POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Towards Efficient Training in Deep Learning Side-Channel Attacks

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Luca Castellazzi**

Student ID: 10527434
Advisor: Prof. Luca Oddone Breveglieri
Co-advisors: Prof. Alessandro Barenghi, Prof. Gerardo Pelosi
Academic Year: 2021-2022

Abstract

Side-Channel Attacks (SCAs) represent a serious threat to data security, offering a non-invasive way of retrieving secrets processed by a microcontroller during the execution of cryptographic algorithms. Recent research enhanced SCAs exploiting Deep learning (DL) networks to retrieve the secrets. The usage of multiple devices for training showed how DL-based SCAs can be effective even in real-world scenarios against an unprotected implementation of the Advanced Encryption Standard (AES), being able to successfully attack a device that is different from the ones used for training. In addition to multiple devices, this work considered several training-phases with multiple encryption keys and different amount of data, showing how these three variables affect the performance of DL-based SCAs in real-world scenarios. Device-Key Trade-off Analysis (DKTA) has been introduced to study the trade-off between number of devices and number of keys, while Device-Trace Trade-off Analysis (DTTA) to investigate the trade-off between number of devices and number of traces. Targeting an unprotected implementation of the AES, DKTA highlighted that the number of keys does not influence the attack, while DTTA showed how important it is to consider as many train-data as possible. Both analyses confirmed that the attack performance increases if multiple train-devices are considered. On the other hand, this is not true when attacking a masked implementation of AES, since the addition of a device, on average, makes the attack unfeasible. In this scenario, DKTA highlighted the importance of considering multiple keys, while DTTA showed that the enlargement of the training dataset can highly boost the performance of the attack.

Keywords: Cryptography, Cyberattack, Deep Learning, Side-Channel Attacks

Abstract in Lingua Italiana

Gli attacchi di tipo Side-Channel (SCA) rappresentano una seria minaccia per la sicurezza dei dati, in quanto permettono di recuperare, in maniera non invasiva, un segreto elaborato da un microcontrollore durante l'esecuzione di un algoritmo di cifratura. Recentemente, l'efficacia di questi attacchi è stata aumentata sfruttando tecniche di Deep Learning, come le reti neurali, per estrarre il segreto. L'utilizzo di più dispositivi durante la fase di training permette alle reti di essere efficienti anche in scenari reali, riuscendo a recuperare il segreto di AES, l'algoritmo standard per la cifratura, attaccando un dispositivo diverso da quelli utilizzati durante il training. Questo studio ha dimostrato, considerando molteplici attacchi e fasi di training, che la performance di un attacco SCA realistico con reti neurali non è influenzata solo dal numero di dispositivi, ma anche dalla presenza di più chiavi di cifratura e dalla quantità di dati utilizzati durante il training. L'introduzione di un'analisi del trade-off tra numero di dispositivi e numero di chiavi (Device-Key Trade-off Analysis, DKTA) e tra numero di device e quantità di dati utilizzati durante il training (Device-Trace Trade-off Analysis, DTTA), ha permesso di studiare il ruolo che queste variabili hanno durante un attacco. Attaccando un'implementazione non protetta di AES, tramite DKTA è stato possibile evidenziare come il numero di chiavi non influenzi l'attacco, mentre con DTTA è stato possibile dimostrare quanto sia importante considerare più dati possibile durante la fase di training. Entrambe le analisi hanno confermato che l'utilizzo di più di un dispositivo durante la fase di training permette di aumentare l'efficacia dell'attacco. Quando invece l'implementazione di AES attaccata prevede una contromisura di tipo masking, l'aggiunta di un dispositivo non permette di recuperare la chiave di cifratura. In questo caso, DKTA ha evidenziato quanto sia importante considerare più di una chiave, mentre DTTA ha mostrato come l'utilizzo di una grande quantità di dati durante il training permetta di avere alte prestazioni nella fase di attacco.

Parole Chiave: Crittografia, Attacco Informatico, Deep Learning, Attacco Side-Channel

Contents

Introduction

Today's digital world highly relies on data and their security is one of the most crucial aspects. Encryption is fundamental to ensure data security, as it hides the information contained in the data, protecting them from unauthorized access.
Everyday-life presents multiple examples of encryption applications. For instance, credit card information is encrypted during every transaction, text messages are encrypted when sent through instant-messaging applications, and Solid State Drives (SSDs), which are used to store data in the vast majority of computers, mount a microcontroller that performs encryption in order to fulfill device and system-level security requirements.

Even if encryption algorithms are proven to be mathematically secure, poor implementations can make them vulnerable to a specific type of attack.

Side-Channel Attacks (SCAs) [35, 36] allow to retrieve a secret encryption key from a device in a non-invasive way. Statistical methods are exploited to extract the correlation (leakage) between device-related physical variables (e.g., Power Consumption, Electro-Magnetic emissions) and processed sensitive information (e.g., intermediate values of the target algorithm), making it possible to retrieve the key.

In Profiled-SCAs [16], the attacker owns a sample of the target-device. In a preliminary phase, called *Profiling Phase*, the attacker exploits the sample-device to generate a model of the correlation between a physical variable and the secret key. Next, in the *Attack Phase*, the attacker applies the model to the actual target-device and retrieves the secret key.

One of the emerging trends in SCA field is the extension of Profiled-SCAs through the usage of Deep Learning (DL) techniques [11, 39]. During the Profiling Phase, Artificial Neural Networks (ANNs) are trained to recognize the leakage patterns from the sample-device and later, in the Attack Phase, they are exploited to retrieve the secret key from the target-device. This kind of attacks is known as Deep Learning based Side Channel Attack (DL-based SCA).

The absence of assumptions about the distribution of the noise in the measurements

and the ability of correctly retrieving the secret with few attack samples are the main advantages of DL-based SCAs over traditional Profiled-SCAs. In addition, the ability of ANNs to perform automatic feature extraction from data allows to completely automate the Profiling Phase.
On the other hand, the usage of ANNs in real-world scenarios makes this approach affected by the so-called *Portability Problem*: if the network focuses only on recognizing noise and details from the sample-device, rather than learning its general behavior, then it would provide wrong predictions during the Attack Phase against a different device. Therefore, the goal is to build a network that is general enough to successfully attack a device which is different from the one used during training.

The Multiple Device Model (MDM) [7] is the state-of-the-art solution to the Portability Problem. It aims at adding diversity to the training dataset considering at least 2 sample-devices during the Profiling Phase. This makes the network more general and able to attack a different, unseen device. In [7], MDM is validated only against 8-bit microcontrollers while targeting an unprotected implementation of the 128-bit version of the Advanced Encryption Algorithm (AES) [21].

Differently from [7], the novel methodology introduced by this work considers DL-based SCAs dependent on three main variables, being the number of sample-devices, the number of keys used during the Profiling Phase and the size of the training set. The idea is that all these variables, and not only the number of devices, influence the attack performance. Studying the impact that the three most important variables have on the attack, this work is able to provide an efficient training configuration to mount effective realistic DL-based SCAs. In particular, the performed experiments consider at most 2 sample-devices, up to 10 keys and a number of measurements that goes from 50,000 to 500,000.
Additionally, for the first time, this work applies the MDM over data coming from 32-bit microcontrollers and against a software masked implementation of AES-128.

More precisely, focusing on a complete characterization of the Profiling Phase of realistic DL-based SCAs, two techniques are introduced:

- *Device-Key Trade-off Analysis (DKTA)*: studies the trade-off between number of devices and number of keys;
- *Device-Trace Trade-off Analysis (DTTA)*: studies the trade-off between number of devices and size of the training set.

Considering multiple attacks with different train-configurations, DKTA and DTTA highlight possible scenarios where the number of keys or the size of the training set can enhance the performance of the attack in presence of limited number of devices. This can be useful in practice, since keys can be automatically generated with a script for free and the measurements can be collected even if a single device is available, while the consideration of multiple sample-devices implies the purchase of additional hardware.

This volume is structured as follows.
Chapter 1 provides a detailed background about Deep Learning, in particular the Multi-Layer Perceptron network, Side-Channel Attacks and masking countermeasures. Chapter 2 focuses on the methodological approach adopted in this work, providing a detailed description of the pipeline followed to mount DL-based SCAs and introducing DKTA and DTTA. Chapter 3 offers a detailed overview of the experimental setup of this work, while the results of the two trade-off analyses are shown and discussed in Chapter 4. In the end, Chapter 5 summarizes the relevant achieved results and suggests possible future developments.

1 | Background and State-of-the-Art

This chapter presents the theoretical and mathematical background needed to mount Deep Learning based Side-Channel Attacks.
Starting from a detailed description of the Multi-Layer Perceptron network, this section proceeds with introducing the Advanced Encryption Standard cipher, Side-Channel Attacks and their countermeasures, and explaining how Deep Learning based Side-Channel Attacks work.

1.1. Deep Learning and Multi-Layer Perceptron

Machine Learning (ML) is a sub-field of Artificial Intelligence (AI) that focuses on automatically extracting relevant information from data, and on using it to make informed decision on new, unseen data. Indeed, ML programs are said to *learn* from data, as highlighted by Mitchell [46]:

> "*A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*"

This *learning* ability of ML programs (also called *models*) is exploited to solve problems that would be too complex to address with traditional programming. For example, the automation of all those actions that humans do without being able to explain how (e.g., tell if a picture shows a cat or a dog) can be performed with ML algorithms, due to the lack of a fixed set of operations that, given an input, produces the desired output.

There are multiple *learning approaches* in Machine Learning, and they depend on the type of problem to solve:

- *Supervised Learning*, used to estimate an unknown function that maps known inputs to known outputs;

- *Unsupervised Learning*, used to learn a more efficient representation of a set of unknown inputs;
- *Reinforcement Learning*, where agents interact with the environment and try to reach a specified goal maximizing the reward that the environment provides them as feedback to each performed action.

Deep Learning (DL) is a sub-field of Machine Learning which focuses more on emulating how humans learn: DL algorithms (known as Artificial Neural Networks (ANNs), or simply *networks*) do not need the input to be refined by removing noise and highlighting only the most relevant features (as done in ML), because they are able to autonomously extrapolate important information directly from raw data.
Indeed, ANNs are proven to be *universal approximators* [30].

Neural Networks can be divided into two main categories:

- *Feed Forward Neural Networks (FFNNs)*: the structure of the network is linear, from an input layer to an output layer, forcing information to be processed only in one direction;
- *Recurrent Neural Networks (RNNs)*: the structure of the network presents loops, which allow information to be stored and used in later iterations.

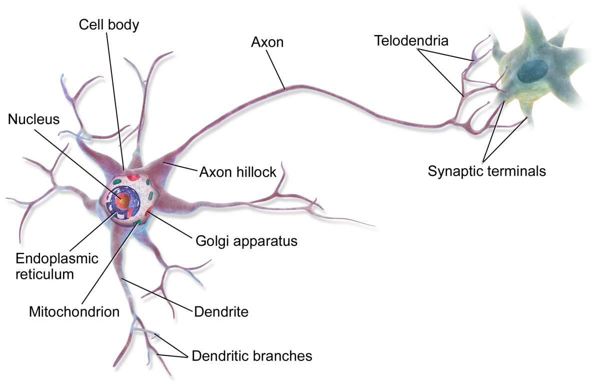
Multi-Layer Perceptron (MLP) [19, 43, 55], Convolutional Neural Networks (CNNs) [25, 37, 59] and AutoEncoders (AEs) [41] are typical examples of FFNNs, while Long-Short Term Memories (LSTMs) [29] are popular RNNs.

Lately, DL has been applied to multiple domains. Among others, Computer Vision and Natural Language Processing are two of the most popular.

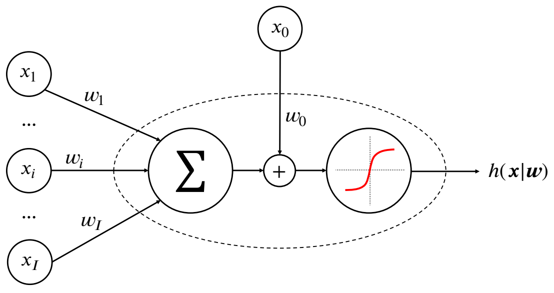
1.1.1. Perceptron

Artificial Neural Networks (ANNs) were introduced to emulate biological neural networks, which constitute the human brain.
As in biological brains, the basic data-processing unit of an ANN is called *neuron*: artificial neurons, known as *Perceptrons* were originally introduced by McCulloch and Pitts [43] and later implemented by Rosenblatt [55].

Figure 1.1 compares a biological neuron and a Perceptron.
A biological neuron, depicted in Figure 1.1a, works by collecting electrical signals and propagating them to the next neuron. Specifically, signals are collected through the *dendrites*, accumulated inside the *nucleus* and propagated through the *axon* if the total



(a) Biological Neuron



(b) Perceptron

Figure 1.1: Comparison between biological neuron and artificial neuron. Subfigure (a), on the left, shows a biological neuron [9], highlighting its principal components, such as *dendrites*, *nucleus*, *axon* and *synapses*. Subfigure (b), on the right, shows the Perceptron, an aritficial neuron.

accumulated signal exceeds a fixed threshold. The resulting signal is then transmitted to another neuron through the *synapses*
Figure 1.1b shows how this process is emulated artificially: a Perceptron receives a vector $\mathbf{x} = [x_1, x_2, ..., x_I]$ of I input signals, accumulates them through a linear combination considering the vector $\mathbf{w} = [w_1, w_2, ..., w_I]$ of *weights*, adds a the fixed *bias* term x_0 with weight w_0, evaluates the result of the accumulation with an *activation function* $h(\cdot)$ and propagates the final outcome to the next neuron.
The following equation summarizes the functioning of a Perceptron:

$$h(\mathbf{x}|\mathbf{w}, x_0, w_0) = h(\sum_{i=1}^{I} w_i x_i + w_0 x_0) = h(\sum_{i=0}^{I} w_i x_i) = h(\mathbf{x}|\mathbf{w}) \tag{1.1}$$

The *bias* value x_0 is usually set to $+1$ and considered as implicit element of vector $\mathbf{x}$. At the same time, w_0 is considered part of vector $\mathbf{w}$.

A Perceptron can be used as simple binary classifiers (able to say if an input belongs to class $+1$ or to class -1) if the following activation function is used:

$$h(x) = sign(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{otherwise} \end{cases} \tag{1.2}$$

Examples of functions that can be approximated by a Perceptron are *logical OR* and *logical AND*.

Equation 1.1 shows that the output of a Perceptron (its prediction) is dependent only on the input $\mathbf{x}$ and on the weights $\mathbf{w}$. Since the input is an external signal and can not be controlled by the Perceptron, it can be said that the weights are the only variable responsible for enabling the *learning ability* of an artificial neuron.

For this reason, the *learning process* of a generic Deep Learning network is identified as the tuning of its weight. This procedure is knonw as network *training*.

In the particular case of a Perceptron used as classifier, the learning process penalizes the weights only in case of misclassification. Referring to Figure 1.1b, it follows the *Hebbian Rule* [14]:

$$w_i^{k+1} = w_i^k + \eta \cdot x_i^k \cdot t^k \tag{1.3}$$

In Equation 1.3, x_i^k identifies the i^{th} input at time-step k, w_i^k refers to the weight associated to it, η is a constant known as *learning rate* and t^k is the desired output at time-step k.

Note that:

- Weights need to be initialized before training, even randomly;
- η sets the *speed* of the learning process, setting the amount of penalization per iteration;
- The presence of t^k during the training suggests that Perceptrons are trained in a *supervised* way.

1.1.2. Multi-Layer Perceptron

The *Hebbian Rule* used to train Perceptrons as classifiers presents a limitation: it always converge only if the input data-points belong to a *linearly separable* dataset. This means that a single Perceptron will always be able to classify only datasets whith two classes, where the decision-boundary is a *linear function*.
As practical example, while both *logic OR* and *logic AND* can be approximated by a single artificial neuron, *logic XOR* can not.

To solve this problem, multiple *layers* of Perceptrons with *non-linear activation functions* have been stacked and connected together [19], building a fully-connected ANN denominated Multi-Layer Perceptron (MLP).

The general structure of a MLP is shown in Figure 1.2. A MLP is composed by a single *input layer*, multiple *hidden layers* and a single *output layer*, where a *layer* is defined as a set of Perceptrons that share the same activation function:

- **Input Layer**: exactly I neurons, where I is the number of inputs, modeled as vector $\mathbf{x} = [x_1, x_2, ..., x_I]$;

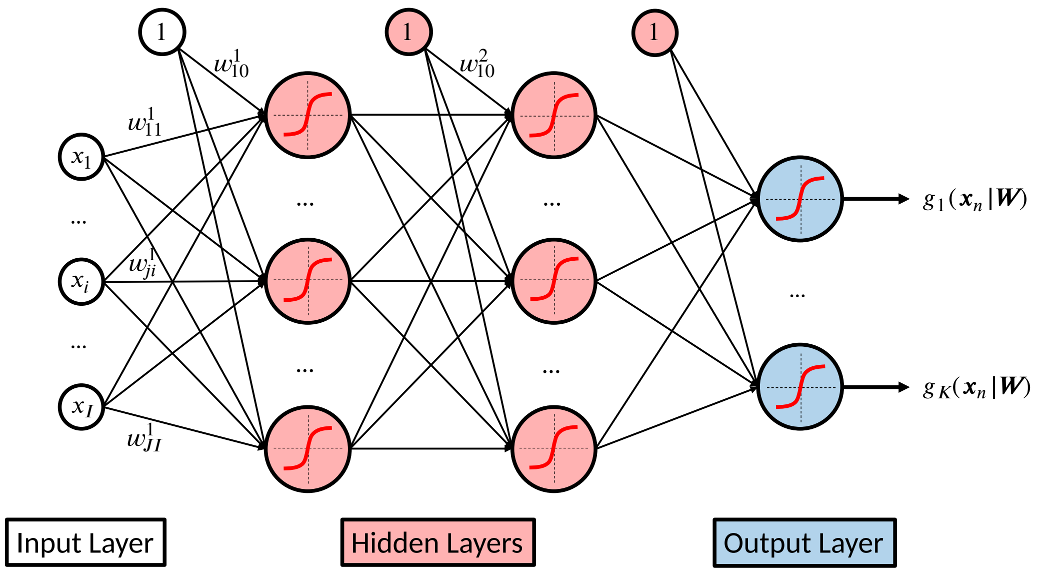
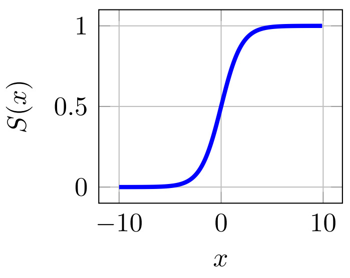


Figure 1.2: Structure of a Multi-Layer Perceptron. Input layer, hidden layers and output layers are highlighted in different colors. Inside each neuron, a generic activation function is represented.

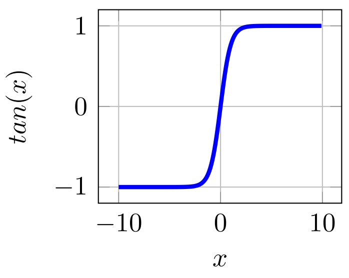
- **Hidden Layers**: variable both in number and size. Their amount and their size determine the network complexity;
- **Output Layer**: exactly K neurons, where K is the number of outputs.

The connections between neurons (which model the *axons* of biological neurons) are *weighted*. If neuron i of layer $l-1$ is connected with neuron j of layer l, then the connection $i \rightarrow j$ is weighted with w^l_{ji}. Therefore, matrix $\mathbf{W}^l = \{w^l_{ji}\}$ contains all the weights present between layer $l-1$ and layer l.
The *bias* related to neuron j of layer l is interpreted as additional element of layer $l-1$, as depicted in Figure 1.2. Conventionally, it is located at the first position of layer $l-1$, and considered as its 0^{th} neuron. As a consequence, its weight is denoted by w^l_{j0} and considered part of matrix $\mathbf{W}^l$.
As in the case of the simple Perceptron, all *biases* are set to $+1$.

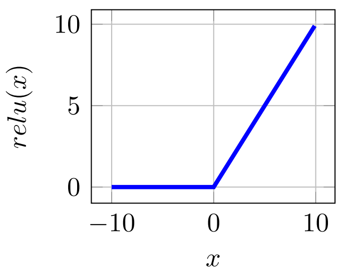
Following the same convention, the *activation function* of layer l is denoted with function $\mathbf{h}^l = h^l_j(\mathbf{h}^{l-1}, \mathbf{W}^l)$, where j indicates the j^{th} neuron of layer l.
Usually, the activation function of the output layer is denoted with $\mathbf{g} = \{g_k(\mathbf{x}|\mathbf{W})\}$, where k indicates the k^{th} output neuron and $\mathbf{W}$ represents the dependence of this function on all the weights of the network, which are combined due to the hidden activation functions.



(a) Sigmoid Function



(b) Tanh Function



(c) ReLu Function

Figure 1.3: Most popular Non-Linear Activation Functions. Subfigure (a), on the left, shows the *sigmoid* activation function, an *s-shaped* function in the interval $[0;1]$. Subfigure (b), in the center, depicts the *tanh* activation function, an *s-shaped* function in the interval $[-1;1]$. Subfigure (c), on the right, shows the *ReLu* activation function. All the functions are presented in the interval $[-10;10]$ for clarity.

Figure 1.3 shows the most popular non-linear activation functions:

- **Sigmoid**: $S(x) = \frac{1}{1+e^{-x}}$
- **Hyperbolic Tangent (*tanh*)**: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **Rectified Linear Unit (ReLU)**: $relu(x) = max(0, x)$

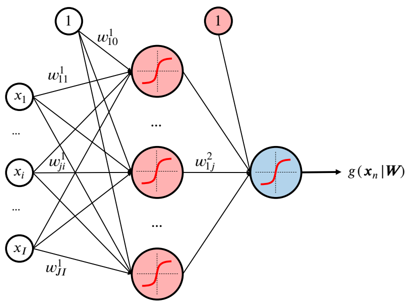
All these functions are suitable for an MLP, as they are *not* linear. Indeed, they make it possible to exceed the limit imposed by simple Perceptrons, thus allowing the recognition of more complex patterns.

MLP Training

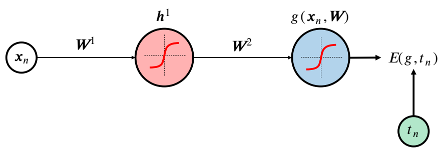
Consider the simple MLP depicted in Figure 1.4a: it features an input layer, a single hidden layer and an output layer with a single neuron.
Its training process can be modeled as follows: given a labeled training set of N data-points (MLPs are *supervised* algorithms) $D = \{(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), ..., (\mathbf{x}_N, t_N)\}$, where $\mathbf{x}_n$ is a data-point and t_n its desired output, the goal is to find MLP weights such that $g(\mathbf{x}_n|\mathbf{W}) \approx t_n$ for $n \in \{1, 2, ..., N\}$.
To solve this problem, the *error* between $g(\mathbf{x}_n|\mathbf{W})$ and t_n must be minimized with respect to $\mathbf{W}$.

Usually, to address this task, the gradient of the function to minimize is computed and then set to zero. Due to the lack of practical closed-form solutions to this problem, the iterative technique known as *Gradient Descent (GD)* [13] is used.
The k^{th} step of GD applied to function $f(\mathbf{x})$ is defined as follows:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha \nabla f(\mathbf{x}^k) \tag{1.4}$$



(a) Simple MLP



(b) Compressed Representation of a simple MLP

Figure 1.4: Structure of a simple MLP with only one hidden layer and only one output-neuron. Subfigure (a), on the left, depicts the complete representation of a simple MLP, showing every single connection between neurons. Subfigure (b), on the right, present a compressed representation of the simple MLP of Subfigure (a), showing the weights in matrix form. In addition, Subfigure (b) shows when the desired output relative to the input is used to compute the error function.

The intuition is given by Figure 1.5: the gradient of a function gives the direction of the steepest ascent of the function, so, to find the minimum, one should move in the direction of the steepest descent, opposite to the gradient. Constant α, called *learning rate*, sets the step of this movement.
Even if this process allows to start from random values and proceed finding the optimal ones, sometimes it can converge to a *local minimum*, usually due to a too high or too low value of the learning rate. To solve this problem, several algorithm, called *optimizers*, have been proposed, with the goal of reducing, dynamically, the value of the learning rate during training. Among these algorithms, Adam [34] and RMSprop [27] are two of the most popular and effective.

Let $E(g, t_n)$ be the error function between MLP output $g(\mathbf{x}_n|\mathbf{W})$ and true target t_n related to input $\mathbf{x}_n$.
In practice, it is too complex to compute its gradient directly. To solve this problem, the so-called *Backpropagation* algorithm has been introduced [56]: the gradient of the error function of a MLP is computed *backwards*, starting from its output layer and proceeding to the input one, exploiting the *Chain Rule of derivatives*.
The Chain Rule states that if function y depends on z and z depends on x, then the derivative of y with respect to x is given by:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z} \cdot \frac{\mathrm{d}z}{\mathrm{d}x} \tag{1.5}$$

The Chain Rule is suitable for the MLP case because the activation function of each layer depends on the output of the previous layer and the weights.

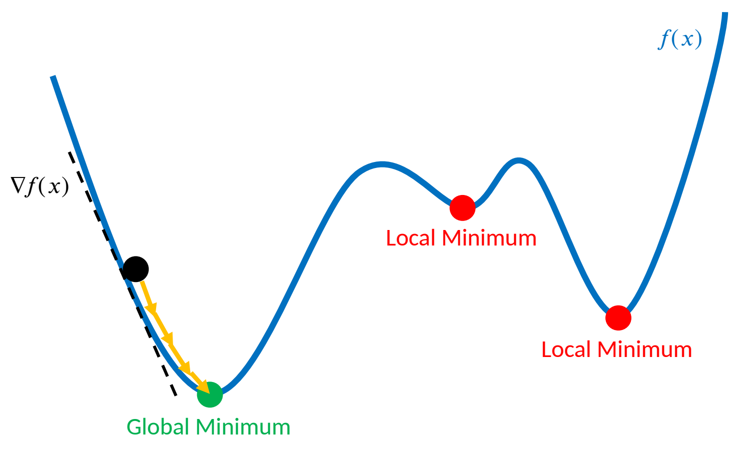


Figure 1.5: Graphical representation of the Gradient Descent technique. Starting from a random point of a curve (black point in the figure), it is possible to reach its minimum (green point) through iterative update of its variables. It may happen that the result of the minimization is a local minimum of the function (red points).

Remembering that each neuron accumulates the inputs through a linear combination and then applies the activation function to it, as described by Equation 1.1, the Backpropagation algorithm applied to the simple MLP depicted in Figure 1.4a and schematized in Figure 1.4b computes the gradients of the error function as follows:

$$\frac{\partial E(g,t_n)}{\partial \mathbf{W}^2} = \frac{\partial g(\mathbf{x}_n|\mathbf{W})}{\partial \mathbf{W}^2\mathbf{h}^1(\mathbf{x}_n,\mathbf{W}^1)} \cdot \frac{\partial \mathbf{W}^2\mathbf{h}^1(\mathbf{x}_n,\mathbf{W}^1)}{\partial \mathbf{W}^2}$$
$$\frac{\partial E(g,t_n)}{\partial \mathbf{W}^1} = \frac{\partial E(g,t_n)}{\partial g(\mathbf{x}_n|\mathbf{W})} \cdot \frac{\partial g(\mathbf{x}_n|\mathbf{W})}{\partial \mathbf{W}^2\mathbf{h}^1(\mathbf{x}_n,\mathbf{W}^1)} \cdot \frac{\partial \mathbf{W}^2\mathbf{h}^1(\mathbf{x}_n,\mathbf{W}^1)}{\partial \mathbf{h}^1(\mathbf{x}_n,\mathbf{W}^1)} \cdot \frac{\partial \mathbf{h}^1(\mathbf{x}_n,\mathbf{W}^1)}{\partial \mathbf{W}^1\mathbf{x}_n} \cdot \frac{\partial \mathbf{W}^1\mathbf{x}_n}{\partial \mathbf{W}^1} \tag{1.6}$$

Note that Equation 1.6 refers to the simplest MLP structure depicted in Figure 1.4a, which was chosen for clarity. However, Chain Rule and Backpropagation are general algorithms and can be applied to any Neural Network structure.

Summarizing, the training process of the MLP depicted by Figure 1.4a can be schematized as follows:

1. Define a labeled dataset $D = \{(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), ..., (\mathbf{x}_N, t_N)\}$
2. Initialize MLP weights, even randomly;
3. *Feed* the MLP with data-point $\mathbf{x}_n$;

4. Compute the error function $E(g, t_n)$;
5. Compute the gradient of $E(g, t_n)$ through Backpropagation;
6. Update the weights through a GD technique;
7. Repeat the process until all data in D has been scanned once.

Each complete scan of the training set is known as *epoch*. A single training process involves multiple epochs, to better tune the weights.

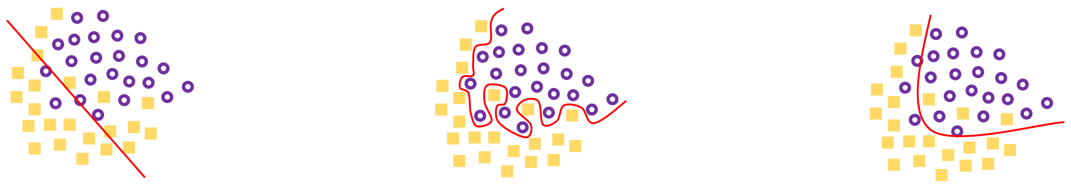
Note that both the desired output t_n and the MLP output function $g(\cdot)$ can be represented as vectors (e.g., when the MLP has multiple output neurons). In that case, also function E is expressed in vectorial form.
However, the overall training process remains the same.

Note also that the presented approach involves the computation of Backpropagation after *a single* data-point is processed (*Online Backpropagation*).
In is possible to compute it after *a subset* of data-points are processed (*Batch Backpropagation*.

Overfitting and Hyperparameter Tuning

The goal of the training process is to produce a MLP that is able to correctly approximate an unknown function. Therefore, the network should be neither to simple, nor too complex: a too simple network would not recognize the function, while a too complex one would replicate it exactly with respect to the training data.
Both behaviors are not recommended for practical application. Indeed, in presence of new, unseen data, both a too simple MLP and a too complex one would provide wrong results: the former, due to the lack of learning during the training phase, the latter, due to its specialization on training data.
A too simple MLP is said to *underfit* the data, while a too complex one is said to *overfit* them. The desired MLP is the one that is *general enough* to learn the unknown function during training and successfully apply it to new, unseen data. Figure 1.6a shows the behavior of an underfitting classifier, Figure 1.6b the behavior of a an overfitting classifier and Figure 1.6c the desired general behavior.

Since the complexity of a MLP is dependent on parameters that are *external* to the network (e.g., number of hidden layers, number of hidden neurons), it can not be adjusted during training, as it only concerns the update of the weights.
Those *external* parameters are known as *hyperparameters*, and are fixed before training



(a) Underfitting Classifier (b) Overfitting Classifier (c) Desired Classifier

Figure 1.6: Possible Behaviors of a Classifier. Subfigure (a), on the left, shows an underfitting classifier which is unable to approximate the equation of the decision boundary between the two classes, resulting in lots of misclassified points. Subfigure (b), in the center, depicts an overfitting classifier which specialized on the training data and perfectly represents the decision boundary between the classes without generating any misclassified point. Subfigure (c), on the right, presents the desired behavior expected from a classifier, because the two classes are clearly identified and the decision boundary is only approximated: the presence of some misclassified points suggests that the classifier is general and flexible enough to classify a new, unseen data-point, even if it is slightly different from the training ones.

by the programmer of the network. The process of selecting the most suitable hyperparameters is called *Hyperparameter Tuning*.

In order to face the problem of Underfitting/Overfitting the data and the process of Hyperparameter Tuning, the available dataset is usually partitioned into three independent subsets:

- *Training set*: dedicated to the update of the weights;
- *Validation set*: useful to provide unbiased evaluations during Hyperparameter Tuning. It is usually considered inside of the training process;
- *Test set*: useful to have a final, unbiased evaluation of the trained model with respect to new, unseen data.

The partitioning process must pay attention to generate independent subsets (without sharing data) and must ensure that the three subsets present the same data-distribution: data-sharing or distribution-mismatch would make MLP predictions biased and not robust, invalidating the training process.

Concerning Hyperparameter Tuning, the following list presents the most popular techniques used to find the best configuration among a manually-defined space:

- *Manual Search*: test only few configurations from the hyperparameter space, adjusting them manually;
- *Grid/Exhaustive Search*: exhaustively test each single configuration of hyperparameters. Efficient only if the hyperparameters space is small;
- *Random Search*: test only a randomly-selected subset of all possible combinations of hyperparameters. Preferred when the hyperparameter space is big.

- *Genetic Algorithm* [22, 45]: emulate *Natural-Selection Principle* considering populations of hyperparameters which evolve during time. From each population, only some configurations of hyperparameters, the ones that allowed to build the most-performing models, are selected and used to generate *offsprings* for the next population. The goodness of hyperparameters is assessed with a *fitness function* which can measure model performance. *Mutations* and selection of poor-performing configurations are added to the procedure to increase the diversity of the population and better simulate natural evolution. After a fixed number of generations, or when a fixed value of model performance is reached, the algorithm stops and the best-performing hyperparameters at that time are selected as final result.

MLP performance and generalization is measured through its error over the validation set, known as *validation error/loss.* Indeed, it provides a measure of how much the MLP is wrong over new, unseen data.

Concerning Underfitting and Overfitting, one should always start with a very simple MLP, and then adjust it according to its performance.
If the MLP starts overfitting, the following techniques can be considered:

- *Increasing training-set size*: more train-data can make the MLP able to better understand data-patterns;
- *Early Stopping* [49]: the training phase is stopped before its natural end (determined by a fixed number of epochs) if there is evidence that the MLP starts overfitting;
- *Data Augmentation* [64]: noisy data are added to the training set in order to make the network robust even in presence of non-ordinary patterns.
- *Regularization*: the addition of a *penalty* in model computations during training allows to avoid the generation of too big parameters, simplifying the model.
- *Batch Normalization* [32]: normalize layers inputs by re-centering and re-scaling. A dedicated layer, called *BatchNormalization layer*, is usually added where normalization is intended to take place.

Regularization is one of the most popular overfitting-limiting techniques, and in the particular case of the MLP, it can be implemented in multiple ways:

- *L1-Regularization* [8]: the penalization term $\frac{1}{2}\lambda\sum\|\mathbf{w}\| = \lambda\sum_{i=0}^{N-1}|w_i|$ is added directly to the error/loss function of the MLP (for vector of weights $\mathbf{w}$ of N elements);
- *L2-Regularization* [8]: the penalization term $\frac{1}{2}\lambda\sum\|\mathbf{w}\|_2^2 = \lambda\sum_{i=0}^{N-1}w_i^2$ is added directly to the error/loss function of the MLP (for vector of weights $\mathbf{w}$ of N elements);

- *Dropout* [28]: during training, disable each neuron of layer l with probability d^l, called *dropout rate*. The goal is to avoid that hidden neurons rely on other neurons. During validation/testing, all neurons are used, but weights are scaled accordingly.

The difference between L1 and L2 Regularization is that while L2 only shrinks the values of the weights towards zero without ever reaching it, L1 let some weights reach zero, providing a sort of automatic feature selection, which makes some inputs irrelevant to the computation of the output.

Both variable λ in L1 and L2 Regularization and variable d in Dropout are considered hyperparameters and should be carefully tuned.

1.1.3. Multi-Layer Perceptron as Classifier

Classification can be defined as the problem of identifying which category, from a fixed set, a given input belongs to. In ML jargon, categories are also known as *labels*.

Mathematically, considering a set of C possible classes $\{c_0, c_1, ..., c_{C-1}\}$ and an input datapoint x belonging to class c_i, with $i \in \{0, ..., C-1\}$, a classification problem is defined as the search of the function $f(\cdot)$ such that:

$$f(x) = c_i \tag{1.7}$$

Function $f(\cdot)$ is therefore an unknown function that maps known inputs to known outputs. As a consequence, *supervised learning* is the most suitable approach to solve a classification problem.
In particular, a MLP can be used to approximate function $f(\cdot)$.

A MLP for classification must feature a specific output activation function and a specific error/loss function. Indeed, the activation function of the output layer defines what the MLP produces, while the loss function computes the error over a prediction, and it is crucial in the Backpropagation algorithm for weights update. Both these tasks depend directly on the problem the MLP is addressing.

For a the generic C-class classification problem defined in 1.7 where each input belongs to *exactly one* class, the considered output activation function is the *softmax function*, defined as:

$$\sigma(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=0}^{C-1} e^{z_j}}, \text{ for } k \in \{0, 1, ..., C-1\} \tag{1.8}$$

In Equation 1.8, $\mathbf{z} \in \mathbb{R}^C$ is a vector of C real numbers and k is the class-index.

The peculiarity of the *softmax* is that it receives as input a vector of real numbers (even negative) and produces a probability distribution as output. This means that the complete output of a *softmax* is a vector of C probabilities, whose sum gives 1.
By implementing it as last layer of a MLP classifier, it provides the probability that the given input belongs to each class. The prediction of the MLP is the class with highest probability.

Concerning the *loss function*, the considered one is the *Cross-Entropy (CE)*, defined as

$$CE(\mathbf{t}, \mathbf{s}) = -\sum_{j=0}^{C-1} t_j \ln(s_j) \tag{1.9}$$

Equation 1.9 is general and refers to the error between a true-distribution $\mathbf{t}$ and a score distribution $\mathbf{s}$.

In the MLP case, $\mathbf{t}$ is an encoded representation of the class of a general input data-point. The most popular encoding technique for classification problems is *One-Hot Encoding*, which, applied to the problem defined by Equation 1.7, would label x with a C-long binary vector with exactly $C-1$ zeros and a unique one, at position i ($i \in \{0, ..., C-1\}$). For example, in a problem where the possible classes are $\{Red, Green, Blue\}$, class *Red* would be represented by $[1, 0, 0]$, class *Green* with $[0, 1, 0]$ and class *Blue* with $[0, 0, 1]$.

On the other hand, $\mathbf{s}$ is the class probability computed in the output layer of the MLP, being it a vector of probabilities relative to the input.

Therefore, summarizing, in the case of MLP classifier with input $\mathbf{x}$, the *Cross-Entropy* function involves $\mathbf{t}$, the vector representing the one-hot encoded class of $\mathbf{x}$, and $\mathbf{s}$, the predicted class probability vector for $\mathbf{x}$.
In this case, CE takes the name of *Categorical Cross-Entropy*, and it is the usual choice for MLPs while solving a classification problem.

1.2. Side-Channel Attacks

Cryptography is defined as the study of methods and techniques used to ensure secure communications between two endpoints. Its goal is to protect the information that is shared during a communication, even in presence of an adversary, whose intention is to retrieve such information.
Encryption is the cryptographic technique that allows to hide information contained in data. One of the first applications of encryption is *Caesar Cipher* [38]: Julius Caesar used to encrypt military-related messages shifting each letter of the alphabet 3 positions to the right (or to the left).

Modern encryption processes present four main elements:

- *Plaintext*: the input of the process, the data to be obscured;
- *Ciphertext*: the output of the process, the obscured version of the plaintext;
- *Algorithm*: the sequence of mathematical operations used to generate the ciphertext starting from the plaintext;
- *Key*: the cryptographic secret used by the algorithm to generate the ciphertext.

Encryption algorithms are divided into two main categories: *symmetric-key* algorithms and *asymmetric-key* algorithms.
In symmetric-key encryption algorithms, such as the Advanced Encryption Standard (AES) [21], the same secret key is used for both encryption and decryption. In a scenario where a communication is secured with symmetric encryption, all the endpoints must share the same secret key. This can be difficult in practice, because an additional trusted channel is needed. For this reason, symmetric encryption is usually adopted to secure static information, such as disk data.
On the other hand, asymmetric-key encryption algorithms, like Rivest-Shamir-Adleman (RSA) [54], require a pair of keys per endpoint. Each pair of keys is composed by a *public key*, which is visible to everyone and used to encrypt, and a *private key*, which is kept secret and used to decrypt. This method allows to solve the symmetric-encryption problem of sharing the secret key, making this kind of algorithms more suitable for communication security.

Modern encryption algorithms are implemented following the Kerckhoffs' Principle [33], which states that *a cryptographic system should be secure even if everything about it, except the key, is public knowledge.*

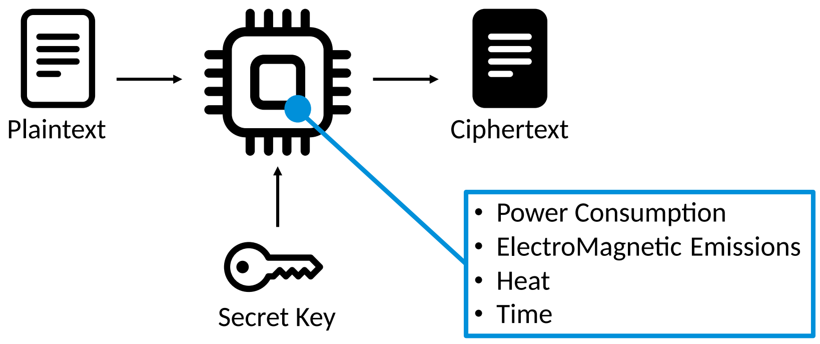


Figure 1.7: Scheme of a Side-Channel Attack. While the target-device is generating the ciphertext starting from the plaintext and the secret key, phisical variables such as Power Consumption, Electromagnetic Emissions, Heat or Time can be measured and later used to recover the secret key. The icons of this picture are taken from the open-source project Bootstrap [48]

Moreover, they are also proven to be mathematically secure against brute-force attacks. Indeed, the secret key is designed specifically to be long enough to make unfeasible, with the current technology, the exhaustive search of its correct value.

Side-Channel Attacks provide a non-invasive way to retrieve a secret encryption key without relying exclusively on brute-force methods.
A Side-Channel Attack (SCA), schematized in Figure 1.7, targets an electronic device (referred to as *target-device*) that is running an encryption algorithm, with the aim of retrieving its secret key, by exploiting the correlation between a physical variable and the secret key itself.

Physical variable refers to any measurable quantity produced by the device during the encryption, such as its Power Consumption (PC), its Electro-Magnetic (EM) emissions, the heat generated throughout the process or the time taken to complete it. Indeed, due to the widely-used Complementary Metal-Oxide-Semiconductor (CMOS) logic, the Power Consumption of an electronic circuit is proportional to the processed data, in particular to its bit-switching activity [47].
As a consequence, targeting the most sensitive operations of an encryption algorithm, it is possible to extrapolate information about the secret key. Therefore, the measured physical variables are called *traces*, while their correlation with the secret key is known as *leakage*, since it exposes the secret outside the device.

In practice, SCAs rarely target directly the secret key. Rather, they focus on predicting the value of a *leakage model*, defined as a function able to approximate the leakage of a device. A following step is dedicated to key-recovery, starting from the predicted leakage model.
The possible leakage models are Hamming Distance (HD), defined as the number of different bits between two binary numbers, Hamming Weight (HW), where $HW(x) = HD(0, x)$ and Identity (ID), defined as $ID(x) = x$.
The leakage model involves also an intermediate value of the targeted algorithm. This value is the result of a sensitive operation in the encryption process. In SCA, an operation is *sensitive* if it causes some memory change (e.g., change the value of a register), for the reasons mentioned before.

The most popular metric used to evaluate a SCA performance is the so-called *Guessing Entropy*.
The Guessing Entropy (GE) is defined as the average *rank* of the correct key among attack predictions [60]. The underlying idea is to consider the result of the SCA as a list of probabilities, one for each possible value of the secret key, structured in decreasing order, and select the position-index of the correct key. Since the list of probabilities is ordered largest to smallest, it is considered as a *ranking* of the possible key values and, therefore, the position index is called *rank*. The value of GE is computed as average over a fixed number of independent experiments involving disjoint sets of traces to attack.
The amount of attack-traces considered in a SCA is crucial for the exploitation of the leakage, since too few traces could make the leakage non-recognizable. Therefore, it can happen that the same SCA presents different GE values for different amounts of traces.
A SCA is said to be *successful* if it reaches $GE = 0$, meaning that its most probable prediction coincides with the correct key. On the other hand, a SCA that reaches $GE = N \neq 0$ suggests the correct key as the $(N+1)^{th}$ most probable value: in this case, the secret key can not be directly retrieved with the attack, but the eventual brute-force space has been reduced.
For this reasons, given multiple SCAs, the best one is the attack that reaches the lowest GE in the least number of traces.

1.2.1. Details on the Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [21] has been the standard algorithm for data encryption since 2001, when it was chosen by the U.S. National Institute of Standards and Technology (NIST).
AES encrypts a fixed-sized plaintext of 128 bits using a secret key of 128, 192 or 256 bits.

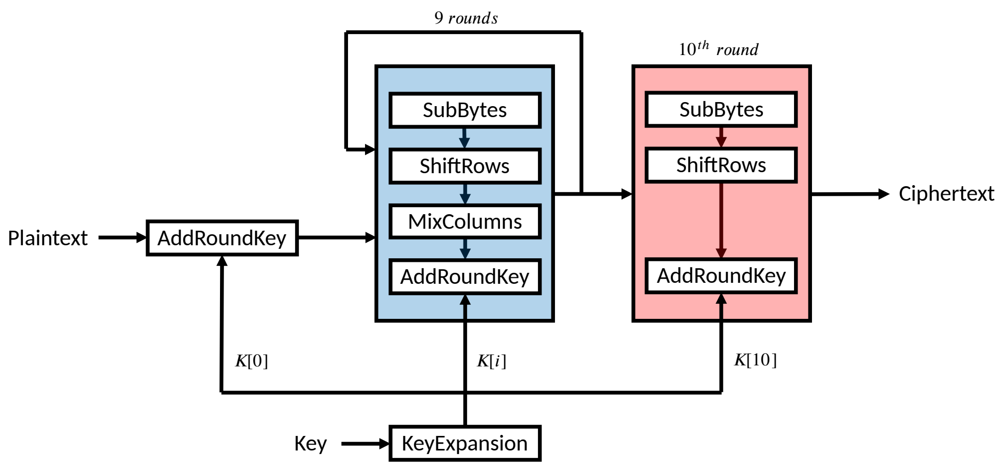


Figure 1.8: AES-128 Algorithm Structure. A 128-bit plaintext is encrypted with a 128-bit secret key, generating a 128-bit ciphertext.

The algorithm is structured on multiple rounds, where 4 main operations are performed sequentially. The number of rounds depends on the the length of the key: 10 rounds for 128-bit key, 12 rounds for 192-bit key and 15 rounds for 256-bit key.

Figure 1.8 shows the structure of AES-128, the version of AES where the key is 128-bit long.
At the beginning of the algorithm, the *KeyExpansion* operation generates 11 round-key starting from the main secret key. The number of round-keys is exactly $n\_rounds + 1$, where the first ($K[0]$ in the figure) is the main secret key without modifications.
AES operates on a 4×4 matrix called *state*. The 16 bytes $b_0, b_1, ..., b_{15}$ of the plaintext are stored in the state in *column-major order* as follows:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix} \tag{1.10}$$

The four main operations of AES are:

- *AddRoundKey*: Each byte of the state is XORed with the corresponding byte of the round-key;

- *SubBytes*: Each byte of the state is replaced with another, according to a lookup table called *SBox*;
- *ShiftRows*: The last three rows of the state are shifted cyclically with increasing offset;
- *MixColumns*: The columns of the state are mixed using an invertible linear transformation.

The *SubBytes* operation is crucial for AES, since it adds non-linearity to the algorithm. This operation is performed over the whole state, proceeding byte-by-byte, indexing the SBox: the table row-index is given by the first 4 bits of the considered byte, while its column-index is given by the remaining 4.

After an initial *AddRoundKey* where the plaintext is XORed with the main secret key, the first 9 rounds feature the same sequence of operations: $SubBytes \rightarrow ShiftRows \rightarrow MixColumns \rightarrow AddRoundKey$. The last round presents the same order, but without the $MixColumns$ operation.

AddRoundKey and *SubBytes* are the most sensitive operations of AES, since the former handles both plaintext and key, and the latter adds non-linearity through the SBox-lookup. This makes them suitable targets of a SCA. Indeed, the most popular intermediates when attacking AES are the result of *AddRoundKey*, known as *sbox-in*, and the result of *SubBytes*, called *sbox-out*.

1.2.2. Direct Side-Channel Attacks

Traditional Side-Channel Attacks see the attacker applying statistical methods to traces collected directly from the target-device, in order to highlight the leakage.
This approach is called *Direct*, because the attacker focuses straight on the target-device.

Simple Power Analysis and Differential Power Analysis are the most popular Direct SCA techniques. Both were introduced by Kocher et al. [35, 36].

Simple Power Analysis (SPA) consists in directly interpreting side-channel traces to gain information about the operations performed by the target-device. For example, the visual inspection of a power-consumption trace collected from a device running AES-128 without countermeasures reveals the presence of exactly 10 peaks, one for each round of the algorithm.
SPA Attacks usually do not reveal directly the secret key, but can be very helpful as preliminary analysis when performing more sophisticated attacks.

On the other hand, Differential Power Analysis (DPA) is a statistical method used to identify data-dependent correlations from power-consumption traces and it can be exploited to directly recover the secret key of an encryption algorithm.
A basic DPA Attack consists in partitioning a set of traces into subsets, calculating the average trace of each subset and finally computing the difference of the average traces, often called *differential trace*. If the partitioning-method is correlated with the measurements, then the resulting difference tends to be a non-zero value, otherwise, it approaches zero.
In order to recover the secret key, the basic DPA procedure is exploited considering traces generated with multiple key-candidates: the candidate which leads to a differential trace with the highest non-zero values (visible spikes) is the guessed key value.

Differential ElectroMagnetic Analysis (DEMA) [3, 50] and Correlation Power Analysis (CPA) [10] are other common Direct SCA techniques, variants of DPA.

1.2.3. Profiled Side-Channel Attacks

A different Side-Channel Attack approach sees the attacker not only accessing the target-device, but also having complete control over a *copy* of it (i.e., a physically different device that comes from the same manufacturer). The copy is often called *clone-device*.
This approach is known as *Profiled*, because a characterization phase exploiting the clone-device precedes the actual attack against the target-device.
Indeed, Profiled-SCAs consist of two steps:

1. *Profiling Phase*: The attacker collects traces from the clone-device and uses them to generate a *model* of the leakage, characterizing its behavior.

2. *Attack Phase*: The attacker applies the produced model to some traces collected from the target-device in order to retrieve its secret key.

The Template Attack (TA) [16] is an example of Profiled-SCAs.
The model of the leakage generated in a TA is called *template*, and consists in the mean and the covariance matrix of the measurements collected from the clone-device. Indeed, every measured signal is considered as a random variable, result of the composition of a constant value and a random noise. The hypothesis that the random noise follows a Multivariate Gaussian distribution allows the computation of the template. During the *Profiling Phase*, a template is generated for each possible value of the considered target (e.g., key-byte value).
In the *Attack Phase*, considering traces collected from the target-device, the values of the Probability Density Function (PDF) relative to each possible target are computed,

exploiting the templates and the Multivariate Gaussian distribution hypothesis. In the end, following the Maximum Likelihood approach, the target which leads to the highest PDF value is the result of the attack.

Recently, the search of a more general and distribution-agnostic solution to Profiled-SCAs led to the usage of Machine Learning (ML) techniques during the Profiling Phase [6, 26, 31]. This technique, which mainly exploits Support Vector Machines (SVMs) [18, 63], makes no assumption about the distribution of the noise and approaches a SCA as a classification problem where the classes are represented by the possible values of the targeted intermediate.
This particular Profiled-SCAs are known as ML-based SCAs.

1.2.4. Side-Channel Attack Countermeasures

Section 1.2 highlights how side-channels such as PC or EM-emissions are strictly related to the operations performed by the device.
Therefore, in order to avoid the presence of leakage, countermeasures can be developed both in hardware and in software.

SCA countermeasures can be divided into three main categories:

- *Hiding* [40]: random delays and noise are manually added during the computation to eliminate/reduce sensitive information contained in the leakage;
- *Masking* [15, 53]: per-execution arbitrary values are used to randomize key-related data, eliminating the correlation between traces and secret;
- *Morphing* [2]: dynamic changes are added to the encryption code, keeping semantic conformity.

In particular, a software *Masking* countermeasure is implemented as follows.
Every sensible variable x of the cryptographic algorithm to secure is split into $d + 1$ quantities $x_0, x_1, ..., x_d$, also known as *shares*. In particular, shares $x_1, ..., x_d$ are called *masks*, while x_0 is the *masked value*. When d masks are considered, the masking technique is said to be *of order d*.
The shares are generated in such a way that, for a group operation $\perp$, the following relation holds:

$$x_0 \perp x_1 \perp ... \perp x_d = x \tag{1.11}$$

Masking countermeasures for AES feature the XOR operation (denoted by $\oplus$) as group operation [57].

The presence of masks allows to make the intermediate results of the considered algorithm independent of sensitive variables. Therefore, a SCA targeting a single intermediate variable is no more effective against this kind of implementation.
As a consequence, in order to reveal the actual leakage, an attacker should exploit, at the same time, information about both the masks and the masked values [44, 53, 57].

More generally, to break a masking countermeasure of order d, an attacker needs to exploit information about all d masks and the masked values at the same time. For this reason, a masking countermeasure of order d is said to be vulnerable to an *attack of order* $d + 1$. This kind of attacks is theoretically possible, but high values of d can make it unfeasible.

To perform an attack of order $d + 1$, one must preprocess the traces to individuate $d + 1$ points of interest, where the leakages can be exploited. With classical statistical methods, this task may require a long time, and surely a huge amount of traces.
DL networks, on the other hand, can automatically perform feature selection without the need of an explicit preprocess in search of points of interest. This speeds up the attack, both in terms of time required to collect the traces and in terms of manual operations to perform.

1.3. Deep Learning based Side-Channel Attacks

As briefly introduced in section 1.2.3, one of the recent trends in SCA field focused on the extension of Profiled-SCAs through the usage of Machine Learning techniques.
This approach relaxes the assumption of Multivariate Gaussian noise in the traces, providing a way to develop distribution-agnostic attacks.
On the other hand, ML-based SCAs do not provide the possibility of completely automate the Profiling Phase. Indeed, ML techniques fit data successfully only if a preliminary feature-extraction phase has been performed manually. In SCA terms, the attacker needs to manually extrapolate the sections of the trace where the leakage is present (usually performed considering the Pearson Correlation Coefficient between a trace and its relative leakage model [10]) and provide them as input to the ML algorithm.

One of the emerging trends in Profiled-SCA field aims precisely at automating the Profiling Phase keeping a distribution-agnostic approach. This trend involves Deep Learning Artificial Neural Networks (ANNs) in order to model the leakage of the device [11, 39]. As a consequence, this new class of attacks is knonw as DL-based SCAs.

The following characteristics make ANNs particularly suitable for the aforementioned goal:

- Their layered structure allows to automatically detect the most relevant features from data without being explicitly programmed to do so, but learning from data themselves and improving with experience;
- Their ability of approximating complex functions allows to relax the assumption of Multivariate Gaussian noise in the traces.

How DL-based SCAs Work

DL-based SCAs approach a SCA as a classification problem, similar to the one described by Equation 1.7.
The data-points to be classified are the traces, while the classes are the possible value that the targeted leakage model can assume.

An ANN is trained to associate to each trace the value of the relative *leakage model*. Therefore, the only difference between DL-based SCAs and traditional classification problems is that DL-based SCAs need an additional step to retrieve the secret key, since it is not the target of the classification performed by the network.
Therefore, the *accuracy* of the ANN can be misleading: as pointed out in [11], in a DL-based SCA the accuracy tells how good a network is in presence of *a single* attack trace. Indeed, it is computed with respect to the predictions of the network, which are not the actual target of the attack. As a consequence, it can happen that good attacks feature a poor network accuracy. For these reasons, accuracy should not be considered when mounting DL-based SCAs.

In practice, a DL-based SCA can be seen as a Profiled-SCA where:

- The Profiling Phase involves the training of the ANN over traces coming from the *clone-device*
- The Attack Phase involves the usage of the trained ANN to make predictions over new, unseen traces coming from the *target-device*

The additional *key-recovery* step needs to combine the predictions related to each single trace coming from the target-device. Indeed, the secret key is the same for each attack-trace, and it is the actual target of the attack.

Concerning the evaluation technique, GE is the usual choice for DL-based SCAs.
However, GE should be avoided while assessing network performance during the Profiling Phase: it requires too much time and resources to be computed multiple times during network training. For this reason, network-related metrics such as the *prediction error* over validation data [42] are exploited to assess network performance.

The Portability Problem

In a realistic DL-based SCA, *clone-device* and *target-device* are two physically different devices, usually coming from the same manufacturer. In addition, they encrypt with different keys, being the targeted secret key unknown to the attacker.
As a consequence, the considered network is trained over data coming from a given device-key configuration, and expected to classify traces coming from a different one.

From a Machine Learning point of view, this approach can be prone to *overfitting*, since the network may *specialize* in correctly classifying only traces coming from the *clone-device*, providing wrong predictions for new, unseen data during the Attack Phase.

For this reason, DL-based SCAs are said to be affected by the *Portability Problem*, defined as the challenge of *building a general enough ANN which is able to successfully attack a device-key configuration that is different from the one used in the Profiling Phase*.
The state-of-the-art solution to the Portability Problem is the Multiple Device Model (MDM) [7], which aims to increase the diversity of the training set considering traces that come from at least two devices, in order to develop a more general ANN.

1.4. Related Work and Contribution

Related Work

Deep Learning was applied for the first time to Side-Channel Attacks by Maghrebi et al. [39]. They not only defined the DL-based SCA pipeline, but also provided a detailed comparison between Template Attacks (TAs) [16], existing SCAs based on Machine Learning techniques [6, 26, 31] and their novel methodology involving several types of neural networks (AutoEncoders [41], Convolutional Neural Networks, Multi-Layer Perceptron and Long-Short Term Memory [29]). The proposed approach was innovative, since it completely automated the Profiling Phase by exploiting the ability of the networks to perform automatic feature-extraction from the traces: the manual selection of the leakage points is not necessary anymore. DL-based SCAs have been proven to outperform both TAs and ML-based SCAs. The same work reported analogous results repeating the experiments against a masked implementation of AES.

A different approach to attack masked implementations of AES with neural networks was proposed by Gilmore et al. [23] and Bae et al. [5]. Assuming that the attacker knows the value of the mask during the Profiling Phase, a network can be trained to retrieve the masked version of an intermediate value, while a different network can be trained to recover the value of the mask. Combining the two results, it is possible to obtain the unmasked intermediate value and consequently the key.

Cagli et al. [11], instead, showed that Convolutional Neural Networks are able to retrieve the secret key even in presence of trace-misalignment, thanks to their ability of perform automatic feature-extraction. In particular, the authors underlined the possibility of completely automate the Profiling Phase eliminating the manual feature-selection phase usually performed before the attack, and pointed out the importance of data-augmentation [64] when the targeted traces present consistent misalignment.

The Portability Problem was tackled for the first time by Bhasin et al. [7]. They pointed out that the performance of a DL-based SCA drops significantly by going from attacking the same device used for training to attacking a different one.
The authors identified the cause of this behavior as overfitting, highlighting how a neural network can easily specialize on the provided training set and therefore provide wrong results for data coming from a different source. The proposed solution aims at increasing the generalization of the considered neural network by diversifying the training set with traces coming from multiple sample-devices. This method is known as Multiple Device Model (MDM), and it is considered the state-of-the-art solution to the Portability Problem.
In particular, MDM suggests to structure a DL-based SCA in the following way:

1. Consider a total of $n \geq 3$ devices;
2. Dedicate $t \geq 2$ devices (*train-devices*) to the Profiling Phase and 1 (*test-device*) to the Attack Phase;
3. Build the training set collecting k traces from each train-device;
4. Build the validation set collecting j traces from each train-device, paying attention to generate a disjoint set with respect to the training one;
5. Train and validate a neural network with the collected traces;
6. Build the test set collecting traces from the test-device;
7. Attack the test-device with the trained network.

Contribution

The novel methodology introduced by this work aims at providing efficient training-configurations to mount effective realistic DL-based SCAs against AES-128. In order to do so, three main variables are considered: the number of train-devices, the number of keys provided during the Profiling Phase and the number of train-traces. Therefore, differently from [7], this work addresses the Portability Problem not only in terms of number of devices, but also considering the impact of other two variables. Moreover, the proposed methods repeat the attacks considering all the possible device-key or device-trace combinations with the available setup, computing, in the end, the average result: this is an additional difference with respect to [7], where MDM was tested only for specific device-key configurations.

In addition, for the first time, this work applies MDM to 32-bit microcontrollers and to a masked implementation of AES-128. Indeed, the effectiveness of MDM was only validated on 8-bit CPUs targeting an unprotected implementation of AES-128 [7, 62].

In the end, the followed approach to attack a masked implementation of AES is the same explained in [39]: the traces are collected and provided as input to the network without performing any manual feature-selection procedure. This automates the Profiling Phase and lets the network recognize and exploit the relevant features. In addition, the attack is made even more realistic since no knowledge about the value of the masks is needed.

2 | Research Methodology

This chapter presents the methodological approach adopted in this research work.
At the beginning, the addressed problem is described, alongside the proposed solution. The following sections are dedicated to explain some practical details of the proposed solution, highlighting relative choices and motivations.

2.1. Addressed Problem and Proposed Solution

As previously introduced in 1.3, the Portability Problem is the major concern when mounting high-performance realistic Deep Learning based Side-Channel Attacks. Its state-of-the-art solution, known as Multiple Device Model (MDM) [7], involves the usage of multiple devices during the Profiling Phase in order to make the DL network more general.

This research work considers the Profiling Phase of DL-based SCAs dependent on three main variables, identified in the following quantities: *number of devices*, *number of keys* and *number of traces*.
Therefore, with respect to the State-of-the-Art [7], the following questions remain unanswered:

- *How does the number of keys used in the Profiling Phase influence the attack performance?*
- *How does the number of train-traces influence the attack performance?*

Moreover, the relationship among variables can be explored, introducing the following questions:

- *Are there possible scenarios where the number of keys used during the Profiling Phase can enhance the attack performance in case of limited number of devices?*
- *Are there possible scenarios where the number of train-traces can enhance the attack performance in case of limited number of devices?*

Finally, the effectiveness of MDM has only been validated on 8-bit microcontrollers [7, 62].

This suggests another question:

- *Does the MDM solution work also on microcontrollers that do not feature an 8-bit architecture (e.g., 32 or 64-bit)?*

This research work, taking inspiration from the work of Bhasin et al. [7] about the MDM, aims at answering all the previous questions, not only in the case of DL-based SCA against a software unprotected version of AES-128, but also targeting a software implementation with masking countermeasures. The considered target-device is a 32-bit microcontroller. Moreover, this work intends to provide efficient train-configurations to mount effective DL-based SCAs in the considered scenarios.

In order to achieve its goals, this work completely characterizes the Profiling Phase of DL-based SCAs, in terms of a trade-off between its most important variables.
This technique is formed by two trade-off analyses, one between the number of devices and the number of keys, called *Device-Key Trade-off Analysis (DKTA)*, and one between the number of devices and the number of traces, called *Device-Trace Trade-off Analysis (DTTA)*.

Since they consider different variables, DKTA and DTTA present different motivations:

- *DKTA* relies on the practical aspect of the attack: considering multiple keys is simpler than considering multiple devices, since the keys can be automatically generated via software for free, while the devices must be purchased.
- *DTTA* relies instead on the Machine Learning *rule-of-thumb* that involves the addition of train-data to prevent overfitting: with more data available it should be easier, for the network, to recognize the targeted general patterns.

From a conceptual point of view, however, the two analyses are equivalent: starting from a fixed number of of train-devices, they track the performance of multiple attacks in terms of Guessing Entropy, where the number of keys (for DKTA) or the number of traces (for DTTA) is gradually increased.

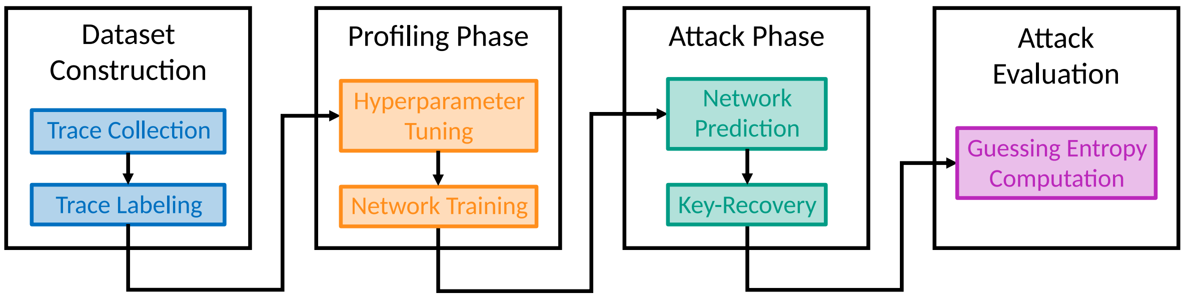


Figure 2.1: DL-based SCA Pipeline. A DL-based SCA is structured in 4 main phases: Dataset Construction, Profiling Phase, Attack Phase and Attack Evaluation.

2.2. Attack Pipeline

This section describes in detail the steps that this work followed in order to mount a DL-based SCA.
Figure 2.1 summarizes the attack pipeline. The first step involves the construction of the datasets, both for training and attack, done by collecting the traces and labeling them. Next, the Profiling Phase is performed: it includes the choice of the DL network, the search of its optimal hyperparamters and its training. Then, during the Attack Phase, the trained network is applied to new, unseen traces coming from the target-device in order to retrieve its secret key. The final step is the evaluation of the attack, performed computing its Guessing Entropy.

2.2.1. Dataset Construction

The first step in the development of a DL-based SCA is the collection of the traces. However, before starting the trace capture, it is important to define the target of the attack in terms of encryption algorithm, sensitive operation and intermediate value, and leakage model.
This work targets AES-128 with and without SCA masking countermeasures. The considered sensitive operation is the *first round SubBytes operation*, and the targeted inter-

mediate value is its output, known as *sbox-out.* Concerning the leakage model, *Identity (ID)* is preferred. The underlying motivations to these choices are the following:

- AES is the standard algorithm for encryption and therefore widely used in a variety of different devices (e.g., SSDs, smart cards);
- When attacking AES with a SCA, the only rounds that can be targeted are either the first or the last. Indeed, from a practical point of view, a SCA should target only those operations that are invertible, allowing the computation of the key starting from the intermediate. The first round is a suitable choice because it involves directly the plaintext, which is known by the attacker. The last round is suitable as well due to the presence of the ciphertext, which is itself known to the attacker. Intermediate round i, with $1 \leq i \leq 9$, should not be attacked because, in order to recover the key (*round-key* referring to Figure 1.8), the output of round $i - 1$ is needed, and it is unknown to the attacker.
- The *SubBytes* operation is one of the sensitive operations of AES. It adds non-linearity to the algorithm performing the SBox-lookup and, therefore, it makes the leakage to be uniformly distributed.
- The Identity leakage model allows a simpler key-recovery step, allowing to target the actual value of the intermediate.

Regarding the type of side-channel to exploit, this work considers the power consumption of the device.
It is important to point out that the traces considered by this work are aligned: the oscilloscope exploits a trigger signal coming from the device during the encryption to know when the targeted operation starts. More details about this feature are in Section 3.1. This scenario can be considered realistic since DL-based SCAs are *Profiled-SCAs* and so, during the Profiling Phase, the attacker has full control of the device, being able to set the trigger at will. Speaking of the positioning of the trigger, in this work it depends on the presence of the SCA masking countermeasure: when the countermeasure is not present, the attack considers short traces involving the measurements of the power consumption during the first round *SubBytes* operation only, while when the countermeasure is in place, the traces are enlarged considering also measurements related to adjacent operations.

This work considers a single preprocessing operation, the *scaling* of the traces. In particular, the traces are scaled to *0-mean* and *unit-variance* to make the Gradient Descent technique converge faster during training, sometimes increasing the perfromance of the

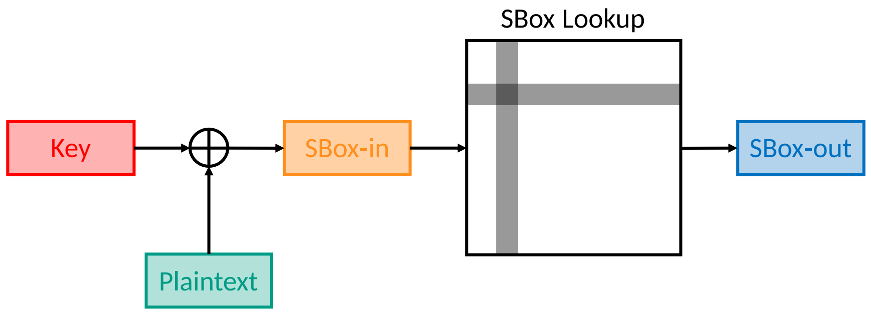


Figure 2.2: Forward Computation of *AddRoundKey* and *SubBytes* operations. Starting from known plaintext and key, it is possible to calculate the value of *sbox-in* through the XOR operation. The value of *sbox-out* is computed performing SBox-lookup with the value of *sbox-in*.

network [32]. The following equation defines how the scaling is performed:

$$x_{new} = \frac{x - \mu}{\sigma} \tag{2.1}$$

In Equation 2.1, x is a raw-trace, x_{new} is its scaled counterpart, μ is the mean value of the trace-set and σ is its standard deviation.
In this work, great attention is paid to the computation of μ and σ. Indeed, they must refer to the training data only, since considering the whole dataset would cause a sort of *information leakage* between training set and validation/test sets, while they should be independent.

The collected traces need also to be labeled with the relative value of *sbox-out*. Indeed, a DL-based SCA is essentially a classification problem where the network has to predict the correct value of the targeted intermediate, and therefore it is handled with a *Supervised Learning* approach. Each attack of this work, for simplicity, targets a single byte of the key, so also the labels need to refer to a single byte of *sbox-out*.
Figure 2.2 shows how to generate the *sbox-out* label for a given trace emulating the first round *AddRoundKey* and *SubBytes* operations of AES: knowing plaintext and key, *sbox-in* can be easily computed XORing them, while *sbox-out* is retrieved indexing the SBox. In particular, the first four bits of *sbox-in* form the row-index, while the last four form the column one.

Even though scaling and labeling are conceptually executed just after the collection of the traces, in the workflow of this research they are performed *on-the-need* while reading data

Hyperparameter Space

Hyperparameter	Values
Number of Hidden Layers	$\{1, 2, 3, 4, 5\}$
Number of Hidden Neurons	$\{100, 200, 300, 400, 500\}$
Dropout Rate	$\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5]$
L2-Regularization Coefficient	$\{0.0, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05]$
Gradient Descent Technique	$\{adam, rmsprop, sgd\}$
Learning Rate	$\{0.0, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005]$
Batch Size	$\{128, 256, 512, 1024\}$

Table 2.1: Hyperparameter Space. For each hyperparameter, this table provides the possible values that it can take. This space is used as reference by the Genetic Algorithm that performs the Hyperparameter Tuning.

from the traces. Concerning scaling, this is done because different attacks can consider different amounts of traces, thus causing a variation in μ and σ. On the other hand, concerning trace-labeling, this is done to reduce the time needed to load entire trace-sets from file.

2.2.2. Profiling Phase

The Profiling Phase starts with the choice of the DL network. This work prefers a Multi-Layer Perceptron (MLP) over a Convolutional Neural Network (CNN) since the traces are already aligned [11] and therefore there is no need of further computations. The MLP features an input layer with as many neuron as the number of samples in the traces, and an output layer of exactly 256 neurons, one for each possible value of the targeted intermediate, which is a byte (and therefore its value is in $[0; 255]$).

Multiple experiments revealed that the network is prone to overfit the data, highlighting the need of regularization techniques. In particular, the simultaneous employment of BatchNormalization layers, Dropout layers, L2-Regularization an a dynamic learning rate that decreases when the network starts showing an overfitting behavior appeared to be the best-performing one.

After the choice of the network, Hyperparameter Tuning is performed to define the optimal structure of the network.
The hyperparameters considered in this work are shown in Table 2.1, alongside their possible values. As a consequence, Table 2.1 is known as *hyperparameter space*, since it

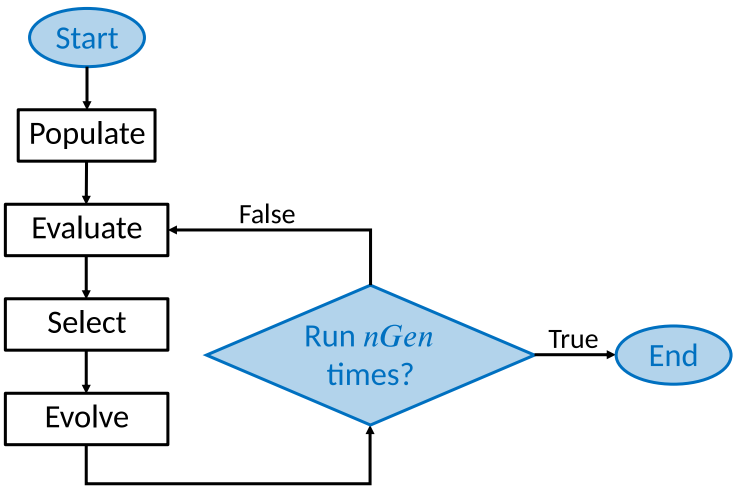


Figure 2.3: Genetic Algorithm Block Diagram. This diagram shows the main operations performed by a Genetic Algorithm. The stopping criterion, the fixed number of generations ($nGen$), is the one considered by this work.

represents the space where to perform the search during Hyperparameter Tuning. The considered hyperparameters of this work are:

- *Number of hidden layers*: depth of the network and its complexity (together with the number of hidden neurons);
- *Number of hidden neurons*: how many neurons consider in each hidden layer, influencing the complexity of the network (together with the number of hidden layers);
- *Dropout rate*: percentage of neurons to *turn-off* per layers during training;
- *L2-Regularization coefficient*: λ coefficient of L2-Regularization [8];
- *Gradient Descent technique*: technique to compute Gradient Descent;
- *Learning Rate*: controls the *speed* of the learning process;
- *Batch size*: how many traces to provide to the network at once during the training process.

The search of the optimal hyperparameters, in this work, is performed with a Genetic Algorithm [22, 45], in particular adapting the work of Matt Harvey et al. [24].
Figure 2.3 summarizes how a Genetic Algorithm works, while Algorithm 1 provides the pseudocode of the Hyperparameter Tuning process.

The algorithm is initialized generating a population of hyperparameters. The size of the population (*popSize*) is an input to the algorithm and never changes. Every configuration belonging to this population features random hyperparameters: their values are randomly

Algorithm 1 Genetic Algorithm for MLP Hyperparameter Tuning.

```
1: $pop \leftarrow$ Populate($hpSpace$, $popSize$)
2: for i $\leftarrow 0$ to $nGen - 1$ do
3:     $sortedPop \leftarrow$ Evaluate($pop$)
4:     $parents \leftarrow$ Select($sortedPop$, $popSize$, $selPerc$, $scProb$)
5:     $pop \leftarrow$ Evolve($parents$, $popSize$, $hpSpace$, $mProb$)
6: end for
7: $bestHPs \leftarrow sortedPop[0]$
8: return $bestHPs$
```

chosen from the hyperparameter space (*hpSpace*) defined by Table 2.1. Then, each configuration of the population is used to build a MLP, which is trained on the training set and evaluated on the validation one. The *validation loss* of each network is stored. Once all the networks are evaluated, the stored validation losses are compared in order to select only part of the configurations, the ones that led to the best-performing MLPs. In order to add diversity to the next population (emulating the natural process of evolution), even some bad-performing configurations are considered. Both the percentage of configurations that are selected (*selProb*) and the chance for a bad-performing configuration to be among the best (*scProb*) are input parameters to the algorithm. The selected configurations are known as *parents* and are combined, in couples, to generate *offsprings*, which compose the new population. An offspring is a configuration of hyperparameters where the values are randomly chosen from the parents. In order to add diversity to the new population, each offspring has a small chance to be mutated, meaning that some random hyperparameters can be replaced by a random value taken from the hyperparameter space. The mutation probability (*mProb*) is an input parameter to the algorithm.
This process of evaluation, selection and evolution is repeated for a fixed number of generations (*nGen*), which is a parameter of the algorithm. The best-performing configuration at the end of the process is the selected set of hyperparameters.

Throughout this work, the process of Hyperparameter Tuning is performed every time a train-device is added or when the target algorithm is changed. In addition, since in DL-based SCAs a single network attacks a single byte of the key, the search is performed also if the targeted byte is changed.
This choice is aligned with the idea of realistic attacks, because it emulates the behavior of an attacker who tries to take full advantage of the available training-setup.

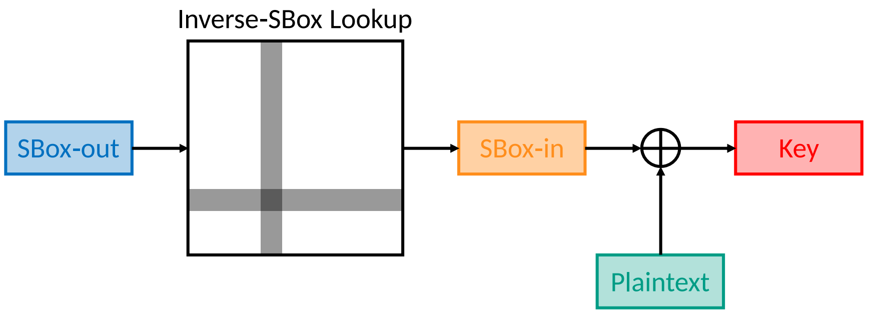


Figure 2.4: Backward Computation of *SubBytes* and *AddRoundKey* operations. Starting from the predicted *sbox-out*, the value of *sbox-in* is retrieved performing table-lookup over the Inverse-SBox. XORing the value of *sbox-in* and the known plaintext it is possible to retrieve the value of the secret key.

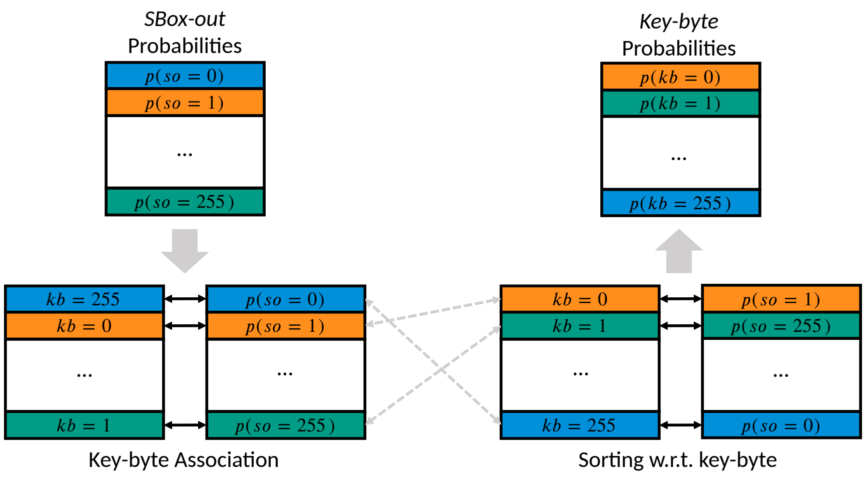
2.2.3. Attack Phase

The peculiarity of DL-based SCAs is that what the networks are trained to predict is *not* the actual target of the attack. Indeed, in this work the MLP predicts the value of *sbox-out* relative to each trace, but the attack aims at recovering the value of the key.
For this reason, the Attack Phase is composed by two steps: *network prediction* and *key-recovery*.

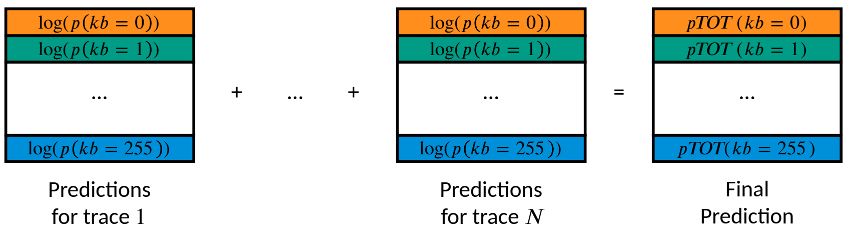
During network prediction, the trained network generated during the Profiling Phase is applied to the traces coming from the target-device: for each trace, the network provides a probability vector over the values of the targeted intermediate, since a DL-based SCA is treated as a classification problem.
On the other hand, during the key-recovery step, the generated probability vectors are combined to retrieve the secret key.

In this work, the key-recovery step is performed by first converting the probability vectors over the *sbox-out* into probability vectors over the *key-byte*, and then combining them together.
This conversion step makes the predictions comparable, being all relative to the same key. Without conversion, the probability vectors would be relative to a different value of *sbox-out*, resulting non-comparable.

The conversion step is feasible because it is possible to compute the key starting from the *sbox-out*, as shown in Figure 2.4: first, the *Inverse-SBox* is indexed with the value of



(a) Conversion of *sbox-out* Probabilities into *key-byte* Probabilities



(b) Computation of the Final Prediction

Figure 2.5: From Network Prediction to Attack Result. Subfigure (a), on the left, shows how *sbox-out* probabilities are converted into *key-byte* probabilities through sorting. Subfigure (b), on the right, summarizes the procedure that allows to combine the predictions over different traces to define the result of the attack, the *final prediction*.

sbox-out to generate *sbox-in* and then, thanks to the fact that the XOR operation is the inverse of itself, the key can be generated XORing *sbox-in* and plaintext, which is known to the attacker.

Figure 2.5a shows how the conversion step is performed, and can be summarized as follows:

1. Consider a probability vector over *sbox-out (so)*;
2. Associate to each element of the vector the value of the relative *key-byte (kb)*, following the procedure depicted in Figure 2.4;
3. Sort the probability vector *with respect to the computed key-byte*;

4. Leave the *key-byte* values out and consider only the sorted probability vector over *sbox-out*;

Even if the result of this procedure is a sorted version of the starting probability vector over *sbox-out*, it can be considered a *full-fledged* probability vector over *key-byte*.

On the other hand, Figure 2.5b shows the combination process of the key-recovery step: the *log-probabilities* are computed and summed up.
The underlying idea is the following: a DL-based SCA over N attack traces can be thought of as a set of N independent events of type *the targeted key-byte has value* x*, with* $x \in [0; 255]$, because the act of predicting *sbox-out* for trace j is independent of the act of predicting *sbox-out* for trace k, with $j \neq k$. Therefore, since the MLP provides a probability vector over the values of *sbox-out* for each trace and the *joint probability* of independent events is the product of the probabilities of the single events, the network outputs related to each single trace can be multiplied. The *log-probabilities* are computed to consider the sum over the product, avoiding possible multiplications by 0, which would produce inconsistent results.

In this work, the vector resulting from the key-recovery step is called *final prediction*, since it is the probability vector over the *key-byte* that is produced at the end of the attack, with the considered number of traces. Element i (zero-based) of this vector is the *total probability (*$pTOT$*)*, with respect to the considered amount of attack traces, that the targeted *key-byte* assumes value i, with $i \in [0; 255]$.
The most probable key-byte is the actual result of the attack, with the considered number of traces.

2.2.4. Attack Evaluation

The *final prediction* vector is usually sorted in decreasing order, creating a ranking among the possible values of the targeted key-byte. This is very useful in order to evaluate the attacks with Guessing Entropy (GE), the state-of-the-art metric for SCA [60]. It is defined as the average position (rank) of the correct key-byte among the *final prediction*.

In this work, the Guessing Entropy is computed as average over 100 independent experiments, where an *experiment* is defined as the generation of a *final prediction* vector, starting from a set of traces. In particular, the 100 sets of traces are disjoint, being a partition of the attack dataset.

Usually, the value of GE tends to vary if different amounts of traces are considered. Therefore, GE is computed for incremental number of traces, starting from a single one.

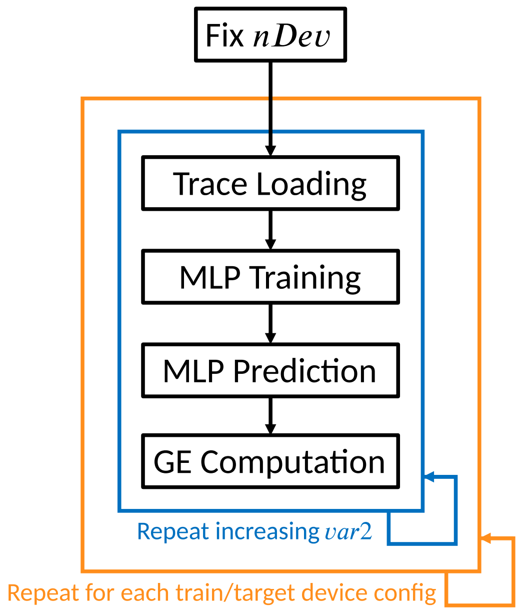


Figure 2.6: DKTA and DTTA Block Diagram. The diagram shows the steps that both analyses implement. Variable *var2* represents the number of keys for DKTA and the number of traces for DTTA.

The final result is a curve which indicates, for every amount of traces, the relative average rank of the correct key-byte among the *final prediction*.
Given two attacks, the one that reaches the lowest GE in the less number of traces is the best. In particular, this work considers a *0-based ranking*, meaning that the goal of an attack is to reach $GE = 0$.

2.3. Trade-off Analysis

This section is dedicated to explain the details of the trade-off analyses introduced by this work to completely characterize the Profiling Phase of DL-based SCAs, the Device-Key Trade-off Analysis (DKTA) and the Device-Trace Trade-off Analysis (DTTA).

The block diagram in Figure 2.6 shows how both DKTA and DTTA work.
In the figure, *var2* refers to either the number of keys (for DKTA) or to the number of traces (for DTTA), while *nDev* refers to the number of train-devices, common in both approaches.
The block diagram explains how DKTA and DTTA allow to completely characterize the Profiling Phase of DL-based SCAs: they allow to track the performance of several attacks, mounted with all possible train-configurations available with the considered setup, making it possible to highlight the trade-off, between the three most important variables, that

leads to the best performance.

The following comments better explain most crucial aspects of DKTA and DTTA, always referring to Figure 2.6.

All train-device(s)/target-device combinations are considered. Both DKTA and DTTA start fixing the number of train-devices, *nDev*. In this way, all the possible train-device(s)/target-device configurations are fixed as well. For example, if the available devices are denoted as A, B and C and the analysis is performed fixing $nDev = 1$, then the possible scenarios are $A$vs$B$, $A$vs$C$, $B$vs$A$, $B$vs$C$, $C$vs$A$, $C$vs$B$, which are the *combinations of two devices (train + target) from set {A, B, C}*. Similarly, when $nDevs = 2$, the possible scenarios are $AB$vs$C$, $AC$vs$B$ and $BC$vs$A$. This peculiar notation identifies an attack, specifying the devices used for training on the left of the *vs* symbol, and the target device (always one) on its right, and it will be extensively used in the next chapters to refer to specific attacks.

Number of keys and number of traces are constantly increased throughout the process. Once the train-device(s)/target-device configurations are fixed, multiple attacks are performed exploiting them. In particular, within the same attack-scenario, multiple attacks are performed, constantly increasing the value of *var2*, being number of keys, for DKTA, or number of traces, for DTTA. For example, fixed the attack-scenario $A$vs$B$ and the number of keys as second variable, the analysis proceeds considering first the attack $A$vs$B$ with 1 key, then $A$vs$B$ with 2 keys, and so on until the max number of keys is reached. At that point, the attack-scenario is changed and the experiments repeated. The only devices that are affected by the change of the second variable are the training ones, being the target fixed for all the experiments. In addition, the keys used with the target-device is never present in the training traces, neither in DKTA, nor in DTTA.

The considered results are averages over all experiments. Since all possible attack-scenarios that can be generated with the available setup and a fixed number of devices are tested, DKTA and DTTA need to average their results. In particular, the results are averaged only with respect to the attack-scenarios, meaning that the differences between attacks with different values *var2* (number of keys or number of traces) are still intact. This approach allows to have an unbiased outcome, useful when multiple scenarios must be compared.
The only downside is that the average is a sort of *high-level metric*, which sometimes can be misleading. For example, the dependency of the attack on a specific train-configuration is almost impossible to spot without considering all possible experiments separately.

The mounted DL-based SCAs are realistic and independent The adopted approach of considering all possible attack-scenarios with the available setup and using a dedicated key for the target-device makes the DL-based SCAs of both DKTA and DTTA *realistic*, since the train-configuration is always different from the attack one. At the same time, they are made *independent* by the fact that the considered DL network is re-built and re-trained every time. These precautions allow DKTA and DTTA to better emulate what an attacker would do in practice in a real-world scenario.

3 | Experimental Setup

This chapter is dedicated to the description of the experimental setup considered during the research.
A *Capturing Setup* is dedicated to the collection of the traces, performed with an oscilloscope that is able to measure the power consumption of the target-device.
A *Deep Learning Setup*, on the other hand, is dedicated to the execution of Deep Learning scripts, providing all the necessary computing power.

An additional section dedicated to the validation of the Hyperparameter Tuning technique implemented in this work is included. Indeed, this procedure allows also to validate the entire setup, from the acquisition of the development of the networks.

All experiments were executed in a laboratory, which is a controlled environment. In particular, both temperature and humidity were constant during the tests.

3.1. Capturing Setup

The Capturing Setup includes all the software and hardware tools that make the collection of traces possible.

Figure 3.1 presents a model of the Capturing Setup, showing how it is structured. The instrumentation involves an *acquisition machine*, an *oscilloscope* and the *Device Under Test (DUT)*.
The acquisition machine is connected with both the DUT and the oscilloscope. A connection is present also between oscilloscope and DUT. The collection process is controlled by the acquisition machine, which first sets up the oscilloscope providing all the needed information to collect the desired traces, and then starts the encryption processes on the DUT. While the DUT is encrypting, the oscilloscope measures its power consumption. At the end of the encryption processes, the oscilloscope transmits the collected power traces to the acquisition machine, which stores the results.

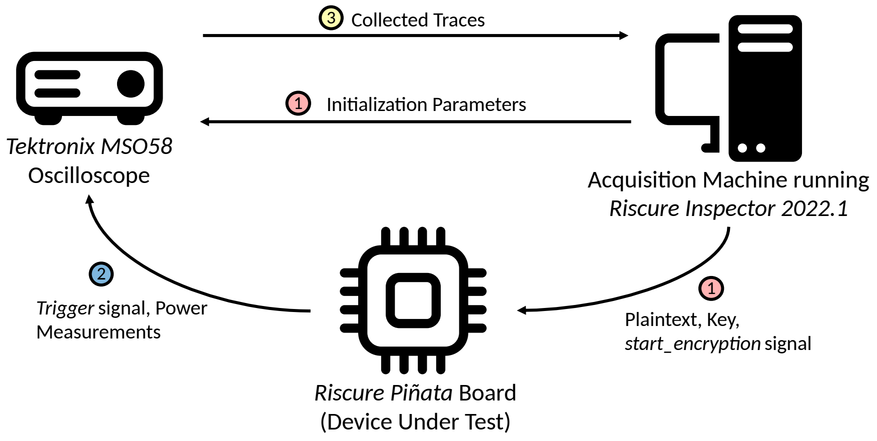

3 Collected Traces
1 Initialization Parameters
Tektronix MSO58 Oscilloscope
Acquisition Machine running Riscure Inspector 2022.1
2 Trigger signal, Power Measurements
1 Plaintext, Key, start_encryption signal
Riscure Piñata Board (Device Under Test)

Figure 3.1: Model of the Capturing Setup. The *acquisition machine* is connected with both the *oscilloscope* and the Device Under Test. The acquisition machine initializes the oscilloscope to start the capture and then begins the encryption process sending plaintext, key and *start-encryption* signal to the Device Under Test. The Device Under Test starts encrypting and the oscilloscope measures its power consumption, aligning the traces thanks to the trigger signal controlled by the Device Under Test. Once the traces are collected, they are transmitted to the acquisition machine, which stores them. The icons of this picture are taken from the open-source project Bootstrap [48].

3.1.1. Acquisition Machine

More in detail, the acquisition machine is a workstation running Windows 10. It features an Intel Xeon W-2125 CPU @ 4GHz and 512GB of RAM.
It runs the *Riscure Inspector* application [51], version 2022.1, which allows the user to have full control over the trace-collection process.

Through *Inspector*, the oscilloscope is initialized with the number of traces to collect, their length and the sampling frequency. In addition, the desired encryption algorithm, plaintext, secret key and *start-encryption signal* are provided to the DUT. While the acquisition machine and oscilloscope exchange data through a network cable, the communication between the acquisition machine and the DUT exploits the Universal Asynchronous Receiver-Transmitter (UART) protocol. In particular, a Universal Serial Bus (USB) port of the acquisition machine is dedicated to send data to and receive data from the UART pins of the DUT. A $USB \rightarrow UART$ adapter cable enables the communication.

Inspector generates `.trs` files as output: this format allows to save not only the raw measurements, but also some metadata, such as plaintext and key. In addition, the *Inspector* allows the user to customize its internal modules to solve more specific problems, exploiting the `Java` programming language.

In this work, three new modules have been introduced:

- `custom_changeKey.java`: allows to change the secret key stored in the DUT with a user-defined value, saving it in an external file;
- `custom_encrypt.java`: forces the DUT to consider the key from an external file while encrypting with unprotected-AES. The key is also stored as metadata in the output `.trs` file;
- `custom_maskedEncrypt.java`: forces the DUT to consider the key from an external file while encrypting with masked-AES. The key is also stored as metadata to the output `.trs` file;

Inspector offers also built-in modules which perform all sorts of SCAs, from the traditional ones like Differential Power Analysis and Template Attacks, to the most recent one involving Deep Learning. These modules are useful when different setups need to be tested.
Moreover, it allows to preprocess the collected traces. For example, it is possible to re-sample them to a different frequency or trim them in order to consider specific sub-traces.

3.1.2. Oscilloscope

Concerning the oscilloscope, it is a *Tektronix MSO58*, with 1GHz as input bandwidth, 6.25GS/s as maximum sample rate and a 12-bit Analog-to-Digital Converter (ADC).

It measures the power consumption of the DUT through a current probe. The probe is used *as* a resistor: it is connected on one side to the DUT, and on the other to the power supply, making it possible to compute the value of the current that flows through it.
In practice, the current probe does *not* involve resistors, as they may lead to incoherent results, depending on the intensity of the current. Indeed, inductors are preferred, since they lead to consistent measurements both when the current intensity is low and when it is high.

Moreover, the oscilloscope is able to provide aligned traces. In order to do so, it exploits a probe, called *trigger probe*, to constantly track the activity of a specific General-Purpose Input/Output (GPIO) pin of the DUT, known as *trigger pin*. The signal present at the *trigger pin* is controlled by the DUT, which can raise it or lower it. Usually, the trigger signal is raised right before the considered encryption operation, and lowered right after it, allowing to individuate such operation among all the measurements.

Another useful feature of the considered oscilloscope is the so-called *Fast-Frame Mode*: the

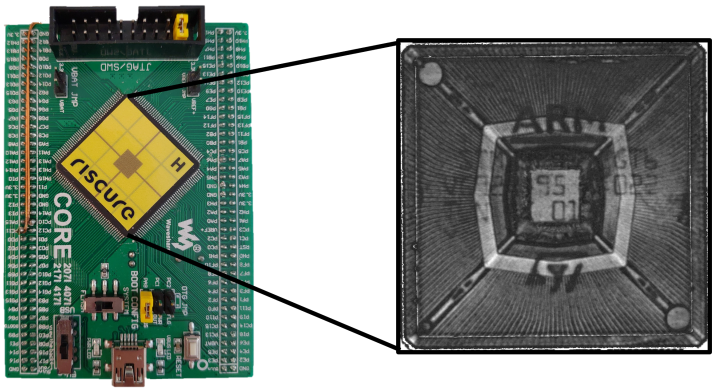


Figure 3.2: Riscure Piñata Board with CSAM. The figure shows a Piñata board, on the left, and a Confocal Scanning Acoustic Microscopy (CSAM) of its CPU, on the right.

user can control how many traces are collected before each transmission to the acquisition machine. By default, this quantity is set to 1, meaning that each single trace is collected by the oscilloscope and immediately forwarded to the acquisition machine. A value of N allows to communicate with the acquisition machine only once N traces have been collected. Considering a value $N > 1$ speeds up the process, because it reduces the number of times data is transferred to the acquisition machine. On the other hand, however, sometimes a too high value of N can lead to data loss. This work considers Fast-Frame Mode with $N = 1000$, the maximum value possible for the configuration *Inspector-Tektronix MSO58*.

3.1.3. Device Under Test

Regarding the Device Under Test, the target of the attacks, it is an STM32F4 with an ARM Cortex-M4 CPU. It features a 32-bit architecture and 168MHz clock-frequency. More in detail, it is a *Riscure Piñata* [52], version 2.3.1, and it is produced by the same company which develops *Inspector*.
It is depicted in Figure 3.2, alongside a Confocal Scanning Acoustic Microscopy (CSAM) of its CPU. The CSAM, performed with a Sanoscan Gen6, is a non-destructive method that uses ultrasonic waves to visualize microscopic structures. Indeed, it is possible to clearly distinguish the die of the CPU, which appears like a small square in the middle of the picture, and all the bounding wires, which connect the die to the outside.

In total, this work considers 3 *Piñata* boards, denoted by $D1, D2, D3$: they are shown in Figure 3.3.

The peculiarity of *Piñata* boards is that they are intentionally modified by their manufacturer in order to be targeted by Side-Channel Attacks. Indeed, using it in cooperation

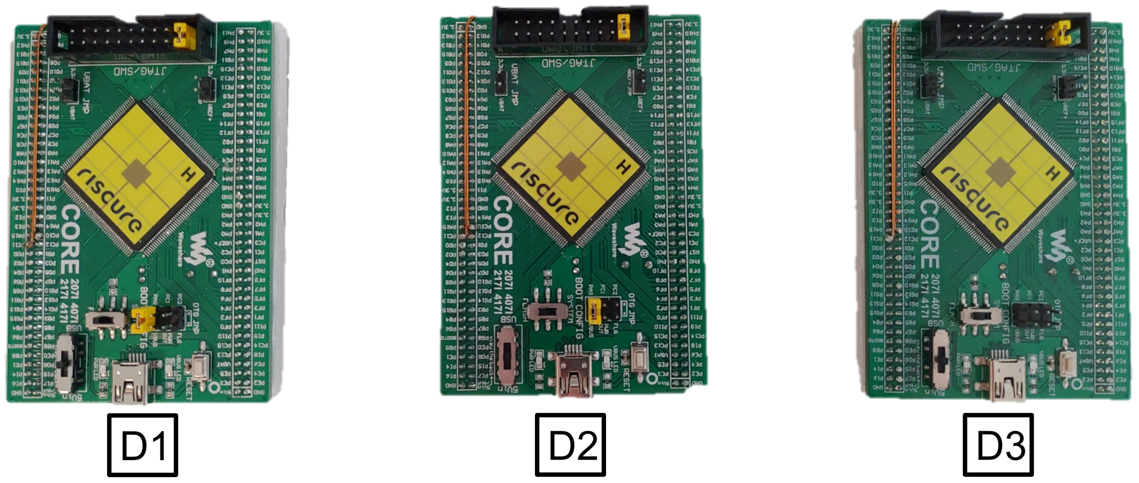


Figure 3.3: Devices considered during the Experiments. The figure shows the three samples of Riscure Piñata, produced by the same manufacturer, which were used during the experiments.

with *Inspector*, it is possible to perform a variety of different attacks, considering either power consumption traces or electro-magnetic ones. In addition, it is possible to attack both software and hardware implementations of AES, with and without countermeasures, besides other encryption algorithms. This work only focuses on software implementations of AES-128, both unprotected and masked.

Regarding AES, by default the *Piñata* firmware sets the trigger signal *around* the whole algorithm: the trigger is raised right before starting the encryption, and it is lowered after the ciphertext is generated. This means that, by default, the traces produced by a *Piñata* comprehend a complete encryption, formed by all the rounds of AES.
This work considers specific operations related either to the first round only, or to the masks. For this reason, the firmware, which is written in `C` language, is customized changing the positions of the trigger signals. In particular, for unprotected-AES, the trigger is set to go *up* right before the *SubBytes* operation of the first round, while it is set to go *down* right after it is completed. For masked-AES, the trigger is raised during mask handling before the first round and it is lowered right before the *MixColumns* operation of the first round.
Every modification of the firmware is followed by the so-called *flashing* of the board: the existing firmware inside the flash memory of the device is overwritten with the customized version. In order to perform this operation, this work uses the `dfu-util` [61].

Figure 3.4 shows the *Piñata* while it is connected to both the acquisition machine and the oscilloscope. In the upper-left of the picture, it is possible to see the end of the $USB \rightarrow UART$ adapter (orange, yellow and black cables) which connects the acquisition machine to the board, while in the upper-right the terminals of the current probe can

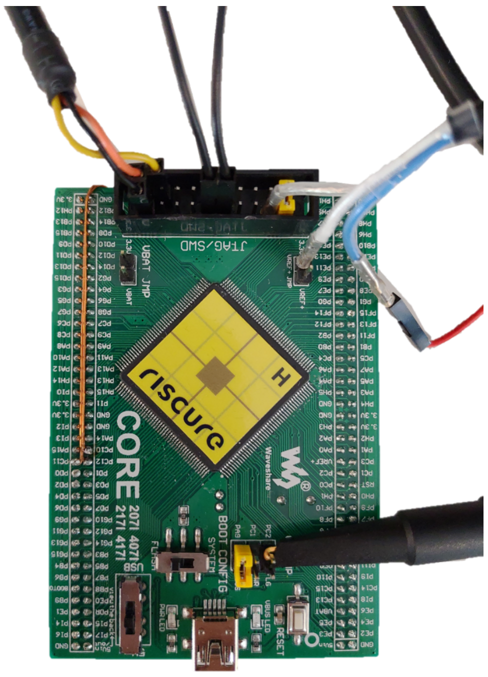


Figure 3.4: Piñata Under Test. This figure shows a Piñata while it is connected to the acquisition machine and the oscilloscope. The final section of the $USB \rightarrow UART$ adapter is visible in the upper-left of the picture, while the terminals of the current probe can spotted in the upper-right. In the lower-right there is the trigger probe, connected to the trigger pin of the Piñata. Ground cables can be seen in the middle of the upper part of the figure.

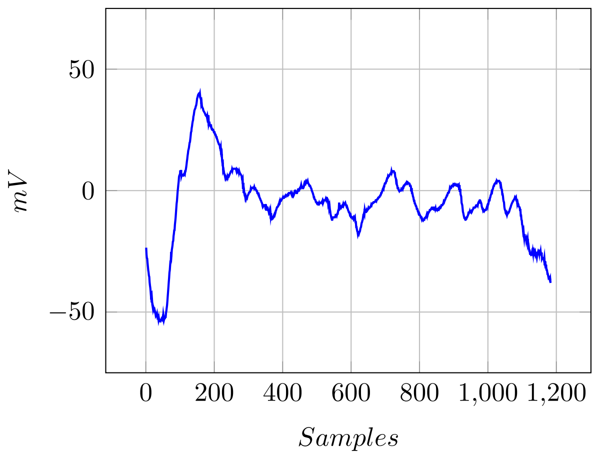
be spotted (white, blue and grey cables). In the lower-right, instead, the trigger probe (black tool) is clearly visible while it is connected to the trigger pin of the *Piñata*. The two black wires visible in the middle of the upper part of the image are ground cables.

3.1.4. Details about Trace Collection

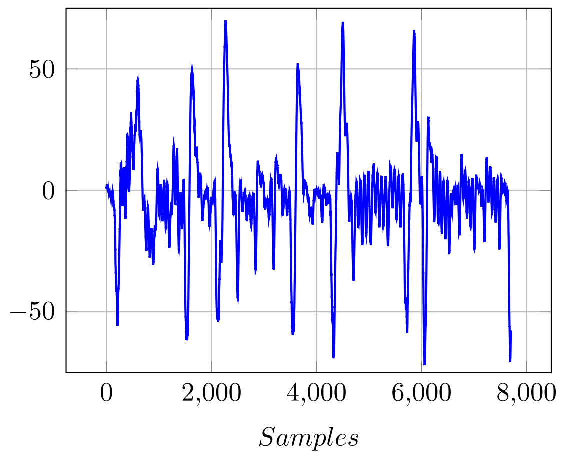
This research considers 3 devices, $D1, D2, D3$, and 11 keys, $K0, K1, ..., K10$. Multiple trace-sets are collected, one for each device/key configuration (Di, Kj), with $i \in \{1, 2, 3\}$ and $j \in \{0, 1, ..., 10\}$. In total, 66 trace-sets are considered, 33 per targeted algorithm (AES with and without countermeasures). Each trace-set features 50,000 traces and can be generated in about 30 minutes.

The 11 keys are randomly-generated with a `Python` script. It generates a 128-bit hexadecimal value for 11 times, saving the result in a `JSON` file. The keys were produced before any experiments, and have been kept unchanged for the entire work.
$K0$ is only used as target-key: it is never used in the training configurations.

The plaintext is randomly-generated before every single encryption. The underlying idea is that, by attacking the *SubBytes* operation and collecting a considerable amount of traces, it is possible to observe the leakage of each possible outcome of the SBox-lookup.



(a) Unprotected-AES Trace



(b) Masked-AES Trace

Figure 3.5: Sample Power Consumption Traces. Subfigure (a), on the left, shows a trace collected during an encryption with unprotected-AES: it refers to the execution of the first round *SubBytes* operation only. Subfigure (b), on the right, presents a trace collected during an encryption with masked-AES: it refers to several computations, some relative to operations over the mask, others relative to the first round of AES, such as *AddRoundKey*, *SubBytes*, *ShiftRows*. Both traces are captured from device $D1$, considering key $K0$.

The traces are captured with a frequency of 500MHz by the oscilloscope and then manually re-sampled at 168MHz using *Inspector*.
The initial sampling frequency of 500MHz satisfies the condition imposed by the Nyquist-Shannon Sampling Theorem [58] to avoid information-loss while converting a continuous signal into a discrete sequence of samples, being at least 2× the target frequency of 168MHz, the clock-frequency of the DUT. Re-sampling, on the other hand, aims at reducing the number of samples per trace minimizing the information-loss. Indeed, too-long traces would either exceed the physical memory limit or exponentially increase the training time of the MLP.

Figure 3.5 shows samples of collected power consumption traces from the configuration $(D1, K0)$. Figure 3.5a presents a trace acquired during an encryption with unprotected-AES. On the other hand, Figure 3.5b shows a trace acquired during an encryption with masked-AES. It is clearly visible that the unprotected trace is shorter than the masked one: specifically, an unprotected trace is formed by 1183 samples, while the masked one by 7700. This happens because the masked traces refer to more operations with respect to the unprotected ones: in order to attack a masked implementation, one needs to exploit the leakage of multiple operations, specifically the ones involving the masks and the masked value [44, 57].
More in detail, each major peak shown in Figure 3.5b coincides with the start of a different operation. In order, from left to right, those operations are:

- *MixColumns* applied to the mask: useful to keep the AES-state protected;

- Key-reshaping: the key is reshaped from 1×16 to 4×4 to match the AES-state;
- Plaintext masking: application of the mask to the plaintext;
- Initial *AddRoundKey*;
- First round *SubBytes*;
- First round *ShiftRows* with an additional operation over the mask to maintain consistency during the rounds.

These operations have been precisely mapped through several experiments, each one considering the trigger *around* a specific operation.

3.2. Deep Learning Setup

Deep Learning models need an high computational power in order to be trained. For this reason, a dedicated desktop computer is considered. It runs Ubuntu 20.04 LTS and features an Intel Core i7-10700K CPU @ 3.8GHz, 66GB of RAM and an Nvidia GeForce RTX 2080 Ti GPU, with 11GB of dedicated GDDR6 memory.

This machine runs also a JupyterLab Server instance, which can be accessed remotely. JupyterLab has been used as test environment, as it offers the possibility of running independent code cells. Mainly, it has been exploited to validate multiple network architectures over the available data. Initially, also some manual Hyperparameter Tuning has been performed, to validate the setup and the feasibility of the proposed solution.

Since Device-Key Trade-off Analysis and Device-Trace Trade-off Analysis both take quite some time, it was not possible to run them on JupyterLab, since it is a web application and needs constant access to Internet to work. Therefore, the main scripts of this work have been run directly on the Linux machine.

All DL networks have been trained within a dedicated `Python` virtual environment, which contains all the required libraries. In particular, the considered Deep Learning libraries are:

- `Tensorflow 2.8.0` [1]: end-to-end platform for Machine Learning;
- `Keras 2.8.0` [17]: simple API to develop Deep Learning networks.

In particular, `Keras` is used as interface mounted on top of `Tensorflow`, to simplify the development of DL models.

3.3. Validation of the Hyperparameter Tuning Process

This section discusses the effectiveness of the adopted Genetic Algorithm for Hyperparameter Tuning described in Section 2.2 when describing the Profiling Phase of a DL-based SCA. In particular, a comparison between it and the exhaustive-search used by the author of [7] is shown.
Moreover, this section allows to validate the entire experimental setup, from the collection of the traces to the training and testing of the networks.

A Genetic Algorithm is particularly suitable for Hyperparameter Tuning because it allows to start from random hyperparameters and gradually improve them, eventually finding the optimal ones.
In order to reach this goal, the procedure must be initialized with multiple parameters, as shown in Algorithm 1. Each parameter influences the performance of the hyperparameter-search as follows:

- *hpSpace*: defines the possible values that the hyperparameters can take;
- *popSize*, *scProb*, *mProb*: control the *exploration* level of the algorithm, allowing it to range within the hyperparameters space;
- *nGen*: is considered the *stopping criterion* for the algorithm;
- *selPerc*: controls the *exploitation* level of the algorithm, making it focus on the best configurations.

Moreover, *popSize* and *nGen* influence also the execution time of the search. Indeed, increasing the population size leads to train and validate more networks each iteration, while increasing the number of generations means increasing the number of iterations, repeating the algorithm several times.

Table 3.1 presents the value of the Genetic Algorithm parameters chosen in this work. The table shows that the considered parameters slightly change from attacking unprotected-AES to attacking masked-AES: the only difference is that the second-chance probability is decreased from 20% to 10%. This choice has been made to avoid algorithm divergence from consistent hyperparameter configurations, which was visible with higher values of *scProb*.

Appendix A reports the value of all the hyperparameters obtained with the Genetic Algorithm. Indeed, it is computed every time the targeted algorithm, the number of devices

Genetic Algorithm Parameters

Parameter	Value for Unprotected-AES	Value for Masked-AES
popSize	15	15
nGen	20	20
selPerc	30%	30%
scProb	**20%**	**10%**
mProb	20%	20%

Table 3.1: Genetic Algorithm Parameters. This table provides the chosen value of each input-parameter to the Genetic Algorithm used for Hyperparmeter Tuning. The parameters used both while targeting unprotected-AES and masked-AES are reported. The only difference is the value of *scProb*, decreased when attacking masked-AES to ensure algorithm convergence.

or the targeted byte is changed.

The Hyperparameter Tuning technique of this work is totally different from the one considered by the authors of the Multiple Device Model [7]: they exploited an exhaustive search over a smaller hyperparameter space.
More in detail, their hyperparameter space was formed by only two variables, being the number of hidden layers $l \in \{1, 2, 3, 4\}$ and the number of hidden neurons $n \in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$. All the other hyperparameters were fixed and not tuned: the considered Gradient Descent Technique was *RMSprop*, the Learning Rate was set to 0.001 and the Batch Size was fixed at 256. Due to the presence of a small hyperparameter space and some fixed values, in the following this approach is referred to as *simplified Exhaustive Search (sES)*.
In addition, regularization techniques are not mentioned in the Hyperparameter Tuning of [7]. On the other hand, the new proposed solution considers *Dropout*, *BatchNormalization* and *L-2 Regularization* during training, alongside a learning rate reduction when the network starts diverging and *Early Stopping*.

The proposed Hyperparameter Tuning approach with a Genetic Algorithm is surely slower than the one used in [7], but, at the same time, it offers more flexibility by avoiding fixed values and by considering a bigger hyperparameter space. However, the consideration of a small hyperparameter space is a forced choice when the tuning algorithm is an exhaustive-search and, as shown in [7], in some cases it can still provide excellent results.

In order to compare the approaches, the following experiment has been performed targeting unprotected-AES:

1. Perform Hyperparameter Tuning with the Genetic Algorithm;
2. Perform Hyperparameter Tuning with the exhaustive-search approach descibed in [7], considering also the same fixed hyperparameters;
3. Train two networks, one with the hyperparametrs from point 1., the other with hyperparameters from point 2., on the same train-set;
4. Attack with both networks the same target-device, which is not used during training;
5. Repeat adding a device to the hyperparameter tuning and to the training phase of the attacks (max 2 devices).

For simplicity, referring to the notation introduced in Section 2.3, the procedure involve only attack $D1$vs$D3$ and attack $D1D2$vs$D3$, without averaging the results of multiple experiments. Moreover, the target of the attacks is the 6^{th} byte, randomly-chosen.

Table 3.2 shows the resulting hyperparameters. Table 3.1a provides the results obtained considering one train-device, while Table 3.1b presents the ones relative to two train-devices. The **highlighted** values are fixed and not tuned. The same configuration of layers and neurons is considered by the Genetic Algorithm (GA) in both scenarios. The same happens with the semplified Exhaustive Search (sES). The networks resulting from GA are more complex than the ones resulting from sES, since the former feature 4 hidden layers of 500 neurons, and the latter only 1 hidden layer of 100 neurons. The deep network structures resulting from the application of GA present also regularization, both in terms of L2-Regularization and Dropout in the case of only one device, and in terms of only Dropout in case of two devices: this technique allows to avoid overfitting even if the structure of the networks is quite complex. *Gradient Descent technique* is always different between GA and sES, while *learning rate* and *Batch Size* differ only when two train-devices are considered.

Figure 3.6 shows the training *history* of the best networks of the two approaches. The *train history* refers to the trend of the *training error* and the *validation error* during the training of a network. In this case, the tracked network is the one generated with the best hyperparameters of each approach.
A robust and general-enough network exhibits a validation error that follows the trend of the training one, which decreases. In addition, the lower the validation error is, the better the network performs in classifying new, unseen traces.
Figure 3.6a and Figure 3.6b show the training histories produced by the best hyperparameters found with GA, while Figure 3.6c and Figure 3.6d show the training histories

Resulting Hyperparameters

(a) 1 Train-Device

Hyperparameter	Results from GA	Results from sES
Number of Hidden Layers	4	1
Number of Hidden Neurons	500	100
Dropout Rate	0.2	–
L2-Regularization Coefficient	0.001	–
Gradient Descent Technique	*adam*	***RMSprop***
Learning Rate	0.001	**0.001**
Batch Size	256	**256**

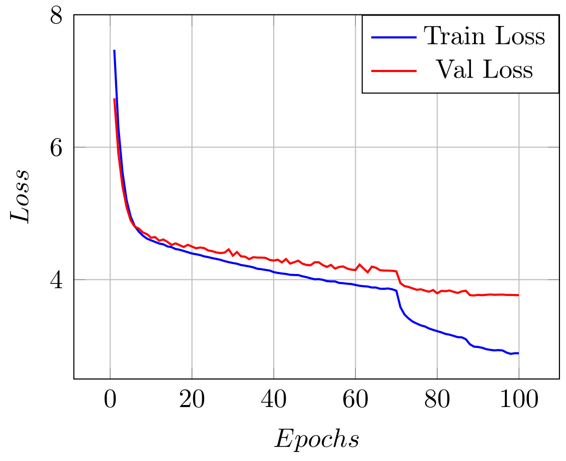
(b) 2 Train-Devices

Hyperparameter	Results from GA	Results from sES
Number of Hidden Layers	4	1
Number of Hidden Neurons	500	100
Dropout Rate	0.4	–
L2-Regularization Coefficient	0.0	–
Gradient Descent Technique	*adam*	***RMSprop***
Learning Rate	0.005	**0.001**
Batch Size	128	**256**

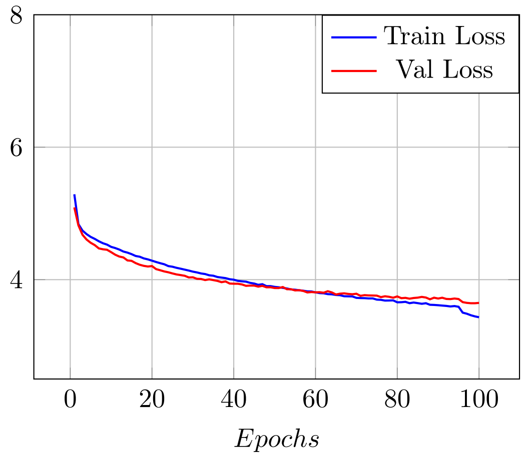
Table 3.2: Resulting Hyperparameters, Genetic Algorithm (GA) and simplified Exhausive Search (sES). Table (a), up, shows the resulting hyperparameters when considering only one train-device. Table (b), down, shows the resulting hyperparameters when considering two train-devices.

referred to sES. It is visible that the validation loss related to the hyperparameters found with GA follows the respective training losses, while the ones relative to sES diverge. This divergent trend can indicate the presence of overfitting, since the network seems to misclassify unknown traces.
Overall, the loss achieved by GA is lower than the one achieved by sES.
The number of epochs is different between GA and sES: this is because GA considers 100 as maximum number of epochs, while sES only 50. This value has been taken from [7], like the other fixed values of the sES approach.

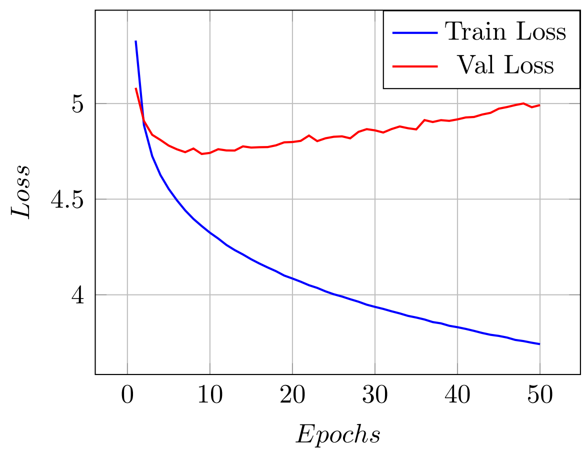
Figure 3.7 shows the results of the attacks performed with the selected hyperparameters. The plots show the Guessing Entropy (GE) of the attacks for an increasing number of attack-traces: the attack that leads to $GE = 0$ in fewer traces is considered the best.



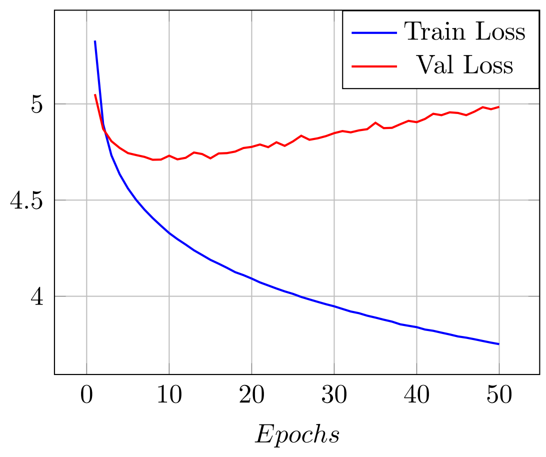
(a) GA Approach, 1 Train-Device



(b) GA Approach, 2 Train-Devices



(c) sES Approach, 1 Train-Device



(d) sES Approach, 2 Train-Devices

Figure 3.6: Training History of the Best-Performing Hyperparameters with both Genetic Algorithm and simplified Exhaustive Search Approaches. Subfigure (a), on the upper left, shows the training and validation loss trends of the best-performing network obtained through Hyperparameter Tuning with Genetic Algorithm and only one train-device. Figure (b), on the upper right, shows the same quantities, but obtained with two train-devices. Subfigure (c), on the lower left, presents both best training loss and best validation loss obtained through Hyperparameter Tuning with semplified Exhaustive Search and only one train-device. Figure (d), on the lower right, tracks the trend of the same variables, but obtained with two train-devices.

Moreover, the plots show the point in which $GE \leq 0.5$, marking it with a dashed vertical line: it is a good approximation for $GE = 0$, so it can be considered as a metric here.

Figure 3.7a shows the results of the attack $D1$vs$D3$: the attack performed with hyperparameters found with the Genetic Algorithm is able to recover the 6^{th} byte in 9 attack-traces, while the one which considers hyperparameters found with the simplified Exhaustive Search is successful in 12.

Figure 3.7b shows the results when considering two train-devices, in particular the scenario $D1D2$vs$D3$: also in this case, GA leads to the best attack, reaching $GE \leq 0.5$ in 7 attack-traces, while sES needs 11 of them.

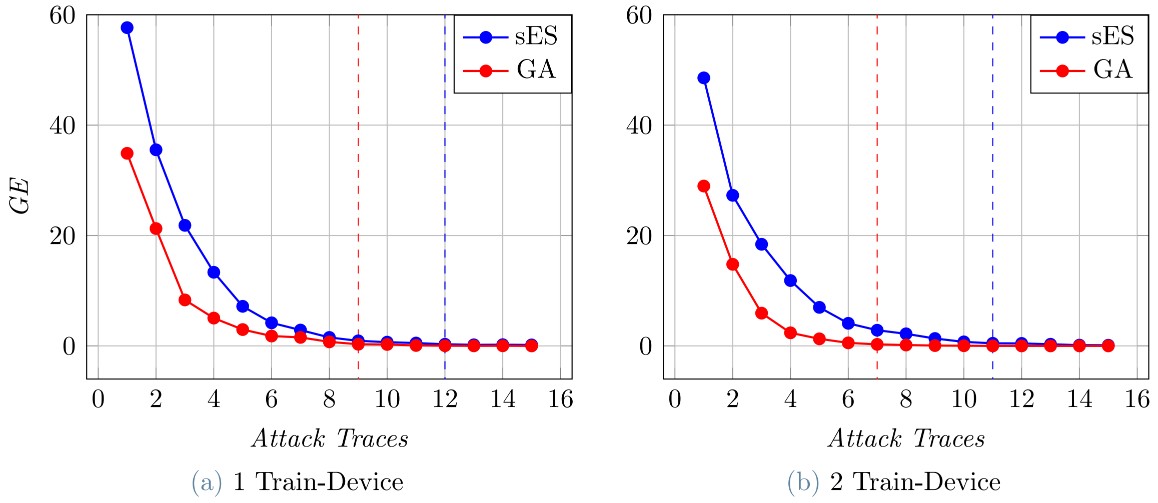


Figure 3.7: Attack Comparison between Genetic Algorithm and simplified Exhaustive Search. The attacks are evaluated in terms of Guessing Entropy over increasing number of attack-traces. Subfigure (a), on the left, shows the results achieved with only 1 train-device. Subfigure (b), on the right, presents the results achieved with two train-devices. The dashed lines indicate the point in which $GE \leq 0.5$, good approximation for $GE = 0$. The Genetic Algorithm allows to reach $GE = 0$ with fewer attack-traces in both cases.

In order to search for the best hyperparameters and therefore perform the proposed attacks, the two considered algorithms took the following times:

- *simplified Exhaustive Search*: about 15 minutes;
- *Genetic Algorithm*: about 7 hours.

Despite taking much more time, the Genetic Algorithm outperformed the simplified Exhaustive Search in both scenarios.
Since this work aims at proposing efficient training configurations to mount effective DL-based SCAs, the Genetic Algorithm is preferred because it leads to the best performance. Therefore, it is used also as Hyperparameter Tuning method to search the best hyperparameters when targeting masked-AES.

4 | Experimental Results

This chapter presents the results obtained throughout this work targeting AES-128 with and without masked countermeasure.
In addition to the outcomes of DKTA and DTTA, all information needed to reproduce the results are provided.

4.1. Results Targeting Unprotected-AES

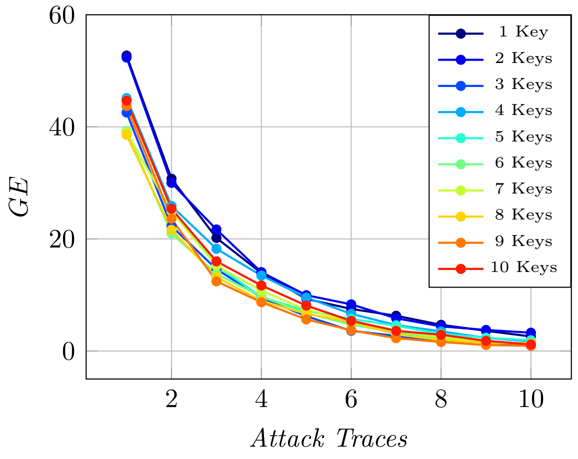
This section shows the results obtained performing Device-Key Trade-off Analysis (DKTA) and Device-Trace Trade-off Analysis (DTTA) against an unprotected software implementation of AES-128. Since DKTA was executed first, its results were considered and exploited while performing DTTA.

All the hyperparameters considered for the experiments are visible in Table A.1 of Appendix A. As pointed out in Section 3.3, the considered networks are quite complex, presenting multiple hidden layers with many neurons, but they do not overfit thanks to regularization, present in terms of Dropout, BatchNormalization and L2-Regularization.

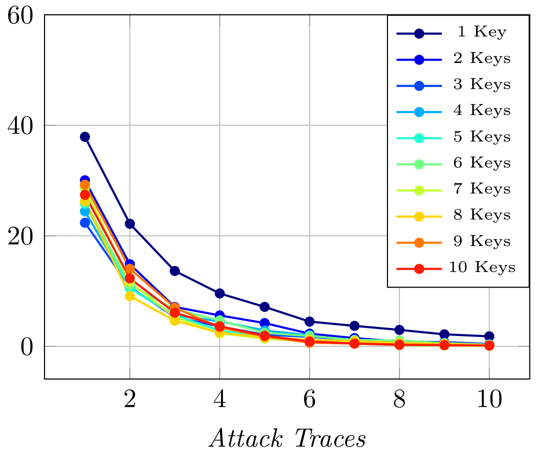
4.1.1. DKTA Results

For simplicity, DKTA was performed targeting byte 0, 5, 11 and 14 only (0-based numbering). Indeed, since a single DKTA alone involves multiple attacks, the time needed to repeat it 16 times in order to target all bytes of the key would have been too long.
The considered bytes were chosen with the constraint of covering all columns and all rows of the AES-state. The following representation shows their position in the AES-state:

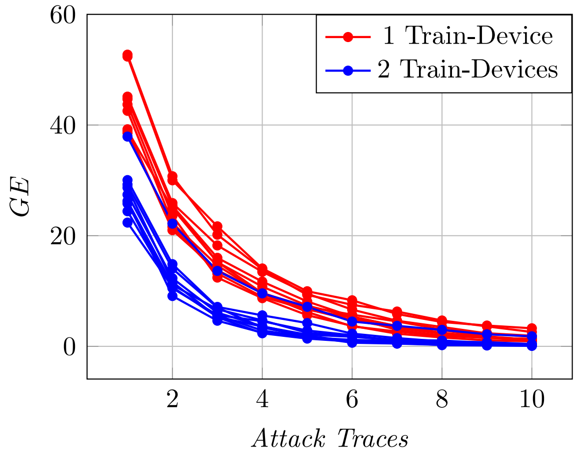
$$\begin{bmatrix} \mathbf{b_0} & b_4 & b_8 & b_{12} \\ b_1 & \mathbf{b_5} & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & \mathbf{b_{14}} \\ b_3 & b_7 & \mathbf{b_{11}} & b_{15} \end{bmatrix} \tag{4.1}$$



(a) Byte 0, Single Train-Device



(b) Byte 0, Two Train-Devices



(c) Byte 0, Comparison

Figure 4.1: DKTA Results Targeting Byte 0 against Unprotected-AES. Subfigure (a), in the upper-left, shows the DKTA results obtained with a single train-device. Subfigure (b), in the upper-right, displays the outcome of DKTA when two train-devices are considered. Subfigure (c), in the middle, overlaps the previous results, highlighting the importance of adding a device.

Each of the the following attacks was performed with a total of 50,000 train-traces, where 45,000 compose the training set and the remaining 5,000 form the validation set. GEs were instead computed as average over 100 experiments.

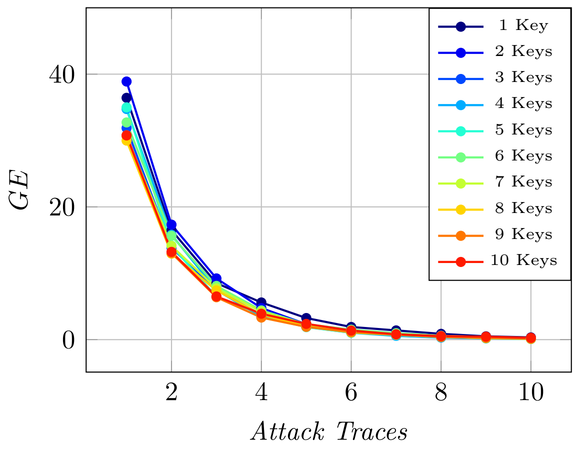
Results Targeting Byte 0

Figure 4.1 shows the DKTA results for byte 0 when considering up to 10 attack-traces. Here and in the following results, 10 is considered as threshold for a very effective attack, since $GE = 0$ in 10 or fewer traces means that the attacker, on average, needs to collect at most 10 traces to directly obtain the correct key-byte from the network, and the time taken to complete this operation is very short (assuming that the attacker is able to

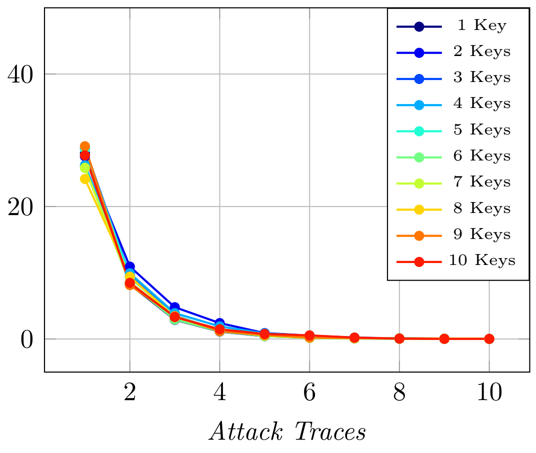
physically connect to the target device).
Figure 4.1a focuses on the attacks with only one train-device: it is visible that GE does not reach 0 in 10 traces. Despite this, the number of keys seems to influence the performance of the attack: the values of the GE curves obtained considering at least 3 keys during the training phase are lower than the respective ones obtained with 1 or 2 keys, for any number of attack-traces.
On the other hand, Figure 4.1b shows the results obtained when two train-devices are considered. In this case, GE seems to go below 1 with 7 attack-traces, reaching 0 in 8. Moreover, the improvement given by the number of keys is even more visible than before: considering only a single key during the training phase does not allow to reach $GE = 0$ within 10 attack-traces, while the addition of a key makes it possible. Considering more than 2 keys seems to be worthless, since the performance does not improve anymore.
Regarding the number of devices, it highly influences the attacks, since GE reaches 0 only when 2 devices are considered. In general, the overall performance of the attacks improves when considering 2 devices. This can be seen comparing visually Figure 4.1a and Figure 4.1b or by observing Figure 4.1c, which overlaps the previously-discussed GEs: the values of the curves obtained with two train-devices are lower than the respective ones obtained with only a single train-device, for any number of attack-traces. The only outlier is the attack performed with two devices and only a single key: its behavior is more similar to the attacks performed with a single train-device, showing a GE that does not reach 0.
Summarizing, targeting byte 0, the performance of the attack increases with both the number of keys, though not monotonically, and the number of devices.

Results Targeting Byte 5

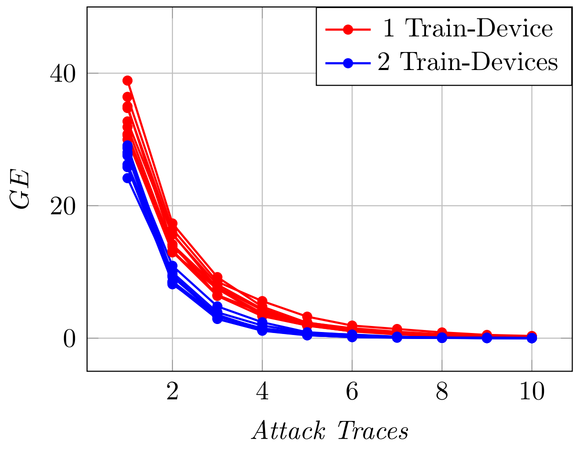
Figure 4.2 presents the results obtained targeting byte 5.
Here the influence of the number of keys over the attack performance is quite low, almost null. Indeed, it can be seen that the GE curves form a single bundle, both when considering one and two train-devices. To be precise, observing Figure 4.2a, when only a single train-device is considered, it can be seen that the performance slightly increases monotonically when more keys are used. This small difference in performance is visible only when using fewer than 6 attack-traces, then all curves overlap. In the end, all GEs reach 0 at the same time in 9 attack-traces, so the number of keys does not really impact the performance.
On the other hand, considering two train-devices as in Figure 4.2b, the curves appear almost overlapped from the beginning: the number of keys does not influence the attacks.
However, targeting byte 5 it is possible to reach $GE = 0$ within 10 attack-traces, even with a single train device. The addition of a second device improves the performance, lowering



(a) Byte 5, Single Train-Device



(b) Byte 5, Two Train-Devices

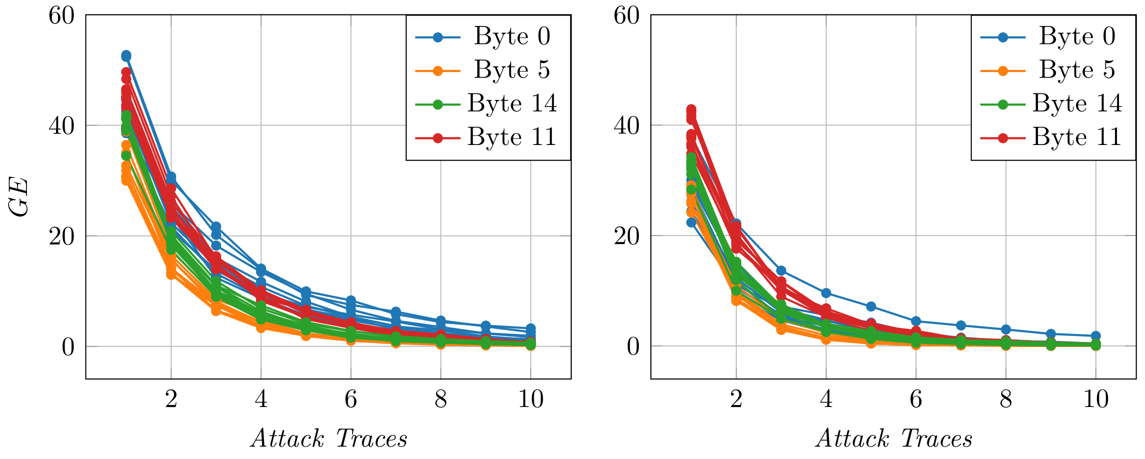


(c) Byte 5, Comparison

Figure 4.2: DKTA Results Targeting Byte 5 against Unprotected-AES. Subfigure (a), in the upper-left, shows the DKTA results obtained with a single train-device. Subfigure (b), in the upper-right, displays the outcome of DKTA when two train-devices are considered. Subfigure (c), in the middle, overlaps the previous results, highlighting the importance of adding a device.

the number of attack-traces needed to directly obtain the key-byte from 9, required with a single device, to only 6.

Also in this case, the performance improvement given by the addition of a device applies for any number of attack-traces. This can be seen from Figure 4.2c, where the results of Figure 4.2a and Figure 4.2b are overlapped: all the curves obtained with two devices feature values that are lower than the respective ones obtained with a single device.



(a) DKTA Comparison, Single Train-Device (b) DKTA Comparison, Two Train-Devices

Figure 4.3: DKTA Results Comparison targeting Uprotected-AES. The plots in this figure overlap all the obtained DKTA results, including the ones relative to byte 11 and 14. Subfigure (a), on the left, shows the comparison between byte 0, 5, 11 and 14 when only a single train-device is considered. Subfigure (b), on the right, displays the results of the attacks against the same bytes, but when two train-devices are used.

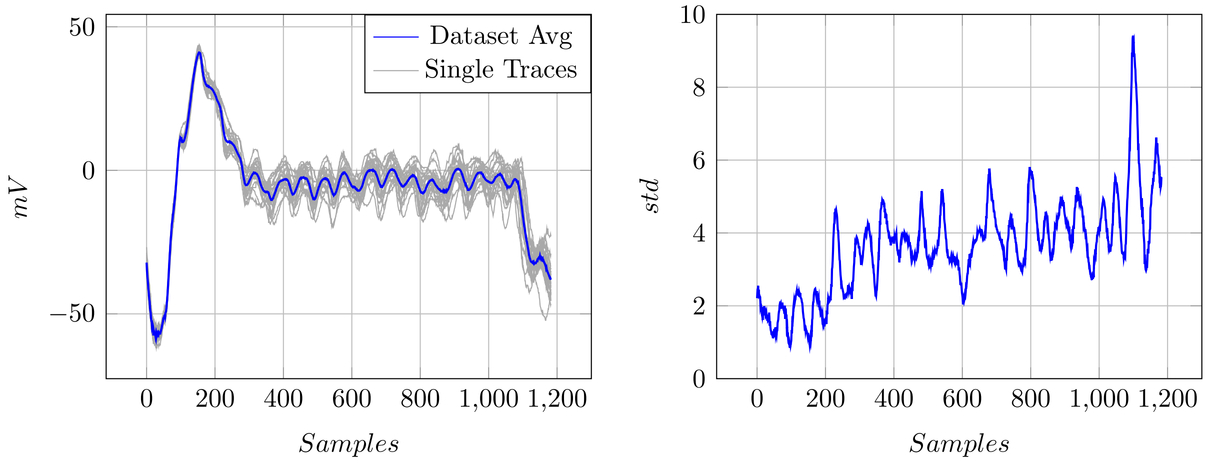
Results Comparison: Byte 0, Byte 5, Byte 11 and Byte 14

Figure 4.3 offers a compacted overview of all DKTA results. Indeed, in addition to the results obtained targeting byte 0 and byte 5, here also the ones relative to byte 11 and 14 are shown. Figure 4.3a shows the results obtained when a single train-device is considered, while Figure 4.3b displays what happens when two train-devices are used.

It can be seen that these figures do not distinguish the GE curves with respect to the number of keys used in the training phase: this choice has been made to highlight the difference in attack performance when targeting different bytes. Indeed, byte 11 and byte 14 *behave* almost like byte 5: the number of keys does not influence the attack, because once a certain number of attack traces is considered, the curves overlap. This is clearly visible because the attacks against byte 5, 11 and 14 exhibit a compact bound of GE curves, while byte 0, the only one affected by the number of keys, shows some outliers.

In Figure 4.3a, it can be seen that the attacks targeting byte 5 and 14 reach $GE = 0$ within 10 attack traces, while the ones targeting byte 0 and 11 do not, even if they are very close to it. This is not the case for Figure 4.3b, where, despite the only outlier generated by the usage of a single key attacking byte 0, all GE converge within 10 traces.

Figure 4.3 also highlights a pattern: the attack performance decreases passing from targeting byte 5 to targeting byte 14 and from targeting byte 14 to targeting byte 11. Indeed, the bundle of GE curves relative to byte 5 presents values that are lower than the ones related to byte 14 for any number of attack traces, and the same happens between byte 14 and byte 11. This behavior can be observed for any number of devices.



(a) Traces Partial Misalignment

(b) Samples Standard Deviation

Figure 4.4: Jitter Effect Visualization. Subfigure (a), on the left, shows the Jitter Effect as partial misalignment of the traces over a small dataset of 20 traces. Subfigure (b), on the right, shows the Jitter Effect quantifying the trace divergence in terms of samples standard deviation.

This performance-drop can be strictly related to the *jitter* phenomenon. In general, the *jitter* is defined as small deviation from the true-periodicity of a signal, in particular the clock one. The effect of this small deviation is the *partial* misalignment of the traces, since they appear aligned and overlapped right after the reception of the trigger signal, but then they start diverging.

Figure 4.4 allows to visualize the *jitter effect* over a small dataset of 20 traces. Figure 4.4a shows that at the beginning, for the first 100/150 samples, the traces appear aligned and almost overlapped, while towards the end, they are completely misaligned. The highlighted trace in the middle is the average of the dataset, and it is exploited to to visualize the divergence among the traces. Figure 4.4b quantifies the misalignment in terms of samples standard deviation: the *jitter effect* is confirmed by the fact that, starting from the beginning of the trace and reaching its end, the standard deviation increases monotonically.

This partial misalignment of the traces makes it harder to recognize the leakage patterns related to those quantities that are computed towards the end of the trace. Since different bytes are handled in different moments in time during the encryption, their computation can be affected by the *jitter* with various extents, making easier or harder their recovery through an SCA.

In particular, even if the AES-state stores the bytes in *column-major* order as shown in Section 4.1.1, the considered *SubBytes* operation scans it in *row-major* order with a `for loop`, proceeding row-by-row.

Therefore, focusing only on the targeted bytes and referring to the AES-state depicted in

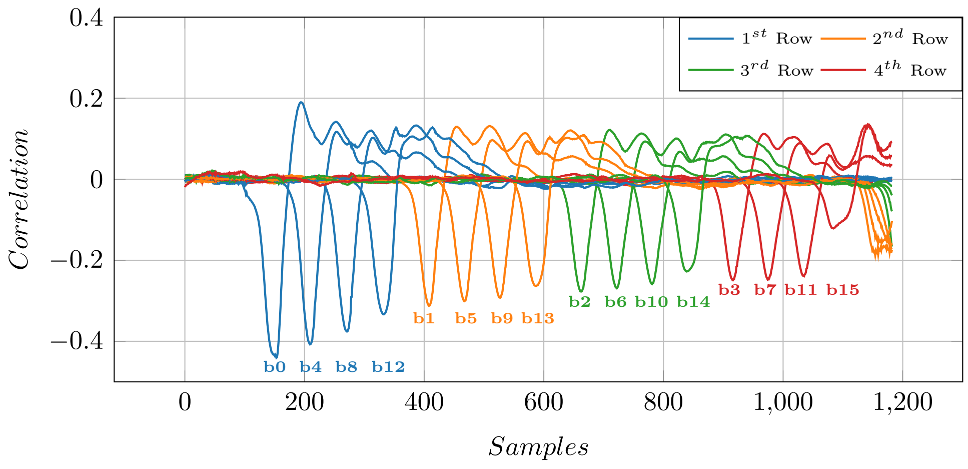


Figure 4.5: Pearson's Correlation Coefficient between Trace and *SBox-out*. This plot shows the intensity of the leakage related to each byte of the key, in terms of Pearson Correlation Coefficient between the trace and the relative *sbox-out*

Section 4.1.1, byte 0 is computed before byte 5, byte 5 is computed before byte 14 and byte 14 is computed before byte 11. This order of computation suggests that byte 0 is the least affected by the *jitter effect*, followed by byte 5, byte 14 and, last byte 11. As a consequence, the leakage related to byte 0 should be stronger than the one related to byte 5 and so on, following the previous order, as far as byte 11. Figure 4.5 confirms this hypothesis, showing how the leakage intensity, calculated as Pearson's Correlation Coefficient between a trace and the relative *sbox-out* [10], decreases almost monotonically while scanning the AES-state by row.
So, the performance drop visible passing from targeting byte 5 to targeting byte 14 and then to targeting byte 11 is likely to be caused by the *jitter effect*, which affects byte 11 more than byte 14, and byte 14 more than byte 5.

Following this reasoning, byte 0 should be the easiest to retrieve, and therefore provide the best-performing attacks. Surprisingly, observing Figure 4.3, this is *not* the case, since the attacks against byte 0 perform worse than all the others when only a single train-device is considered, and worse than the ones related to byte 5 when two train-devices are used. This unexpected result can be explained by the *warm-up problem* of the instruction cache of the processor. Indeed, byte 0 is computed first at the beginning of the `for-loop` implementing *SubBytes*, and in that moment the loop-instructions are loaded directly from the flash memory of the device, being not yet stored in the instruction cache. From the second iteration on, those instructions are already present in cache and loaded from there. The fact that the instructions need to be loaded directly from the flash memory may alter the power consumption during the computation of byte 0, making it slightly harder to recover it with a SCA.
This process is known as *warm-up problem* since the at the beginning of the loop the

cache does not contain any loop-iteration, resulting empty, *cold*, while from the second iteration on, it results *warmed-up*, being full of instructions.

DKTA Summary

In summary, DKTA shows that the number of devices highly influences the attack, because the addition of a device always leads to a performance improvement. On the other hand, on average, the number of keys appears almost irrelevant, since it influences the performance of the attack in a consistent way only in the specific case of attacking byte 0 with two train-devices.

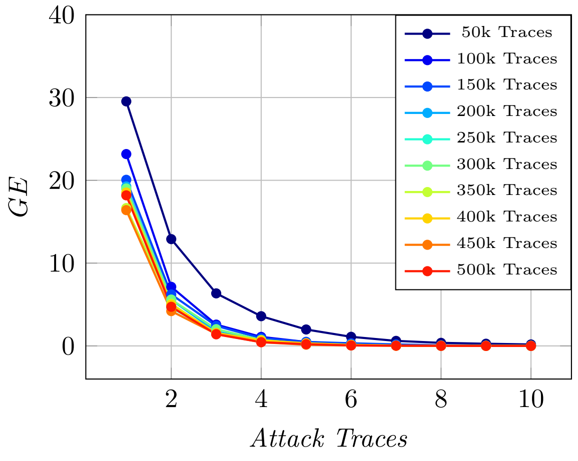
However, the overall performance of the attacks is very good, because, on average, it is possible to retrieve the targeted key-byte within only 10 attack-traces. In particular, targeting byte 5 and 14, this is achieved even performing the training phase with a single train-device, showing that, with the considered setup, the networks are very little affected by the Portability Problem.
In addition to the considered setup, the presence of a moderate Portability Problem with respect to the experiments shown in [7] may be explained also considering the utilized MLPs: they are very different from the one used in the considered state-of-the-art paper [7], since they feature *Dropout*, *BatchNormalization* and *L2-Regularization*. The addition of these countermeasures to overfitting may have increased the generalization level of the networks, making it possible to attack with more effectiveness a device-key configuration which is different from the one used during training.

Moreover, even in presence of a moderate Portability Problem, the Multiple Device Model is proven to boost the performance of the attacks, allowing to retrieve the targeted key-byte in few attack-traces. This result validates the effectiveness of MDM also on 32-bit microcontrollers.

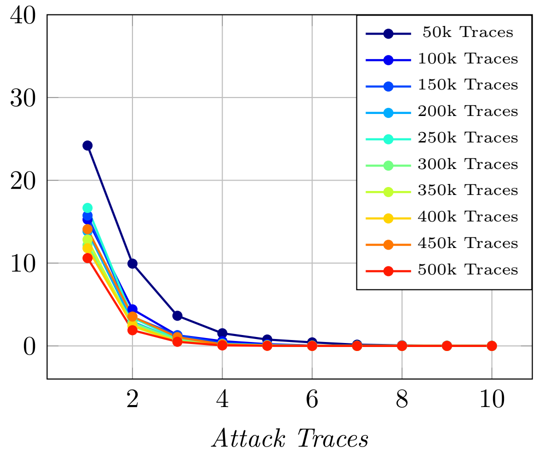
4.1.2. DTTA Results

For simplicity, DTTA was performed targeting only byte 5, since it does not represent a particular case:

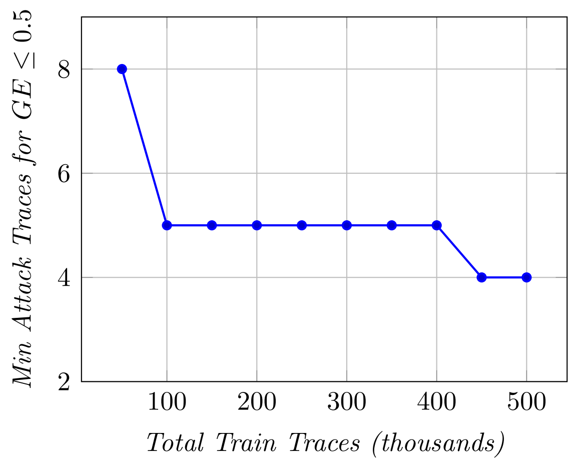
- it belongs to the 2nd row, so neither the first, computed at the very beginning of *SubBytes*, nor to the last, computed at the end;
- it is relatively affected by the *jitter effect*;
- its leakage is neither too visible, nor negligible.



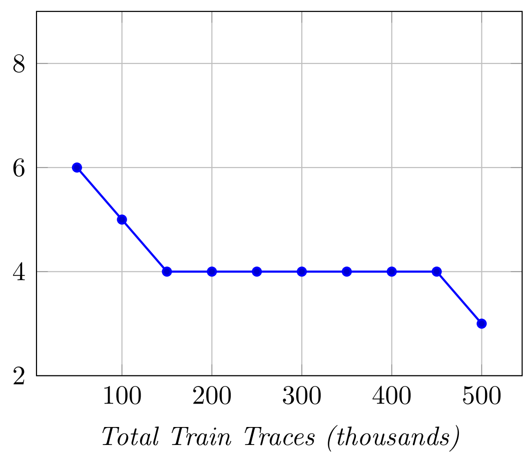
(a) Byte 5, Single Train-Device



(b) Byte 5, Two Train-Devices



(c) Min Attack Traces, Single Train-Device



(d) Min Attack Traces, Two Train-Devices

Figure 4.6: DTTA Results Targeting Byte 5 against Unprotected-AES. Subfigure (a), in the upper-left, shows the DTTA results obtained with a single train-device. Subfigure (b), in the upper-right, displays the outcome of DTTA when two train-devices are considered. Subfigure (c), in the lower-left, and Subfigure (d), in the lower-right, provide the trend of the minimum number of attack-traces needed to achieve $GE \leq 0.5$, showing Subfigure (a) and Subfigure (b), respectively, from another perspective.

In addition, relying on DKTA results, 10 keys are used during training. Indeed, by attacking byte 5, the number of keys does not influence the performance, so it makes no difference to use 1 or 10 keys.
On the other hand, since the collection of the traces was performed per device-key configuration, the higher the considered number of keys, the higher the number of available traces. As a consequence, considering two devices, the available traces are doubled: in order to provide comparable results, both DTTA with a single train-device and DTTA with two train-devices were performed considering the total amount of traces available with 10 keys and a single device, being 500,000.

Figure 4.6 shows the results of DTTA with the number of train-traces ranges from 50,000

to 500,000, every 50,000. To be precise, only the size of the training set varies, while the amount of traces dedicated to the validation set is always constant at 5,000. In addition, also in this case GEs are computed as average of 100 experiments.
Figure 4.6a shows the DTTA results obtained when only a single train-device is considered, while Figure 4.6b presents the DTTA results obtained with two train-devices. In both cases, the attacks are very good, since all of them reach $GE = 0$ in less than 10 traces. In particular, in both situations, the attack performance increases with the number of train-traces, almost monotonically. There is a quite big performance-boost when passing from 50,000 to at least 100,000 train-traces, since a gap is clearly visible in the plots between the GE curve obtained with 50,000 train-traces and the bundle of GE curves generated with a number of train-traces which ranges between 100,000 and 500,000.
Figure 4.6a and Figure 4.6b show an additional important result: the GE curve obtained with a single device and 100,000 train-traces is very similar to the one obtained with two train-devices and only 50,000 traces. This suggests that in case of constraints about the number of devices, the performance can still be boosted considering more train traces.

In order to quantify and better understand the influence that the number of train-traces has on the attack performance, especially when the GE curves form a single bundle, like when the number of train-traces ranges between 100,000 and 500,000, the number of minimum attack-traces needed to reach $GE \leq 0.5$ can be exploited. Figure 4.6c and Figure 4.6d show the trend of this quantity with respect to the number of train-traces. $GE \leq 0.5$ provides a good approximation for $GE = 0$, since it indicates that GE is closer to 0 than to 1. In particular, a threshold is necessary because GE refers to a position in a ranking and, in practice, it makes sense only if it is an integer value.
Figure 4.6c shows the minimum number of attack-traces to reach $GE \leq 0.5$ when a single train-device is considered. With 50,000 train-traces, it is possible to retrieve the correct key-byte in 8 attack-traces, but considering only 50,000 train-traces more, the same goal is achieved in only 5 attack-traces. The performance remains constant until 450,000 or 500,000 train-traces are considered, when the number of needed attack-traces drops to 4.
Figure 4.6d, on the other hand, focuses on the case of two train-devices. Here the number of attack-traces needed to obtain $GE \leq 0.5$ decreases monotonically with the increase of the number of train-traces when it ranges from 50,000 to 150,000, passing from 6 to 5 to 4. It then remains constant at 4 until 500,000 train-traces are used, where it drops to 3, achieving the best overall result.
In both cases, with a $10\times$ increase in the number of train-traces there is a $2\times$ increase in attack-performance, since the number of attack-traces needed to reach $GE = 0$ is halved.

DTTA Summary

In summary, DTTA shows that the number of train-traces highly influences the attack, because the enlargement of the training set usually leads to a performance-boost. In addition, the number of devices is again shown to be crucial in DL-based SCA against unprotected-AES, since the addition of a device always leads to better and quicker attacks. The combination of more train-traces and multiple train-devices allows to mount high-performing attacks, able to recover the targeted key-byte in only 3 attack-traces.

Also DTTA highlights the presence of moderate Portability Problem and validates the Multiple Device Model on 32-bit microcontrollers: the attacks are very good even if a single train-device is considered, but their performance can still be improved adding a second train-device.

Moreover, the previous considerations about the considered MLPs hold also in the case of DTTA: it is possible that the added overfitting-countermeasures, *Dropout*, *BatchNormalization* and *L2-Regularization* increased the generalization of the models, reducing the impact of the Portability Problem.

4.2. Results Targeting Masked-AES

This section shows the results obtained performing Device-Key Trade-off Analysis (DKTA) and Device-Trace Trade-off Analysis (DTTA) against a software masked implementation of AES-128. Also in this case, DKTA was executed first, so its results were considered and exploited while performing DTTA.

In this case, the targeted byte are byte 5 and byte 11.
In particular, byte 11 was attacked in order to validate the results obtained with byte 5.

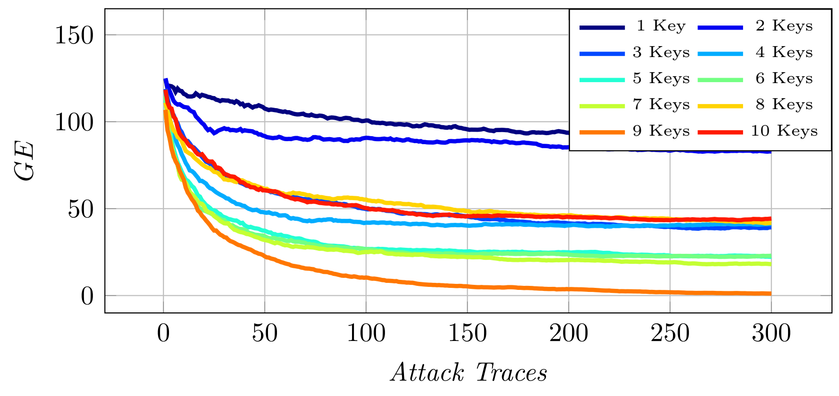
All the hyperparameters considered for the experiments are visible in Table A.2 of Appendix A. Comparing Table A.2 and Table A.1, it is visible that the networks dedicated to attack masked-AES are much simpler than the ones used against the unprotected implementation. The most likely hypothesis is that more complex network structures were discarded due to overfitting in favour of more more general, and consequently simpler, ones. Indeed, it can be seen that regularization is still present, in terms of Dropout, L2-Regularization and BatchNormalization.

4.2.1. DKTA Results

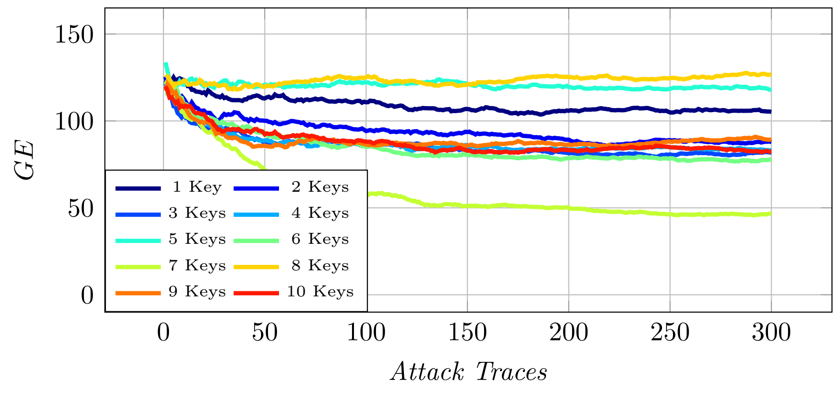
This section shows the DKTA results against a software masked implementation of AES-128, targeting byte 5.
The results of the same analysis performed targeting byte 11 are reported as validation support to the ones relative to byte 5.

Average Results Targeting Byte 5

Figure 4.7 shows the DKTA results for byte 5 in the cases of training with both one and two devices.
From Figure 4.7a, it can be seen that the considered MLP, trained with data coming from a single device, can successfully retrieve byte 5 from a masked implementation of AES, since the GE curve obtained with 9 keys reaches 0 within 300 attack-traces. It is important to point out that the MLP was able to perform the attack without the need of manual feature-extraction from the traces.
The number of attack-traces needed to succeed against a masked implementation is quite high, if compared to the unprotected scenario. This is due to the masking countermeasure: it eliminates the first-order leakage, forcing the network to collect information from much more traces before being able to recognize any leakage-pattern.
This result suggests that, also when targeting a masked implementation of AES, the



(a) Byte 5, Single Train-Device



(b) Byte 5, Two Train-Devices

Figure 4.7: DKTA Results Targeting Byte 5 against Masked-AES. Subfigure (a), up, shows the DKTA results obtained with a single train-device. Subfigure (b), down, displays the outcome of DKTA when two train-devices are considered.

involved networks are very little affected by the Portability Problem, with the considered setup.

Unfortunately, when two train-devices are considered, all the attacks fail, as depicted in Figure 4.7b. This behavior suggests that when attacking a masked implementation of AES, adding a train-device not only decreases the performance of the attack, but makes it unpractical. The most likely hypothesis is that considering both the masking countermeasure and a second data source as the second train-device, too much diversity is added to the training set, making the network unable to fully recognize the leakage patterns.

Regarding the number of keys, the attacks are highly influenced by it, especially in the single-device scenario. Indeed, referring to Figure 4.7a, the attack performance depends on the number of keys, though the increase is not monotonic with it: for example, 1 or 2 keys lead to an attack performance that is much worse than the one obtained with 5, 6 or 7 keys, even if there is not convergence to $GE = 0$; at the same time, while 9 keys allow to retrieve the correct key-byte, 10 keys do not.

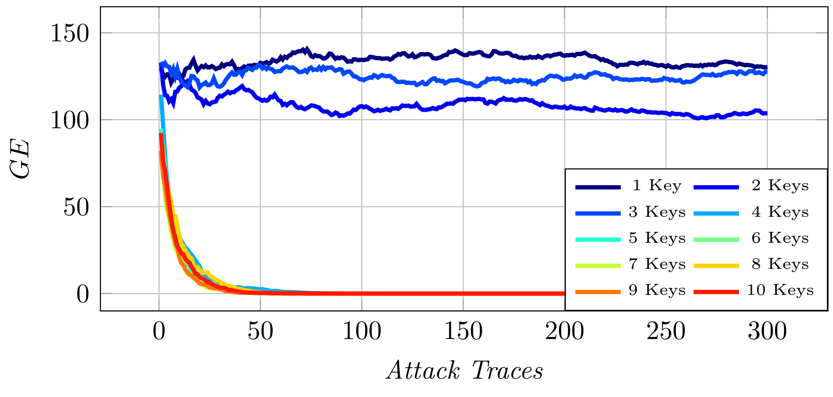
On the other hand, when considering the two-device scenario, this dependence is less visible, since all the performances are quite low. The best result, however, is achieved with a total of 7 keys, even if GE reaches a minimum value close to 50 in 300 attack-traces.

Analysis of the Single Attacks Targeting Byte 5

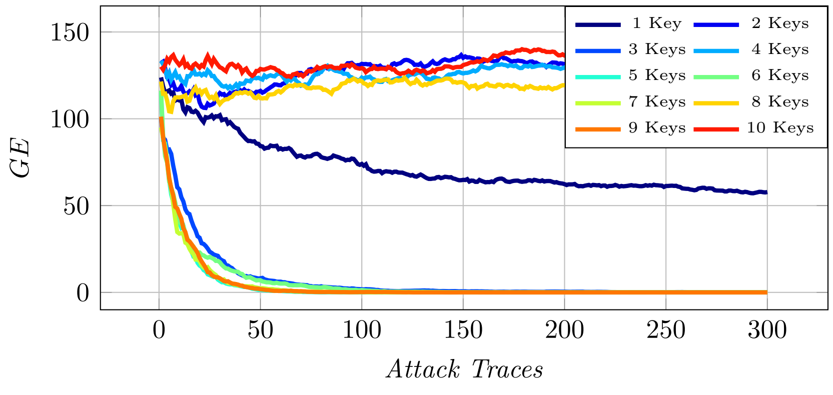
Figure 4.7 represents the *average* result of the attacks against the masked implementation of AES. A more detailed analysis, involving the study of each single attack performed to execute DKTA, revealed that there is a strong dependency between the choice of the train-devices and the attack performance. In particular, if the train-devices coincide with the ones used to during Hyperparameter Tuning, then the attacks perform better. This result can be interpreted as different manifestation of the Portability Problem, which is not related to the difference between train-device and target-device, but to the difference between train-device and device used to perform Hyperparameter Tuning.
In this work, Hyperparameter Tuning is always performed with $D1$ for the one-device scenario and with $D1$, $D2$ for the two-device case: this choice was made randomly, without any constraint.
Figure 4.8 and Figure 4.9 prove the highlighted dependency providing the results obtained with the attacks $D1$vs$D2$, $D2$vs$D1$ and $D3$vs$D2$ for the single-device scenario and $D1D2$vs$D3$, $D1D3$vs$D2$ and $D2D3$vs$D1$ for the two-device scenario, respectively.

Considering the attack $D1$vs$D2$, reported in Figure 4.8a, it is possible to perform a very good attack against a masked implementation of AES, reaching $GE = 0$ in about 50 traces. Here, the number of keys highly influences the attack. With 1 or 3 keys, the network behaves like a random classifier, providing the correct key-byte in a position between 120 and 140, with 2 keys the performance slightly increases, but GE ends up at 90 even if 300 attack-traces are used. With 4 keys or more, the convergence to $GE = 0$ is almost immediate, since it takes about 50 attack-traces. In this case the train device, $D1$, coincides with the device used during Hyperparameter Tuning. On the other hand, in a scenario where $D1$ is no more used as train-device, such as $D2$vs$D1$ in Figure 4.8b and $D3$vs$D2$ in Figure 4.8c, the performance drops and, in particular, there is no more a monotonic increase of performance when adding keys.

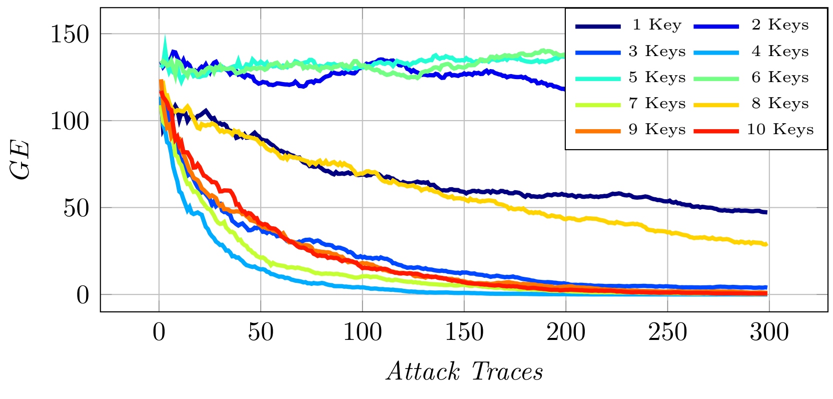
Regarding the two-device scenario, attack $D1D2$vs$D3$ reported in Figure 4.9a, where the train-devices are exactly the ones used during Hyperparameter Tuning, shows that it is possible to mount a successful DL-based SCA with two train-devices against a masked implementation of AES. In particular, the correct key-byte can be retrieved in 150 attack-traces, 3× the amount needed, in the best case, with a single train-device. Also in this case, however, the attack performance does not increase monotonically with the number



(a) Byte 5, $D1$vs$D2$



(b) Byte 5, $D2$vs$D1$

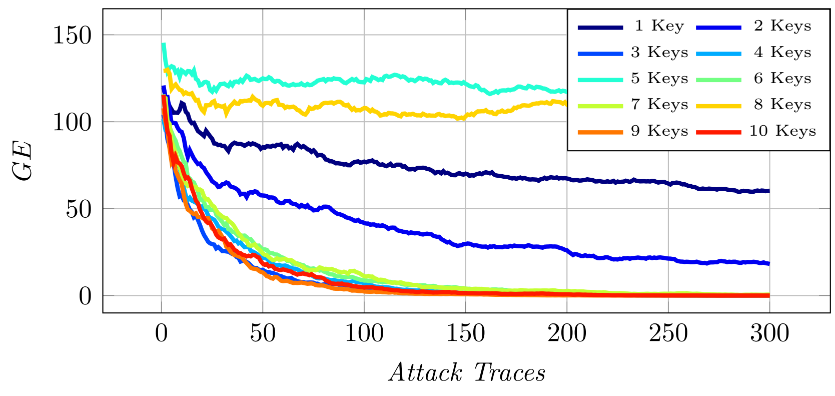


(c) Byte 5, $D3$vs$D2$

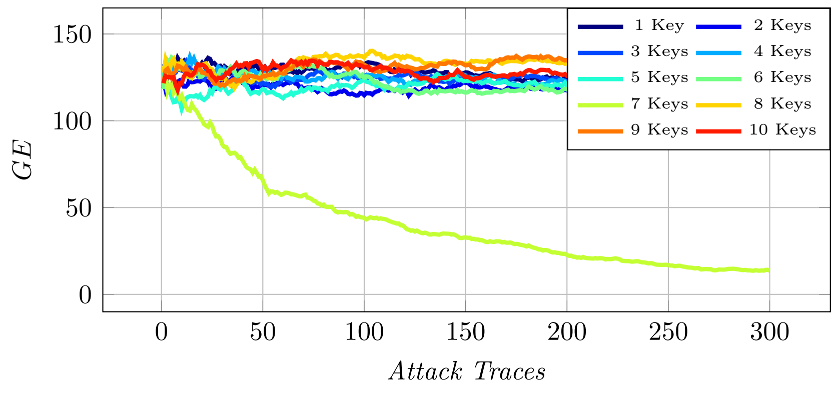
Figure 4.8: Single Attacks Targeting Byte 5 with a Single Train-Device, Masked-AES. This figure displays three of the six attacks performed during DKTA targeting byte 5 with a single train-devices. Subfigure (a), up, shows the results of the attack $D1$vs$D2$. Subfigure (b), in the middle, shows the results of the attack $D2$vs$D1$. Subfigure (c), down, shows the results of the attack $D3$vs$D2$.

of keys, but considering more than 1 or 2 keys usually ensures convergence to $GE = 0$, even if divergent outliers are obtained with 5 and 8 keys.

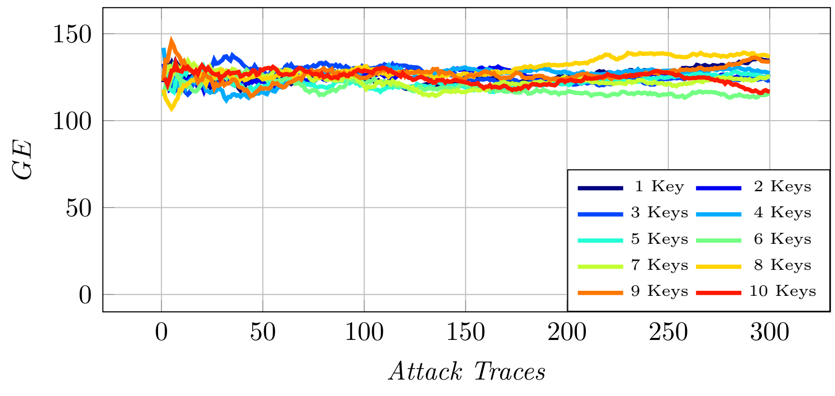
On the other hand, in any scenario where either $D1$ or $D2$ is not considered as train-device, as depicted in Figure 4.9b, relative to attack $D1D3$vs$D2$ and in Figure 4.9c, relative to attack $D2D3$vs$D1$, the performance drops considerably: the networks always behave like



(a) Byte 5, $D1D2$vs$D3$



(b) Byte 5, $D1D3$vs$D2$

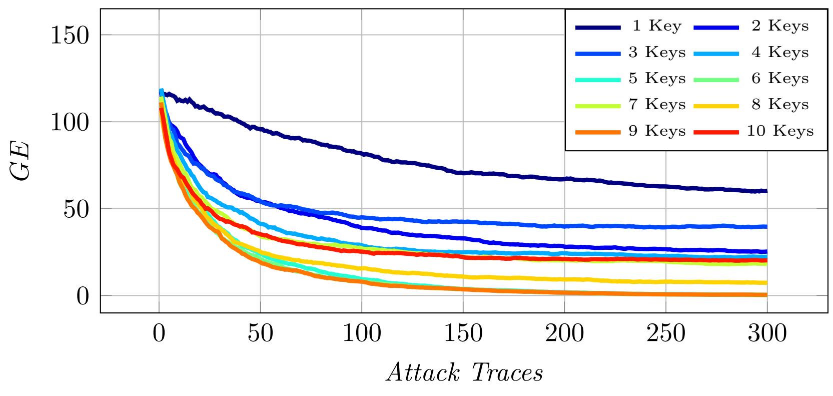


(c) Byte 5, $D2D3$vs$D1$

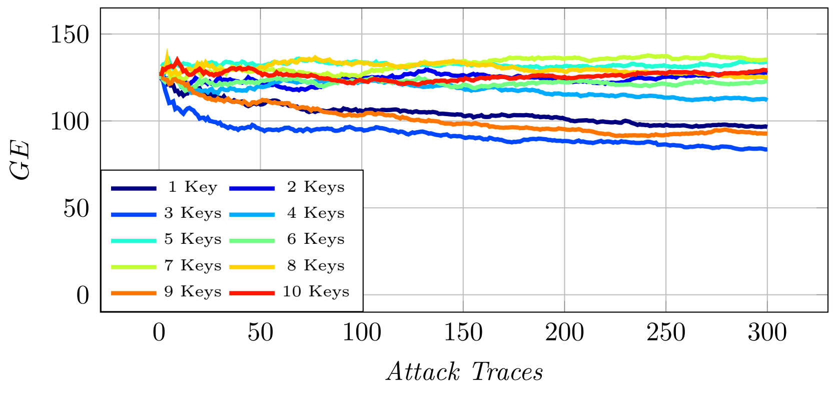
Figure 4.9: Single DKTA Attacks Targeting Byte 5 with Two Train-Devices, Masked-AES. This figure displays all the attacks performed during DKTA targeting byte 5 with two train-devices. Subfigure (a), up, shows the results of the attack $D1D2$vs$D3$. Subfigure (b), in the middle, shows the results of the attack $D1D3$vs$D2$. Subfigure (c), down, shows the results of the attack $D2D3$vs$D1$.

a random-guesser, providing the correct key-byte between position 120 and 140. The only outlier is the attack $D1D3$vs$D2$ with exactly 7 keys, visible in Figure 4.9b, where GE reaches a value less than 20 in 300 attack-traces.

From the attacks with a single train-device, one may deduce that $D1$ and $D2$ are more similar than $D1$ and $D3$ or $D2$ and $D3$, as the attacks $D1$vs$D2$ and $D2$vs$D1$ perform



(a) Byte 11, Single Train-Device



(b) Byte 11, Two Train-Devices

Figure 4.10: DKTA Results Targeting Byte 11 against Masked-AES. Subfigure (a), up, shows the DKTA results obtained with a single train-device. Subfigure (b), down, displays the outcome of DKTA when two train-devices are considered.

better than $D3$vs$D2$. Even if it is plausible since they are physically different devices and therefore underwent the same manufacturing process in different moments, this seems not the case, as the only successful attack with two train-devices is obtained just exploiting traces coming from both $D1$ and $D2$ to target $D3$: in case of almost complete similarity between only $D1$ and $D2$, this attack would have failed, since the train-traces would have been so different from the attack-ones that no leakage pattern would have been recognized during the attack phase.
In addition, since the devices are never changed during the research, in case of strong differences between them, also the results related to unprotected-AES would have been affected by this behavior.

Average Results Targeting Byte 11

The results relative to DKTA targeting byte 11 are reported in Figure 4.10.
Regarding the case of a single train-device, shown in Figure 4.10a, the results obtained targeting byte 11 are quite similar to the ones obtained with byte 5. Indeed, also in this

case the MLP is able to retrieve the correct key-byte from a masked implementation of AES, showing that the Portability Problem is minimal with the considered setup. In addition, the attack performance can be enhanced, as before, by considering multiple keys. The results are very similar also when considering two train-devices, as depicted in Figure 4.10b: the attacks are not able to reach $GE = 0$, even if 300 attack-traces are exploited.
These results validate the hypothesis of performance-drop when multiple devices are considered targeting a masked implementation of AES.

DKTA Summary

In summary, an MLP is shown to be able to recover the correct key-byte from a masked implementation of AES-128 running on a 32-bit microcontroller without the need of manual feature-extraction from the traces. In particular, it is possible to *break* the masking countermeasure with about 300 traces if a single train-device and 9 keys are considered.
The addition of a device during training decreases the performance of the attack, making it impossible to retrieve the correct key-byte.
The number of keys influences the attack, but the performance does not increase monotonically with it.

The Portability Problem is still present, but it appears not to be caused by the consideration of a target-device which is different from the train-one. Indeed, it is more related to the choice of the train-devices: if they coincide with the ones used during Hyperparameter Tuning, then the attack performs better, otherwise, it may become unfeasible.
Indeed, considering the same devices for both training and Hyperparameter Tuning, successful attacks are performed both with one and two train-devices, recovering the correct key-byte in about 50 and about 150 traces respectively.

However, in the case of two train-devices, when the same devices are used both in training and Hyperparameter Tuning, even if the attack is successful, the achieved performance is still lower than the one achieved with a single device. Therefore, the MDM technique should be avoided when targeting a masked implementation of AES. However, the MDM technique should be avoided when targeting a masked implementation of AES: even if it allows to perform successful attacks considering as train-devices the same ones used during Hyperparameter Tuning, their performance is still much lower than the one achieved with a single train-device.

4.2.2. DTTA Results

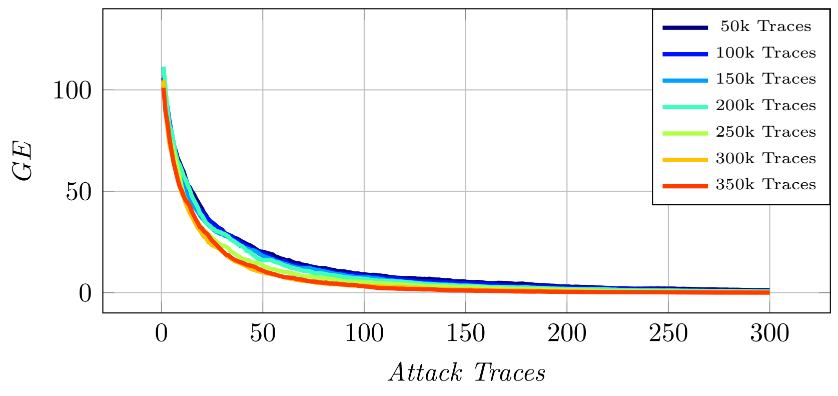
This section shows the DTTA results obtained against the software masked implementation of AES-128, targeting byte 5 and byte 11.

Relying on the results of DKTA, the attacks with a single train-device were performed considering 9 keys, while the attacks with two train-devices used 7 keys. Indeed, 9 keys and 7 keys led to the best performance, on average, when training the MLP with traces coming from a single device and two devices, respectively.
Since the collection of the traces was performed per device-key configuration, in order to provide comparable results, a total of 350,000 traces was considered, as it is the maximum achievable with 7 keys and a single device. The number of traces starts at 50,000 and reaches 350,000, increasing with *step* 50,000.

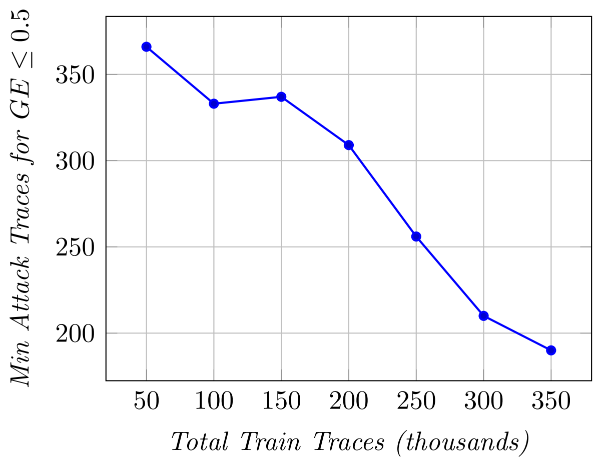
Average Results Targeting Byte 5

Figure 4.11 shows the DTTA results when targeting byte 5 against a software masked implementation of AES-128.
Figure 4.11a presents the case of single train-device: all the attacks are very good, converging to $GE = 0$ within 300 attack-traces. In particular, the performance of the attack appears to increase monotonically with the number of traces, highlighting how the size of the training set influences the attack.
On the other hand, Figure 4.11c shows the DTTA results in a scenario where two train-devices are considered. In this case, the attacks are not able to reach $GE = 0$, even if 300 attack-traces are exploited. The number of traces does not influence the attack performance, since all GE curves, in the end, converge to a value close to 40.

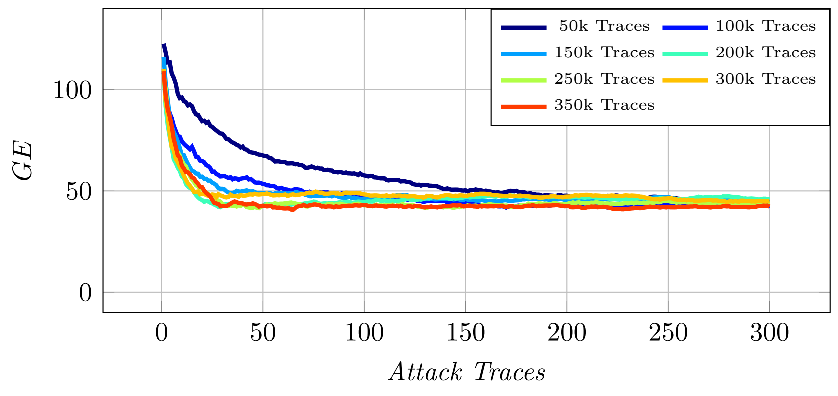
Figure 4.11b explains the DTTA results obtained with a single train-device in terms of minimum number of attack-traces needed to achieve $GE \leq 0.5$, which is a good approximation for $GE = 0$. It can be seen that, similarly to the case of DTTA against unprotected-AES, the overall performance-boost given by the addition of train-traces is about $2\times$: considering 350,000 train-traces it is possible to achieve $GE \leq 0.5$ with only 190 attack-traces, close to half the total amount needed to achieve the same goal with 50,000 train-traces. Moreover, it can be seen that the trend of the minimum number of attack-traces decreases when the number of train-traces increases, confirming that the attack performance monotonically increases with the size of the training set.
The first four points in the plot of Figure 4.11b present value higher than 300, while the plot in Figure 4.11a only shows the trend of the GEs for a maximum of 300 attack-traces: actually, the total amount of attack-traces considered while computing the GEs was 500,



(a) Byte 5, Single Train-Device



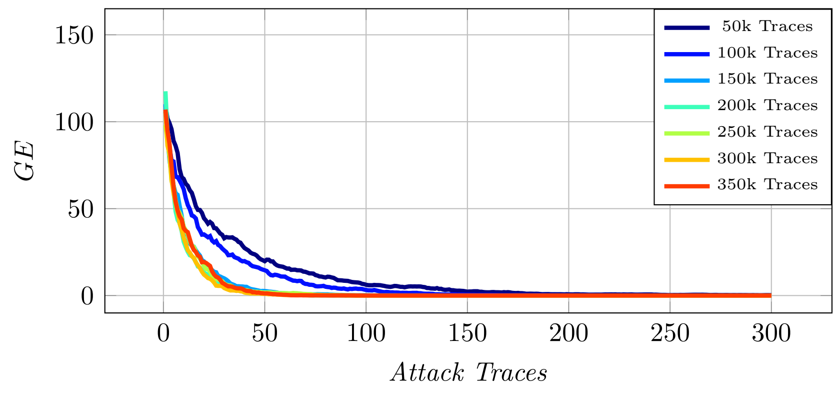
(b) Min Attack Traces, Single Train-Device



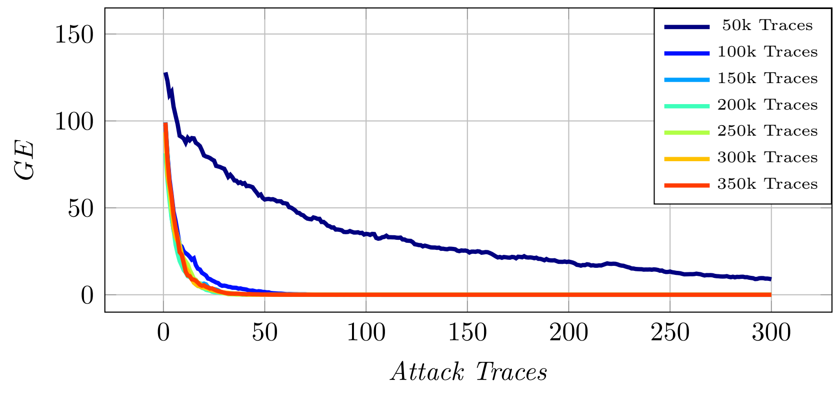
(c) Byte 5, Two Train-Devices

Figure 4.11: DTTA Results Targeting Byte 5 against Masked-AES. Subfigure (a), up, shows the DTTA results obtained with a single train-device. Subfigure (b), in the middle, shows the minimum number of attack traces needed to perform a successful attack with a single train-device. Subfigure (C), down, displays the outcome of DTTA when two train-devices are considered.

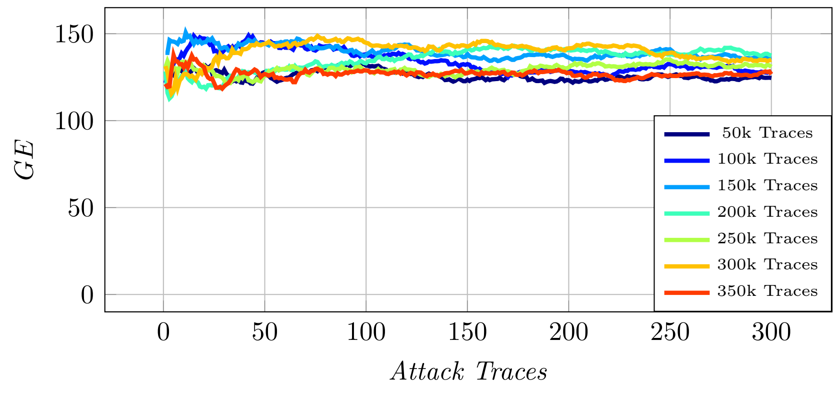
so it is possible to have values greater than 300. The choice of plotting the values of GEs for only 300 attack-traces is only graphical, since after this threshold there are no major differences in the results.



(a) Byte 5, $D1D2$vs$D3$



(b) Byte 5, $D1D3$vs$D2$



(c) Byte 5, $D2D3$vs$D1$

Figure 4.12: Single DTTA Attacks Targeting Byte 5 with Two Train-Devices, Masked-AES. This figure displays all the attacks performed during DKTA targeting byte 5 with two train-devices. Subfigure (a), up, shows the results of the attack $D1D2$vs$D3$. Subfigure (b), in the middle, shows the results of the attack $D1D3$vs$D2$. Subfigure (c), down, shows the results of the attack $D2D3$vs$D1$.

Analysis of the Single Attacks Targeting Byte 5

Considering once again Figure 4.11c, it is important to point out that it shows the *average* result obtained with two train-devices. Figure 4.12 shows the single attacks mounted to execute DTTA in case of two train-devices.

Figure 4.12a depicts the results related to $D1D2$vs$D3$, where the train-devices coincides with the ones used during Hyperparameter Tuning: increasing the number of train-traces it is possible to successfully retrieve the correct key-byte in about 50 attack-traces. In particular, the most important performance-boost is obtained passing from 50,000 or 100,000 train-traces to at least 150,000, since $GE = 0$ is reached with about 50 attack-traces instead of 150/200.

Figure 4.12b is relative to attack $D1D3$vs$D2$. The baseline for this attack, obtained with 50,000 traces, does not reach $GE = 0$ even with 300 attack-traces. However, it was the best result achieved with this device-configuration during DKTA: it is the visible outlier in Figure 4.9b. DTTA shows that it is possible to enhance its performance considering more train-traces. In particular, with 100,000 of them it is possible to obtain $GE = 0$ in 60/70 attack-traces, while with at least 150,000 the performance is further increased, converging to 0 in less than 50 attack-traces.

Concerning attack $D2D3$vs$D1$, depicted in Figure 4.12c, it can be seen that it is not influenced by the addition of train-traces, altering, in fact, the average result of DTTA int the two-device scenario. The network acts like a random-guesser, providing the correct key-byte around position 128 among prediction. Indeed, it appears unable either to learn the leakage pattern, or to transfer the information gained during training to the attack.

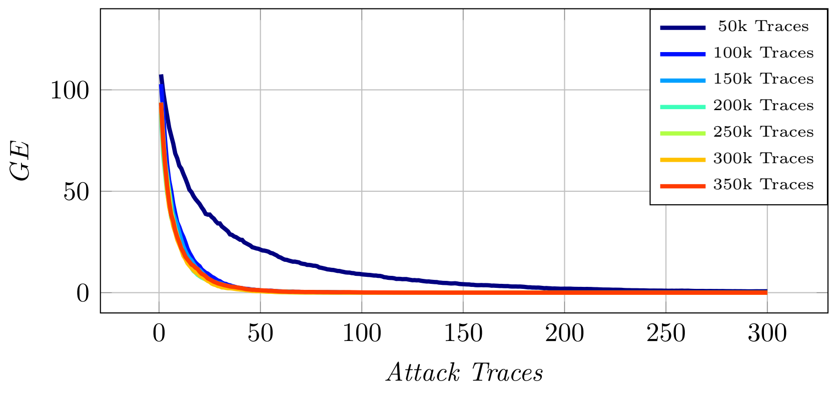
In the end, the high performance of attack $D1D2$vs$D3$ compared to the low performance of attack $D2D3$vs$D1$ confirms that the Portability Problem can manifest also in terms of train-device choice, highlighting the importance of considering the same devices used during Hyperparameter Tuning also in the case of DTTA. However, as pointed out by attack $D1D3$vs$D2$, sometimes it is possible to achieve excellent results even with different train-devices, as long as the size of the dataset is increased.

Average Results Targeting Byte 11

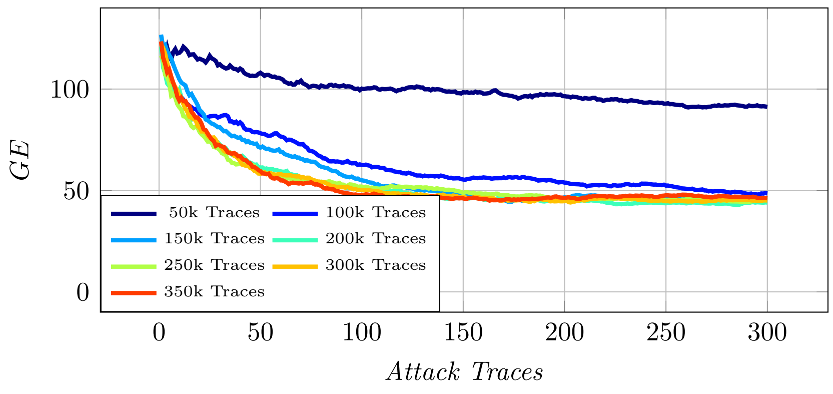
The results relative to DTTA targeting byte 11 can be seen in Figure 4.13.

When a single train-device is considered, as shown in Figure 4.13a, the results obtained targeting byte 11 are very similar to the ones obtained with byte 5, since the the MLP is able to quickly recover the correct key-byte and, at the same time, the attack performance can be enhanced by considering more of train-traces. Actually, the attacks to byte 11 converge to $GE = 0$ faster than the ones targeting byte 5, recovering the key-byte in just about 50 traces *on average* when at least 100,000 train-traces are considered. In addition, the results show that considering more than 100,000 train-traces is worthless in this case, since the relative curves are almost overlapped from the beginning.

On the other hand, considering the case of two train-devices depicted in Figure 4.13b, the results are almost identical, since in both cases the attacks are not able to recover the



(a) Byte 11, Single Train-Device



(b) Byte 11, Two Train-Devices

Figure 4.13: DTTA Results Targeting Byte 11 against Masked-AES. Subfigure (a), up, shows the DTTA results obtained with a single train-device. Subfigure (b), down, displays the outcome of DTTA when two train-devices are considered.

correct key-byte, even if 300 train-traces are considered. In addition, even the convergence point of the GE curves is almost the same, being around 40.
Therefore, also considering DTTA, the results obtained targeting byte 11 validate the ones obtained with byte 5, once again highlighting the presence of minimal Portability Problem with the considered setup and demonstrating that considering multiple device decreases the performance of the attacks against a masked implementation of AES.

DTTA Summary

In summary, DTTA shows how the number of train-traces highly influences the performance of the attack, even in presence of masking countermeasure.
On average, the best performing train-configuration involves a single device and 350,000 traces, which makes it possible to recover the correct key byte in 190 attack traces, increasing the performance by a factor of almost 2× if compared to the result achieved with only 50,000 traces. In particular, in this case, it can be observed that the attack performance increases monotonically with the number of train-traces.

On the other hand, if an additional train-device is considered, then the average performance drops consistently, since attacks do not reach $GE = 0$ even if with 300 attack-traces.

The importance of considering, during the attack, the same train-devices used during Hyperparameter Tuning is highlighted also by DTTA: the attack $D1D2$vs$D3$ reaches $GE = 0$ once again, and its performance is further increased by the addition of train-traces. Moreover, DTTA showed how it is possible to limit the dependence on the train-devices used during Hyperparameter Tuning by increasing the training set size, as visible from attack $D1D3$vs$D2$.

In the end, the Portability Problem is clearly visible only in the specific scenario $D2D3$vs$D1$. Even if this is an isolate case, it confirms the unreliability of the MDM technique in case of masking countermeasure: adding a device, on average, makes the attack unfeasible.

4.3. Reproducibility

The collected traces are available on Zenodo [12].
Since a total of 3 devices, 11 keys and 2 implementations of AES were considered, the data is provided as 2 sets of 33 datasets, one per device-key configuration. Each dataset contains 50,000 traces.
The length of the traces depends on the targeted algorithm: the traces related to unprotected-AES involve 1183 samples, while the ones relative to masked-AES are formed by 7700 samples.
The size of the validation set was fixed to 5000 traces, both in DKTA and DTTA. Moreover, the Guessing Entropy was computed always as average over 100 independent experiments, each one involving 500 attack-traces: the sets of traces were built partitioning the test set, meaning that no traces were repeated among the experiments.
The targeted key was always $K0$, meaning that it was never used during training.

The code used throughout the research is available at
`https://github.com/luca-castellazzi/EfficientTraining_DLSCA`.
The project was developed in `Python 3.8.10`. The `requirements.txt` file present in the linked repository contains all the considered libraries, with the specific version used throughout this work.
The code repository features also a `JSON` file containing the secret encryption keys used in the experiments, since they were randomly-generated with a `Python` script.
Moreover, also two `Jupyter Notebooks` are provided together with the code. The first one shows how to retrieve the 6^{th} byte of $K0$ (byte 5) with a DL-based SCA, expalining it step-by-step, starting from model training and terminating with the key-byte recovery. The second one, instead, performs a complete attack against $K0$, training 16 networks and attacking each byte of the secret key. Both attacks are performed against the unprotected implementation of AES, validating the setup and the attack pipeline.

The machine used to run all the `Python` scripts presents the following specifications:

- *Operating System*: Ubuntu 20.04;
- *CPU*: Intel Core i7-10700K CPU @ 3.8GHz;
- *RAM*: 66GB;
- *GPU*: Nvidia GeForce RTX 2080 Ti with 11GB of dedicated GDDR6 memory.

Therefore, a different setup may lengthen or shorten the time needed to execute the experiments, mainly the process of Hyperparameter Tuning.

5 | Conclusions and Future Work

This work provides a complete characterization of Deep Learning based Side-Channel Attacks against software implementations of AES-128, both with and without masking countermeasures, running on 32-bit microcontrollers, in terms of trade-off between the most important variables of the Profiling Phase, being the number of devices, the number of keys and the number of traces.
Device-Key Trade-off Analysis (DKTA) is introduced to study the influence that number of devices and number of keys have on the attack performance, while Device-Trace Trade-off Analysis (DTTA) to focus on the impact that number of devices and number of traces have during the attack.

Considering a software unprotected implementation of AES-128, the number of devices and the number of traces are shown to highly influence the attack, since a considerable performance improvement can be achieved either by adding a device or increasing the number of traces. In particular, a 10× increase in the number of train-traces leads to a 2× performance improvement, highly reducing the time taken to complete the Attack Phase.
On the other hand, the number of keys appears to be much less relevant, influencing the performance only in the specific case of an attack targeting byte 0 with two train-devices, where the consideration of multiple keys gives a visible performance-boost.

Considering a software masked implementation of AES-128, instead, all the variables influence the attack: while an increase in terms of number of keys or number of traces usually increases the performance, the usage of an additional train-device makes it unfeasible, on average, to recover the correct key-byte.
Increasing the number of keys appears to be useful when the training set size is fixed and a single train-device is used, even if the attack performance does not increase monotonically. On the other hand, a monotonic performance improvement can be achieved by increasing the number of traces when the number of keys is fixed and a single train-device is considered. In particular, a 2× performance improvement has been observed with a 7× increase in the number of train-traces.

Furthermore, a detailed analysis of each single attack performed with both one and two train-devices highlighted that there is a performance improvement if the train-devices coincide with the ones used during Hyperparameter Tuning.

The obtained results allow to advance in the research field of DL-based SCAs.
First of all, it is proven that an increase in the number of train-devices leads to a performance improvement also attacking a software unprotected implementation of AES-128 running on a 32-bit microcontroller: this fact validates, for the first time, the MDM introduced in [7] on architectures more complex than the 8-bit ones. In addition, the effectiveness of this technique is consistently validated by averaging the results obtained with all the possible combinations of train-devices/target-device available with the considered setup.
Then, the successful attacks against the software masked implementation of AES-128 show for the first time that the approach introduced in [39] of automatic feature-extraction when SCA-countermeasures are in place works even if the traces come from 32-bit microcontrollers.
Furthermore, it is proven that DL-based SCAs depend on three main variables, being the number of train-devices, the number of keys and the number of train-traces. The number of devices and the number of traces, among the three, are the most influencing, since their increase usually makes the attack better.
Moreover, the fact that it was possible to perform successful realistic DL-based SCAs even considering a single train-device suggests that the Portability Problem may be dependent on the considered setup. However, it is also possible that the Hyperparameter Tuning process influences the attack performance in portability scenarios: indeed, the hyperparamter search of this work considers also regularization coefficients and *Early Stopping*, increasing the complexity of the process with respect to the one used in [7].
Finally, this work highlights a possible different manifestation of the Portability Problem in case of attack against a software masked implementation of AES-128: it is no more dependent on the difference between the train and the target device, but rather between the train-devices and the ones used during Hyperparameter Tuning. In particular, the best performance is achieved by using the same devices for both the tuning process and the training during the Profiling Phase.

Even if this work focuses only on attacking the most basic AES implementation alongside its masked version, the proposed methodology can be applied to any encryption algorithm.
Therefore, future developments related to this work can target different and even more complex versions of AES.

For example, DKTA and DTTA can be applied to:

- Hardware implementations of AES;
- AES with different modes of operation [4, 20];
- Pipelined implementations of AES.

In addition, this work can be further extended considering a different side-channel, such as Electro-Magnetic (EM) emission. This scenario would allow to relax the assumption, for the attacker, of physical access to the pins of the target-device, needed to connect the current probe. Indeed, in order to capture EM emissions, one can simply put a probe close to the device, without the need of accessing the pins. In this case, the noise present in the measurements may be the major concern, especially if the attack is performed in a non-controlled environment.

Bibliography

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from `https://www.tensorflow.org`.

[2] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 77–82, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2002.

[4] Sultan Almuhammadi and Ibraheem Al-Hejri. A comparative analysis of AES common modes of operation. In *30th IEEE Canadian Conference on Electrical and Computer Engineering, CCECE 2017, Windsor, ON, Canada, April 30 - May 3, 2017*, pages 1–4. IEEE, 2017.

[5] Daehyeon Bae, Jongbae Hwang, and JaeCheol Ha. Breaking a masked AES implementation using a deep learning-based attack. In *ACM ICEA '20: 2020 ACM International Conference on Intelligent Computing and its Emerging Applications, GangWon Republic of Korea, December 12 - 15, 2020*, pages 22:1–22:5. ACM, 2020.

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

[7] Shivam Bhasin, Anupam Chattopadhyay, Annelie Heuser, Dirmanto Jap, Stjepan Picek, and Ritu Ranjan Shrivastwa. Mind the portability: A warriors guide through realistic profiled side-channel analysis. *IACR Cryptol. ePrint Arch.*, page 661, 2019.

[8] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, January 2006.

[9] Bruce Blaus. Blausen 0657 multipolarneuron.png. `https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png`, September 2013. Licensed under Creative Commons Attribution 3.0, `https://creativecommons.org/licenses/by/3.0/`.

[10] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

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

[12] Luca Castellazzi. Towards Efficient Training in Deep Learning Side- Channel Attacks, April 2023. `https://doi.org/10.5281/zenodo.7817187`.

[13] A. CAUCHY. Methode generale pour la resolution des systemes d'equations simultanees. *C.R. Acad. Sci. Paris*, 25:536–538, 1847.

[14] Snehashish Chakraverty, Deepti Moyi Sahoo, and Nisha Rani Mahato. *Hebbian Learning Rule*, pages 175–182. Springer Singapore, Singapore, 2019.

[15] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[17] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[18] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[19] George V. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[20] Morris Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices, 2010-01-18 2010.

[21] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[22] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing.* Natural Computing Series. Springer, 2003.

[23] Richard Gilmore, Neil Hanley, and Máire O'Neill. Neural network based attack on a masked implementation of AES. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 106–111. IEEE Computer Society, 2015.

[24] Matt Harvey and Norman Heckscher. Evolve a neural network with a genetic algorithm. `https://github.com/harvitronix/neural-network-genetic-algorithm`, 2017. Published under MIT License.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

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

[27] Geoffrey Hinton. Lecture 6 of the 'Neural Network for Machine Learning' course on Coursera, 2018.

[28] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

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

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[33] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, pages 161–191, 1883.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

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

[36] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *J. Cryptogr. Eng.*, 1(1):5–27, 2011.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[38] Dennis Luciano and Gordon Prichett. Cryptology: From caesar ciphers to public-key cryptosystems. *The College Mathematics Journal*, 18(1):2–17, 1987.

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

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

[41] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *Proceedings of the 21th*

International Conference on Artificial Neural Networks - Volume Part I, ICANN'11, page 52–59, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. A comprehensive study of deep learning for side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):348–375, 2020.

[43] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[44] Thomas S. Messerges. Using second-order power analysis to attack DPA resistant software. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2000.

[45] Melanie Mitchell. *An introduction to genetic algorithms.* MIT Press, 1998.

[46] Tom Mitchell. *Machine Learning.* McGraw-Hill Education, 1997.

[47] Len Luet Ng, Kim Ho Yeap, Magdalene Wan Ching Goh, and Veerendra Dakulagi. Power consumption in cmos circuits. In Prof. Hai-Zhi Song, Dr. Kim Ho Yeap, and Dr. Magdalene Goh Wan Ching, editors, *Electromagnetic Field in Advancing Science and Technology*, chapter 5. IntechOpen, Rijeka, 2022.

[48] Mark Otto et al. Bootstrap project. Icons available at `https://icons.getbootstrap.com`.

[49] Lutz Prechelt. Early stopping — but when? In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade: Second Edition*, pages 53–67, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[50] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[51] Riscure. Inspector side channel analysis. `https://www.riscure.com/security-tools/inspector-sca`.

[52] Riscure. Piñata (training target). `https://www.riscure.com/products/pinata-training-target`.

[53] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[54] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[55] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[57] Kai Schramm and Christof Paar. Higher order masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[58] C.E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[59] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

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

[61] Tormod Volden et al. dfu-util. `https://sourceforge.net/p/dfu-util/dfu-util/ci/master/tree`, 2012.

[62] Huanyu Wang, Sebastian Forsmark, Martin Brisfors, and Elena Dubrova. Multi-source training deep-learning side-channel attacks. In *50th IEEE International Sym-*

posium on Multiple-Valued Logic, ISMVL 2020, Miyazaki, Japan, November 9-11, 2020, pages 58–63. IEEE, 2020.

[63] Jason Weston and Chris Watkins. Support vector machines for multi-class pattern recognition. In *7th European Symposium on Artificial Neural Networks, ESANN 1999, Bruges, Belgium, April 21-23, 1999, Proceedings*, pages 219–224, 1999.

[64] Sebastien C. Wong, Adam Gatt, Victor Stamatescu, and Mark D. McDonnell. Understanding data augmentation for classification: when to warp?, 2016.

A | Selected Hyperparameters

This appendix shows all the results of the Hyperparameter Tuning processes performed throughout this work with the Genetic Algorithm.

Hyperparameters to Attack Unprotected-AES

(a) 1 Train-Device

Hyperarameter	Byte 0	Byte 5	Byte 11	Byte 14
Number of Hidden Layers	4	4	4	4
Number of Hidden Neurons	500	500	200	300
Dropout Rate	0.1	0.2	0.2	0.4
L2-Regularization Coefficient	0.01	0.001	0.001	0.0
Gradient Descent Technique	*adam*	*adam*	*adam*	*adam*
Learning Rate	0.001	0.001	0.005	0.005
Batch Size	128	256	1024	128

(b) 2 Train-Devices

Hyperarameter	Byte 0	Byte 5	Byte 11	Byte 14
Number of Hidden Layers	3	4	4	3
Number of Hidden Neurons	400	500	500	300
Dropout Rate	0.2	0.4	0.4	0.5
L2-Regularization Coefficient	0.0005	0.0	0.005	0.0
Gradient Descent Technique	*adam*	*adam*	*adam*	*RMSprop*
Learning Rate	0.005	0.005	0.0005	0.005
Batch Size	128	128	512	1024

Table A.1: Hyperparameters to Attack Unprotected-AES. These tables show the hyperparameters selected by the Genetic Algorithm to attack unprotected-AES targeting byte 0, 5, 11 and 14. Table (a), up, shows the result of Hyperparameter Tuning when a single train-device is considered. Table (b), down, presents the hyperparameters selected in case of two train-devices.

Hyperparameters to Attack Masked-AES

Hyperparameter	**1 Train-Device**		**2 Train-Devices**	
	Byte 5	**Byte 11**	**Byte 5**	**Byte 11**
Number of Hidden Layers	2	3	2	4
Number of Hidden Neurons	100	100	200	100
Dropout Rate	0.3	0.3	0.5	0.5
L2-Regularization Coefficient	0.001	0.001	0.005	0.05
Gradient Descent Technique	*RMSprop*	*adam*	*adam*	*adam*
Learning Rate	0.005	0.0005	0.0005	0.0005
Batch Size	128	256	256	256

Table A.2: Hyperparameters to Attack Masked-AES. This table shows the hyperparameters selected by the Genetic Algorithm when targeting byte 5 and byte 11 against a masked implementation of AES, both with a single train-device and two train-devices.

List of Figures

List of Tables

Acknowledgements

Thanks to my advisor, Prof. Luca Oddone Breveglieri, and co-advisors, Prof. Alessandro Barenghi and Prof. Gerardo Pelosi, for the availability, the collaboration and the valuable advice.
A special thanks to Dr. Niccolò Izzo for the collaboration, the availability and for solving any doubts I had during the development of the Thesis, and to Dr. Paolo Amato and Dr. Danilo Caraccio for the opportunity and for believing in me.
Finally, a huge thanks to my family for always supporting me during my studies.