P| = \text{max_traces} - \text{min_traces}$ and steps, max_traces, min_traces $\in \mathbb{Z}$. Putting this in words, range function creates a sub set that contains $\text{steps} \leq |t| \leq \text{max_traces} - \text{min_traces}$ groups with an amount of steps traces. Guessing entropy relies on an optimized brute force. It XORs all the possible key candidates $k \in K$ with the plain text of a trace. In the second loop function get_p t creates a $p$ set that contains the plain text of the traces in $t_i$ ($t_{i,\text{max}} - t_{i,\text{min}} \in \mathbb{Z}$). Each plain text $p_i$ is XORed with all the key candidate. After that the guessing entropy is computed.

**Algorithm 1. Fragment of the Original Guessing Entropy**

1: procedure ComputeOriginalGuessingEntropy
2:     
3:     for $t_i \leftarrow \text{range}(|\text{min_traces}|, |\text{max_traces}|, \text{steps})$ do
4:         
5:             for $p \leftarrow \text{get_p}(t_{i,\text{min}}, t_{i,\text{max}})$ do
6:                 
7:                     for $k \leftarrow \text{range}(2^{|p|})$ do
8:                         S-box($p_i \oplus k$)

Note that the number of iteration in this implementation corresponds to the following expression:

$$\left(\frac{|\text{max_traces} - |\text{min_traces}|}{\text{steps}}\right) \times (t_{i,\text{max}} - t_{i,\text{min}}) \times 256$$

(2)

As said guessing entropy is a optimized brute force-base algorithm. To compute it, we need the classifier to calculate the probabilities, add up those probabilities to all key candidates. This process is done incrementally. Line 3 in Algorithm 1 increments the traces by the steps factor until we reach the maximum number or traces. The only way to optimize this part of the algorithm is by implementing job routines based on threads or cores processors. Another implementation should imply hardware dedicated to matrix operations (like GPUs), but some consideration about memory management should be done. Still, we consider that those options could improve our proposal and we are considering implementing it for future works.

In this first implementation we were able to improve the performance by removing the last nested loop and applying a vector operation that computes the S-box function of all key candidate in one pass (see Algorithm 2). In particular, the original algorithm computes the bit-wise XOR operation per trace required to compare all bytes in the keyspace with the predicted trace label i.e., S-box{0, 255} ⊕ plaintext done byte per byte. In contrast, our algorithm defines a vector of all bytes ($\tilde{k} = [0, \ldots, K]$) and XORes the trace’s plain text as a scalar multiplication substituting the per-value by a per-block operation. Then, the vector feeds the algorithm that computes the key rank ($\text{rank}_{k^*}$), also using a per-block operation. The resulting throughput has a higher capability than the per-value approach. In Eq. (2) this correspond to remove the last factor of the expression.

Although we achieved a good performance using the same programming language, i.e., Python as in [2], we consider that it is possible to improve further performance by using a more efficient programming language.

4.2 Early Stopping Algorithm Overview

A well-trained deep learning model for SCA evaluation would result in a guessing entropy value that converges to zero as the number of traces increases. We represent this convergence as the limit expressed in Eq. (3), defined by a lower bound $v$, which represents the smallest number of traces where the desired guessing entropy value $w$ is met

---

2. Publicly available at https://github.com/ANSSI-FR/ASCAD

3. More details about the algorithm can be found in [2].
we determine that our early stopping framework should stop the training since the patience criterion was met. Although patience is a common parameter in most early stopping algorithms, the patience parameter is subject to the number of attacks performed in each epoch. Our experiments show that performing several attacks per epoch increases the guessing entropy reaches its maximum of convergence (zero in the best case scenario), and the convergence persists at this value for the available traces, indicating that the model is stable. If the model keeps its stability through several epochs, it is said that the model has reached an optimal state. In the following, we discuss how the area of hit depicted in Fig. 2 is the base of our early stopping algorithm.

**Algorithm 2.** Fragment of the Optimized Guessing Entropy

```
1: procedure COMPUTE_OPTIMIZED_GUESSING_ENTROPY
2:   ... 
3:   for ti ← range(min_traces, max_traces, steps) do
4:     ... 
5:     for p ← get_pred(ti, min, max) do
6:       ... 
7:       get_pred(S-box([0, ..., 255] ⊕ p)))
```

In a real-world evaluation, the value of \( v \) is not established regularly. Instead, the evaluator lets the model find the value. It also implies that the \( w \) value could not be zero. In such a case, the \( v \) value would indicate where the model starts stabilizing to that number. Nevertheless, it is commonly expected that \( w = 0 \) to estimate (properly) the number of leakage traces required to exploit the device’s cryptographic algorithm. To define the parameter \( w \) makes sense for those cases when the number of attack traces \( N_a \) is limited. For instance, let us take from ASCAD\(^{10}\) the maximum number of attack traces (10,000). Then, a classifier to evaluate those leakage traces might generate a guessing entropy not converging towards zero but some other positive value. Consequently, the early stopping strategy would never trigger the evaluator’s conditions.

On the other hand, some evaluations require the specification of a maximum number of traces. In those cases, parameter \( v \) is set to determine a threshold of traces the evaluator requires. Given these two modes of evaluation, we determine that our early stopping framework should support them by defining use cases with regard to the values of \( w, v, \) and \( N_a \). We will come to this later in the paper when we explain the soft and the greedy case.

Given parameters \( w, v, \) and \( N_a \), our framework determines when the classifier reaches an optimal state based on three aspects, the area of hit, persistence and patience. We define all of them as follows;

**Definition 1.** Area of hit \((w, v, N_a)\) such that \( v, w, \) and \( N_a \in \mathbb{Z} \). Given \( w \), the target value for the guessing entropy function, \( N_a \) maximum number of traces and optionally \( v \). A deep learning neural network, reaches the area of hit with parameters \((w, v, N_a)\) at epoch \( \epsilon \), when Eq. \((3)\) holds for a minimum number of traces \( v \) until \( N_a \). When \( v \) is not specified, then it is established by the deep learning neural network performance.

As Definition 1 suggests, \( w \) and \( N_a \) define the area of hit where the guessing entropy should remain stable to meet the convergence conditions. Hence, the parameter \( v \) gets the value from the number of traces the guessing entropy reaches \( w \). Through epochs, the classifier could change the value of \( v \). For an evaluation where \( v \) is not specified, a classifier that changes \( v \) is not a problem. Because the evaluator only wants to know if the classifier is able to exploit the leakage with \( w \) values of entropy, no matter the number of traces. However, the opposite occurs if we specify \( v \).

Before declaring a hit, the framework should determine the stability of the guessing entropy. Suppose that the guessing entropy reaches \( w \), then we set \( v \). But before going to the maximum number of traces \( N_a \), the guessing entropy diverges from \( w \) as it did in the example in Fig. 1 after the second windows of optimal state. In this situation, we cannot ensure that the classifier has reached an optimal state since it miss-classifies some labels. This leads us to the two remaining definitions;

**Definition 2.** Persistence: Denoted as \( \rho(P_{ge}) = \kappa \) defines the percentage of the number of traces the guessing entropy below \( w \) after reaching \( v \). Where \( P_{ge} \) is a vector whose elements are the values of the guessing entropy and its index the number of traces with \( v \) offset i.e. \( |P_{ge}| = N_a - v \). Thus \( \kappa \) is the percentage of traces within the area of hit.

According to Definition 2, for our framework to declare a hit, the classifier should keep the guessing entropy for a \( \kappa \) percentage of the traces. We designed our framework to be flexible in evaluating the persistence. It features two modes of persistence that we will explain in Section 4.2.2. The persistence monitors the stability in one epoch per maximum number of traces. However, our framework should monitor the stability through the epochs to avoid false positives. Stability in this context refers to the times (measured in epochs) the guessing entropy curve meet the persistence criterion.

**Definition 3.** Patience \((P_a)\) is a value that establishes the number of consecutive epochs for the guessing entropy to stay in the area of hit, where \( P_a \in \mathbb{Z} \).

In other words, if the model does hit the persistence \( P_a \) times, then the framework will stop the training since the stability criterion was met. Although patience is a common parameter in most early stopping algorithms, the patience of our early stopping strategy is subject to the number of attacks performed in each epoch. Our experiments show that performing several attacks per epoch increases the
difficulty to meet the patience criterion. Because the guessing entropy vector on each epoch has variance and so the average of those attacks tends to go outside the area of hit. We will come to this later.

4.2.1 The Soft and Greedy Use Cases

Notice that Fig. 2 has two guessing entropy curves for Model A and B, intending to illustrate two possible use cases of the persistence. The use case from guessing entropy Model A is when the evaluator sets the parameters $w$ and $N_a$. Wherever the guessing entropy Model A intersects with the $w$-line, we get the value of $v$ and the area of hit. This case is called soft case because $v$ takes different values through training epochs. In this scenario, any guessing entropy curve should be less than $w$ for a classifier to hit the persistence and patience conditions as $v$ is dynamically set using the intersection (GE, $w$).

Further, let us suppose we have to outperform the guessing entropy of Model A, meaning we should find a deep learning model whose guessing entropy converges within fewer traces. According to Fig. 2, this would correspond to the Model B guessing entropy, for instance. For scenarios like this, our proposed strategy allows the evaluator to set the parameter $v$ in advance. Contrary to the former scenario, when $v$ is previously set, its value does not dynamically change during training. Moreover, the area of hit merges with the area of requested convergence (see Fig. 2).

Therefore, We condition any neural network to have a guessing entropy that touches both areas; the area of hit and the area of requested convergence. Any neural network model that achieves this goal is considered better than any other model with a guessing entropy whose convergence happens after the $v$ value. There are several factors to consider this condition as challenging to meet. One of those factors—the most relevant for our case—is due to the fluctuations of the guessing entropy during the evaluation process. When the neural network wrongly classifies the correct key with a single trace, the guessing entropy rapidly diverges with large values, losing any chance to meet the persistence and patience conditions. We named this the greedy case because it represents a more challenging goal to achieve.

4.2.2 Persistence Modes

We feature the persistence in two modes; (i) full and (ii) binary persistence. Full persistence acts exactly how Definition 2 conceptualizes the persistence. To claim a hit, the guessing entropy curve should not go outside the area of hit at any number of traces (100% of the traces after $v$). In contrast, binary persistence allows defining a percentage of traces the guessing entropy curve should keep in the area of hit. For instance, a value of 0.95 in binary persistence mode means that 5% of traces are allowed outside the area of hit. Binary persistence helps in situations when the deep learning model miss-classify a wrong key as the correct key. As a result, it starts diverging [35]. Hence, to bypass this error, we included the binary persistence.

4.3 Finding Optimality With Grid Search

A good approach is to combine the greedy use case with an “optimal model searching algorithm” such as grid search. Other options include random search, or bayes search [1], [36]. Grid search is a well-known algorithm used in machine learning that might derive an optimal model from a set of hyper-parameters. As shown in [37], grid search has the limitation that there are no implemented metrics for seeking...
optimal models for side-channel analysis. Consequently, the
common practice is to let the grid search pass through the
whole set of hyper-parameters and then evaluate the perfor-
manence of all possible models leading to an inefficient prac-
tice. Combining our strategy and grid search, one can
efficiently stop the search, avoiding further computation
after finding the best model. We will discuss this in more
detail in the experiments section (Section 5).

5 EXPERIMENTAL RESULTS

5.1 Comparison With Key-Rank Strategy

Using the soft case, we compare our strategy with the early
stopping strategy defined in [19] that we call key-rank
strategy.

5.1.1 Soft Case

This experiment uses the model defined in [38] trained with
ASCAD\textsuperscript{F}. According to the study, the model’s guessing
entropy converges to zero after 191 traces using 50 epochs, a
batch size of 50, and using One cycle policy to control the
learning rate. We called this model Model\textsubscript{1}. TABLE 1 sum-
marizes the Model\textsubscript{1} architecture. For training it, we used
$|P| = 45,000$. Values of the learning rate, epoch, and batch
size as in the original work. Our early stopping strategy
requires attack traces, so we took $|A| = 10,000$. We set
parameters $N_a = 5,000$, $w = 0$ and persistence mode = full.
Notice that for the sake of completeness, we let the training
finish.

Fig. 4a depicts how the key-rank strategy suggests the
model has reached its optimal state, while Fig. 4b depicts
our strategy results. Notice that the key-rank strategy has
the lowest peak after 10 epochs, exactly where the plot in
Fig. 4b starts with the lowest values of guessing entropy.
However, two observations are essential at this point. First,
by letting the training continue, we observed that a few
episods ahead (around epoch 13), the model performed bet-
ter than in epoch 10. Moreover, it keeps this convergence up
to epoch 20, while the key-rank strategy immediately sug-
gests that the model starts moving from this optimal state to
a likely overfitting state. This latter suggestion leads us to
our second observation: according to how the evaluator sets
the patience of our proposed strategy, a single hit like in
key-rank strategy would not stop the training. Actually, by
using the key-rank strategy, there is no way to monitor the
stability of the convergence leading to uncertain outcomes.

The window of “time” that the key-rank strategy has is
too short, implying that we cannot ensure having reached

TABLE 1
Deep Learning Architecture Summary for Model\textsubscript{1}, Each Table
Row Represents a Layer of the Model

| Layer type     | Details                        |
|----------------|--------------------------------|
| Conv           | # kernels 4, kernel size 1, Selu|
| Pooling        | Average, kernel size 2         |
| Batch normalization |                      |
| Fully-connected | # units 10, Selu               |
| Fully-connected | # units 10, Selu               |
| Fully-connected | # units 256, Softmax           |


AUTHORzed licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
5.2 Training Time Comparison

We perform two experiments that show the difference in training time when using our optimized version of the guessing entropy and the original version from [2]. The first experiment resembles the case when an evaluator uses a guessing entropy algorithm to attack the leakage of a device. We use an already trained deep learning classifier and perform a battery of 10 attacks. Each attack uses \( N_a = 5000 \) traces, and we measure the computation time in steps of 100 traces. We tested it on a Laptop PC Intel(R) Core(TM) i7-9850H CPU2.60GHz 2.59 GHz, 16,0 GB RAM, Windows 10 Pro x64 OS. Fig. 7 shows the computation time of the implementation from [2] and our proposal. Our algorithm outperforms implementation from [2] while our proposal keeps the computation time below 1 second for more than 5000 traces, and the implementation in [2] quickly increases the computation time within fewer traces.

The second experiment showcases the scenario of a training process. We perform two training processes. Both processes use our early stopping framework, but one with our optimized version and the other with the original version of the guessing entropy algorithm. We train the deep learning model defined in the previous section. Since we want to measure the time of the training process; then, The framework does not stop any of the two processes. The number of attacks used per epoch is one, and the values of the framework’s parameters are the same as the ones used in the experiment of the previous section. As expected, our early stopping framework finds the right epochs when the model is optimal regardless of the version of the guessing entropy algorithm (see Fig. 8). However, the training time will be different because the first experiment of this section indicates that our optimized version outperforms the original. More precisely, the original version increases the training time by 58% (see TABLE 2). This difference in time will become more significant when training a deep learning model using more traces.

5.3 Comparison With Six-Sigma Method to Find an Optimal Model

5.3.1 Greedy Case

This experiment showcases the scenario where we have a previous model we are trying to outperform. To accomplish this, we will use the greedy case. Parameter \( v \) should be established to a value lower than the previous model has found. We use the results presented in [37] where the authors applied a customized version of the Six-sigma algorithm.
methodology [39], [40] over a standard grid search algorithm to find a model whose guessing entropy converges earlier than Model\textsubscript{v
1}. Their goal was similar to ours since we are trying to optimize the grid search by stopping it when we find a “better model”. Hence, we will compare their approach to optimizing a grid search algorithm by using six-sigma in contrast to our approach using our framework. We will not consider that their approach implies more human intervention than ours. We will focus entirely on the results. They showed that by adding two fully-connected layer (of 10 units) more, the better model (called Model\textsubscript{v
2} see TABLE 3) converges with fewer traces, i.e., around 150 instead of 191 traces (see Fig. 10). Nevertheless, our experiment shows that it is only necessary to train Model\textsubscript{v
1} using the correct number of epochs to outperform its previous results.

We need a space of hyper-parameters as we use grid search. In particular, we define a set of four hyper-parameters; each one takes two possible values, as shown in TABLE 4. Given that, each hyper-parameter should take two values each time. All the possible combinations give us 16 possible training processes.

Our aim is that the early stopping strategy stops the grid search when any training processes derive a model that meets the stop conditions. The soft case would also do in this scenario, but only if the goal is to come across a model whose guessing entropy converges at any number of traces. However, in some circumstances, like looking for a model that outperforms the previous ones, we must define a minimum of traces, i.e., we should set a value for \( v \) in advance. In such circumstances, a standard grid search algorithm will go over all 16 training processes, which is helpful if the evaluator is looking to test all of them. Otherwise, it represents a costly process in terms of time. In contrast, our grid search version stops searching after a single model achieves the required performance.

For our experiment, we fix \( v = 100 \) as the number of traces where the guessing entropy should already be less than \( w \) (touching the area of the requested convergence) and \textit{persistence mode} = \textit{binary} with a value of 0.95. Note that we repeat five times the grid search by letting our strategy perform different numbers of attacks to the average the guessing entropy curve (see Fig. 9). Indeed, an averaged guessing entropy encourages the stability criterion. However, the number of epochs increases according to the number of attacks because of the variance. Consequently, it could quickly overfit the model due to the additional epochs. This experiment shows that as long as the number of attacks per epoch performed by our framework guarantees the stability of the convergence, that number of attacks should be enough [41].

In this particular result, all five repetitions are from the first training process (first combination of hyper-parameters), and all of them outperform the guessing entropy reference from [38] with less than 40 traces in less than 50 epochs. Since the “best” model (Model\textsubscript{v
1} architecture) was in the first training process, we stopped the grid search after 1 training process out of 16 possible ones (best case scenario).

### TABLE 2

| Training Time Differences Between Our Optimized Guessing Entropy Algorithm and the Original Version on the Second Experiment |
|---------------------------------------------------------------|
|                | Original GE | Our optimized GE |
| Training time   | 11.55 min   | 5.02 min        |

### TABLE 3

| Deep Learning Architecture Summary for Model\textsubscript{v
2}, Each Table Row Represents a Layer of the Model |
|---------------------------------------------------------------|
| Layer type | Details |
|------------|---------|
| Conv       | # kernels 4, kernel size 1, SeLU |
| Pooling    | Average, kernel size 2 |
| Batch normalization | |
| Fully-connected | # units 10, SeLU |
| Fully-connected | # units 10, SeLU |
| Fully-connected | # units 10, SeLU |
| Fully-connected | # units 10, SeLU |
| Fully-connected | # units 256, Softmax |

### TABLE 4

| Variables/Hyper-Parameters and Values Define the Hyper-Parameters Space of the Grid Search |
|------------------------------------------------------------------------------------------|
| Variable/hyper-parameter | Values |
| Architecture               | \{Model\textsubscript{v
1}, Model\textsubscript{v
2}\} |
| Batch size                 | (50, 100) |
| Epochs                     | (50, 100) |
| Optimizer                  | \{RMSprop, Adam\} |
Results from [37] support this experiment’s conclusions. Notice that according to their results, the models trained using 25 epochs performed better than those trained using 50 (see Fig. 10). Furthermore, our results show that we also outperformed the result from [37] in both searching time and guessing entropy convergence. In this particular case, overfitting was the problem that limited the model’s performance in both works [37] and [38].

5.4 Additional Experiments Using Grid Search

Let us consider the following scenario: an evaluation requirement is to ensure that a device’s leakage cannot be exploited in less than a given number of traces. This type of evaluation could be requested after applying a countermeasure that blocks the device after a given number of consecutive transactions. Let us take the number of traces in the previous experiment as the threshold. Hence, we need a model to converge toward zero in less than 40 traces. In our case, we set the parameter \( v \) to 10 traces, meaning that the goal is to find a model that exploits the leakage using 10 traces only. We will use our grid search algorithm to look for the deep learning model.

In this case, we will dynamically create the architecture of the all deep learning model candidates using the VGG approach [42] which follows several rules that have been adapted for side-channel analysis [2], [32]. However, we modified some of those rules to meet some state-of-the-art criteria issued in [38], [43]:

1) The model comprises several convolutional blocks followed by a fully-connected layer and an output layer with the Softmax activation function.

2) Convolutional and fully-connected layers use the SeLU activation functions.

3) In a convolutional block, after a convolutional layer, an average pooling layer follows. Additionally, a batch normalization layer is attached to every odd-numbered convolutional block.

4) The filter size for a convolutional layer is fixed to size 3, except for the first convolutional block, whose size is 1.

5) The number of filters in a convolutional block \( j \) increases according to the rule:

\[
    n_{filter,i} = \min(2^i \cdot 512)
\]

for every layer \( i \geq 1 \), where \( n_{filter,1} = 4 \)

We define the hyper-parameter space as batch size={50, 70, 100}, epochs={70, 100} number of block={6, 7, 8}. Given those multi-value hyper-parameters, our hyper-parameter space will perform 18 iterations (3 deep learning models that will be trained 6 times each). The worst-case scenario is that our grid search algorithm finds that the last deep learning model is the one we are looking for. However, we ensure that if that is not the case, the training will stop at any moment when the grid search finds the model.

We conducted the whole experiment two times. The first time we used our framework and stopped the grid search. The second time we used the framework and stopped the grid search.
after finding the optimal model. The evaluation elapses 14.96 min. The second time we used our framework but not our grid search. Furthermore, we used the original guessing entropy algorithm. This time the evaluation takes around 6.13 hours. In both experiments, the model that met the conditions was the second model. In the second case, we had to wait for the 18 training processes end to evaluate the performance of each of the models. Fig. 11 depicts the results.

6 Conclusions and Future Works

This paper introduced an optimized early stopping strategy for deep learning models used in side-channel attacks. Our proposal defined patience and persistence as criteria for monitoring the guessing entropy from two different axes: (i) its stability through the training epochs and (ii) the required number of attack traces. Our proposal reliably recognizes when the deep learning model reaches its optimal state by keeping track of the guessing entropy from these two different perspectives. It prevents the model from overfitting, reducing the uncertainty of getting a faulty model.

Our proposal relies on the guessing entropy, and we present evidence that guessing entropy is a feasible side-channel analysis metric to evaluate the performance of an attack. However, the metric might exhibit false outcomes during a side-channel evaluation; on top of that, it might be considered that its time complexity is too heavy to be computed during training. Our early stopping strategy assists the evaluator in avoiding those issues. To overcome the computation overhead, we developed an optimized version of the guessing entropy algorithm. Concerning false outcomes, our proposed strategy can monitor the guessing entropy in two different modes of persistence, allowing the evaluator to design stopping conditions that best suit the case.

This work clears the way for further studies to improve or develop new early stopping strategies based not only on the guessing entropy, but also on new metrics such as the one addressed in [44] and [45]. In future works, we plan to investigate other optimization techniques to reduce the computation time even more. Finally, we consider that our early stopping strategy algorithm can be the base for a score function for more efficient hyper-parameter searching algorithms.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.
[2] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, “Deep learning for side-channel analysis and introduction to ASCAD database,” J. Cryptographic Eng., vol. 10, no. 2, pp. 163–188, 2020.
[3] N. Mukhtar, L. Batina, S. Picek, and Y. Kong, “Fake it till you make it: Data augmentation using generative adversarial networks for all the crypto you need on small devices,” in Proc. Cryptographers’ Track RSA Conf., 2022, pp. 297–321.
[4] G. Perin, L. Chmielewski, L. Batina, and S. Picek, “Keep it unsupervised: Horizontal attacks meet deep learning,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2021, no. 1, pp. 343–372, Dec. 2021.
[5] J. Rijsdijk, L. Wu, G. Perin, and S. Picek, “Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2021, no. 3, pp. 677–707, Jul. 2021.
[6] D. v. d. Valk, S. Picek, and S. Bhasin, “Kilroy was here: The first step towards explainability of neural networks in profiled side-channel analysis,” in Proc. Int. Workshop Constructive Side-Channel Anal. Secure Des., 2020, pp. 175–199.
[7] D. van der Valk, M. Kreck, S. Picek, and S. Bhasin, “Learning from a big brother-mimicking neural networks in profiled side-channel analysis,” in Proc. IEEE/ACM 57th Des. Automat. Conf., 2020, pp. 1–6.
[8] A. Dubey, A. Ahmad, M. A. Pasha, R. Cammarota, and A. Aysu, “ModuloNET: Neural networks meet modular arithmetic for efficient hardware masking,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2022, no. 1, pp. 506–556, Nov. 2022.
[9] B. Hettwer, S. Gehrer, and T. Güneysu, “Deep neural network attribution methods for leakage analysis and symmetric key recovery,” in Proc. Int. Conf. Sel. Areas Cryptography, 2019, pp. 645–666.
[10] Y. Zhou and F.-X. Standaert, “Deep learning mitigates but does not annihilate the need of aligned traces and a generalized reset model for side-channel attacks,” J. Cryptographic Eng., vol. 10, no. 1, pp. 85–95, 2020.
[11] R. Roelofs et al., “A meta-analysis of overfitting in machine learning,” Proc. Adv. Neural Inf. Process. Syst., vol. 32, pp. 9757–9765, 2019.
[12] X. Ying, “An overview of overfitting and its solutions,” in Proc. J. Phys.: Conf. Ser., 2019, Art. no. 022022.
[13] S. Yeom, I. Giacomelli, M. Fredriksson, and S. Jha, “Privacy risk in machine learning: Analyzing the connection to overfitting,” in Proc. IEEE 31st Comput. Secur. Found. Symp., 2018, pp. 268–282.
[14] B. Hettwer, S. Gehrer, and T. G"uneyusu, “Profiled power analysis attribution methods using convolutional networks with development knowledge,” in Proc. Int. Conf. Sel. Areas Cryptography, 2018, pp. 479–498.
[15] F.-X. Standaert, F. Koeune, and W. Schindler, “How to compare profiled side-channel attacks?,” in Proc. Int. Conf. Appl. Cryptography Netw. Secur., 2009, pp. 485–498.
[16] M. O. Choudary and M. G. Kuhn, “Efficient stochastic methods: Profiled attacks beyond 8 bits,” in Proc. Int. Conf. Smart Card Res. Adv. Appl., 2014, pp. 98–110.
[17] S. Pick, A. Heusen, A. Jovic, S. Bhasin, and F. Regazzoni, “The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2019, pp. 209–237, 2018.
[18] G. Perin, B. Ege, and L. Chmielewski, “Neural network model assessment for side-channel evaluations,” Cryptology ePrint Archive, p. 722, 2019. [Online]. Available: https://eprint.iacr.org/2019/722.
[19] D. Robissout, G. Zaid, B. Colombier, L. Bossuet, and A. Habrard, “Online performance evaluation of deep learning networks for profiled side-channel analysis,” in Proc. Int. Workshop Constructive Side-Channel Anal. Secure Des., 2020, pp. 200–218.
[20] F.-X. Standaert, T. G. Malkin, and M. Yung, “A unified framework for the analysis of side-channel key recovery attacks,” in Advances in Cryptology – EUROCRYPT 2009, A. Joux Ed., Berlin, Germany: Springer, 2009, pp. 485–498.
[21] G. Perin, I. Buhan, and S. Picek, “Learning when to stop: A mutual information approach to prevent overfitting in profiled side-channel analysis,” in Proc. Int. Workshop Constructive Side-Channel Anal. Secure Des., 2021, pp. 53–81.
[22] R. Shwartz-Ziv and N. Tishby, “Opening the black box of deep neural networks via information,” 2017, arXiv:1703.00810.
[23] T. Yu and H. Zhu, “Hyper-parameter optimization: A review of algorithms and applications,” 2020, arXiv:2003.05689.
[24] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” Proc. Adv. Neural Inf. Process. Syst., vol. 24, pp. 2546–2554, 2011.
[25] L. Wu, G. Perin, and S. Picek, “I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis,” Cryptology ePrint Archive, p. 1293, 2020. [Online]. Available: https://eprint.iacr.org/2020/1293.
[26] P. Liashchynskyi and P. Liashchynskyi, “Grid search, random search, genetic algorithm: A big comparison for NAS,” 2019, arXiv:1912.06059.
[27] M. Azouaoui et al., “A systematic appraisal of side-channel evaluation strategies,” in Proc. 6th Int. Conf. Secur. Standardisation Res., 2012, pp. 106–116.
[28] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, “A testing methodology for side channel resistance validation,” NIST Non-Invasive Attack Testing Workshop, vol. 7, pp. 115–136, 2011.
[29] P. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” in Advances in Cryptology — CRYPTO ’99, M. Wiener Ed., Berlin, Germany: Springer, 1999, pp. 388–397.
[30] H. Maghrebi, T. Portigliatti, and E. Prouff, “Breaking cryptographic implementations using deep learning techniques,” in Proc. Int. Conf. Secur. Privacy Appl. Cryptography Eng., 2016, pp. 3–26.

[31] L. Masure, C. Dumas, and E. Prouff, “A comprehensive study of deep learning for side-channel analysis,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2020, pp. 348–375, 2020.

[32] J. Kim, S. Pikec, A. Heuser, S. Bhaskar, and A. Hanjalic, “Make some noise unearthing the power of convolutional neural networks for profiled side-channel analysis,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2019, no. 3, pp. 148–179, May 2019.

[33] J. Daemen and V. Rijmen, The Design of Rijndael. Berlin, Germany: Springer, 2002.

[34] J. Blomer, J. Guajardo, and V. Krummel, “Provably Secure Masking of AES,” in Selected Areas in Cryptography. Berlin, Germany: Springer, 2004, pp. 69–83.

[35] L. Wu et al., “On the attack evaluation and the generalization ability in profiling side-channel analysis,” Cryptology ePrint Archive, p. 899, 2020. [Online]. Available: https://eprint.iacr.org/2020/899

[36] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” J. Mach. Learn. Res., vol. 13, pp. 281–305, Feb. 2012.

[37] S. Paguada, U. Rioja, and I. Armendariz, “Controlling the deep learning-based side-channel analysis: A way to leverage from heuristics,” in Proc. Int. Conf. Appl. Cryptography Netw. Secur., 2020, pp. 106–125.

[38] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, “Methodology for efficient CNN architectures in profiling attacks,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2020, pp. 1–36, 2020.

[39] J. Rioja, S. Paguada, L. Batina, and I. Armendariz, “The uncertainty of side-channel analysis: A way to leverage from heuristics,” J. Emerg. Technol. Comput. Syst., vol. 17, no. 3, pp. 1–27, May 2021.

[40] C.-S. Cheng, S.-C. Lee, P.-W. Chen, and K.-K. Huang, “The application of design for six sigma on high level smart phone development,” J. Qual., vol. 19, pp. 117–136, 2012.

[41] L. Wu, G. Perin, and S. Picek, “On the evaluation of deep learning-based side-channel analysis,” Cryptology ePrint Archive, Report 2021/952, 2021, https://ia.cr/2021/952

[42] K. Simonyan and Y. Bengio, “Very deep convolutional networks for large-scale image recognition,” 2014, arXiv:1409.1556.

[43] S. Paguada and I. Armendariz, “The forgotten hyperparameter: Introducing dilated convolution for boosting cnn-based side-channel attacks,” in Proc. Int. Conf. Appl. Cryptography Netw. Secur., 2020, pp. 217–236.

[44] J. Zhang, M. Zheng, J. Nan, H. Hu, and N. Yu, “A novel evaluation metric for deep learning-based side channel analysis and its extended application to imbalanced data,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2020, no. 3, pp. 73–96, Jun. 2020.

[45] G. Zaid, L. Bossuet, F. Dassance, A. Habrard, and A. Venelli, “Ranking loss: Maximizing the success rate in deep learning side-channel analysis,” IACR Trans. Cryptographic Hardware Embedded Syst., vol. 2021, no. 1, pp. 25–35, Dec. 2021.

Lejla Batina (Senior Member, IEEE) received the PhD degree from KU Leuven, Belgium, in 2005. She is a professor in embedded systems security with the Radboud University in Nijmegen, the Netherlands. Her research group consists of 12 researchers and 8 PhD students who have graduated under her supervision. She has also studied with the Eindhoven University of Technology, (Professional Doctorate in Engineering in 2001) and worked as a cryptographer for SafeNet B.V. (2001-2003). She is an editorial board member of top journals in security, IEEE Transactions on Information Forensics and Security and ACM Transactions on Embedded Computing Systems.

Ileana Buhã received the PhD degree from Twente University, The Netherlands, in 2008. She is currently an assistant professor of cryptographic engineering in the Digital Security group. Her research interest focuses on developing tools to help designers of cryptographic algorithms develop secure implementations. She spent more than 10 years in the security evaluation industry. She serves on several program committees of conferences that specialized on hardware security (CHES, COSADE, CARDIS, FDT, DATE, SPACE, etc.). She was also appointed as the general chair for CHES 2018 and program co-chair for CARDIS 2022.

Igor Armendariz received the PhD degree from the University of the Basque Country, in 1996, where he also was a professor in computer science (1996–2000). He is a researcher with Ikerlan Technological Research Center within the Cybersecurity in Embedded Systems team. He had a scholarship Award with CEIT (1991–1994), working as a researcher in the same center later (1995–1996). He began working in the Communication Department of Ikerlan (2000–2015), and currently, he is part of the Industrial Cybersecurity department. Cybersecurity specialist acc. to IEC-62443-4-1, IEC-62443-4-2 (TUV Rheinland, Components) #230/19.

Servio Paguada received the BEng degree in system engineering from Universidad Nacional Autonoma de Honduras, in 2011, in the same year, he also received the BSc degree in mathematics with informatic applications from the same university, the master’s degree in information technology management from Universidad Tecnologica Centroamericana, in 2014, and the MSc degree in embedded systems from Mondragon University, Basque Country, Spain, in 2016. He is currently working toward the PhD degree with the Radboud University in Nijmegen, the Netherlands. He is also part of the PhD students group with IKERLAN Technology Research Centre, Arrasate Mondragon, Gipuzkoa, Spain. His current research interest is optimizing profiled side-channel analysis applied to embedded systems.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.