Adversarial Examples as a defense against Side-Channel Attacks

A novel approach for countermeasure design

Alexandros Korpas Kamarianos

Adversarial Examples as a defense against Side-Channel Attacks

A novel approach for countermeasure design

by

Alexandros Korpas Kamarianos

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on July 14, 2021 at 10:00 AM.

Student number:	4907167	
Supervisor:	Dr. S. Picek,	TU Delft
Thesis committee:	Prof.dr.ir. R.L. Lagendijk,	TU Delft
	Dr. S. Picek,	TU Delft
	Dr. J. Urbano,	TU Delft

TU Delft Delft University of Technology

Abstract

Over the last decades, side-channel attacks (SCAs) have been proven as a substantial weakness of cryptographic devices, while the recent growth of deep learning (DL) dramatically improved the performance of SCA. More and more researches present ways to build lightweight deep neural network (DNN) models that can retrieve the secret encryption key by analyzing a few power traces of the captured devices. In the meantime, traditional countermeasures are rendered more or less ineffective against these sophisticated SCA.

This research aims to present a novel approach in building SCA countermeasures using adversarial examples, cleverly crafted inputs that trick DNN models and force them to misclassify. As modern SCAs rely on DNN models, an adversarial-based countermeasure could alter the side-channel leakage so that the attacking model cannot identify the correct key. To investigate this approach, we add artificial adversarial noise on an unprotected dataset using evolutionary computation techniques and see how SCAs are affected. We present different methods on how this noise can be designed and how the different parameters of each method affect its performance. Our results indicate that an adversarial-based countermeasure can decrease sufficiently the performance of the attack.

Furthermore, we compare our adversarial-based countermeasures with traditional countermeasures. We show that our proposal can provide equivalent or better protection in some cases, while it presents worse performance than others. Through our systematic research, we identify the reasons behind this weakness and find solutions for future work.

Additionally, we investigate how such a countermeasure can perform against non-profiled SCAs. We show that our adversarial-based countermeasure is effective against this specific family of SCAs, even though it is designed against DNN-based attacks. Comparing this performance to traditional SCA countermeasures, we see that our proposal can be an alternative to noise-based countermeasures, while it fails to provide better security compared to time-based countermeasures.

Preface

Conducting a master's thesis during the COVID-19 was the most challenging period I have encountered. Nevertheless, this journey was something more than rewarding, growing academically and personally. Starting with very little knowledge about hardware security, side-channel attacks, and deep learning, I learned so much about these topics that evolved me as potential security professional. Most importantly, I had the chance to understand more about myself and appreciate the importance of well-being and personal relationships. In this period, there have been many people who helped and supported me, and I would like to take the opportunity to thank them on record.

First, I would like to thank Stjepan for supervising this project. Being always available when I need him, he provided me with guidance on approaching this topic. His advice and insights helped me to get through all the difficulties that I have encountered. A big ”thank you” for being my supervisor.

Second, I would like to thank people for the AISy Lab for helping me in many phases of this research. Lichao, Guilherme, and Luca, I am genuinely grateful for supporting me in any possible way.

Next, I would like to thank all the friends I have made the last three years, during my studies. I could not miss mentioning Fiske, Stefanos, and Chris, who I worked with in many courses and made this journey enjoyable.

Last but not least, I want to thank my family and my girlfriend. Their constant support helped me to get through this challenging period. Everything would be much harder if their were not next to me. Thank you for everything.

Alexandros Korpas Kamarianos
Delft, July 2021

Contents

1
Introduction

1.1. Motivation

Through the last decades, embedded devices are becoming a big part of our daily life. Fitness trackers, smart cards, and digital locks are only a few examples that almost anyone uses daily. Usually, these devices act as an identification key, exchange sensitive information with other devices or handle essential operations, like payments. As malicious entities often target them, they need to perform cryptographic operations to provide the required security level. At the same time, these devices commonly have limited resources. Their constant adoption in our lives and their limitations stimulate the academic community to research secure and efficient cryptographic algorithms for these devices.

An encryption algorithm commonly used by embedded devices is the Advanced Encryption Standard (AES). AES is a symmetric encryption algorithm that is considered secure, as the known cryptanalytic attacks can not recover the full encryption key in a timely manner [78]. However, it is shown that AES implementations are vulnerable to side-channel attacks (SCAs) [5]. This family of attacks does not focus on the algorithm itself but aims to exploit leakages of the running device, such as power consumption or electromagnetic emissions of the microprocessor during the encryption algorithm's execution. These attacks are usually divided into non-profiled and profiled attacks. In the former case, the attacker analyzes the leaked information directly from the target device and tries to recover the encryption key. In the latter case, the attacker first builds a profiler trained to successfully attack a clone device, identical to the target device. When it is time to attack the target device, the attacker can retrieve the encryption key by capturing and analyzing less information from the target device. As these attacks can be practical and surprisingly cheap to perform [22], various countermeasures have been proposed to make the attack harder by providing higher defense or forcing the attack to acquire significantly more leaked information to succeed.

In the meantime, the evolution of deep learning has been proven as a potent tool for various applications. A properly trained deep neural network (DNN) can solve quickly and reliably many classification problems, such as image and text recognition [70, 81], fraud detection [65, 82], spam filtering [14, 85], and regression problems, like traffic prediction [54, 86] and product recommendations [80, 80]. Therefore, it is not surprising that the adaptation of deep learning techniques in the profiled side-channel improved their performance majorly.

As the interest around the deep neural networks is high and the reasons behind their remarkable performance are not fully clear, the academia and malicious entities actively explore in-depth its strengths and weakness. Szegedy et al. [77] presented an exciting discovery: the so-called adversarial examples, namely input that is intentionally designed to make a machine learning model mispredict. Usually, adversarial examples are generated by adding small but cleverly crafted perturbations to the network's original input. Therefore, the human eye can barely distinguish the difference between the two inputs, while the deep neural network fails to predict the correct class/value.

1.2. Research Questions

If we connect dots, an appealing observation emerges intuitively. While the application of the deep neural networks in the profiled SCA rendered the current countermeasures less effective, a novel adversarial-based countermeasure could be a promising alternative. The main questions that we identify are the following:

RQ1: Can an adversarial-based countermeasure be designed so that DNN-based side-channel attacks become impractical regarding the observations needed for secret key retrieval?

RQ2: What is the added value of using our proposed countermeasure compared to existing SCA countermeasures?

RQ3: Can this adversarial-based countermeasure provide security against non-profiling side-channel attacks too?

Implementing an adversarial-based countermeasure is by itself an important contribution. On the one hand, it can force the DNN classifier to mispredict the key, providing solid defense. At the same time, this countermeasure could possibly work against non-profiled SCAs, even though it is designed for DNN-based side-channel attacks. Our ultimate goal is to compare the provided security by an adversarial-based countermeasure with that of existing countermeasures.

1.3. Outline

The following chapters present our research on the above research questions. First, chapter 2 exhibits the relevant background knowledge required for the comprehension of this report. Then, in chapter 3, we show previous work related to our topic, and we formulate our research questions and subquestions. In chapters 4 to 6, we present our experiments and conclusions of our research. Last, we summarize our conclusions and contributions in chapter 7, giving some ideas on how this project could be used for future research.

2
Background

This chapter provides background information for this research project. First, we present the basic cryptography concepts, focusing on the current encryption standard (AES). Next, we present the evolutionary computation and the differential evolution algorithm. Followingly, the concepts of machine learning, deep learning, and adversarial examples are explained. Furthermore, we present the side-channel attacks, which is the research area of this project. Last, we give a short overview of the used datasets for our experiments.

2.1. Cryptography

The term cryptography originates from the Ancient Greek words *κρυπτός*, which can be translated as *secret* or *hidden*, and *γράφειν*, which means *to write*, and is about constructing protocols that protect the confidentiality, the integrity, and the authenticity of sensitive information. While the first cryptography schemes were invented in ancient times, cryptography was majorly advanced during the 20th century; first because of the high demands of private communication during World War I and World War II, and then because of the rapid technological evolution and the advance of computational resources. Nowadays, cryptography is an essential component of secure digital communications ensuring the confidentiality and the integrity of transmitted information throughout the internet [19].

Cryptography consists of two main processes; encryption and decryption. The former transforms a readable message (plaintext) into an incomprehensible one (ciphertext), usually by following a series of mathematical operations. At the same time, the latter is the reserve process where the ciphertext is transformed back to the initial message [71, p. 119]. These two processes, especially the second one, require a secret key so that a third party, usually called an adversary, cannot retrieve the initial message. According to the number of the used keys, we define two main cryptography schemes; *symmetric-key cryptography* and *asymmetric-key cryptography* [71, p. 119].

During symmetric-key cryptography, a sender, usually named Alice in the literature, encrypts a message using a private key and sends it to the receiver, usually called Bob. Assuming that Bob is aware of the private key, he can decrypt the received ciphertext and read the message. On the other hand, during asymmetric-key cryptography, Bob generates a pair of keys; a public key used only for encryption and one private key used only for decryption. When Bob shares the public key with Alice, she can encrypt a private message that only Bob can decrypt using his private key.

In principle, both approaches might be considered mathematically secure –in practice, their security relies on the used algorithm– as the adversary, often named Eve, cannot retrieve the transmitted messages without knowing the private key [38]. However, the two approaches present significant practical differences. Even though symmetric-key encryption is highly efficient computationally, it requires an existing secure channel where Alice and Bob can exchange the symmetric private key. On the other hand, asymmetric-key cryptography is very convenient as Bob can share his public key to Alice without the need for a secure channel, but the mathematical nature of the asymmetric key adds a computational overhead to the decryption phase [38]. Thus, a *hybrid cryptography* scheme is used in practice, where Alice and Bob are using the convenience of asymmetric-key cryptography to agree on a symmetric-key algorithm and a private key and, then, the efficiency of symmetric-key cryptography to exchange their messages.

An essential axiom in cryptography is *Kerckhoffs's principle* [71, p. 179] which states that "a cryptosystem should be secure even if everything about the system, except the key, is public knowledge". That axiom is very important for our project, too, as we will show later in this report that side-channel attacks can retrieve the private key of cryptosystems that are considered mathematically secure.

2.1.1. Advanced Encryption Standard

The Advanced Encryption Standard (AES) is an encryption standard established by the U.S. National Institute of Standards and Technology (NIST) in 2001 [23]. It is widely used for the encryption of electronic data transmitted throughout the Internet. It is a subset of the Rijndael algorithm family, developed by Vincent Rijmen and Joan Daemen, proposed to NIST's AES selection competition in which it was selected among fifteen other competing algorithms [15]. AES defines three encryption algorithms, AES-128, AES-192, and AES-256, where the number on the name indicates the private key's bit size.

AES is a symmetric-key block cipher, where its input is in fixed-sized blocks of 128 bits. The algorithm operates on a $4x4$ array of bytes called "state" and initialized by the input block. AES consist of a certain amount of rounds determined by the size of the key; ten rounds for AES-128, 12 rounds for AES-192, and 14 rounds for AES-256. First, the algorithm uses the `KeyExpansion` operation, which derives some 128-bit rounds keys –as many as the rounds of the algorithm plus one– using the cipher key and a fixed key scheduling routine. During the rest of the algorithm, there are in total four primitive operations that modify the last computed state in a certain way, that are described briefly below:

- `SubBytes`: a non-linear substitution step where each byte of the input state is replaced by another byte using an 8-bit substitution table, usually called SBOX. This SBOX is derived by the multiplicative inverse over $GF(2^8)$ and provides non-linearity to this operation.

- `ShiftRows`: a transposition step where the last three rows are shifted cyclically for one, two, and three steps accordingly.

- `MixColumns`: a linear mixing operation that operates on the columns of the state, where the four bytes of each state's column is multiplied with a fixed $4x4$ array.

- `AddRoundKey`: each byte of the state is combined with a byte of the round key using bitwise xor.

In total, the order of the executed operations is described followingly, and it is schematically presented in figure 2.1. First, before the first round, the algorithm executes the operations `KeyExpansion` and `AddRoundKey`. Then, from the first round to the round before the last one, it repeats cyclicly the operations `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. Finally, during the last round, the algorithm skips the operation `MixColumns` operations, and he only executes the operations `SubBytes`, `ShiftRows`, and `AddRoundKey`.

The first key-recovery attack on full AES, namely when all rounds are completed, was presented by Bogdanov et al. [4]. In this work, the authors designed a biclique attack that recovers the key almost four times faster than a brute-force attack. More specifically, this attack requires $2^{126.2}$, $2^{190.2}$, and $2^{254.6}$ operations to recover an AES-128, AES-192, and AES-256 key, respectively. These results were improved later by Tao and Wu [78] to $2^{126.0}$, $2^{189.9}$, and $2^{254.3}$, accordingly, which is the best known key-recovery attack so far against full AES. Since the acceleration of this attack is tiny, AES is considered very secure from a mathematical perspective. However, as we will show in later sections, AES is quite vulnerable to side-channel attacks.

2.2. Evolutionary Computation

Evolutionary computation is a family of algorithms inspired by the Darwinian evolution theory that combine "probabilistic and deterministic driving forces for solving optimization, classification and modeling problems" [66]. Evolutionary algorithms have been extensively used for the optimization of many real-world applications, such as local inventory routing [35], vehicle routing and scheduling [56], and task scheduling [24].

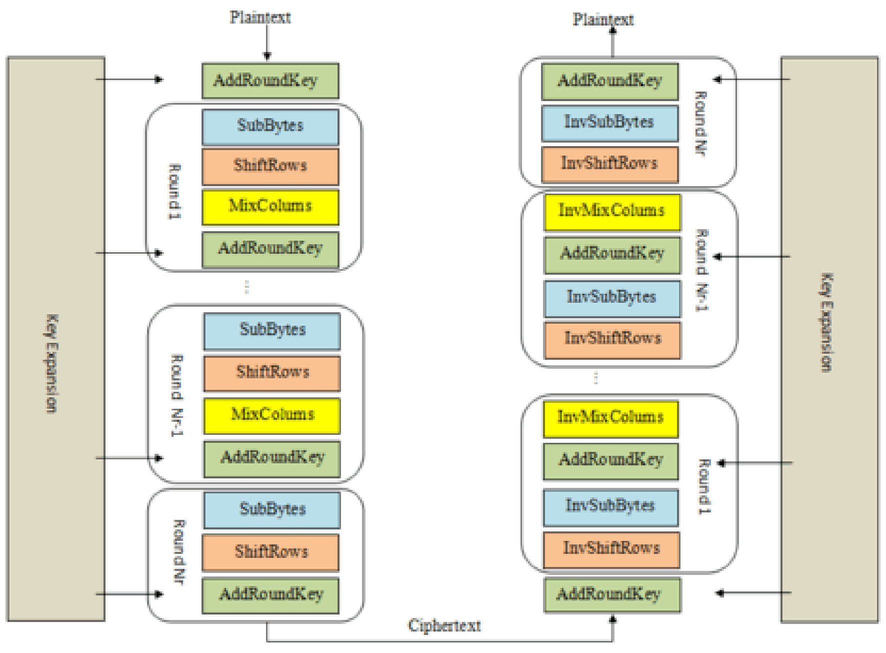


Figure 2.1: High-level description of AES [11]

First, the evolutionary algorithms (EA) randomly generate an initial population of candidate solutions (individuals). Then, this population is evolved using specific processes called *mutation*, *recombination*, and *selection*. Finally, after some iterations, the candidate solutions are evolved toward the optimal state. Below, we present the operation of these processes briefly.

Mutation Mutation operation refers to the biological mutation that happens naturally on a cell's DNA or RNA, usually during its reproduction. Similarly, in evolutionary computation, the EA mutates the parameters of the population's individuals by adding a random number.

Recombination Recombination (or crossover) operation refers to the biological process where two organisms combine their genetic material to generate new offspring that contain genetic traits that differ from their parent. In evolutionary computation, EA randomly combines the parameters of two individuals to breed new candidate solutions.

Selection Selection refers to the natural selection or "survival of the fittest" process. In evolutionary computation, EAs use the objective function that has to be optimized to evaluate the fitness of the candidate solutions. Then, they eliminate replacing the least-fit individuals of a population with the ones generated by the mutation and crossover processes.

There are many subcategories and algorithms in the field of evolutionary computations. In this project, we will use the Differential Evolution algorithm that is presented below.

2.2.1. Differential Evolution

Differential Evolution (DE) was firstly introduced by Storn and Price [75]. It is a global optimization algorithm that is used for continuous and discrete optimization problems. In their work, the authors introduce a new mutation technique called *differential mutation*. Then, they present several versions of the DE algorithm that use different variations of differential mutation and recombination. In this section, we use the classic DE (also called *DE/rand/1/bin* by the authors) [75, p. 209] as a black-box algorithm for our optimization problems. Below, we present in detail how classic DE works.

Problem statement We define a function $f : \mathbb{R}^D \to \mathbb{R}$ (objective function) that has to be minimized. Goal is to find a solution $m \in \mathbb{R}^D$ that $f(m) \leq f(p)$ for all $p \in \mathbb{R}^D$.

Parameter selection We choose the parameters $Cr \in [0,1]$, $F \in [0,2]$, $N \geq 4$. Cr is called *crossover probability*, F is called *differential weight*, and N is the population size.

Initialization We first initialize randomly all the population's individuals within the search-space $\mathbb{R}^D$.

Execution step For each individual $x \in \mathbb{R}^D$ (called target vector):

- We pick three *different* individuals $a, b, c \in \mathbb{R}^D$ from the population at random (a is called base vector).
- We pick a random index $j_{rand} \in 1, 2, ..., D$.
- We generate a new individual $u \in \mathbb{R}^D$ (called trial vector) as follows:
  - We compute the a random mutant $v = a + F(b - c)$
  - For $j \in 1, 2, ..., D$, we compute:

$$u_j = \begin{cases} v_j & \text{if } U(0,1) \leq Cr \text{ or } j = j_{rand} \\ x_j & \text{otherwise} \end{cases}$$

- If $f(u) \leq f(x)$, we replace the target vector x in the population with the trial vector u.

Theoretically, we can repeat the execution step until the global minimum m is found. However, this is not doable in practice since DE cannot guarantee that the trial vector will become the global minimum. Therefore, we usually run the DE algorithm for a certain number of iterations. As a result, we consider that the returned value is close but not always equal to the global optimum m.

2.3. Machine Learning

The term "Machine Learning" (ML) refers to the field of artificial intelligence where an algorithm extracts knowledge from given data and learns how to perform a specific task[57]. Machine learning has presented a remarkable performance in various tasks, such as speech recognition [18, 58], anomaly detection [1], spam filtering [30], and medical prognosis and diagnosis [45]. In practice, ML algorithms can identify patterns on a dataset that can be either labeled or unlabeled. In the former case, the ML algorithm combines the given data and the labels to identify the patterns (supervised learning). In the latter case, the ML algorithm extracts knowledge using only the given data (unsupervised learning). This project focuses on supervised learning and, more specifically, its application to a specific type of task called classification tasks.

In a classification task, the goal is to identify how a given input x, called a feature vector, and consists of n features ($x = (x_1, x_2, ..., x_n)$), is mapped to a class y. This mapping is described by an unknown function f where $y = f(x)$. The goal of the classification machine learning algorithms is to build a function $\hat{f}$ that returns a predicted class $\hat{y}$ ($\hat{y} = \hat{f}(x)$) and approximates the function f. The algorithm's performance is measured using the accuracy metric that represents the percentage of correct predictions ($y = \hat{y}$) over several predictions.

The way machine learning uses a dataset to solve a task differs from algorithm to algorithm. In this project, we focus on a specific family of machine learning, deep learning. In the next section, we present in detail how deep learning works. Last, we present a major weakness of the deep learning (and machine learning) algorithms, the adversarial examples.

2.3.1. Deep Learning

In the last decade, deep learning has been rapidly developed and succeed in outperforming traditional machine learning. Traditional ML algorithms suffer from handling high-dimensional, which results in an exponential increase of the execution time. Therefore, a feature selection is a necessary preprocessing step for these algorithms to decrease the dimensions of the input vector. At the same time, deep learning algorithms perform fast and scalable operations. As a result, deep learning achieves high performance in a short amount of time without the need for a prior feature selection [27]. Deep learning is widely used in various tasks, such as image recognition [20], autonomous driving [25], and natural language processing [53].

Deep learning is inspired by the biological neural networks of the human brain. The fundamental components of deep learning are called neurons, while a group of neurons is a called layer. A neuron is a computational unit with m inputs and an output and has a pair of *trainable* parameters; a weight vector associated with each input and a bias. When a neuron receives an input signal $x = (x_1, x_2, \dots, x_m)$, it uses its weight vector $w_k = (w_{k_1}, w_{k_2}, \dots, w_{k_m})$ and its bias b_k (when applicable) to compute the summation $v_k = \sum_{i=1}^{m} x_i w_{k_i} + b_k$. Then, the result v_k is passed through a non-linear function called activation function ϕ. The result of the output function is the output y_k of the neuron ($y_k = \phi(v_l)$. Figure 2.2 visually presents the structure of a single neuron.

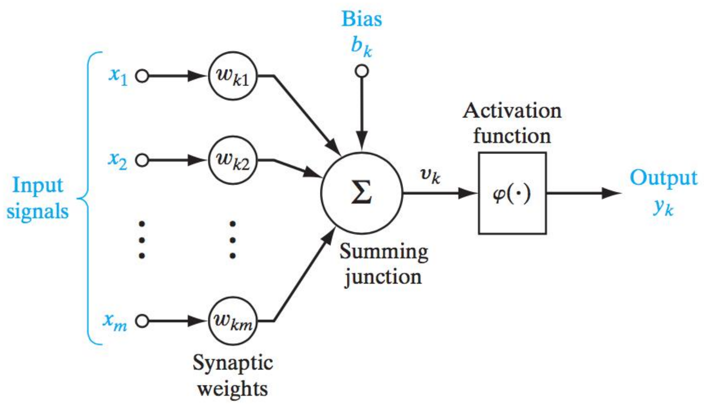


Figure 2.2: Computation performed by a single neuron[32]

A structure of connected layers is called a neural network. Each layer's neurons are connected with neurons from another layer depending on the neural network's architecture. The most common neural network architectures are the multilayer perceptron (MLP) and convolutional neural network (CNN) architectures.

In the rest of this section, we first present the different activation functions of the neurons used in this project. Then, we present the MLP and CNN architectures. Finally, we present how deep learning uses a dataset to extract knowledge during the training phase and tunes the trainable parameters of a layer to build a robust neural network classifier.

Activation Functions

As explained above, a neuron first computes the weighted sum of the input signals and then adds a bias. This process is entirely linear. At the same time, most of the classification problems are non-linear. Therefore, neurons apply a non-linear function on this summation that introduces the non-linearity property, which results in the neuron's output. This non-linear operation is called the activation function.

There are several activation functions. Below, we present the most common ones that are also used in this project: the ReLU, ELU, SELU, sigmoid, tanh, and softmax functions.

Rectified Linear Unit (ReLU) The ReLU function is a simple activation function that returns the input if it is positive or, otherwise, zero. Its mathematical definition is presented in equation 2.1.

$$ReLU(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = max(0, x) \tag{2.1}$$

Exponential Linear Unit (ELU) Similar to ReLU, ELU returns the input if it is positive. However, for negative inputs, ELU approaches smoothly $-\alpha$. Its mathematical definition is presented in equation 2.2.

$$ELU(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2.2}$$

Scaled Exponential Linear Unit (SELU) SELU is very similar to ELU, with the only difference that it is scaled using a parameter λ. Also, the α and λ are fixed parameters determined by the input distribution. For standard scaled inputs (mean 0, stddev 1), the values are $\alpha = 1.6732$, $\lambda = 1.0507$. Its mathematical definition is presented in equation 2.3.

$$SELU(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2.3}$$

sigmoid Sigmoid is an S-shaped curve with values in the range $(0, 1)$, presented in equation 2.4.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

Hyperbolic tangent (tanh) Hyperbolic tangent is another S-shaped curve with values in the range $(-1, 1)$. Its mathematical definition is presented in equation 2.5.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

Softmax Contrary to the other activation functions that use only the output of a single neuron, softmax uses the output of all the neurons of the layer. It is used to scale the outputs of the layer's neurons so that they are all between zero and one, and their sum is one. It is usually used as the activation function of the output layer in classification problems to convert the output of the hidden layers to a probability for each class. Its mathematical definition is presented in equation 2.6.

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.6}$$

Multilayer perceptron (MLP)

The architecture of an MLP model is standard. We logically divide the layers of the MLP model into three categories: the input layer, the hidden layer(s), and the output layer. The input layer's size (i.e., the total neurons) is equal to the feature vector's size. Next, the hidden layers are fully connected; namely, all the layer nodes are connected with nodes of the previous layer. In case that there is exactly one hidden layer, the neural network is called a swallow. Otherwise, it is called a deep neural network (DNN). Finally, the output layer is also a fully connected layer with a size equal to the number of class classification classes. The structure of a small MLP neural network is presented in figure 2.3.

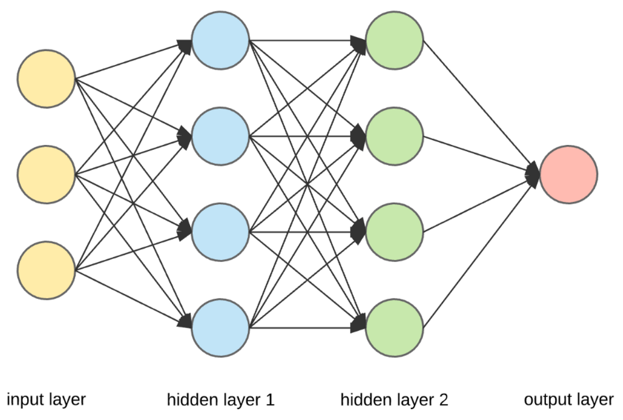


Figure 2.3: A simple MLP neural network

Convolutional neural network (CNN)

CNN models have an input and output layer with the same specification as the MLP models. However, the structure of the hidden layers is not standard. The minimum requirement for a model to be a CNN model is that a least one convolutional layer [27, p. 321]. The rest of the hidden layers is usually a combination of different types of layers. Below, we present the type of layers relevant to this project: the convolutional, pooling, flatten, and fully connected layers.

Convolutional layer Unlike the fully connected layers, convolutional layers' neurons are not connected with all the neurons of the previous layer. Instead, they perform the convolution operation. As presented in figure 2.4, the convolutional layer uses a small matrix with trainable parameters called kernel or filter. Then, it takes the input, divides it into equally sized chunks, and computes the sum of the element-wise product of the two matrices. An important parameter of the convolutional layer is the stride, which is the step that the kernel slides for each convolution operation.

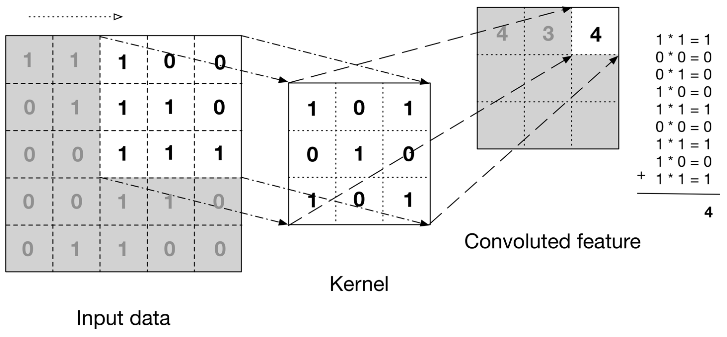


Figure 2.4: Example of a 2D convolution[87]

Pooling layer A convolution layer is usually followed by a pooling layer [27, p. 330]. The pooling layer combines small chunks of the input's features by computing their maximum (max pooling) or average (average pooling) value. That allows the CNN model to be more generalized and more robust to small changes of the input. In figure 2.5, we present examples of maximum and average pooling.

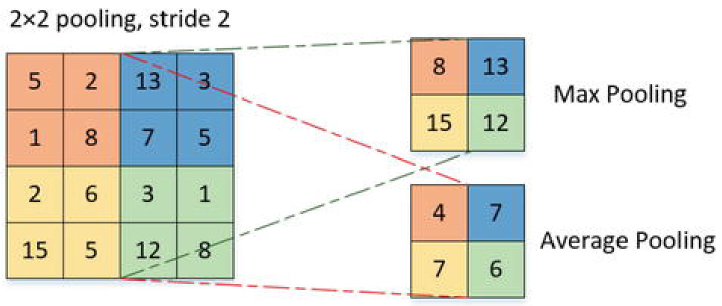


Figure 2.5: Example of max and average pooling[21]

Flatten layer As its name indicates, the flatter layer reduces the depth of the layer's input to one. It is usually used after the convolutional and pooling layers as they usually operate with multidimensional vectors. In this way, the input is flattened and prepared to be processed by the next layers.

Fully-connected layer As presented in the MLP architecture, fully connected layers are those that their neurons are connected with all the input features.

Training & Testing phases

As explained earlier, a classification problem aims to build a function $\hat{f}$ that approximate the behavior of an unknown function f. In deep learning, this function $\hat{f}$ is represented by a DNN model. Upon its creation, the parameters of the model (neuron's weight and bias) are randomly initialized. It is logical that these random parameters are insufficient to approximate the function f sufficiently. Therefore, the model's parameters need to be tweaked so that its predictions approximate the behavior of the unknown function f. That can be achieved through the training process.

The training process of a DNN model is an optimization problem; we want to minimize the distance between the functions f and $\hat{f}$. To achieve this, we use a loss function to measure this distance using the model's parameters. Then using an optimization algorithm called gradient descent (equation 2.7, we can iteratively update the model's parameters so that the loss is minimized. This optimization algorithm uses a process called back-propagation that calculates the derivative of the loss function with respect to trainable parameters. Then, the gradient descent algorithm subtracts the derivative from the original parameters[1]. By repeating this process multiple times (epochs), gradient descent succeeds in minimizing the loss function.

$$w_{n+1} = w_n - \gamma \nabla L(w_n) \tag{2.7}$$

The parameter γ is called the learning rate and affects how big or small steps are the gradient descent steps in each iteration. A common practice is to use an optimizer method that updates the learning on each training epoch.

After the training phase, there is the testing phase. First, the model uses the input of a testing dataset to make its predictions. Then, using the true labels of the testing dataset, the model's accuracy can be computed.

Hyperparameters

Next to the model's parameters, another equally important factor for the model's performance is its hyperparameters. On the one hand, there are the architecture hyperparameters related to the neural network structure (number of hidden, layer's size, activation function, type-specific parameters). On the other hand, there are the training hyperparameters related to the training process (epochs, learning rate, optimizer).

Architecture hyperparameters are non-trainable and cannot change after the neural network is built. At the same time, they are essential for the performance of the neural network. Therefore, there has been a lot of research on hyperparameter optimization.

2.3.2. Adversarial Examples

Adversarial examples (AE) refer to specially crafted input designed in a way that looks normal to a human but forces a machine learning model to misclassify. They have been firstly introduced by Dalvi et al. [16] for traditional machine learning, while Szegedy et al. [77] presented their first application against deep learning models.

Adversarial examples have to look like the original input. Thus, the added perturbation has to be small. The l_p distance metric is usually used to measure the added perturbation, which is presented in equation 2.8. The most commonly used are the l_0, l_2, and l_∞. The l_0 metric represents the number of changed features in the adversarial example. The l_2 metric describes the euclidean distance between the original input and the adversarial example. The l_∞ metric represents the maximum change among all the features of the adversarial example [88].

$$l_p = \|x\|_p = (\sum_{i=1}^{n} \|x_i\|^p)^{\frac{1}{p}} \tag{2.8}$$

The generation of an adversarial example is an optimization problem. The goal of this optimization is to minimize the DNN's prediction for the true class. Thus, an adversarial method uses an optimization technique to distort the original input and force the DNN model to misclassify. In the rest of this section, we present two adversarial example generation methods; the fast gradient sign method and the one-pixel attack.

[1]A derivative of a function indicates the direction that the function increases. Thus, its negative shows the direction that function decreases

Fast gradient sign method

The most simple and fast adversarial example method is the *fast gradient sign method* (FGSM) presented by Goodfellow et al. [28]. In practice, FGSM uses the gradient descent method to compute the gradients of the neural network and design an adversarial example using the equation 2.9, where x' is the adversarial example, x is the original input, y is the original output, θ is the model's parameters, J is a loss function, ϵ is the magnitude of the added distortion.

$$x' = x + \epsilon * sign(\nabla_x J(\theta, x, y)) \tag{2.9}$$

An example of this method is illustrated in figure 2.6. In this example, we see that the original image is classified correctly as a panda. After adding a small noise that is not visible to a human eye, the adversarial example is classified incorrectly as a gibbon.

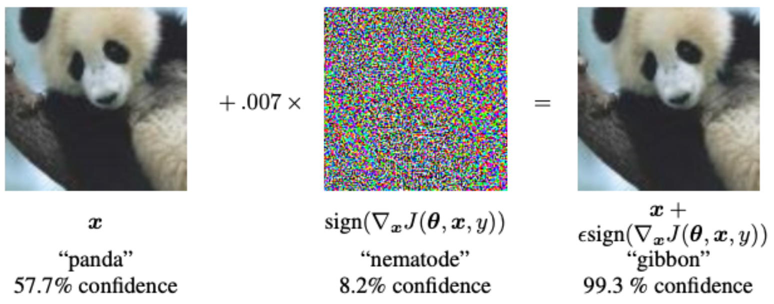

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 2.6: An example of the FGSM method

One-pixel attack

Another interesting method for AE generation is the one-pixel attack presented by Su et al. [76]. In this project, the author presents a method that changes only a single pixel of the original image to design an adversarial example. The process to find which pixel needs to change and the final color is a discrete optimization problem, as the pixel position is not continuous. For this reason, the authors use the different evolution algorithms presented in section 2.2.1. In figure 2.7, we see some examples from the original paper.

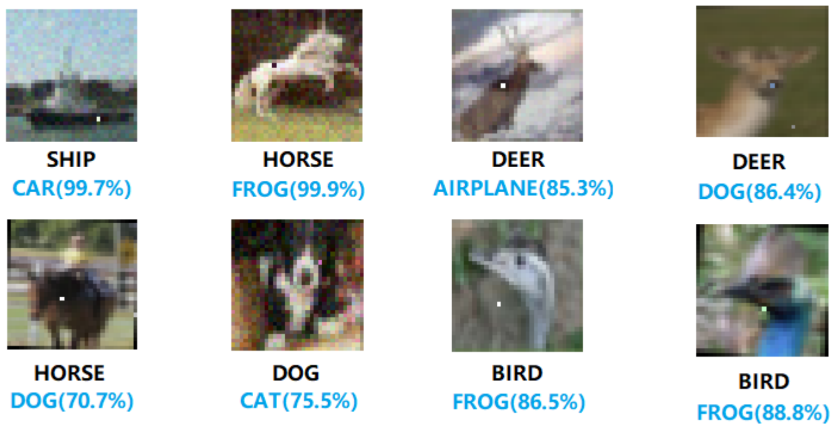


Figure 2.7: Examples of the one-pixel attack. The black label is the true class of the input, while the blue label is the predicted class, alongside with the prediction confidence. [76]

2.4. Side-channel attacks

Side-channel attacks (SCAs) are a family of passive attacks where the attacker does not focus on weaknesses related to a targeted cryptosystem, but he uses information leaked by the running computing system in secondary channels. These channels can be electromagnetic emissions, power consumption, timing information, or even sound generated by the targeted device. For the recovery of the full key, side-channel attacks are following the divide-and-conquer approach, namely, each byte of the secret key (subkey) is retrieved separately.

Side-channel attacks are mainly divided into two categories; the non-profiled and the profiled side-channel attacks. In both categories, the targeted device is considered as an encryption oracle, i.e., the attacker can request from the cryptosystem to encrypt messages of his choice while he gathers information from the leakage channel, for example, the power consumption of the microprocessor. However, the main difference between the two categories is the way that this information is used for the extraction of the secret key. Briefly, in the former category, the attacker uses a certain heuristic function to analyze the acquisitions from the targeted device in addition to the known plaintext and ciphertext to find key candidates. On the other hand, in the latter attacks, before the attacking phase, the attacker uses a clone device that has the same technical specifications as the target devices to create a profiling model. Since the attacker can manipulate the encryption key of the clone device, he can train his profiling model to match the leaked information of his device with the various keys. Then, using a small number of acquisitions from the targeted device, his profiling model can retrieve the secret key.

Side-channel attacks are known to the scientific communities for years [43]. The advance of machine learning and, more significantly, deep learning have only improved their efficiency in the last decade. At first, published works aimed DES, AES predecessor, but since the launch of the AES and its wide adoption, AES has been heavily scrutinized and has been proven extremely vulnerable to side-channel attacks that exploit the EM radiation or the power consumption of the device's processor during the encryption. In practice, these attacks exploit the non-linearity of the `SubBytes` operation in the first round. Since the attacker can feed the cryptosystem with plaintexts of his choice, this non-linear operation evinces the position of the operation on the captured observations and allows the attacker to model the leakage according to the used plaintext and eventually retrieve the secret key. Due to the popularity of these attacks, this project focuses on side-channel attacks on AES that attack at that certain step of the algorithm.

The nature of this attack requires knowledge of the system and usually physical access to the device so that the leaked information can be captured and analyzed for the extraction of the cryptosystem's secret key. As Kerckhoff's principle demands that the security of a cryptosystem should rely exclusively on the secrecy of the private key, all these requirements should consider public and, thus, these leakages render side-channel attacks a serious threat of the modern implementations.

2.4.1. Leakage Models

The power consumption of electronic devices depends on two factors. First, there is the static power consumption that is the power that the device demands to run and is relatively constant. Then, there is the dynamic power consumption that occurs by processing data during the runtime. Every time the processor needs to flip a bit from zero to one or vice versa, a small but significant current must charge or discharge the data lines. Side-channel attacks aim to model the dynamic power consumption using functions called leakage models.

A leakage model focuses on a specific point of the encryption algorithm where a register changes from a state A to another state B and assumes that the leaked observations of the power consumption are proportional to that transition. Regarding AES, side-channel attacks usually focus on the output of the SBOX during the first round, usually called intermediate value. Depending on the used leakage model, this value is mapped to a class representing the device's leaked information.

The common leakage models used in the side-channel attacks are the identity (ID) model and the Hamming Weight (HW). The ID model is directly related to the intermediate value, and, thus, it has 256 classes. On the other side, the HW model classifies the intermediate value depending on the amount of one's of its binary representation, and it is used mostly because of its small number of classes. As the possible amount of ones are within the range $[0, 8]$, this model has nine classes which decrease the attack's complexity. The significantly larger amount of classes of the ID model compared to the HW model leads to more observations for constructing a strong leakage model. However, the ID model is usually stronger as it provides higher precision. For these reasons, non-profiled attacks often use the HW model as the attack needs to be fast, while profiled attacks are using the ID model since the processed data are usually imbalanced.

2.4.2. Non-profiled Attacks

Some of the very first non-profiled side-channel attacks that have been proposed for DES are the simple power analysis (SPA) and differential power analysis (DPA) [42]. SPA relies on the visual interpretation of the power consumption of the targeted device that can provide useful information about the order of the execution of certain operations, such as repeating patterns that can reveal the number of iterations of a for-loop. On the other hand, DPA performs statistical analysis on the observations taking advantage of the data dependencies in the power consumption patterns [73]. With the introduction of AES, Brier et al. [5] proposed the correlation power analysis (CPA) that extends the aforementioned attacking methods by using them, and it is considered the most effective non-profiled attack on AES. The key component of this attack is the Pearson correlation function presented in equation 2.10.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}} \tag{2.10}$$

For a correlation power analysis, the attacker uses the target device to encrypt N different plaintexts and records the power consumption during each encryption. Then, using the divide-and-conquer strategy, he attacks each byte of the key using the following approach:

1. For all the possible options of the subkey (candidates), the attacker:
   (a) computes the modeled power consumption for each trace based on the selected leakage model using the corresponding plaintext and the key candidates.
   (b) computes the Pearson correlation between the modeled power consumption and the actual power consumption for every tracepoint.
   (c) sets as the correlation of the specific candidate the maximum observed correlation (absolute value).
2. The attacker picks the candidate with the highest correlation for his guess.
3. The attacker repeats steps 1 and 2 to get the best candidates for each subkey.
4. The attacker constructs his guess for the full key using the best subkey candidates.

2.4.3. Profiled Attacks

Profiled attacks take advantage of the attacker's knowledge regarding the specifications of the targeted device and introduce a prior phase to the attacking phase, called the profiling or training phase. In profiled attacks, the attacker builds a profiler using a clone device that runs the same encryption algorithm on the same (or equivalent) hardware. According to Kerckhoff's principle, these specifications should be considered known, and the cryptosystem's security should not rely on these.

Profiled attacks consist of two phases, the profiling phase and the attacking phase. During the profiling phase, the attacker uses the clone device to encrypt plaintexts of his choice using keys, again, of his choice that can be either constant or variable, while he captures the side-channel observations (profiling traces). Using the profiling traces, he uses a learning algorithm to train a profiler who learns how a trace should "look like" for each class of the leakage model. Coming to the attacking phase, where the key is unknown, the attacker uses the targeted device as an encryption oracle and encrypts plaintexts of his choice, and captures the side-channel observations (attacking). As the attacker knows the plaintext, he can predict the probability of each leakage model's class for a given trace using the trained profiler.

The main difference between the different types of profiled attacks lies in how the profiler is learned. Below, we present briefly the most common learning algorithms used for the training of the profiler.

Template attacks Template attacks are one of the oldest profiled attacks, firstly introduced by Chari et al. [10] in 2002. Template attacks describe the side-channel attack as a multivariate Gaussian distribution problem and construct a probability density function using the captured traces that describes the probability of each leakage model's class based on the values of a small number of tracepoints (POIs). The selection of the POIs can be made using various feature selection algorithms (sum of differences, Pearson correlation, etc.) that are out of this project's scope.

Traditional Machine Learning As presented in section 2.3, machine learning algorithms present remarkable performance on the classification problems in different domains such as image, sound, and natural language processing. Therefore, many researchers used the most common classification machine learning algorithms, such as Random Forest, Decision Trees, k-Nearest Neighbors, and Support vector machines [33, 34, 49, 50, 60, 62]. Applying these classification algorithms to the side channel domain is straightforward, as the training and testing phases are used for the profiling and the attacking phases, respectively. The adaptation of classic machine learning into the side-channel analysis domain improved the performance of the SCAs significantly by reducing the required traces to guess the correct key [34, 59].

Deep Learning In general, statistical and machine learning algorithms suffer on scalability and need to execute a feature selection algorithm in the preprocessing phase to keep the input dimensionality low[51]. On the other hand, deep learning techniques can process larger input in a shorter time while presenting better performance than their predecessors. Thus, the adaption of deep learning in the side-channel analysis domain stands to reason. By mapping the training and testing phases with the profiling and the attacking phases, respectively, the Deep neural networks (DNN) have been extensively used in the last few years SCAs [6, 55, 61, 64]. Section 3.1 presents in detail the related work in the application of deep learning in the side-channel domain.

2.4.4. Performance Metrics

To evaluate the performance of side-channel attacks, there are two main metrics; the guessing entropy (GE) and the success rate (SR). For the computation, the evaluator performs the attack, but instead of using the best candidate as a prediction of the secret subkey, he uses a guess vector, which is an ordered list of all the candidates (leakage model's classes) based on their correlation (for CPA attacks) or their likelihood (for profiled attacks) in decreasing order. These metrics are usually presented in plots where the x-axis represents the number of the used attacking traces and the y-axis the measured metric. In this way, the evaluator can measure the performance of the attack in relation to the used attacking traces. Even though these metrics refer to attack to the whole secret key, it is common that the evaluator executes the performance evaluation only for one subkey under the assumption that the results would be similar for the rest of the subkeys.

The GE and SR metrics were firstly defined mathematically under the SCA context by Standaert et al. [74]. In this section, we knowingly avoid the formal definition and explain their meaning in simple words, so it is more understandable to the reader.

Guessing entropy The guessing entropy describes the trials that an attacker needs to find the secret (sub)key. The guessing entropy is equal to the average rank of the true (sub)key in the guess vector over a relatively large number of attacks (usually 100). Rank is the position of the true (sub)key in the guess vector after a single attack with starting from either 0 or 1 (in our project, we use the second approach).

Success Rate The success rate describes the percentage of an attack to retrieve the correct (sub)key with the first trial. To compute the success rate, the evaluator executes a relatively large number of attacks (usually 100) and computes the percentage of those whose true (sub)key is placed first for the guess vector.

2.4.5. Countermeasures

Since the first appearance of the side-channel attacks, there has been a lot of research on defensive techniques that protect the secret key entirely or decrease the performance of the attack by increasing the amount of the required attacking traces for the key retrieval. However, as we explain in section 3.1, the introduction of deep learning techniques in SCAs decreased the performance of these countermeasures significantly. The SCA countermeasures are divided into two families, hiding and masking countermeasures, and presented below.

Hiding countermeasures

Hiding techniques are those that aim to decrease the side-channel leakage by altering the captured observations in a way that decreases their correlation with sensitive information. These techniques affect the captured traces, they can often be simulated for research purposes by applying them a posteriori in a clean dataset. Some of the most common hiding techniques that will concern us during this particular project are presented below.

Gaussian noise The presence of Gaussian noise in observations is natural, as all the electronic components in a device's circuit and the measurement equipment can produce Gaussian noise. That noise decreases the signal-to-noise ratio (SNR), decreasing signal processing algorithms' performance [17].

While this effect is negative in other domains, it is often used for security purposes in the side-channel analysis domain. This is because the reduction of the SNR leads to the proportional drop of the correlation between the captured power traces and the sensitive information. Hence, the attacker usually needs to capture more observations from the targeted device to run a successful attack. This countermeasure is implemented by the introduction of a Gaussian noise generator inside the device's microprocessor.

Desynchronization An important pre-processing step for side-channel attacks is synchronizing the captured traces, namely the alignment of similar patterns on the time domain. That step aims to align the same parts of the executed algorithm, increasing the correlation of the sensitive data and improving the attack's performance. For a successful synchronization, a reference point is required, which is usually an easily detectable pattern of one trace. Nevertheless, the identification of such as pattern is not always easy, mainly for two reasons. First, this pattern has to be easily distinguished from others; otherwise, different steps of the algorithm –and thus different sensitive data– with low correlation will be aligned together. Furthermore, these patterns have to be around at the same area of the x-axis, as an exhaustive pattern matching across the whole length of each trace would forbid the application of this pre-processing step in a large number of traces. Consequently, synchronization of a dataset is not a trivia task, even in the case of an unprotected device.

Regarding the side-channel domain, one could exploit the complexity of synchronization and introduces delays of arbitrary length before each execution of the encryption algorithm. That technique is called desynchronization and, in essence, forces the attacker to execute the difficult and time-consuming step of synchronization to perform a successful attack. To implement this countermeasure, the defender introduces small arbitrary delays before each encryption [83].

Random delay interrupts (RDIs) Similar to the desynchronization technique, random delay interrupts add randomness in the time domain of the traces. However, instead of adding a global delay to every trace, the RDIs split the traces into fragments, and the random delay is placed locally in between these fragments. Hence, the misalignment of similar patterns is increased, and the correlation of the captured traces drops significantly [31]. This countermeasure is applied at the software level, where the algorithm randomly decides to execute delay routines of arbitrary length during the execution of sensitive parts.

Clock jitters A clock jitter is the natural deviation of the true periodicity of a microprocessor's clock from its theoretical frequency. However, they are also a classical hardware countermeasure against side-channel attacks that aim to introduce randomness in the time domain of every point of the captured traces. That leads to significant deformation of the clean trace and, as a result, to a proportional drop in the correlation of the captured observations with the sensitive data [6].

Masking countermeasures

Masking techniques aim to decrease the side-channel leakage by applying a mask to the sensitive parts of the encryption algorithms (intermediate values in case of AES). The application of the masks randomizes the mapping of the sensitive information with the leakage [7, 13, 72]. A simple and popular masking technique that provides security against first-order attacks, like those described above, is to perform an XOR operation between the sensitive data and the mask. For example, in the case of AES, the masked intermediate value for the i-th byte of the key/input can be computed as:

$$masked\_SBOX = SBOX[plaintext_i \oplus roundkey_i] \oplus mask_i \tag{2.11}$$

That mask can be random, fixed, or rotated [3, 72, 79]. As a result, the attacker has to design a leakage model that includes the mask, increasing the space complexity of the side-channel attack. Furthermore, the defender can adopt masking techniques with multiple masks, providing higher-order defense to his device [40, 68, 69]. More formally, the usage of N masks provides N-order security against the traditional attacking techniques described above. Consequently, the attack has to design a $N+1$-order attack, namely to recover each mask separately beside the secret key.

2.5. ASCAD

For this project, we use the ASCAD dataset [2, 64], a public dataset widely used by the SCA research community. However, the extracted traces on the ASCAD dataset are protected using first-order masking. Thus, we use the raw dataset used to extract the ASCAD traces, which is also provided publicly by the authors.

ASCAD uses a masked implementation of AES running on an ATMega8515 microprocessor. However, the first two bytes of the mask are `0x00`. That means that we can consider the first two bytes unprotected and, using the ASCAD's raw traces, we can extract an unprotected dataset.

The authors provide two different raw datasets. The first version consists of 60,000 traces (100,000 trace points each) where the key was constant during the whole acquisition process. The second version consists of 300,000 traces (250,000 trace points each) where only $1/3$ of the traces have the same key, and the rest have a variable key. It is considered easier to attack a fixed key dataset rather than a random key dataset. That happens because the registers that hold the secret key are not changing during the execution, and, thus, the traces are cleaner. As a result, the attacker builds a strong profiler that can break the specific key. However, this scenario is not realistic, as the attack does not know the secret key during the training phase. Thus, he usually trains his profiler using a random key. Hence, in our experiments, we use the version with the random key.

3 Related Work

This chapter aims to present the related work in academia, as well as the questions that this particular project is about to examine. At first, we present how deep neural networks managed to overpower proposed countermeasures against side-channel attacks and what is the direction of the current research on DNN-based SCAs. Furthermore, we present works that have a similar direction to our project and point the unanswered questions that motivate us to conduct our research.

3.1. Side-channel attacks based on deep learning

Classical machine learning techniques have been extensively used for side-channel attacks by [34, 36, 49, 59]. These works test how several ML classifiers, such as SVMs, Random Forest, and Naive Bayes, are performing on SCAs, concluding that they can perform better than template attacks most of the time in many different cases, e.g., when the traces are noisy. However, with the introduction of effective countermeasures, such as the masked implementation of AES, the proposed attacks became more time-consuming and thus less efficient [50]. These countermeasures increased the order of the attack; namely, the attacker first needs to remove the mask and then recover the key, which usually requires more power traces and more computational power for a successful attack. Hence, the researchers had to appeal to other more powerful techniques.

With the significant advance of deep learning in the last decade, it stands to reason that the researchers examined DL methods in the side-channel analysis domain. Gilmore et al. [26] were the first that used an MLP to speed up the mask removal. Soon other works followed that showed that DNN-based classifiers, such as CNNs and LSTMs, outperform traditional machine learning models [6, 55], as these models can learn from the raw input and locate the most interesting features without the need of prepossessing techniques. Interestingly, their results show that they retrieve the key directly without prior mask profiling in the case of masking countermeasures [61].

Another win of the deep learning technique against traditional countermeasures for SCAs is presented in the work of Wu and Picek [83]. First, the authors simulated datasets by applying the most common hiding countermeasures a posteriori to a clean dataset and show that the simulated datasets are sufficiently protected against different side-channel attacks. Then, using a deep learning technique called denoising autoencoder as a preprocessing step before the attack, they show that the effectiveness of these countermeasures can be reduced and make the attack feasible.

A notable application of DNN in SCAs is the research of Kim et al. [41]. In this project, the authors propose an attacking strategy where they add noise to a VGG-like architecture. More specifically, after the first batch normalization of the input, they add gaussian noise with mean zero and standard deviation α. This approach improves the performance of the attack significantly and reduces overfitting that commonly happens during the training of the DNNs.

As the usage of DNN in SCAs appears to be promising, Prouff et al. [64] decided to release a public dataset named ASCAD that could work as a standard reference between different researches, similarly to the role of popular datasets on other domains such as the MNIST dataset on the image recognition domain. In other words, this project aims to be a basis for comparing different DNN models proposed by the research community by simply comparing their SCA performance metrics on the ASCAD. Furthermore, they propose two different DNN architectures, an MLP and a CNN, and they tweak different hyperparameters at a time to maximize their performance. This particular research mainly inspires the architecture of the DNN models and the discovery of their hyperparameters in our work.

3.2. Adversarial-based countermeasures

Although adversarial examples for deep learning models were recently introduced in the research community [77], much research has been conducted on various domains and several scopes. Many works are trying to explain the nature of their existence or their transferability [28]. Furthermore, several projects are present different types of attacks [9, 44, 46]. Moreover, some projects present a detection mechanism, namely to distinguish original input from adversarial examples [39, 52]. We can observe a similar pattern in the works mentioned above; adversarial examples are investigated as a weakness exploited from the attacker's side for malicious purposes.

The first time, to the best of our knowledge, that adversarial examples were used for defensive purposes was in the brief research conducted by Inci et al. [37]. The authors used Hardware Performance Counters (HPCs) of the targeted CPU as a leakage channel, namely special purpose registers that provide low-level information about the CPU, such as the number of cache hits and misses. These observations allowed them to retrieve secret keys from various ciphers from the OpenSSL library by using various ML and DNN classifiers. Then, they applied several methods adversarial methods to the captured observations and generated adversarial examples. Their approach was reasonably effective, according to their findings, as the attacker's classifier failed to predict most of the time correctly. The most interesting point of their work was their approach to the generation of adversarial examples. The targeted device posses a classifier that constantly uses various defenses against adversarial examples. Since this classifier is used for the generation of the perturbation, these defensive techniques result in crafting powerful adversarial examples that can force the attacker's classifier to misclassify.

Another work that proposed an adversarial-based countermeasure is the one by Picek et al. [63]. The authors designed an adversarial-based countermeasure that uses the gradient-based optimization technique [8]. This countermeasure works as a noise generation device that operates in parallel with the execution of the cipher and aims to alter the observed power traces of the device. For their tests, they used power traces from a protected implementation of AES that uses a mask. As they wanted to test their implementation on an unprotected dataset, they considered the mask known, and they removed it by using it on the dataset labels. After the mask removal, they trained several models on the clean training data and tested their performance on the testing data. Then, they used their adversarial-based countermeasure to add small perturbation on the testing data and remeasured the performance of their attacking model. Even though their results were remarkable and their countermeasure was effective in all cases, their choice of using their countermeasure only on the testing data veers significantly from the conventional threat model of profiled SCAs. As the attacker is supposed to train his attacking model on an identical device to the targeted one, the perturbation generated by the adversarial-based countermeasure should be applied to both training and testing data. In this particular project, we aim to answer our question questions under that scope.

A different approach is presented by Gu et al. [29]. In their research, the authors tried to use existing techniques for the generation of adversarial examples that change either the whole input or a single feature, using the one-pixel attack technique [76]. However, both of their trials showed that the countermeasure is ineffective. The attack's performance remains the same when the distortion is applied in both training and testing data. Hence, they followed a different strategy; they used adversarial techniques to add noise instructions on the encryption algorithm's machine code. However, we consider unsatisfying their motivation on why their initial trials failed since they did not examine these approaches in-depth. This particular project aims to dive into these approaches and look for solutions that do not require changing the source code of the cipher but only cloak the leakage of the device.

3.3. Research Questions

Revising the related work by academia, we identify some undiscovered directions that we would like to explore. Thus, we formulate the following research questions that this particular project aims to answer.

RQ1: Can an adversarial-based countermeasure be designed so that DNN-based side-channel attacks become impractical regarding the observations needed for the secret key retrieval?

- **RQ1.1:** Which adversarial method can be used for such a countermeasure?
- **RQ1.2:** What are the minimum parameters of that method so that the distortion is minimal but the attack is still infeasible (or difficult enough)?
- **RQ1.3:** How does an adversarial-based countermeasure designed to provide performance against a specific attacking model perform against other attacking models (transferability)?

RQ2: What is the added value of using our proposed countermeasure compared to existing SCA countermeasures?

- **RQ2.1:** How an adversarial-based countermeasure performs compared to existing defensive techniques countermeasures?
- **RQ2.2:** What are the advantages and disadvantages of using our proposed countermeasure instead of existing techniques?

RQ3: Can this adversarial-based countermeasure provide security against non-profiling side-channel attacks too?

- **RQ3.1:** Does the adversarial-based countermeasure that is generated for DNN-based SCAs protect against CPA attacks?
- **RQ3.2:** Does this adversarial-based countermeasure provide any added value for non-profiled attacks compared to existing SCA countermeasures?

In the following chapters, we will examine the above questions in depth. In particular, chapter 4 provides the answer for **RQ1** by presenting the methodology for the development of our proposed countermeasure, and the results of our experiments that show its effectiveness. In chapter 5, we examine the **RQ2** by comparing the performance of our proposal with well-known countermeasures proposed by the research community. Chapter 6 presents how our adversarial-based countermeasure performs in non-profiled attacks seeking an answer for our **RQ3**.

4 Countermeasure Implementation

This chapter aims to answer the first research question. First, we present the notation that we use across this chapter, hoping to increase its readability. Second, we present the methodology of the preliminary steps for our experiments, such as extracting the used dataset and the design of the DNN models. Last, we present in detail how we implement our proposed adversarial-based countermeasure, and we test its performance against DNN-based side-channel attacks.

4.1. Notation

This section aims to present the various notations used in this chapter. To describe our dataset, we denote with X our leakage observations (inputs) and with Y their labels. To represent the dimensions of our observations, we use N for the number of observations (traces) and M for the sample size of each observation (features or tracepoints). To indicate the observations in a certain time sample t, we use the indexed variable X_t. Furthermore, we use the small letters x and y to denote a single observation and label of our dataset, while the x' represents an adversarial example. With δ we represent the added perturbation on an observation x and it holds that $\delta = x' - x$. Moreover, we denote with P and K the sets of plaintexts and keys used by the cryptosystem while the observations were captured. Both of these variables have a size of 16 bytes, where each byte is defined in the domain $\mathbb{B} : \{0, 1, \ldots, 255\}$. The indexed P_i and K_i represent a collection of all the i-th bytes of P and K accordingly. Last, the secret key that our side-channel attacks are aiming to retrieve is indicated with k^*. A summary of these notations is presented in table 4.1.

Notation	Description
N	Total observations
M	Number of features per observation
X	Dataset inputs, $X \in \mathbb{R}^{N\times M}$
Y	Dataset labels
x	Single observation $x \in \mathbb{R}^{N\times 1}$
y	Label for observation x
x'	Adversarial example based on observation x
δ	Added perturbation so that $x' = x + \delta$
P	Dataset's plaintexts, $P \in \mathbb{B}^{N\times 16}$
K	Dataset's keys, $K \in \mathbb{B}^{N\times 16}$
k^*	Secret key

Table 4.1: Notations

Furthermore, for the description of our models' architecture, we use the name of the model followed by a followed by a tuple with its hyperparameters. More precisely, we use the notations MLP($n_{\text{layers}}, n_{\text{units}}, \alpha$) and CNN($n_{\text{blocks}}, n_{\text{filters},1}, n_{\text{kernels}}, \tau, n_{\text{dense}}, n_{\text{units}}, \alpha$) to describe MLP and CNN models respectively. Similarly, we use the notation `Training`($n_{\text{epochs}}, b, O, l$) to refer to the training hyperparameters of each model. The interpretation of each hyperparameter will be explained extensively on the corresponding chapter.

Last, we denote ordinary mathematical terms with their commonly used notation when it is possible. For example, with the notation $N(\mu, \sigma)$, we refer to the normal distribution with mean μ and standard deviation σ.

4.2. Setup

Before implementing our adversarial-based countermeasure and testing its performance, we need a dataset with observations on AES's unprotected version. This dataset will be the base for comparison; we will modify it by applying our proposed adversarial-based countermeasure, and we will execute DNN-based side-channel attacks. Ideally, we will see that the modified dataset that includes our proposed countermeasure is more difficult to be attacked. That would confirm the effectiveness of our proposal. Furthermore, we would like to test our models on different DNN architectures, as we want our countermeasure's effectiveness to be independent of the model that the attacker uses. Thus, we need to define a set of models that will work as a reference for our experiments. The following sections present our approach for the extraction of our dataset and selecting our reference models.

4.2.1. Feature selection and dataset extraction

Our experiments will be based on the ASCAD dataset [64]. However, as we want to run our experiments on an unprotected AES implementation and the provided dataset contains features focused on the third subkey, we need to extract a new dataset for our project. To do so, we will use the raw traces that Prouff et al. [64] used to extract their dataset. These traces are captured from a masked AES-128 implementation (1/3 fixed key, 2/3 variable key) where the first two bytes of the mask are `0x00`. That allows us to consider the first two bytes of the key as unprotected (for more details on the raw and ASCAD datasets, see section 2.5). In total, the dataset consists of $300,000$ observations, where each one of those has a size of $250,000$ features. For a successful attack on the first subkey of the secret key k^*, we need to find a target area that leaks information about the key's first byte. We can do that by performing correlation analysis.

Our goal is to detect areas on the captured observations correlated with the label, resulting from a combination of plaintext and encryption keys. For our experiments, we choose to work with the identity leakage model; thus, our labels are calculated using equation 4.1.

$$Y = AES\_SBOX(P_0 \oplus K_0) \tag{4.1}$$

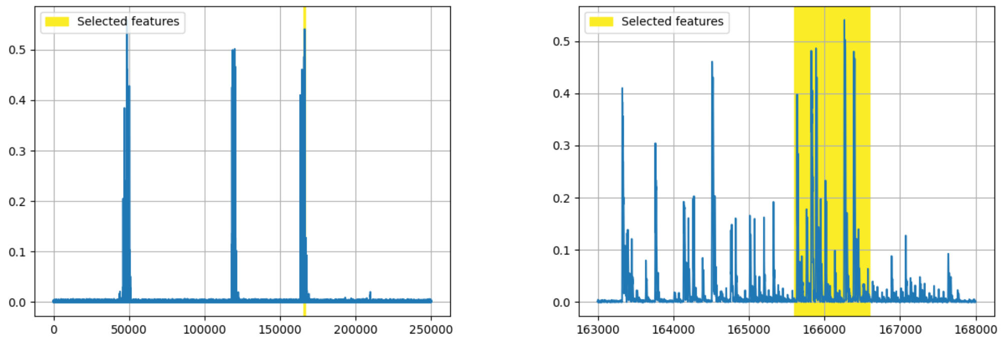
Such detection can be done using Pearson's correlation function (see equation 2.10). Applying this formula on the observations X_t of a certain time sample t and the labels Y for the first subkey, we can identify if a change of the label results in a similar change to the observation on that time sample; if no, then the output would be close to 0. Any other value reveals a small or big correlation between the two samples. Thus, repeating this for all the time samples of our observation, we can find areas that present high correlation, which are the areas that we are interested in. A detailed description of our correlation analysis for the first subkey is presented in algorithm 1.

Algorithm 1: Correlation analysis procedure on ASCAD's raw traces

```
    Input  : $X \in \mathbb{R}^{N \times M}$, $P \in \mathbb{B}^{N \times 16}$, $K \in \mathbb{B}^{N \times 16}$
    Output : $\texttt{corr} \in \mathbb{R}^{1 \times M}$
1   Initialize corr array with $M$ zeros.
2   $Y = AES\_SBOX(P_0 \oplus K_0)$
3   for $i \leftarrow 0$ to $M - 1$ do
4       $\texttt{corr}_i \leftarrow |Pearson(X_i, Y)|$
5   end for
6   return $corr$
```

The correlation analysis results are displayed in figure 4.1a. It is visible that three areas present a high Pearson correlation. We choose to attack the last area presented in higher resolution in figure 4.1b. In particular, we select the $1,000$ features in the interval $[165600, 166600)$, as it looks the densest area.

Even though the dataset with the raw traces provides $300,000$ observations, for our purposes, we select $50,000$ observations with a variable key for our training dataset and $10,000$ traces with a fixed key for our testing dataset. We consider that the training size is sufficient for an attacker to build a profiling model, while an attack should not be considered feasible if the attacker needs more than $10,000$ to retrieve the key. For the labels of our dataset, equation 4.1 is used.



(a) Full input size

(b) Zoom at the interval [163000, 168000)

Figure 4.1: Correlation analysis for the first subkey

Figure 4.2 presents the performance of an MLP attacking model (we will explain its architecture further in the next section) on our dataset. For the attack –and for any attack that we execute in this particular project, we first normalize the profiling and attacking data before the training and testing phase. This preprocessing step speeds up the training and increases the chances of finding a global optimum [48]. As a normalization technique, we use the z-normalization method presented in equation 4.2 with the mean and the standard deviation of observations X to be denoted as $\bar{X}$ and S, respectively.

$$Z = \frac{X - \bar{X}}{S} \tag{4.2}$$

Our model succeeds in achieving $GE = 1$ and $SR = 100\%$ using three attacking traces. Ideally, we would like to have a successful attack using more traces from our testing dataset. That would give us a better resolution when we compare our adversarial-based countermeasure performance. A common practice to achieve that is to add Gaussian noise to our observations. To be specific, we modify our observations X by adding random values from a Gaussian distribution with a mean equal to zero and standard deviation σ, namely:

$$X^* = X + N(0, \sigma) \tag{4.3}$$

Originally, the clean dataset has values in the range $[-116, 85]$. Thus, we add Gaussian noise with $\sigma = \{10, 20, 30\}$, which corresponds to the 5%, 10%, and 15% of the initial dataset's value range, respectively, and repeat the attack using the same MLP model on the modified datasets. In figure 4.2, the results of these attacks are also presented. Indeed, the attack is relatively harder after the noise addition. We arbitrarily choose to run our experiments using the modified dataset with $\sigma = 30$, as we consider that it provides that necessary resolution for evaluating our countermeasures. For the rest of our report, we assume that this noise is generated naturally by the device, and thus, we use these observations as our clean dataset.

4.2.2. Selection of Attacking Models

The generation of an adversarial example demands the existence of a pre-trained model. As an adversarial example is considered a clever perturbation that forces a model to misclassify, that model needs to be trained with our training dataset before the adversarial examples generation. Furthermore, this project aims to find out if and how an adversarial-based countermeasure can make the side-channel attacks harder. Therefore, these models will be used to compare the evaluation metrics of the attack

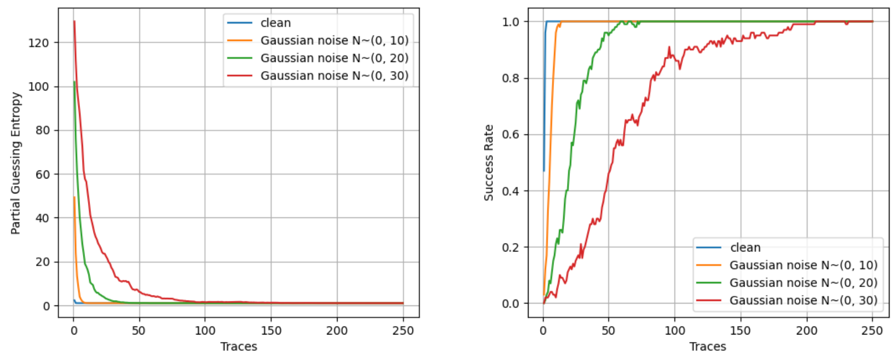


Figure 4.2: Performance of attack on our dataset

with and without the usage of the proposed countermeasures. Hence, we will need a set of attacking models that present decent performance when attacking our testing dataset for our experiments.

Prouff et al. [64] presented an interesting methodology to find attacking models that present good performance on the ASCAD dataset. In their methodology, they attack directly on the ASCAD dataset. However, a common practice in deep learning is to standardize the input of a DNN for practical reasons, as explained in the previous section. Their choice of not applying standardization led to significantly large models in terms of total parameters. For this reason, we decided to perform our own exploration for attacking models by applying their methodology after we use the z-normalization algorithm(see equation 4.2) to standardize our input. That will help us to find smaller models that can be trained faster and speed up our experiments.

To find our reference models, first, we define some basic architecture models for those, and we will explore how changing different hyperparameters can improve their performance. The most common classes of DNN used in side-channel attacks are the MLP models and the CNN models. Thus, we define two skeletons that we will use to construct our reference models, one for each class, that are presented below. The hyperparameters that are changed during our exploration are given with variables, while those that remain the same are defined with constant values.

MLP architecture

Our MLP model follows the standard structure of this kind of DNN classifiers, which is presented in figure 4.3. In detail, it consists of an input layer, n_{layers} fully connected layers, and an output layer. The input layer has a length equal to our input size, while the dense layers have size n_{units} units and use an activation function α. The output layer has a respective size to our leakage model and uses the activation function `softmax`, commonly used in classification problems. Regarding the training parameters, we use our whole training dataset (50,000 traces), and we train for n_{epochs} epochs, and we split the input into batches of size b. Finally, the models are training using the optimizer O with a learning rate l.

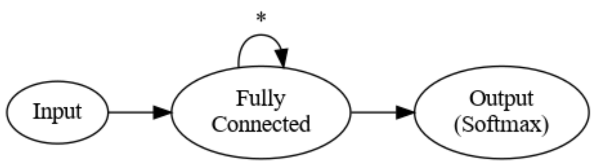


Figure 4.3: Architecture of our MLP reference models

To explore how changing different hyperparameters affect the attack's performance, we define a base MLP model MLP_{base} where its hyperparameters are randomly defined within the corresponding exploration space. Then, we train multiple models by changing one of these hyperparameters and keeping the rest constant, and we measure their performance. To compare the extracted models, we need to define a point that an attack is considered successful. For this purpose, we select the amount of attacking traces needed, so the guessing entropy and the success rate of the attack are stabilized to the values $GE = 1$ and $SR = 100\%$. Thus, we compare and select the specific hyperparameter's best value based on how many attacking traces are needed for the attack to be successful. We update our base model, and we move to the tuning of the next hyperparameter. Note that we select the value that reduces the model's total size in case of similar performance.

Initially, we define the MLP$_{base}$ with random values for its hyperparameters that are MLP$(3, 400,$ $\texttt{relu})$, while its training hyperparameters are $\texttt{Training}(10, 20, \texttt{RMSprop}, 10^{-3})$. After the hyperparameter exploration that we described, we conclude to the best MLP model, that is denoted as MLP$_{\text{best}}$ for the rest of the report, with architecture hyperparameters MLP$(5, 100, \texttt{tanh})$ and training hyperparameters $\texttt{Training}(10, 20, \texttt{RMSprop}, 10^{-4})$. Table 4.2 displays the examined range for each of the hyperparameters during our benchmarks, as well as the value we select for our MLP$_{\text{best}}$, while appendix A.1 presents in detail the results of the hyperparameter exploration.

Parameter	Reference	Range	Best
Architecture hyperparameters			
Number of fully connected layers	n_{layers}	$[2..6]$	5
Units per layer	n_{units}	$\{100, 200, 300, 400, 500\}$	100
Activation function	α	$\{\texttt{relu}, \texttt{sigmoid}, \texttt{tanh}\}$	$\texttt{tanh}$
Training hyperparameters			
Epochs	n_{epoch}	$\{10, 20, 30\}$	10
Batch size	b	$\{1, 50, 100, 200, 400\}$	200
Optimizer	O	$\{\texttt{RSMprop}, \texttt{Adam}, \texttt{SGD}\}$	$\texttt{RSMprop}$
Learning rate	l	$\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$	10^{-4}

Table 4.2: Benchmarks summary for MLP architecture

CNN architecture

In contrary to the MLP models, the CNN models do not follow a standard architecture. For our project, we use the architecture proposed by Prouff et al. [64] presented in figure 4.4. Our CNN model consists of an input layer with a length equal to the size of the dataset's input. It has n_{blocks} convolution blocks, where each one of these includes a convolution layer and a pooling layer. The convolution layers have a convolution window of n_{kernels} kernels, output size of $n_{\text{filters, i}}$ filters and they use an activation function α. For the calculation of the convolution layer's output size $n_{\text{filters, i}}$, we define an output size $n_{\text{filters, 1}}$ for the first layer and for each next layer we double it, i.e. $n_{\text{filters, i}} = 2 \times n_{\text{filters, i-1}}$. We name the type of the pooling layer as τ. Then, a flatten layer converts the data from the previous layers into a 1-dimensional array. Next, there are n_{dense} fully connected layers with size of n_{units} units and an activation function α. Last, there is an output layer that uses the activation function $\texttt{softmax}$. Regarding the training hyperparameters, similarly to our MLP model, we train for n_{epochs} epochs with a batch of size b using the optimizer O with learning rate l.

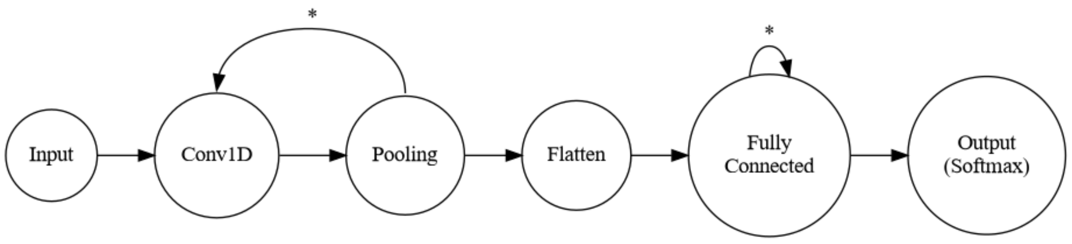


Figure 4.4: Architecture of our CNN reference models

We follow the same hyperparameter tuning process, as explained in the previous section for the MLP model. Initially, we define the CNN$_{base}$ with random values for its architecture and training hyperparameters that are CNN$(4, 32, 7, \texttt{average}, 2, 512, \texttt{relu})$ and $\texttt{Training}(10, 20, \texttt{RMSprop}, 10^{-3})$, respectively. After the hyperparameter exploration that we described, we conclude to the best MLP model, that is denoted as CNN$_{\text{best}}$ for the rest of the report, with architecture hyperparameters CNN$(3, 64, 9,$ $\texttt{average}, 3, 512, \texttt{relu})$ and training hyperparameters $\texttt{Training}(10, 20, \texttt{RMSprop}, 10^{-5})$.

Table 4.3 displays the examined range for each of the hyperparameters during our benchmarks, as well as the value we select for our reference CNN model, while appendix A.2 presents in detail the results of the hyperparameter exploration.

Parameter	Reference	Range	Best
Architecture hyperparameters			
Number of convolution blocks	n_{blocks}	$[2..5]$	3
Output size of the first convolution layer	$n_{\text{filters, 1}}$	$\{16, 32, 64, 128\}$	64
Length of the 1D convolution window	n_{kernels}	$\{5, 7, 9, 11\}$	9
Type of pooling layer	τ	{`average`, `max`}	`average`
Number of fully connected layers	n_{dense}	$[1..3]$	3
Units per FC layer	n_{units}	$\{128, 256, 512, 1024, 2048\}$	512
Activation function	α	{`relu`, `sigmoid`, `tanh`}	`relu`
Training hyperparameters			
Epochs	n_{epoch}	$\{10, 20, 30\}$	10
Batch size	b	$\{1, 50, 100, 200, 400\}$	200
Optimizer	O	{`RSMprop`, `Adam`, `SGD`}	`RSMprop`
Learning rate	l	$\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$	10^{-5}

Table 4.3: Benchmarks summary for CNN architecture

4.3. Design of adversarial-based countermeasures

Before presenting our research on the adversarial-based countermeasure, it is crucial to establish the threat model and the assumptions taken into consideration. First, we visualize our countermeasure as a noise generator unit inside the crypto device's microprocessor that works parallel with the encryption process. For this project, we want to investigate the ideal scenario: how added noise should look to the original traces to provide security against DNN-based side-channel attacks. Thus, we consider that our noise generator has unlimited capabilities and can insert noise of a certain amplitude to a specific time sample (feature) of our input.

There are two options regarding the threat model of our experiments. We can apply the countermeasure only on the attacking phase or in both the profiling and attacking phases. The first approach is trivial and investigated by Picek et al. [63], showing positive results. We find the latter to be more attractive for two reasons. On the one hand, it is the most common threat model of profiling attacks on protected devices. The attacker has to use a clone device with the same configuration and specification as the targeted one. Thus, the attacker has to train his profiler using a device that is protected by the adversarial-based countermeasure. On the other hand, this approach has already been explored by Gu et al. [29] showing negative results, but we believe that there is still room for investigation, as we will explain later in this section. Therefore, we use the following threat model in our experiments. First, the attacker trains a profiling model using a device that applies the proposed adversarial-based countermeasure. Then, the attacker uses his trained model to execute the attack against the targeted device by capturing a set of power traces that include the noise generated based on our countermeasure.

The noise generation as a countermeasure against side-channel attacks is not a new idea. However, our goal is to describe a method that is calculated cleverly instead of adding random noise. This approach can have multiple benefits compared to the old-fashioned approach. When adding random noise to the observations, there is no guarantee that the outcome would be perturbed enough to force the attacking model to misclassify. On the other hand, an adversarial-based noise cloaks the original input and leads to a false prediction by definition. Another significant benefit is that such an adversarial-based noise generator is more cost-efficient than a random one. The former can identify which features need to change to trick the attacking model, while the latter modifies all the input features under the same distribution. Thus, it is certain that some features could be modified with less magnitude or not modified at all. This redundant noise is translated to extra power consumption and higher operational costs. The rest of this section aims to research the specifications of such an adversarial-based noise generator that can protect against DNN-based side-channel attacks.

4.3.1. Methods

For the generation of the adversarial-based noise, we consider two approaches; we can use a method that adds either small noise to all the features or larger noise to a subset of the input's feature. Both approaches have already been investigated by previous researches, as presented in chapter 3.

Picek et al. [63] followed the first approach and showcased that such a countermeasure could work when used only in the attacking phase. At the same time, their approach seems ineffective when adversarial examples are used during the training phase [29]. A possible explanation for that effect would be that, after the adversarial training, the attacking model becomes more generalized and thus more robust against small perturbations on the whole input. In the second approach, Gu et al. [29] used the one-pixel attack [76] to modify a single input feature. They concluded that this method could not provide additional security against SCAs when used in both phases. However, they did not investigate the latter approach in-depth, and we consider that there is room for further research. In detail, it would be interesting to explore if we can use the one-pixel attack to build an effective adversarial-based countermeasure against DNN-based side-channel attacks when applied to multiple features of the input, and, if so, what is the minimal bounds of the added noise.

Hence, we decide to build an adversarial-based method that applies high magnitude distortion to a small number of input features. In more formal words, our method generates an adversarial example x' for input x using a trained DNN model by changing k features of the original input and reduces the secret key's k^* confidence. The generation of such adversarial examples can be translated to an optimization problem; the position and the magnitude of the noise are the problem's parameters and the DNN classifier that returns the key's confidence is the objective function that needs to be minimized.

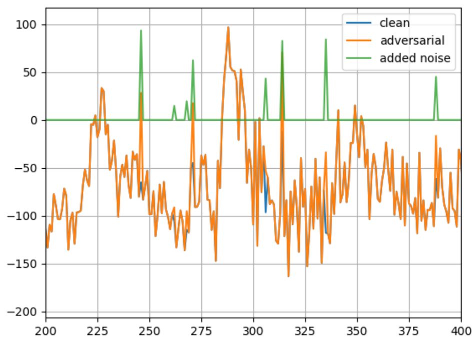
Many adversarial methods could be used as a basis for our adversarial-based countermeasure. The most common methods, such as the fast gradient method [8] and the basic iterative method [47], use the gradient descent optimization algorithm, which can quickly converge to a solution to continuous optimization problems. However, our optimization problem is discrete, as it requires the modification of features that are spread throughout the input's length. Thus, we decide to implement our countermeasures based on the one-pixel attack [76], as its approach could be considered an analogy of our problem in the image domain. This method uses the differential evolution optimization algorithm, which is suitable for discrete optimization problems. As presented in section 2.2.1, this algorithm uses evolutionary computation techniques to optimize an objective function. As explained above, our objective function is DNN classifier and the position and the magnitude of the adversarial noise are the problem's parameters. Below, we introduce three different approaches to selecting the affected features and their new value.

Normal k-Point Protection

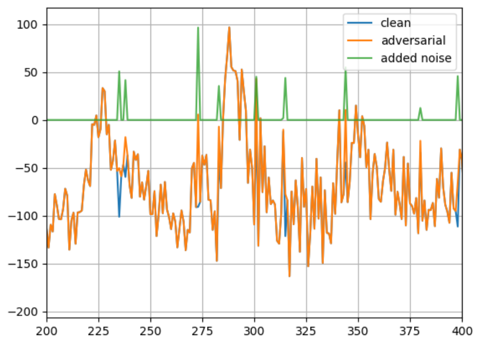
The normal k-Point Protection method is a full adaptation of the one-pixel attack on our input; we change the name to comply with our domain and our defensive purposes. In detail, this method uses the differential evolution as an optimization algorithm to find *which* k features will have to change and *how* so the attacking model fails to predict the correct label. The major change compared to the authors' method is that instead of replacing the original value of a certain feature of the input with one from a given range, our algorithm *adds* a perturbation within given bounds to the original value of the input's feature. To refer to this method, we define the notation $kPP_{\text{normal}}(model, k, bound_{\text{lower}}, bound_{\text{upper}})$, where $model$ is the DNN model that the method uses for the generation of the adversarial examples, k the number of the affected features, and $bound_{\text{lower}}, bound_{\text{upper}}$ the lower and upper bound of the added noise accordingly. For simplicity reasons, we omit the $bound_{\text{lower}}$ when the lower bound is set to zero using the notation $kPP_{\text{normal}}(model, k, bound_{\text{upper}})$. In figure 4.5a, we present a random trace from our dataset before and after the application of the normal k-Point Protection method with parameters $kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 40, 100)$.

Random k-Point Protection

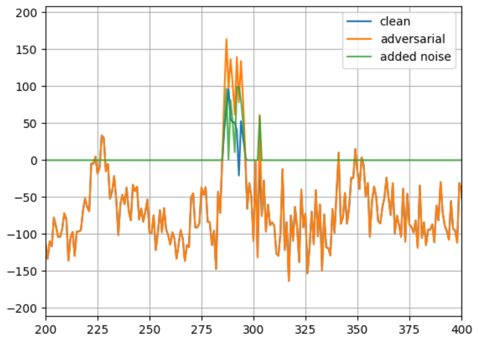
The random k-Point Protection method follows a different approach regarding the selection of the modified points. This method picks randomly a set input's features that remains the same during the execution of the optimization algorithm. In other words, the DE algorithm tries to optimize the perturbation on certain k features of the input that are chosen randomly at the beginning of the adversarial example generation. As this random subset is recomputed upon each method's call, each input (row) of the dataset is changed on different features. Similarly to the previous method, we refer to



(a) Normal k-Point Protection



(b) Random k-Point Protection



(c) Correlation k-Point Protection

Figure 4.5: Comparison of a trace before and after the proposed methods

this method using the notation $kPP_{\text{random}}(model, k, bound_{\text{lower}}, bound_{\text{upper}})$, or simply $kPP_{\text{random}}(model, k, bound_{\text{upper}})$ when the lower bound is set to zero. In figure 4.5b, we present a random trace from our dataset before and after the application of the random k-Point Protection method with parameters $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 40, 100)$.

Correlation k-Point Protection

The correlation k-Point Protection follows another approach when it comes to how the distorted features are selected. The algorithm computes the Pearson correlation of each feature with the label and then selects the top k features with the highest correlation. Finally, the optimization algorithm computes the required perturbation for the generation of the adversarial example. This method presents a few noteworthy points. First, this method cannot be executed on a single trace and requires a larger amount of traces so that the correlation analysis is reliable. Finally, unlike the random variant, the k modified features will be in the same position for each dataset entry. Similarly to the previous methods, we refer to this method using the notation $kPP_{\text{corr}}(model, k, bound_{\text{lower}}, bound_{\text{upper}})$, or simply $kPP_{\text{corr}}(model, k, bound_{\text{upper}})$ when the lower bound is set to zero. In figure 4.5c, we present a random trace from our dataset before and after the application of the correlation k-Point Protection method with parameters $kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 40, 100)$.

4.3.2. Experimental Process

To confirm the effectiveness of our proposed adversarial-based countermeasure, we investigate how the aforementioned approaches can decrease the performance of DNN-based side-channel attacks under specific metrics. First, we execute the attacking scenario against the clean dataset. Then, we use our methods (separately) to generate our protected datasets, and we execute the same attacking scenario against them. Last, we use a set of metrics to quantify the performance of our adversarial-based countermeasure based on these attacks.

Part of our research question is to explore how different parameters of our methods affect their effectiveness. Thus, we apply our methods with different values for each parameter and, we evaluate their importance on the methods' effectiveness using the above metrics. The first parameter of our methods is the trained model that operates as the objective function of our optimization problem. For our experiments we use our MLP_{best} and CNN_{best}. For the number of the affected features, we select a k value that corresponds from 1% to 10% of the input size, namely $k = \{10, 20, ..., 100\}$. Last, other important parameters of our method are the bounds of added perturbation, in which the differential evolution algorithm will explore solutions. We want our method not to add any noise to a feature if required. Thus, we set the lower bound of the added noise to be zero for all of our experiments. At the same time, we set the upper bound of the added noise to be from 100 to 500 with step 100, i.e. $bound_{\text{upper}} = \{100, 200, 300, 400, 500\}$. As our clean dataset's features have values within the range $(-250, 250)$ approximately, the overall noise added to a feature corresponds to $20\%, 40\%, 60\%, 80\%$, or 100% of the original range, respectively. Regarding the differential evolution algorithm parameters, we keep the ones use on the original attack, as described in section 2.3.2. In particular, the population size is set to 400, the crossover probability Cr is set to 1, the differential weight F is set to 0.7, and the maximum iterations are 75.

An important aspect that we have to remark is that our countermeasures are stochastic; differential evolution uses randomness to initialize the first round's population, and random k-Point Protection is choosing the affected features randomly explained above. Hence, we create a set of instances for each value of k to guarantee that these factors do not bias our experiments' conclusions. To summarize, we generate 1,500 modified datasets, using 3 methods, 2 base models, 10 values for k, 5 values for the noise upper bound, and repeating each execution 5 times.

Another noteworthy point is the label that we use to generate the adversarial examples of the testing dataset. In machine learning, the methods that compute the adversarial perturbation for the testing samples commonly use the trained model's predictions as a label. This label is required by the method, so it is aware of the predicted class and computes a distortion that forces the trained model to predict any other. On the contrary, for our adversarial-based countermeasures, we are using the true labels. Our motivation on this choice is that the perturbation is calculated by the same device that executes the encryption algorithm. Thus, the secret key is known to the device and can generate an adversarial example. Also, our model threat demands that the training and the testing datasets are built under the same procedure.

Using the modified datasets, we execute a profiling attack using 2,000 from our testing dataset. We consider that an attack is successful when the attacking model achieves $GE = 1$ and $SR = 100\%$. When that it is not achieved using at most that many traces, the attack is considered infeasible. The examined research question is to test our proposed methods' transferability. We run the attacks twice, once using the same model that the countermeasure uses to generate adversarial examples and once using a different model to execute the attack. For the latter scenario, when the method uses the MLP_{best} model, we attack using the CNN_{best} model and vice versa.

As mentioned above, it is important to set specific metrics that will allow us to evaluate the performance of our adversarial-based countermeasure. Commonly, the evaluation of an adversarial method is done based on two factors. First, generated adversarial examples have to be effective, namely to trick the classifier and misclassify. Second, the generated adversarial examples have to be indistinguishable from the original samples. This last property is important, usually in many domains. An image of a panda, for example, still looks like a panda after the distortion but is classified wrongly by a trained DNN classifier. Hence, we need to adapt these two properties to the side-channel analysis domain to evaluate our defensive techniques properly.

The effectiveness of an adversarial-based countermeasure depends on its effect on SCA performance. Thus, we define a set of SCA metrics that will facilitate us to evaluate what is the performance drop after applying our methods. More precisely, we define the $n_{success}$ metric, which represents the number of traces that need to be processed by the attacking model so it can achieve $GE = 1$ and $SR = 100\%$. In case that $n_{success} > 2000$, we consider that this attack is infeasible. Moreover, we define the metrics GE_x and SR_x representing the achieved guessing entropy and success rate when the attacking model process $x\%$ of the total attacking traces (2,000 in our experiments). In our experiments, we measure for $x = \{10, 50, 100\}$.

On the other side, the indistinguishability property is not directly transferable to our domain. In the side-channel analysis domain, it is not important to keep the traces similar, as the human eye cannot classify the original input correctly. However, the ”minimality of perturbation” introduced by the indistinguishability property can be interpreted as ”minimality of power consumption” generated by our adversarial-based noise generator. Hence, we can use metrics commonly used to measure that property to simulate the additional power consumption that our proposed countermeasure introduces to the original traces. This parameter can be a secondary evaluation metric for our adversarial-based countermeasure, as we would ideally want to achieve sufficient protection with the lowest possible introduced additional consumption.

In the machine learning domain, indistinguishability is usually measured using the l_0, l_2, and/or l_∞ distance metrics. As explained in section 2.3.2, the first norm measures the number of affected features in the original input without taking into account the added perturbation. This metric is represented by our method's parameter k, which defines the amount of the modified tracepoints. The l_2 distance metric measures the metric that describes the euclidean distance, namely the minimum distance in the dataset's space, between the original and the adversarial. The l_∞ distance metric describes the maximum added noise among all the tracepoints. Out of these three metrics, we believe that the l_0 and l_∞ cannot be used to measure the additional power consumption that our adversarial-based countermeasure introduces, as they don't take into account the total added noise. On the other hand, l_2 seems to fits our needs entirely as it can describe how small or big our added noise needs to be, so we construct the perturbed traces. Hence, by comparing the distances of these modified datasets from the clean dataset, we can safely assume that the one with a smaller l_2 metric consumes less power for its generation. The average l_2 distance metric of our modified datasets is presented in detail in appendix B, and we refer to them during the following sections when necessary.

4.3.3. Results

This section presents the experiments' results for each of the three proposed methods in separate subsections to increase our results' readability.

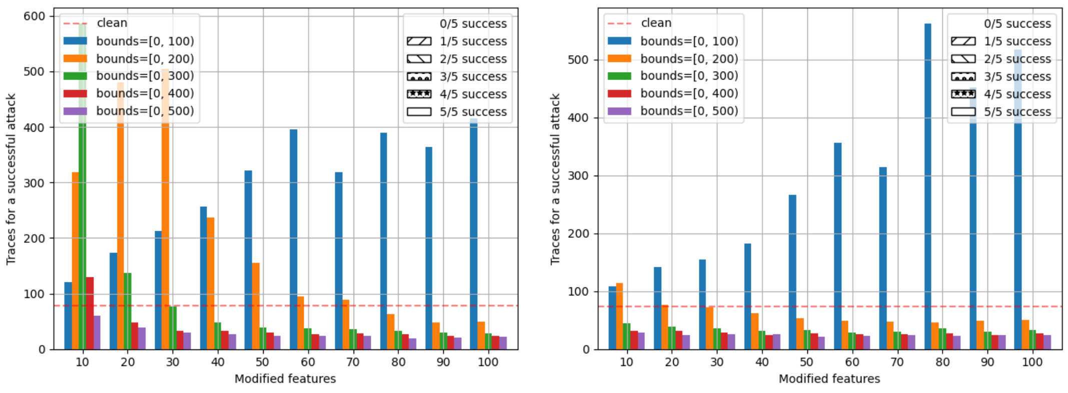
In each subsection, we first present the performance of our reference models (MLP$_{\text{best}}$ and CNN$_{\text{best}}$) against the modified datasets that consist of adversarial examples based on the MLP$_{\text{best}}$ model. Next, we use our reference models to attack against datasets generated using the CNN$_{\text{best}}$ model. Using the $n_{success}$ metric, we first identify which configurations are decreasing the performance of the attacks. Then, using this metric, we select the configurations that provide a decent level of protection. Next, we highlight which method's parameters present better performance using the SCA and, when necessary, the distance metrics. Last, we present our conclusions about the specific method's performance regarding the provided security and transferability.

As described in the previous section, for each examined combination of parameters, we generate five instances. Thus, our metrics represent the average value of the five measurements. Regarding the $n_{success}$ metric, we first measure how many traces are needed for an attack against each instance for the key retrieval, and we compute the average. If an attack is infeasible using the maximum amount of the attacking traces (2,000 traces in this chapter), we do not take this execution to calculate the average, and we denote it properly on our results. If all of the attacks against a specific configuration are infeasible, we use the notation ">2000" in our results. About the GE_x and SR_x metrics, we compute the average value of the GE and SR when x% of the attacking traces are processed.

Normal k-Point Protection

According to figure 4.6a, the normal k-Point Protection that uses the MLP$_{\text{best}}$ model for the noise generation provides notable security when the added noise is bounded in a small range, while the attack becomes easier when the noise is bounded in larger ranges. In detail, when the countermeasure tries to find noise within the range $[0, 100)$, the more features we change, the better protection we achieve. When the upper bound of the noise is set to 200 or 300, we observe that the method's performance peak is achieved when we change 30 and 10 features, respectively, and drops when the method alters more features. Last, we see that when the upper bound is bigger, the method is not effective, and, in particular, the attack becomes easier when the number of the affected features increases. Overall, we notice that, following this approach, this method can make the attack about six to seven times harder, in the cases of $kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 10, 300)$, $kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 20, 200)$, $kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 30, 200)$.

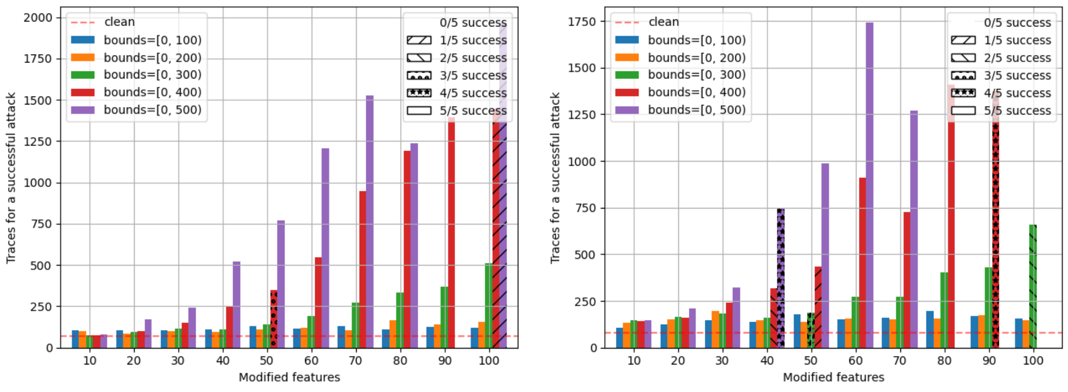
Regarding its transferability, figure 4.6b shows that this implementation can still provide security against CNN$_{\text{best}}$-based attacks, when the upper bound is set to 100, in proportion to the number of features that are modified. However, when the upper bound is increased, the countermeasure is ineffective, and the attack becomes even easier as the number of the required traces for a successful attack drops. The maximum protection is observed in the case $kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 80, 100)$, where the attack is successful after processing around 580 attacking traces.



(a) Attack using the same model (MLP_{best}) (b) Attack using different model (CNN_{best})

Figure 4.6: Performance of normal k-Point Protection using the MLP_{best} model. Bar's pattern describes the number of feasible attacks within 2,000 attacking traces. If a bar is missing, then all of the corresponding attacks were infeasible.

On the other hand, when the normal k-Point Protection method generates the adversarial examples using the CNN_{best} model, the results are quite different. More precisely, as presented in figure 4.7a in the cases that the upper bound of the noise is 100 and 200, we do not observe any significant change in the attack's performance. On the other hand, when the upper bound is increased, we see that the attack becomes harder and harder, proportionally with the number of affected features. In particular, we notice that when the upper bound is set to 500, our method needs to change only 70 features to make the attack infeasible, which is significantly better performance compared to the implementations that we present in figure 4.6.



(a) Attack using the same model (CNN_{best}) (b) Attack using different model (MLP_{best})

Figure 4.7: Performance of normal k-Point Protection using the CNN_{best} model. Bar's pattern describes the number of feasible attacks within 2,000 attacking traces. If a bar is missing, then all of the corresponding attacks were infeasible.

The results about this approach's transferability are presented in figure 4.7b. Unlike the case that our method generated the modified datasets based on the MLP_{best} model, we observe that, in this situation, the results are consistent with each other. To be more precise, smaller bound ranges do not affect the performance of the attack, whereas the attack becomes harder when the bound range and the modified features are increased.

In table 4.4, we present in detail the SCA metrics for the selected configurations that present the notable performance, as presented above. As we can identify, methods that are using the MLP_{best} model cannot provide significant performance, as the GE and SR converge close to 1 and 100%, respectively, when only 10% of the total attacking traces is used for the attack. Regarding the CNN$_{\text{best}}$-based configurations, we see that the selected configurations offer strong protection against the MLP$_{\text{best}}$ model, as none of these attacks can retrieve the correct subkey using 2,000 traces. Furthermore, attacks based on the CNN$_{\text{best}}$ model are always successful only in the case of $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 100, 400)$, while in the case of $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$ the attack always fails.

Method	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 80, 100)$	MLP$_{\text{best}}$	389.2 (5/5)	1.79	0.82	1.00	1.00	1.00	1.00
	CNN$_{\text{best}}$	561.6 (5/5)	2.86	0.71	1.00	1.00	1.00	1.00
$kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 90, 100)$	MLP$_{\text{best}}$	363.2 (5/5)	1.54	0.87	1.00	1.00	1.00	1.00
	CNN$_{\text{best}}$	452.6 (5/5)	2.24	0.78	1.00	1.00	1.00	1.00
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 100, 400)$	MLP$_{\text{best}}$	>2000	57.70	0.04	13.59	0.39	4.45	0.63
	CNN$_{\text{best}}$	1445.0 (5/5)	22.40	0.25	1.14	0.94	1.00	1.00
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 80, 500)$	MLP$_{\text{best}}$	>2000	71.79	0.03	27.86	0.15	13.38	0.31
	CNN$_{\text{best}}$	1236.0 (1/5)	34.89	0.14	2.59	0.68	1.05	0.96
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$	MLP$_{\text{best}}$	>2000	78.02	0.03	38.97	0.17	23.23	0.36
	CNN$_{\text{best}}$	>2000	47.25	0.09	7.32	0.54	1.61	0.84
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 100, 500)$	MLP$_{\text{best}}$	>2000	89.05	0.02	47.11	0.09	28.95	0.24
	CNN$_{\text{best}}$	1966.0 (1/5)	42.31	0.12	4.18	0.68	1.13	0.92

Table 4.4: SCA metrics for attacks against datasets generated by normal k-Point Protection. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">2000" denotes that all five attacks were unsuccessful.

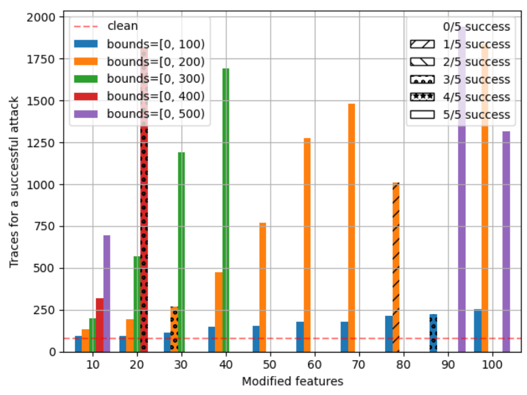
Overall, we observe that the general superiority of the CNN models against the MLP models is present in our results. On the one side, datasets that achieve effectiveness against the MLP$_{\text{best}}$ model fail to protect the CNN$_{\text{best}}$ model. On the other side, when normal k-Point Protection uses the CNN$_{\text{best}}$ model for generating the modified datasets, the method can provide security against both models. Furthermore, the method's performance regarding the feasibility of the attack in the two cases differs majorly. In the former case, the attack was successful after using less than 600 traces in the best scenario, while in the latter case, the proposed countermeasure was achieved to prevent the attack. Last, the best performance of the normal k-Point Protection is observed using the parameters $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$, as the attack is infeasible regardless the using attacking model.

Random k-Point Protection

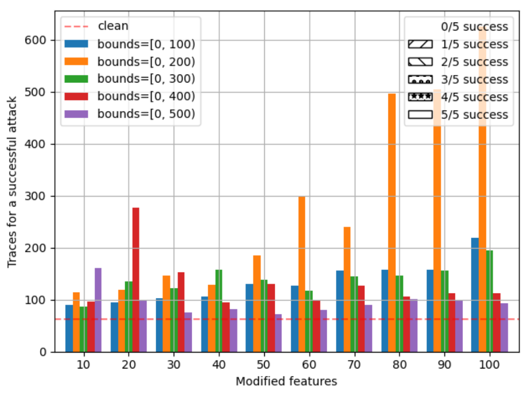
As we can observe in figure 4.8a, the random k-Point Protection performance is quite notable in the case that we use the MLP$_{\text{best}}$ model both for the generation of the adversarial examples and for the attacks. More precisely, we notice that in all the cases except the ones that the added noise is within the range $[0, 100)$, the attack becomes significantly harder or, in most cases, infeasible. By picking the smaller amount of affected features for these upper bounds, we can see that the best combinations of the parameters that the attack is infeasible are the $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 90, 200)$, $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 300)$, $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 30, 400)$, and $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 20, 500)$. In table B.3, we see that the corresponding average l_2 distance metric of these implementations is 1087.43, 1216.76, 1259.31, and 1287.60, accordingly. We see that these combinations combine sufficient protection with a relatively small l_2 metric compared to other combinations, and they can be selected as the top-performing configurations for this approach.

However, when we use the CNN$_{\text{best}}$ model for the attack to examine this approach's transferability, the results are again different. As presented in figure 4.8b, the implementations in which the upper bound is set to 200 perform better than others, but in this case, the attack is feasible. In the best scenario, when this method affects 100 features and the upper bound of the added noise 200, the attack requires 350 traces approximately to be successful, which is significantly lower compared to the attack using the MLP$_{\text{best}}$ model. Regarding the implementations with different $bound_{\text{upper}}$, they barely provide more security, as the performance metrics are not changed significantly.

In the scenarios that our method is applied using the CNN$_{\text{best}}$ model (figure 4.9a), we notice a similar behavior with the one observed in the method normal k-Point Protection. More precisely, small values for the upper bound of the added do not provide significant security regardless of the number of modified features, whereas when this parameter is increased, the attack performance drops. Nevertheless, the amount of features that need to be altered is decreased compared to the previous method.

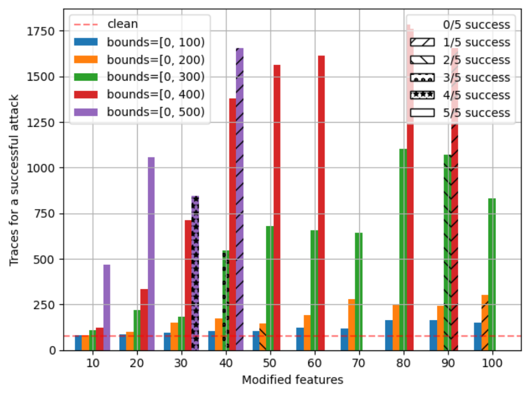


(a) Attack using the same model (MLP_{best})

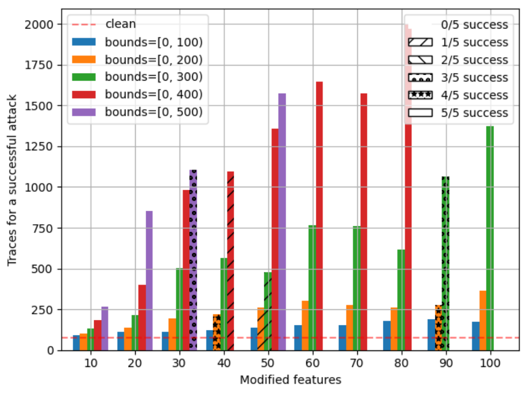


(b) Attack using different model (CNN_{best})

Figure 4.8: Performance of random k-Point Protection using the MLP_{best} model. Bar's pattern describes the number of feasible attacks within 2,000 attacking traces. If a bar is missing, then all of the corresponding attacks were infeasible.



(a) Attack using the same model (CNN_{best})



(b) Attack using different model (MLP_{best})

Figure 4.9: Performance of random k-Point Protection using the CNN_{best} model. Bar's pattern describes the number of feasible attacks within 2,000 attacking traces. If a bar is missing, then all of the corresponding attacks were infeasible.

In particular, we can make the attack infeasible when we add noise of range $[0, 400)$ in 70 features or noise of range $[0, 500)$ in 50 features. The l_2 metric of these configurations is 1932.63 and 2042.48 based on table B.4, respectively, which is quite similar. Hence, we conclude both of these configurations provide significant security with the same power consumption. The same observations apply to figure 4.9b, which presents this method's performance when we attack using the MLP_{best} model.

Based on the above observations, we select the implementations that perform better against both attacking models. Table 4.5 presents the SCA metrics for the attacks against the corresponding generated datasets. First of all, we clearly notice that the configurations that use the MLP_{best} are inferior regarding the transferability of the adversarial examples. Moreover, regarding the configurations that use the CNN_{best} model and the upper bound of the noise is set to 400, only the $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$renders all the attack infeasible. Last, we notice that the $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$is superior to others with the same $bounds$. On the one hand, the configurations that change fewer features present a significantly higher SR_{100} for the two attacks. In comparison, the ones that alter more features introduce more power consumption without a respective drop in the performance of the attacks. Hence, the $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$present the top performance for the random k-Point Protection. These two methods have euclidean distance $l_2 = 2307.70$ and $l_2 = 2413.72$. As the former introduces smaller consumption while it provides the worst performance and the latter opposite, we nominate both methods as the top-performing configurations. The suggestion that the first can be used when power consumption is critical for the encryption device.

Method	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 80, 200)$	MLP_{best}	1010 (1/5)	43.37	0.11	4.83	0.68	1.74	0.89
	CNN_{best}	496 (5/5)	4.10	0.73	1.00	1.00	1.00	1.00
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 90, 200)$	MLP_{best}	>2000	56.09	0.07	13.81	0.42	5.77	0.74
	CNN_{best}	504 (5/5)	5.43	0.75	1.00	1.00	1.00	1.00
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 100, 200)$	MLP_{best}	1839 (1/5)	54.01	0.07	14.37	0.38	5.81	0.73
	CNN_{best}	624 (5/5)	4.85	0.62	1.00	1.00	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 400)$	MLP_{best}	1573 (1/5)	47.81	0.09	8.44	0.56	2.89	0.79
	CNN_{best}	>2000	54.17	0.07	10.20	0.43	2.78	0.76
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 80, 400)$	MLP_{best}	1994 (1/5)	69.13	0.05	29.67	0.26	16.43	0.45
	CNN_{best}	1782 (1/5)	46.48	0.08	8.53	0.50	2.08	0.82
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 90, 400)$	MLP_{best}	>2000	64.70	0.03	23.47	0.19	9.43	0.41
	CNN_{best}	1657 (1/5)	60.61	0.08	17.71	0.32	4.85	0.59
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$	MLP_{best}	>2000	72.93	0.02	25.30	0.15	11.53	0.35
	CNN_{best}	>2000	66.85	0.03	20.06	0.22	5.54	0.49
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 50, 500)$	MLP_{best}	1572 (1/5)	58.06	0.07	17.84	0.39	8.23	0.67
	CNN_{best}	>2000	64.85	0.06	16.59	0.30	3.73	0.64
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 60, 500)$	MLP_{best}	>2000	85.98	0.02	43.96	0.10	26.21	0.22
	CNN_{best}	>2000	53.47	0.07	10.62	0.42	3.95	0.74
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$	MLP_{best}	>2000	97.86	0.01	59.02	0.06	39.63	0.15
	CNN_{best}	>2000	78.65	0.02	37.09	0.11	18.94	0.27
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 80, 500)$	MLP_{best}	>2000	91.41	0.01	60.12	0.03	38.27	0.07
	CNN_{best}	>2000	75.86	0.03	30.17	0.16	15.57	0.39
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 90, 500)$	MLP_{best}	>2000	101.52	0.01	72.56	0.04	56.06	0.07
	CNN_{best}	>2000	95.52	0.02	50.11	0.07	24.92	0.13
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 500)$	MLP_{best}	>2000	108.98	0.01	82.50	0.02	67.55	0.04
	CNN_{best}	>2000	88.66	0.02	47.22	0.05	28.83	0.16

Table 4.5: SCA metrics for attacks against datasets generated by random k-Point Protection. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">2000" denotes that all five attacks were unsuccessful.

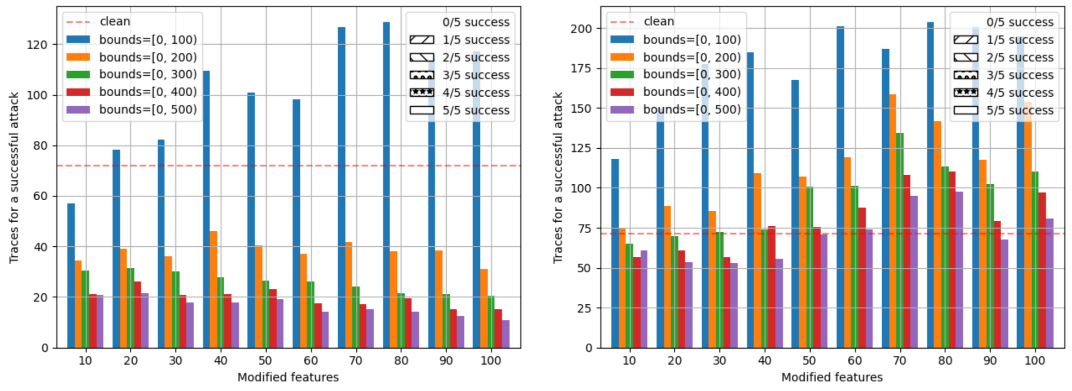
Regarding the used model for the generation of the adversarial examples, our observations are similar to those about the normal variant of the method. Using the CNN_{best} model upon the generation of the modified datasets results in stronger adversarial examples that are transferable to attacks with the MLP_{best} model. On the other hand, adversarial examples that are generated using the MLP_{best} model cannot provide sufficient protection against attacks that are based on the CNN_{best} model.

Correlation k-Point Protection

The performance of the MLP_{best}-based correlation k-Point Protection is presented in figure 4.10. In figure 4.10a, we see that this specific method fails woefully, as the performance of the attack remains almost steady when the upper bound of the added noise is set to 100 and arises when this parameter increased. Surprisingly, the results are slightly different when using the CNN_{best} model for the attack (figure 4.10b. In this case, the smaller values for the upper bound provide a small security level that cannot be compared to what the other methods can provide, while larger ranges seem not to affect the attack's performance.

In figure 4.11, we observe that the correlation k-Point Protection can provide small security against the side-channel attacks based on either the MLP_{best} or CNN_{best}. Figure 4.11a shows that when the upper bound is set to 100 or 200, the performance of attack using the CNN_{best} is hardly increased. In contrast, the attack becomes harder when the added noise is bounded within larger ranges. However, this approach's security level is still away from what other methods can offer when using the CNN_{best} model for the generation of the adversarial examples. We can derive similar outcomes from figure 4.11b.

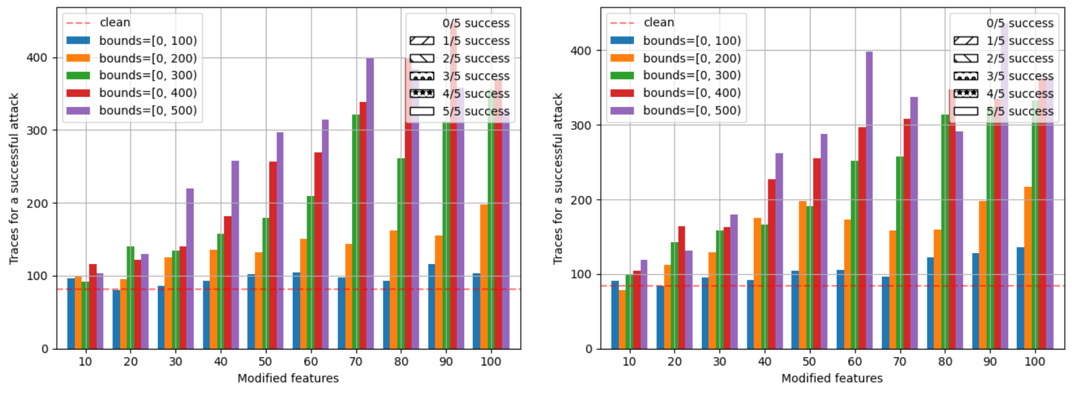
Overall, we conclude that this approach cannot increase the security of the target devices against DNN-based side-channel attacks. We will present possible reasons for this behavior in the next section, presenting our general conclusions derived from our experiments.



(a) Attack using the same model (MLP$_{\text{best}}$)

(b) Attack using different model (CNN$_{\text{best}}$)

Figure 4.10: Performance of correlation k-Point Protection using the MLP$_{\text{best}}$ model. Bar's pattern describes the number of feasible attacks within 2,000 attacking traces. If a bar is missing, then all of the corresponding attacks were infeasible.



(a) Attack using the same model (CNN$_{\text{best}}$)

(b) Attack using different model (MLP$_{\text{best}}$)

Figure 4.11: Performance of correlation k-Point Protection using the CNN$_{\text{best}}$ model. Bar's pattern describes the number of feasible attacks within 2,000 attacking traces. If a bar is missing, then all of the corresponding attacks were infeasible.

4.3.4. Conclusions

Before extracting our conclusions from the aforementioned results, it is important to recall how the proposed adversarial-based countermeasures operate. The difference of the three proposed methods is the way that they select which features will be distorted; the normal method lets the differential evolution algorithm decide the positions, the random method chooses uniformly at random these features, while the correlation method affects the top k features ordered by their correlation with the label.

Looking at the bigger picture, we can conclude that our methods can provide better security when using the CNN$_{\text{best}}$ model for the generation of the adversarial examples instead of the MLP$_{\text{best}}$ model. This behavior can be explained by the fact that CNN models outperform MLP models in general. Their convolution blocks can combine information of nearby features and make more accurate predictions. Thus, when our methods use a CNN model to generate the protective adversarial examples, these examples are more robust and can lead to a stronger drop of the true label's predicted probability. As a result, the generated adversarial examples manage to trick both CNN-based and MLP-based attacks.

Another notable observation is that there are some cases in which our adversarial-based countermeasure not only fails to decrease the performance of the attack, but the attack becomes easier. Our explanation on this effect is that our methods operate in these specific cases leaks information that the attacking models can exploit and, thus, they retrieve the secret subkey faster. This effect is met more or less in every method except for the random k-Point Protection method, and it reveals that this leakage is related to the added noise's positions. When these are chosen randomly, the attacking model cannot exploit that information, and the countermeasure works properly. In particular, in the case of the correlation k-Point Protection, the addition of large noise to the top k highly correlated features leads to a significant drop in their correlation with the true label. As a result, the DNN models seem to ignore these features and consider the next highly correlated features. Hence, the input's dimension is decreased, and the attack can be succeeded with fewer observations than the attacks using the clean dataset.

Finally, we can notice that, in some cases, smaller ranges of the added noise perform better than cases with larger ranges. One could say that this behavior is odd since the smaller ranges are a subset of the larger ones, and he would expect that the differential evolution algorithm would converge to the same solutions. However, our explanation for this behavior is that the differential evolution finds indeed solutions that minimize the prediction of the true label. However, these adversarial examples are used to train the attacking models and, thus, they become robust against them. Thus, in these cases, the adversarial example seems to have no effect on the attack's performance or even make it easier, for reasons explained above.

5
Performance compared to existing countermeasures

This chapter aims to answer the second research question, namely, to compare the performance of our adversarial-based countermeasure with other existing countermeasures proposed by the research community. First, we explain the motivation and the importance of such a comparison. Then, we present how we simulate the countermeasures under investigation. We further analyze the experimental process, the evaluation criteria for the comparison, and the results. Last, based on these results, we derive our conclusions regarding the second research question.

5.1. Motivation

In chapter 4, we presented a novel approach to design countermeasures against DNN-based side-channel attacks using adversarial example generation techniques. Besides the effectiveness, which is, of course, crucial, it is equally important to investigate what is the provided security compared to other countermeasures that have been proposed in the past by the research community. That comparison will reveal the added value that our solution provides against the existing ones.

First of all, it is important to classify our adversarial-based defense technique into either the hiding or the masking family of SCA countermeasures. By observing our proposed countermeasure's overall behavior, we notice that it considers a noise generator that cloaks the original traces and hides the leaked information on the power consumption channel. That approach is followed by hiding countermeasures, whereas masking introduces an additional secret entity that aims to increase the defense order. Hence, from our perspective, we classify our adversarial-based countermeasure as a hiding technique.

All things considered, the comparison of our countermeasure with other hiding techniques occurs intuitively. In contrast, comparing our proposed countermeasure with masking techniques might sound a bit odd at first look. However, we consider that this comparison is equally important. It can reveal the benefits and drawbacks of our technique compared to a family of countermeasures considered the most effective for protecting against side-channel attacks.

5.2. Simulation of protected datasets

To perform our comparison, we need datasets of power traces protected using the SCA countermeasures under examination. These datasets have to be extracted under the same conditions as our clean dataset. In detail, the source code, the targeted device, and the capturing setup have to be the same as the one used to extract the raw traces used for our clean dataset. If these conditions are not met, our comparison will be biased by external factors, and our outcomes will be unreliable.

Such acquisition of datasets is extremely tough. Thus, an alternative approach is needed. In the following sections, we present our approach to building datasets protected with the existing SCA countermeasures under examination.

5.2.1. Hiding countermeasures

Regarding the hiding countermeasures, a common approach is to simulate their behavior by applying a proper algorithm a posteriori to a clean dataset, similarly to what we did for our adversarial-based countermeasure. Wu and Picek [83] present how common hiding countermeasures can be simulated using this methodology. The rest of the present subsection presents their simulation algorithms in detail, explaining how a single clean trace can be altered to simulate a specific countermeasure properly. To apply their simulation techniques to our whole clean dataset, we use the process that is presented in algorithm 2.

Algorithm 2: Apply a countermeasure to a dataset

```
   Input  : $traces$: original traces
   Output : $new\_traces$: modified trace
1  $new\_traces \leftarrow []$                                // container for new trace
2  $i \leftarrow 0$
3  while $i < \texttt{len}(traces)$ do
4      $new\_traces[i] \leftarrow \texttt{addCountermeasure}(trace)$    // apply countermeasure to each trace
5      $i \leftarrow i + 1$
6  end while
7  return $new\_traces$
```

Gaussian noise

The addition of Gaussian noise as a countermeasure is already presented in section 4.2.1, where we used it to decrease the side-channel attacks' performance to the clean dataset. For clarity purposes, we present the simulation algorithm (see algorithm 3). The idea is quite simple; for each feature of the original trace, we select a random value from the Gaussian distribution with zero mean and standard deviation of std add we add to the original value of the feature.

Algorithm 3: Add Gaussian noise to a trace

```
   Input  : $trace$: original trace, $std$: the standard deviation of the Gaussian distribution
   Output : $new\_trace$: modified trace
1  Function addGaussianNoise (trace, std) is
2      $new\_trace \leftarrow []$                            // container for new trace
3      $i \leftarrow 0$
4      while $i < \texttt{len}(trace)$ do
5          $level \leftarrow \texttt{randomNormal}(0, std)$
6          $new\_trace[i] \leftarrow trace[i] + level$         // add noise to the trace
7          $i \leftarrow i + 1$
8      end while
9      return $new\_trace$
10 end Function
```

Desynchronization

To simulate the desynchronization countermeasure, we randomly select a desynchronization level from zero to $desync\_level$ for each trace. Then, we copy the original trace contents to a new container using the desynchronization as a shift offset (see algorithm 4). Applying this countermeasure to our clean dataset will lead to having uneven input lengths across the traces. To overcome this, we pad with zeros until the size of every trace is the same.

Random delay interrupts (RDIs)

For the random delay interrupts simulation, Wu and Picek [83] use the Floating Mean method [12]. Also, the authors enrich this method to add RDIs to a feature with a probability of 50%. Furthermore,

Algorithm 4: Add desynchronization to a trace

```
Input  : $trace$: original trace, $desync\_level$: maximum desynchronization level
Output : $new\_trace$: modified trace
Function addDesync (trace, desync_level) is
    $new\_trace \leftarrow []$                                // container for new trace
    $level \leftarrow$ randomUniform$(0, desync\_level)$
    $i \leftarrow 0$
    while $i + level <$ len$(trace)$ do
        $new\_trace[i + level] \leftarrow trace[i]$           // add desynchronization to the trace
        $i \leftarrow i + 1$
    end while
    return $new\_trace$
end Function
```

they add a small amplitude to the delayed tracepoints to simulate realistically the microprocessor's operations. The pseudocode for the addition of RDIs to our clean dataset is presented in algorithm 5. As this countermeasure randomly affects the time domain, the generated traces won't be equal in length. To fix this issue, we pad all the shorter traces with zero until all traces are even.

Algorithm 5: Add RDIs to a trace

```
Input  : $trace$: original trace, $a, b$: Floating Mean method parameters, such that $a > b$,
         $rdi\_amplitude$: injected peak when RDIs are applied
Output : $new\_trace$: modified trace
Function addRDI (trace, a, b, rdi_amplitude) is
    $new\_trace \leftarrow []$                                // container for new trace
    $i \leftarrow 0$
    while $i <$ len$(trace)$ do
        $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
        $rdi\_occurrence \leftarrow$ randomUniform$(0, threshold \times 2)$
        if $rdi\_occurrence > threshold$ then
            $m \leftarrow$ randomUniform$(0, a - b)$
            $rdi\_num \leftarrow$ randomUniform$(m, m + b)$     // number of RDIs to be added
            $j \leftarrow 0$
            while $j < rdi\_num$                               // add RDIs to the trace
             do
                $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
                $new\_trace[i] \leftarrow new\_trace[i].append(trace[i] + rdi\_amplitude)$
                $new\_trace[i] \leftarrow new\_trace[i].append(trace[i + 1])$
                $j \leftarrow j + 1$
            end while
        end if
        $i \leftarrow i + 1$
    end while
    return $new\_trace$
end Function
```

Clock-jitters

Clock jitters can be simulated by arbitrarily adding or removing features from our input [83]. In detail, while we iterate through our input's features, we select a random number r within a certain range $[-level, level]$ representing how many features will be added or removed. If r is negative, we remove the next r points of the original trace; otherwise, we add r artificial features to the original trace. For the amplitude of the added tracepoints, we calculate the mean amplitude of the tracepoints before and after the added features. Similar to the two previous techniques that affect the time domain of the original input, the modified traces are padded with zeros to preserve an equal input size (see algorithm 6).

Algorithm 6: Add Clock-jitters to a trace

```
Input  : $trace$: original trace, $mean$: the mean of the Gaussian distribution,
         $clock\_jitters\_level$: maximum size of a clock jitter
Output : $new\_trace$: modified trace
Function addClockJitter (trace, clock_jitters_level) is
    $new\_trace \leftarrow []$                                  // container for new trace
    $i \leftarrow 0$
    while $i < \texttt{len}(trace)$ do
        $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
        $r \leftarrow \texttt{randomUniform}(-clock\_jitters\_level, clock\_jitters\_level)$   // level of clock jitters
        if $r < 0$ then
            $i \leftarrow i + r$                                 // skip points
        else
            $j \leftarrow 0$
            $average\_amplitude \leftarrow (trace[i] + trace[i+1])/2$
            while $j < r$ do
                $new\_trace \leftarrow new\_trace.append(average\_amplitude)$   // add points
                $j \leftarrow j + 1$
            end while
        end if
        $i \leftarrow i + 1$
    end while
    return $new\_trace$
end Function
```

5.2.2. Masking techniques

As we explain on section 2.4.5, this technique protects the intermediate value by applying an operation with a mask that randomizes the relation between the sensor data and the observations. Thus, the simulation of a masking countermeasure and its application on a clean dataset, as we did in the previous subsection, is not possible. Thus, we will use the original ASCAD, which consists of tracepoints focused on the third masked byte, for this comparison [64]. As our clean dataset and ASCAD are extracted from the same device, we assume that a masked version of our clean dataset would have the same protection level as ASCAD.

5.3. Results

In this section, we compare the performance of our adversarial-based countermeasure with the defensive techniques presented above. First, we present our experimental process and the metrics we use to derive our conclusions. Then, we perform our experiments separately for each countermeasure under examination and present our results and outcomes. Finally, at the end of the present section, we present our general conclusions regarding our second research question.

5.3.1. Experimental process

Our goal is to compare the performance of the methods presented in section 4.3 with existing defensive techniques. Thus, we need to select implementations of that section that present the maximum level of protection based on our SCA metrics. Out of the three approaches on selecting the affected features, we rule out the correlation variant. As presented, this approach did not succeed in protecting our clean dataset against the performed side-channel attacks. Furthermore. in the previous chapter, we concluded that the usage of the CNN$_{\text{best}}$ model for the generation of the adversarial examples could build stronger protection against both attacking models. On the contrary, the configurations where the MLP$_{\text{best}}$ model is used seem to create adversarial examples that protect the same attacking model. Hence, as the usage of the latter model cannot provide sufficient overall protection, we do not take into account these configurations in our comparison. Overall, we select the implementations $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$, $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$, and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ to perform our comparisons, as these values of affected features and bounded noise make the attacks infeasible (using 2,000 attacking traces) and keep l_2 distance metric relatively small.

In this section, we decide to enrich the attacking scenarios by performing side-channel attacks using different leakage models. It is expected that an attacking model has a different performance according to the leakage model used for the attack. If the attack is on a protected dataset, this difference can help us compare the effectiveness of each countermeasure under investigation against multiple leakage models. For our experiments, we perform attacks using the identity and Hamming weight leakage models, introduced in section 2.4.1.

Moreover, we extend our attacking models by adding two state-of-the-art models from recent literature. The first model that we include in our experiments is introduced by Rijsdijk et al. [67], where authors present a method for generating neural networks using Reinforcement Learning for hyperparameter tuning. The other model we use in our attacks is presented by Wu et al. [84], where the authors used Bayesian Optimization to tune the attacking model's hyperparameters. In both works, the authors use their techniques to produce models for the random key ASCAD, which is extracted from the same raw traces as our clean dataset but focuses on the third subkey that is protected using first-order masking. Thus, their common origin is sufficient to consider that these models will have a relatively strong performance against our clean and simulated datasets. Note that the authors generate different models according to the used leakage model. Thus, in our experiments, we use the respective model for each leakage model. We denote the two state-of-the-art models as CNN$_{\text{Rijsdijk}}$ and MLP$_{\text{Wu}}$, while we describe in detail their structure in appendices C.3 and C.4 respectively. The addition of these two state-of-the-art models will help us investigate our defensive techniques' transferability even further.

The evaluation of our experiments is performed using the metrics presented in section 4.3. In particular, we measure the metric $n_{success}$, which represents the number of the required traces to achieve $GE = 1$ and $SR = 100\%$. Moreover, to evaluate the behavior of each countermeasure to the number of the processed traces, we use the metrics GE_x and SR_x with $x = \{10, 50, 100\}$ that represent the achieved guessing entropy and success rate after processing $x\%$ of a fixed amount of traces.

In tables 5.1 and 5.2, we present the aforementioned metrics for side-channel attacks against the clean dataset and the protected datasets using selected implementations of our adversarial-based countermeasure, respectively. However, before moving to the main goal of this chapter, which is to compare our countermeasure with existing hiding and masking defensive techniques, it is essential to evaluate the performance of our methods against the newly introduced attacking and leakage models.

Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
clean	ID	MLP$_{\text{best}}$	80	1.00	1.00	1.00	1.00	1.00	1.00
		CNN$_{\text{best}}$	71	1.00	1.00	1.00	1.00	1.00	1.00
		CNN$_{\text{Rijsdijk}}$	252	1.05	0.97	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	352	1.67	0.82	1.00	1.00	1.00	1.00
	HW	MLP$_{\text{best}}$	157	1.00	1.00	1.00	1.00	1.00	1.00
		CNN$_{\text{best}}$	158	1.00	1.00	1.00	1.00	1.00	1.00
		CNN$_{\text{Rijsdijk}}$	432	1.83	0.77	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	301	1.03	0.97	1.00	1.00	1.00	1.00

Table 5.1: SCA metrics for attacks against our clean datasets

First of all, we observe that the performance of our methods drops significantly when the leakage model of the attack is changed from ID to HW. For the attacks that use the MLP$_{\text{best}}$ attacking model, we observe that the attacks are still infeasible within 2,000 attacking traces, but the change of the leakage model increases the success rate by 30% to 50%, depending on the targeted dataset. However, in the case of the CNN$_{\text{best}}$ attacking model, the performance of the adversarial-based methods drops majorly. As we see in table 5.2, when the CNN$_{\text{best}}$ is combined with the ID leakage model, it fails to succeed within 2,000 traces, while it can guess the secret subkey correctly when combined with the Hamming

Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$	ID	MLP_{best}	>2000	78.02	0.03	38.97	0.17	23.23	0.36
		CNN_{best}	>2000	47.25	0.09	7.32	0.54	1.61	0.84
		$\text{CNN}_{\text{Rijsdijk}}$	283 (5/5)	1.22	0.93	1.00	1.00	1.00	1.00
		MLP_{Wu}	186 (5/5)	1.02	1.00	1.00	1.00	1.00	1.00
	HW	MLP_{best}	>2000	52.73	0.06	11.25	0.39	2.44	0.70
		CNN_{best}	452 (5/5)	2.37	0.75	1.00	1.00	1.00	1.00
		$\text{CNN}_{\text{Rijsdijk}}$	514 (5/5)	3.11	0.65	1.00	1.00	1.00	1.00
		MLP_{Wu}	979 (5/5)	8.81	0.41	1.01	0.99	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$	ID	MLP_{best}	>2000	72.93	0.02	25.30	0.15	11.53	0.35
		CNN_{best}	>2000	66.85	0.03	20.06	0.22	5.54	0.49
		$\text{CNN}_{\text{Rijsdijk}}$	257 (5/5)	1.06	0.97	1.00	1.00	1.00	1.00
		MLP_{Wu}	128 (5/5)	1.00	1.00	1.00	1.00	1.00	1.00
	HW	MLP_{best}	>2000	41.70	0.07	4.62	0.56	1.29	0.88
		CNN_{best}	428 (5/5)	1.87	0.79	1.00	1.00	1.00	1.00
		$\text{CNN}_{\text{Rijsdijk}}$	459 (5/5)	2.52	0.68	1.00	1.00	1.00	1.00
		MLP_{Wu}	697 (5/5)	6.14	0.51	1.00	1.00	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$	ID	MLP_{best}	>2000	97.86	0.01	59.02	0.06	39.63	0.15
		CNN_{best}	>2000	78.65	0.02	37.09	0.11	18.94	0.27
		$\text{CNN}_{\text{Rijsdijk}}$	243 (5/5)	1.10	0.96	1.00	1.00	1.00	1.00
		MLP_{Wu}	133 (5/5)	1.00	1.00	1.00	1.00	1.00	1.00
	HW	MLP_{best}	>2000	62.56	0.04	15.12	0.29	3.69	0.60
		CNN_{best}	451 (5/5)	2.19	0.79	1.00	1.00	1.00	1.00
		$\text{CNN}_{\text{Rijsdijk}}$	473 (5/5)	2.43	0.68	1.00	1.00	1.00	1.00
		MLP_{Wu}	723 (5/5)	6.70	0.46	1.00	1.00	1.00	1.00

Table 5.2: SCA metrics for attacks against datasets generated by our adversarial-based countermeasure. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">2000" denotes that all five attacks were unsuccessful.

weight leakage model. This performance drop can be explained by the fact that our methods design adversarial examples to hide the true label of the dataset. Thus, when the dataset is re-labeled, the effectiveness of the adversarial example drops. What is important to explore is how the change of the leakage model can affect the performance of other existing countermeasures.

Another important observation is the performance of the state-of-the-art models against our adversarial-based countermeasure. We see that the performance of the $\text{CNN}_{\text{Rijsdijk}}$ and MLP_{Wu} attacking models is not affected by the presence of our adversarial-based countermeasure. On the one hand, the $\text{CNN}_{\text{Rijsdijk}}$ model can successfully retrieve the secret subkey using the same amount of attacking traces regardless of the used leakage model when it attacks either the unprotected dataset or the adversarial-protected datasets. That behavior is impressive if we consider that this model is not optimized for these specific datasets. On the other hand, our methods are also ineffective against the MLP_{Wu} attacking model. When the ID leakage model is used, MLP_{Wu} retrieves the secret subkey using fewer traces than the attack against the unprotected dataset. When the HW leakage model is used, the performance of the attack slightly drops, but still, the key is retrieved before processing 50% of the attacking traces.

In the next sections, we investigate the effectiveness of each one of the countermeasures presented in section 5.2. First, we simulate a protected dataset using the corresponding defensive technique[1]. Then, we measure the performance of the attacks against the protected countermeasure using the metrics presented above. Last, we use these metrics and the ones presented in tables 5.1 and 5.2 to compare the defensive technique under investigation with our adversarial-based countermeasure. For better visualization, we provide figures that present the achieved guessing entropy and success rate for the attacks on the countermeasures under examination in each section.

5.3.2. Hiding countermeasures

In this section, we compare our adversarial-based defensive technique with selected existing hiding countermeasures. In particular, the countermeasures that we consider are Gaussian noise, desynchro-

[1]The values of the parameters for the simulation are explained in the corresponding section

nization, RDIs, and clock jitters.

Regarding the simulation of datasets that are protected with these hiding countermeasures, we apply the techniques presented in section 5.2.1. It is important to pick appropriate values for the algorithms' parameters to perform a balanced comparison of the attacks on the simulated datasets. For this reason, we use the SNR metric that represents the leaked information of a dataset [64]. In detail, we first measure the SNR values for our generated datasets by the methods $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$, $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$, and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$, which are 0.0104, 0.0110, and 0.0098, respectively. Then, we pick a value for the simulation parameter(s) using the "trial and error" approach that generates a simulated dataset with a similar SNR. However, as we will see later in this section, sometimes we cannot achieve the desired SNR metric following the "trial and error" approach because it doesn't change proportionally to the changes on the simulating parameters, but it remains constant. In these cases, we consult the simulations presented by [83] and pick similar values since it is proven that these values can provide against side-channel attacks on ASCAD.

Gaussian Noise

To perform our comparison, we need first to generate a dataset protected with the Gaussian noise countermeasure. We simulate such a dataset by running the simulation technique presented in algorithm 3. To determine the value of the standard deviation of the Gaussian noise, we take the following steps. First, we measure the SNR of the datasets that are protected with our adversarial-based countermeasure. Then, through test and trial, we find a value for the standard deviation that generates a dataset protected by the Gaussian noise countermeasure with the same SNR. This approach will allow us to perform a fair comparison as we compare scenarios with similar leaked information. After following this methodology, we conclude that the appropriate standard deviation is $std = 80$, which generates a dataset with $SNR = 0.1034$ and $l_2 = 2529.01$.

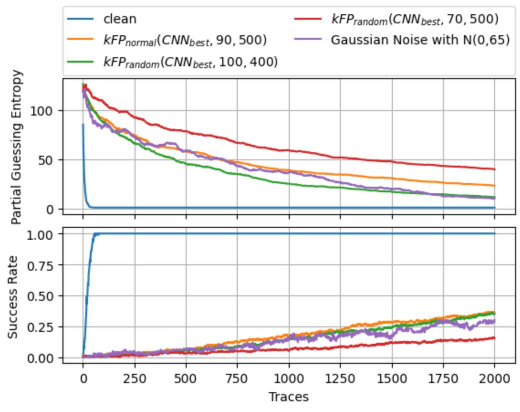
We attack the generated dataset using our attacking models. In table 5.3, we present the performance of these attacks based on our metrics, while in figure 5.1, we present the progress of the guessing entropy and success rate over the amount of processed attacking traces. For reference purposes, we include in the plots the corresponding metrics for the attacks on the clean dataset and the datasets generated by our selected adversarial-based methods.

Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
Gaussian noise ($N(0,80)$)	ID	MLP$_{\text{best}}$	>2000	79.87	0.05	36.76	0.14	9.98	0.30
		CNN$_{\text{best}}$	>2000	44.54	0.13	3.50	0.62	1.18	0.91
		CNN$_{\text{Rijsdijk}}$	>2000	65.15	0.05	11.57	0.29	2.00	0.71
		MLP$_{\text{Wu}}$	>2000	73.63	0.03	25.05	0.08	11.04	0.27
	HW	MLP$_{\text{best}}$	>2000	53.46	0.05	5.08	0.58	1.44	0.85
		CNN$_{\text{best}}$	>2000	42.75	0.04	7.36	0.40	2.59	0.70
		CNN$_{\text{Rijsdijk}}$	>2000	85.37	0.00	43.68	0.07	24.73	0.07
		MLP$_{\text{Wu}}$	>2000	64.02	0.05	23.46	0.12	5.76	0.51

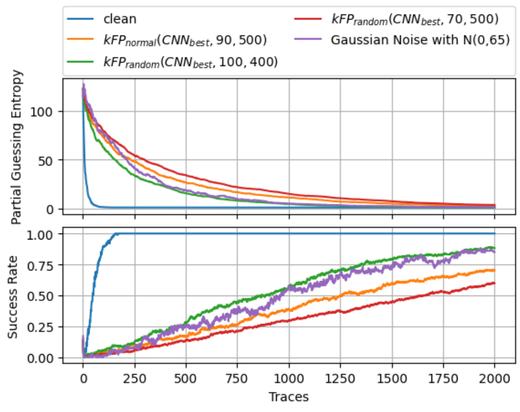
Table 5.3: SCA metrics for attacks against datasets generated by Gaussian noise

Regarding the attacks based on the MLP$_{\text{best}}$ model, we observe in figures 5.1a and 5.1b that the Gaussian noise countermeasure performs similarly to our methods. In particular, for that attacks that use the ID leakage model, the Gaussian noise has almost identical metrics with our $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ which performs better than the other two implementations of our methods. At the same time, using the HW leakage model for the attacks, Gaussian noise offers similar protection to our $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$. This entails that the Gaussian noise countermeasure performs worse than our $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$ and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ under this attacking scenario.

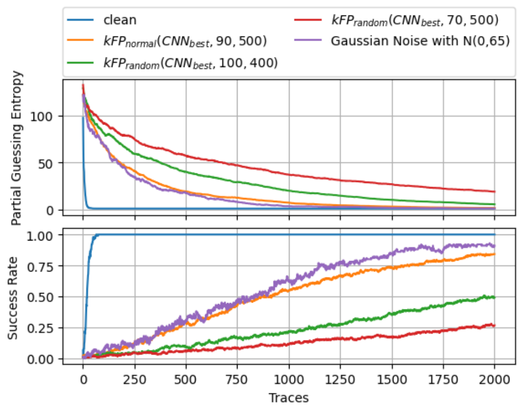
As to CNN$_{\text{best}}$-based attacks, our $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$ and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ seem to perform better than Gaussian noise when the leakage model is the identity model. In particular, we observe in figures 5.1c and 5.1d that in this specific setup success rate of the attack after processing all the 2,000 traces is 80% when the dataset is protected using Gaussian noise or the $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$ method. At the same time, it is 49% and 39% when the $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$ or $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ methods are applied on the clean dataset, respectively. However, the results are different when the attacks are using the Hamming weight model. As we already mentioned in the previous section, our protection techniques fail to provide significant protection against this



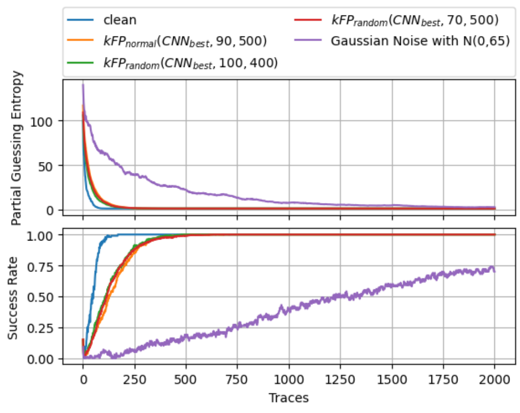
(a) Attack using MLP_{best} (ID leakage model)



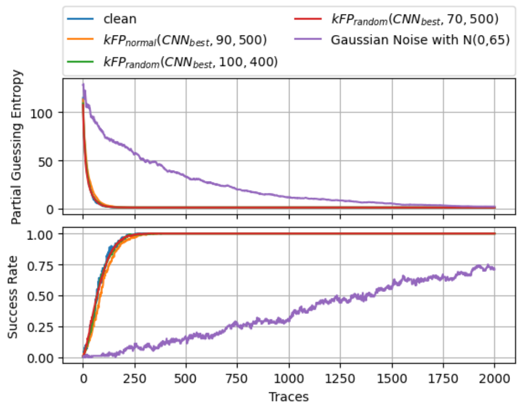
(b) Attack using MLP_{best} (HW leakage model)



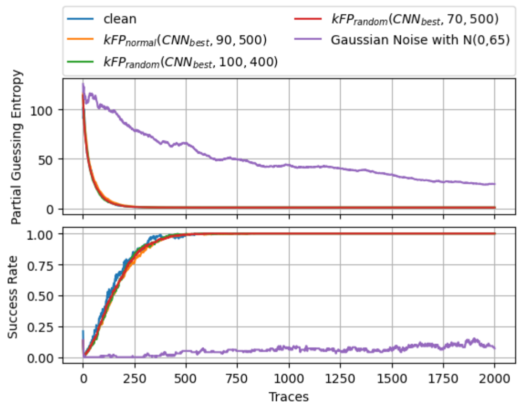
(c) Attack using CNN_{best} (ID leakage model)



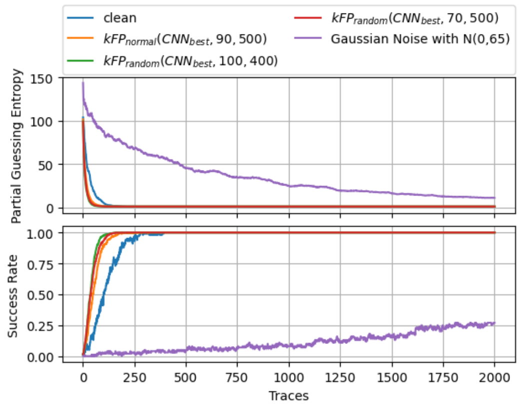
(d) Attack using CNN_{best} (HW leakage model)



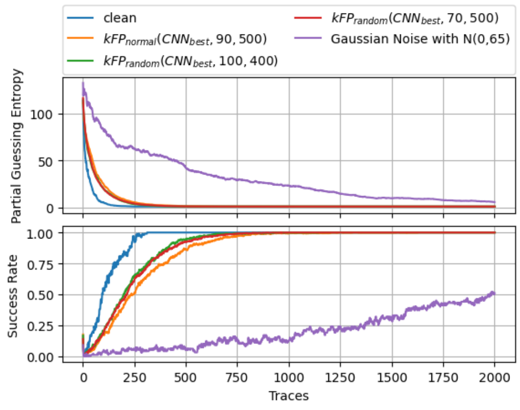
(e) Attack using $\text{CNN}_{\text{Rijsdijk}}$ (ID leakage model)



(f) Attack using $\text{CNN}_{\text{Rijsdijk}}$ (HW leakage model)



(g) Attack using MLP_{Wu} (ID leakage model)



(h) Attack using MLP_{Wu} (HW leakage model)

Figure 5.1: Comparison of our proposal with Gaussian noise

leakage model when the CNN$_{\text{best}}$ model is used for the attack. On the contrary, the Gaussian noise countermeasure offers significant protection even against this leakage model.

As we see in figures 5.1e to 5.1h, Gaussian noise can protect sufficiently the secret key against the state-of-the-art attacking models. In particular, all of the attacks we performed against this countermeasure using the CNN$_{\text{Rijsdijk}}$ and MLP$_{\text{Wu}}$ models are infeasible regardless of the used leakage model. Hence, we see that the Gaussian countermeasure is much stronger than our methods under these attacking scenarios.

Overall, we conclude that Gaussian noise provides more generic protection than our methods. In the examined attacking scenarios, our methods succeed to perform better than the Gaussian noise countermeasure for attacks that use the MLP$_{\text{best}}$ model and for the scenario that is used for the generation of the adversarial examples; namely, attacks that use the CNN$_{\text{best}}$ attacking model and the identity leakage models. For the rest of the scenarios, Gaussian countermeasure was achieved to protect the secret key in the attacking scenarios, while our methods failed.

Desynchronization

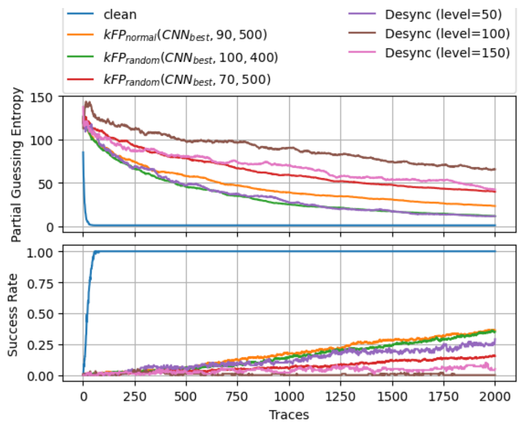
Similar to the process we followed above, we first create a dataset that is protected using the desynchronization countermeasure. However, the described approach for determining the parameters using SNR seems to be not applicable in this case. In particular, the SNR metric drops below the desired level even by using a small desynchronization. Thus, we follow a different approach to consult other researchers who used the same simulation technique for this countermeasure. In [83], the authors propose that a maximum desynchronization of 50 points can provide sufficient protection on ASCAD against DNN-based SCA. As this dataset is protected with first-order masking, we can safely assume that it requires less desynchronization than our clean dataset to get protected. Thus, we decide to simulate this countermeasure by running algorithm 4 using $level = \{50, 100, 150\}$, which generates three datasets with SNR of 0.067, 0.066, and 0.065 respectively.

Table 5.4 summarizes the performance of the desynchronization countermeasure based on our metrics, while figure 5.2 presents the progress of the GE and SR metrics compared to our methods.

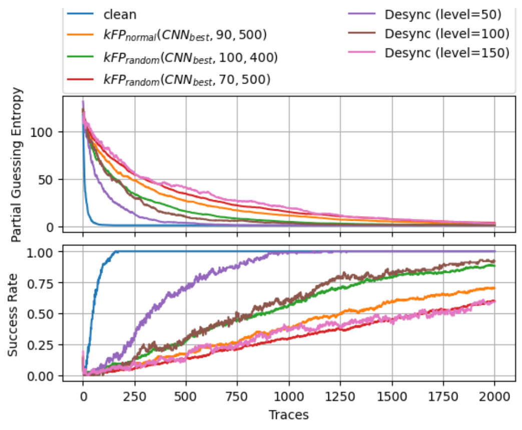
Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
Desync ($level = 50$)	ID	MLP$_{\text{best}}$	>2000	74.33	0.01	27.41	0.11	11.55	0.29
		CNN$_{\text{best}}$	>2000	107.30	0.00	92.44	0.03	70.46	0.01
		CNN$_{\text{Rijsdijk}}$	79	1.00	1.00	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	1305	16.85	0.19	1.17	0.96	1.00	1.00
	HW	MLP$_{\text{best}}$	1113	18.65	0.23	1.02	0.99	1.00	1.00
		CNN$_{\text{best}}$	298	1.79	0.76	1.00	1.00	1.00	1.00
		CNN$_{\text{Rijsdijk}}$	296	1.10	0.93	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	1301	15.19	0.20	1.03	0.98	1.00	1.00
Desync ($level = 100$)	ID	MLP$_{\text{best}}$	>2000	116.75	0.02	90.08	0.00	65.47	0.00
		CNN$_{\text{best}}$	>2000	122.21	0.00	91.10	0.01	77.67	0.01
		CNN$_{\text{Rijsdijk}}$	65	1.00	1.00	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	39.66	0.03	2.94	0.66	1.05	0.96
	HW	MLP$_{\text{best}}$	>2000	40.83	0.09	3.27	0.60	1.11	0.92
		CNN$_{\text{best}}$	642	6.46	0.47	1.00	1.00	1.00	1.00
		CNN$_{\text{Rijsdijk}}$	219	1.02	0.99	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	1558	18.82	0.19	1.17	0.95	1.00	1.00
Desync ($level = 150$)	ID	MLP$_{\text{best}}$	>2000	91.35	0.02	69.91	0.02	42.18	0.05
		CNN$_{\text{best}}$	>2000	115.14	0.00	89.03	0.02	73.43	0.01
		CNN$_{\text{Rijsdijk}}$	70	1.00	1.00	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	66.32	0.06	11.23	0.31	2.58	0.62
	HW	MLP$_{\text{best}}$	>2000	65.49	0.05	19.06	0.38	3.08	0.59
		CNN$_{\text{best}}$	1734	26.52	0.19	1.10	0.93	1.00	1.00
		CNN$_{\text{Rijsdijk}}$	319	1.07	0.98	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	37.02	0.07	2.79	0.69	1.04	0.97

Table 5.4: SCA metrics for attacks against datasets protected using desynchronization

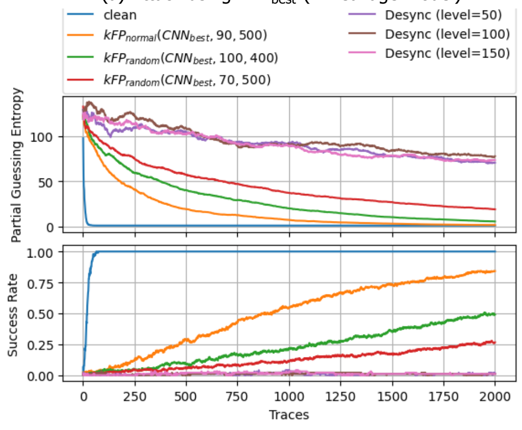
Regarding the attacks based on the MLP$_{\text{best}}$ model, we observe that our countermeasure can perform better than the desynchronization countermeasure in most of the cases. In detail, when the attacks are using the ID leakage model, we observe that desynchronization of 50 and 100 points have



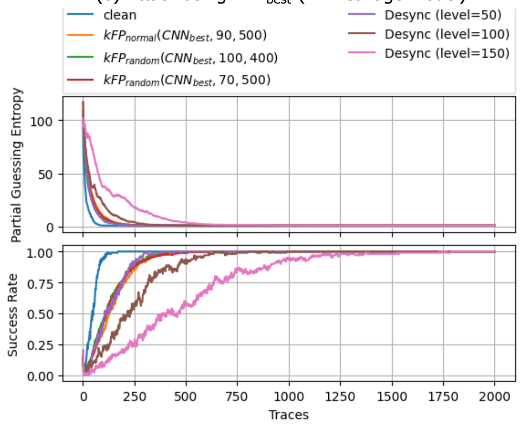
(a) Attack using MLP_{best} (ID leakage model)



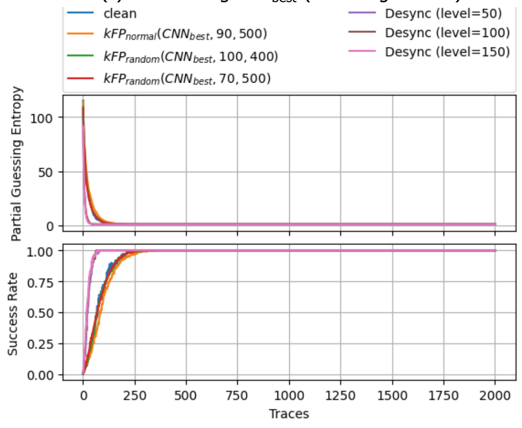
(b) Attack using MLP_{best} (HW leakage model)



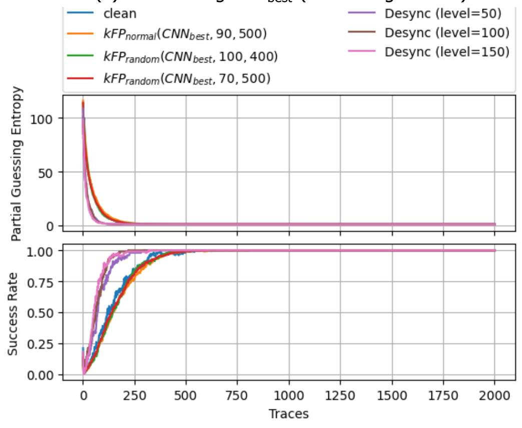
(c) Attack using CNN_{best} (ID leakage model)



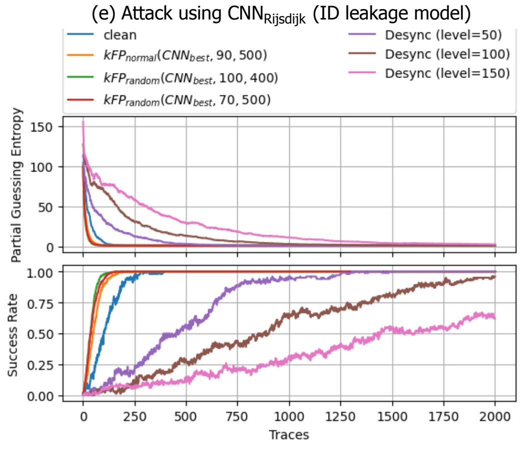
(d) Attack using CNN_{best} (HW leakage model)



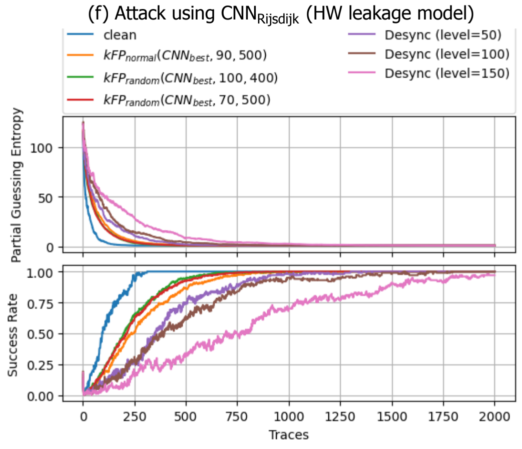
(e) Attack using $\text{CNN}_{\text{Rijsdijk}}$ (ID leakage model)



(f) Attack using $\text{CNN}_{\text{Rijsdijk}}$ (HW leakage model)



(g) Attack using MLP_{Wu} (ID leakage model)



(h) Attack using MLP_{Wu} (HW leakage model)

Figure 5.2: Comparison of our proposal with desynchronization

worse performance than our countermeasures. In comparison, desynchronization of 150 points performs similarly to our $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$, which presents the best performance in this attacking scenario. Similar observations can be derived regarding the attacking scenario where Hamming weight is used as a leakage model.

On the contrary, desynchronization seems to work better than our methods when the attacks use the CNN_{best} and identity as attacking and leakage model, accordingly. In particular, the success rate of the attacks against this countermeasure is 0%, 2%, and 8% when the maximum desynchronization is set to 50, 100, and 150 points, respectively. At the same time, the best performance among our methods in this attacking scenario is met in $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ with $SR = 39\%$. When the leakage model changes to HW, we observe that the protection of this countermeasure drops significantly, as it happens to our methods. However, it still performs better than our countermeasures.

Moving to the attacks that use the $\text{CNN}_{\text{Rijsdijk}}$, we observe that desynchronization is less effective than our methods. In particular, $\text{CNN}_{\text{Rijsdijk}}$ guesses the secret subkey successfully using fewer traces compared to when attacking the clean dataset, regardless of the level of the desynchronization.

About the attacks based on the MLP_{Wu} model, the performance of the method depends on the use used leakage model. On the one hand, using the ID leakage model for the attack, we see that the SR slowly converges to 100% for $level = 50$ after processing around 50% of the traces, while for $level = \{100, 150\}$ the SR_{100} is 96% and 62%, accordingly. At the same time, our methods fail to protect this attacking setup. On the other hand, using the HW leakage model decreases the performance of the attack. For $level = \{50, 100\}$ the SR metric approaches 100% after processing around 1,000 attacking traces, while our methods converge to this percentage using 750 traces. For $level = 150$, the final SR is 97%.

To summarize, we could say that the desynchronization countermeasure does not provide any added value compared to our proposed countermeasure. On the one side, when our methods are effective, the performance of the desynchronization is similar or worse, except for the attacking scenarios that use the CNN_{best} model. On the other hand, when our methods are ineffective, desynchronization provides insignificant protection. The attack is usually feasible within 2,000 traces or the final SR is close to 100%, or it fails too. At the same time, it seems that desynchronization leaked information that was exploited by the powerful $\text{CNN}_{\text{Rijsdijk}}$ and managed to improved its performance.

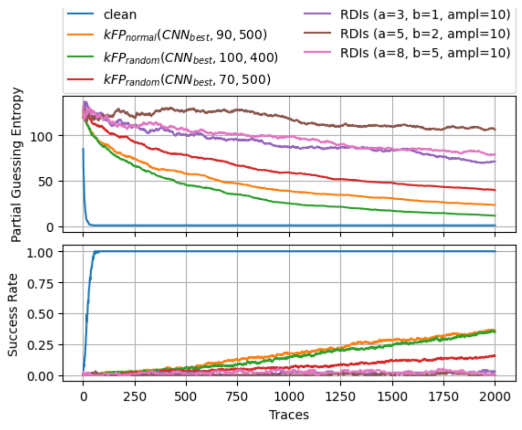
RDIs

For the simulation of the RDI countermeasure, we use algorithm 5. Regarding selecting its parameters, we refer again to the work of Wu and Picek [83], as the usage of the SNR is not conclusive for this case as well. In their work, the authors use $(a, b, rdi\_amplitude) = (8, 5, 10)$ to create a protected dataset using RDIs. As their results indicate that these parameters can provide strong protection against the dataset, we will use them for our experiments too. The execution of the simulation algorithm with these parameters produces a dataset with 5358 points per trace, which is more than 5 times greater than the clean dataset, $SNR = 0.071$, and euclidean distance $l_2 = 5023.14$. As we would like to have a fair comparison with our methods, we decide to generate two more datasets with smaller interrupts using the parameters $(a, b, rdi\_amplitude) = \{(3, 1, 10), (5, 2, 10)\}$. Our simulation using the first set of parameters produces a dataset of input size 1764 tracepoints, $SNR = 0.069$, and $l_2 = 2816.86$, while the second configuration creates a dataset of input size 3141 tracepoints, $SNR = 0.075$, and $l_2 = 3867.59$.

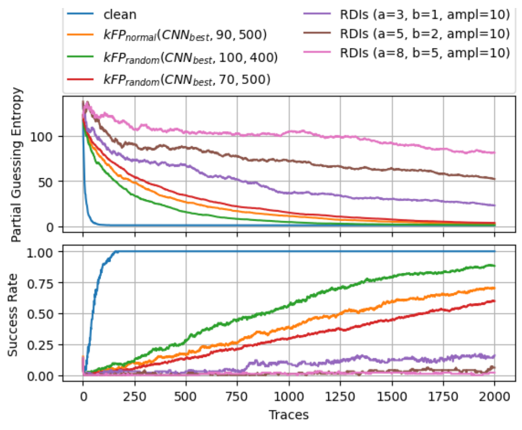
In table 5.5, we present the measured metrics for our attacking scenarios. In contrast, the progress of the achieved guessing entropy and success rate is presented in figure 5.3. As we can see, this particular countermeasure can offer extremely high protection against most attacking scenarios, especially when using parameters that tend to generate large delay interrupts.

Regarding the comparison with our methods, we see that the RDIs can perform significantly better. In detail, we see that the only attacking scenario where our methods can compete with the RDIs is the one where the MLP_{best} and ID models are used as the attacking and leakage models, respectively. In this case, our $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$ is performing similarly to the three simulated RDIs, while the $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$ and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$ are performing worse. When the leakage model changes to HW, then the protection of our methods is significantly decreased, but the attacks against the RDI-protected datasets still have a success rate of less than 30%.

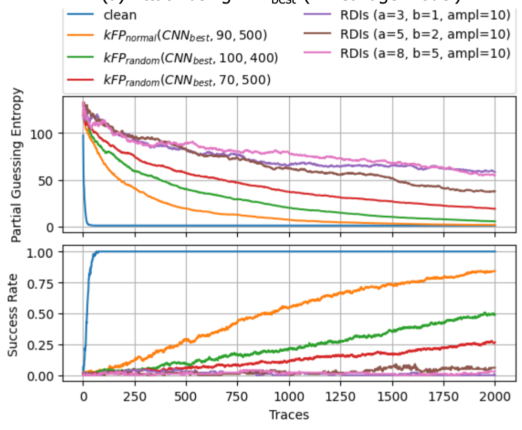
Similar observations can be derived from figure 5.3c, where the attack is performed with the CNN_{best} attacking model and the identity leakage model. For the attacks that combine the CNN_{best} model with



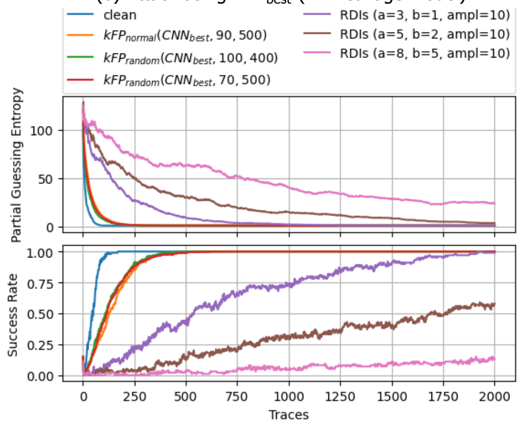
(a) Attack using MLP_{best} (ID leakage model)



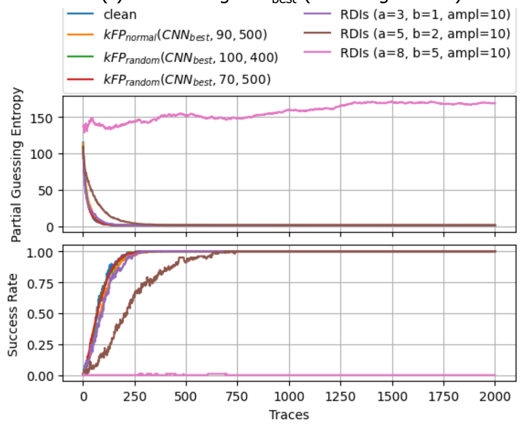
(b) Attack using MLP_{best} (HW leakage model)



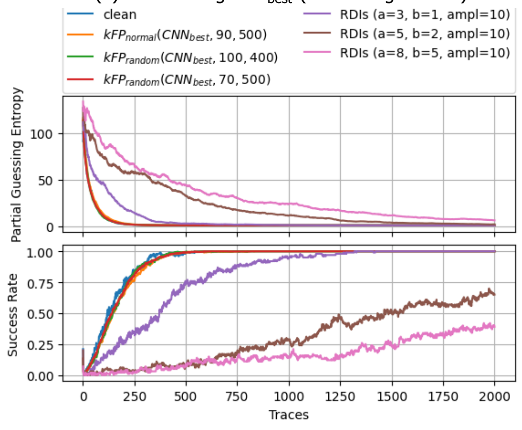
(c) Attack using CNN_{best} (ID leakage model)



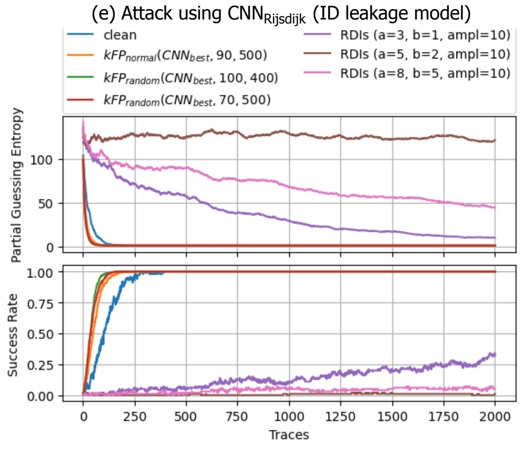
(d) Attack using CNN_{best} (HW leakage model)



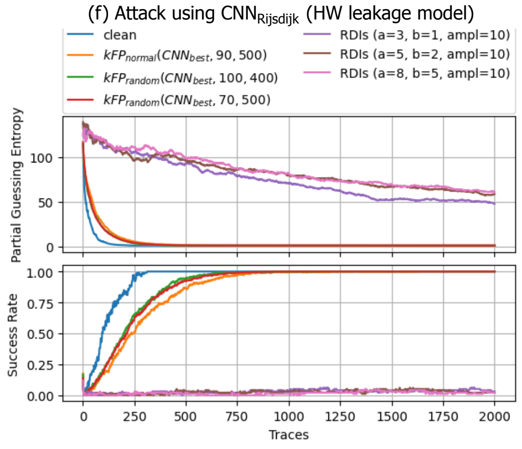
(e) Attack using $CNN_{Rijsdijk}$ (ID leakage model)



(f) Attack using $CNN_{Rijsdijk}$ (HW leakage model)



(g) Attack using MLP_{Wu} (ID leakage model)



(h) Attack using MLP_{Wu} (HW leakage model)

Figure 5.3: Comparison of our proposal with RDIs

Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
RDI ($a = 3, b = 1,$	ID	MLP$_{\text{best}}$	>2000	112.79	0.00	86.88	0.02	71.09	0.03
$rdi\_amplitude = 10$)		CNN$_{\text{best}}$	>2000	97.36	0.03	65.71	0.02	58.26	0.00
		CNN$_{\text{Rijsdijk}}$	285	1.34	0.89	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	76.27	0.04	28.85	0.11	9.97	0.32
	HW	MLP$_{\text{best}}$	>2000	76.14	0.03	36.69	0.09	22.96	0.16
		CNN$_{\text{best}}$	1918	34.93	0.14	2.11	0.74	1.01	0.99
		CNN$_{\text{Rijsdijk}}$	1319	19.46	0.27	1.27	0.94	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	108.36	0.02	70.65	0.03	47.72	0.03
RDI ($a = 5, b = 2$	ID	MLP$_{\text{best}}$	>2000	126.26	0.00	118.25	0.00	106.09	0.00
$rdi\_amplitude = 10$)		CNN$_{\text{best}}$	>2000	98.24	0.01	62.05	0.01	37.48	0.06
		CNN$_{\text{Rijsdijk}}$	690	6.42	0.50	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	125.64	0.00	128.81	0.00	121.52	0.01
	HW	MLP$_{\text{best}}$	>2000	90.69	0.01	72.01	0.01	52.23	0.06
		CNN$_{\text{best}}$	>2000	57.89	0.02	14.90	0.31	3.52	0.58
		CNN$_{\text{Rijsdijk}}$	>2000	56.99	0.04	11.43	0.28	2.09	0.65
		MLP$_{\text{Wu}}$	>2000	104.79	0.00	79.77	0.03	58.48	0.02
RDI ($a = 8, b = 5,$	ID	MLP$_{\text{best}}$	>2000	110.11	0.00	98.54	0.02	78.89	0.00
$rdi\_amplitude = 10$)		CNN$_{\text{best}}$	>2000	96.71	0.01	78.47	0.01	54.89	0.03
		CNN$_{\text{Rijsdijk}}$	>2000	139.61	0.00	159.44	0.00	168.84	0.00
		MLP$_{\text{Wu}}$	>2000	91.88	0.01	68.01	0.06	44.24	0.05
	HW	MLP$_{\text{best}}$	>2000	116.29	0.00	104.62	0.00	81.33	0.02
		CNN$_{\text{best}}$	>2000	75.69	0.01	40.34	0.05	23.76	0.13
		CNN$_{\text{Rijsdijk}}$	>2000	71.41	0.03	23.90	0.15	6.33	0.39
		MLP$_{\text{Wu}}$	>2000	108.43	0.01	81.32	0.01	60.90	0.02

Table 5.5: SCA metrics for attacks against datasets protected using RDIs

the Hamming weight leakage model, as noted already, the performance of our methods drops majorly. At the same time, RDIs offer strong protection against the executed side-channel attacks. The top performance is observed for the simulation with parameters $(a, b, rdi\_amplitude) = (8, 5, 10)$.

The power of the CNN$_{\text{Rijsdijk}}$ model is once again illustrated in figure 5.4e. Using the ID leakage model, this attacking model can guess the secret key correctly after processing 223 and 888 attacking traces when attacking the two datasets with the smaller RDIs. However, the attack is completely failed when it targets the simulated dataset with parameters $(a, b, rdi\_amplitude) = (8, 5, 10)$. We see similar results when the leakage model is changed to HW.

Finally, we see that RDIs can provide better protection than our methods against the MLP$_{\text{Wu}}$ model. In particular, RDIs succeed in decreasing the SR_{100} attacks of most of the attacks to almost 0%, even in the case that the RDIs are small.

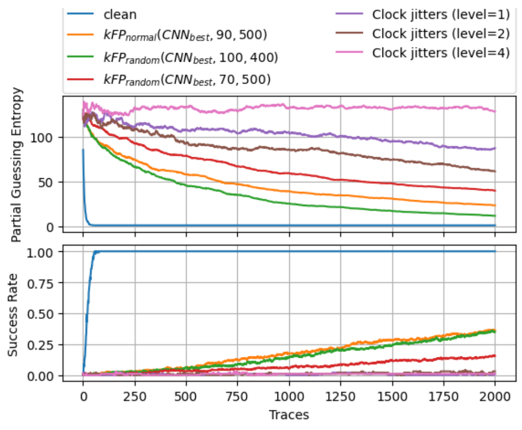
In conclusion, our methods cannot compete against RDI countermeasure since our methods failed to provide similar protection. However, RDIs increase the execution time and, as a result, the device's power consumption. Based on the input size and the l_2 metric, we estimate that the simulated dataset with $(a, b, rdi\_amplitude) = (8, 5, 10)$ increased execution time by fives times introducing double power consumption compared to our methods. Considering that, a defender could prefer to use our methods when the power consumption of the encrypting device is a critical factor.

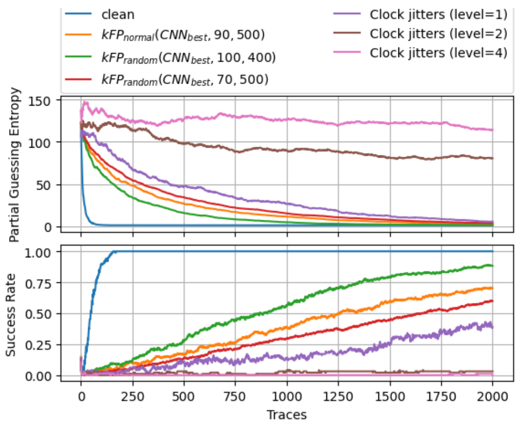
Clock-jitters

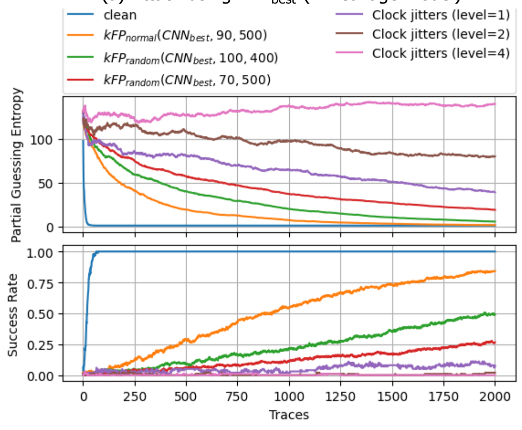
Similar to the approach we followed in the previous section, we choose the parameters for algorithm 6 based on [83]. In this work, the authors generate a dataset protected with the clock-jitter countermeasure by setting $level = 4$, which generates a dataset with input size 1263 tracepoints and $SNR = 0.0069$. Again, to explore the effectiveness of the $level$ parameter to the performance of the countermeasure, we generate two more datasets with $level = \{1, 2\}$. The input length of these two datasets is 1092 and 1162, and the SNR metrics are 0.0069 and 0.0068, respectively.

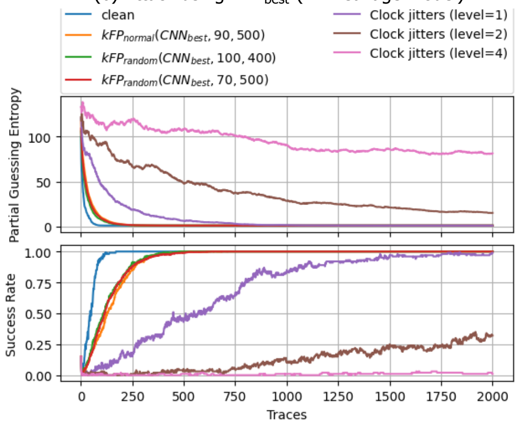
In table 5.6, we present the SCA metrics of the executed attacks on the simulated datasets. In contrast, the progress of the achieved guessing entropy and success rate is presented in figure 5.4. Based on these metrics, we could say that the performance of the clock jitters is almost identical to the performance of the RDIs.

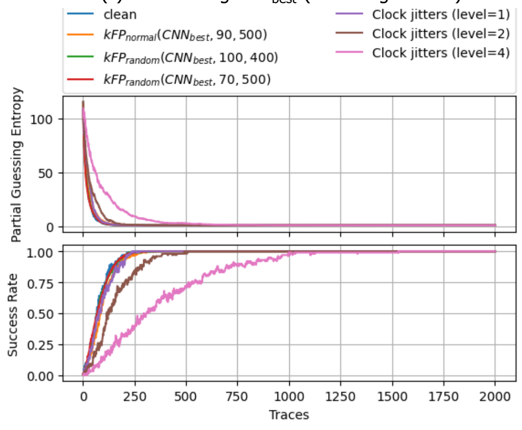
This similarity in the results indicated that clock jitters outperform our adversarial-based countermeasures (figure 5.4) since they provide stronger secret key protection without introducing extra power

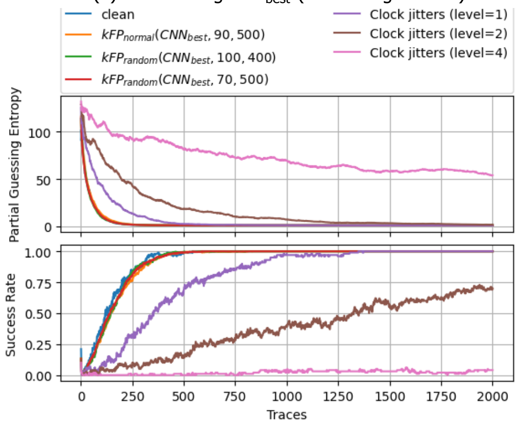

(a) Attack using MLP$_{best}$ (ID leakage model)

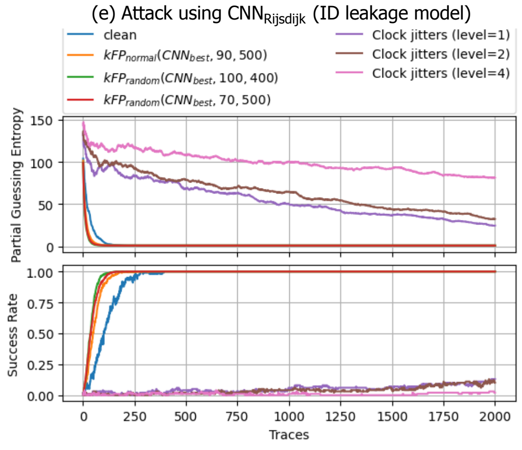

(b) Attack using MLP$_{best}$ (HW leakage model)


(c) Attack using CNN$_{best}$ (ID leakage model)


(d) Attack using CNN$_{best}$ (HW leakage model)


(e) Attack using CNN$_{Rijsdijk}$ (ID leakage model)


(f) Attack using CNN$_{Rijsdijk}$ (HW leakage model)


(g) Attack using MLP$_{Wu}$ (ID leakage model)

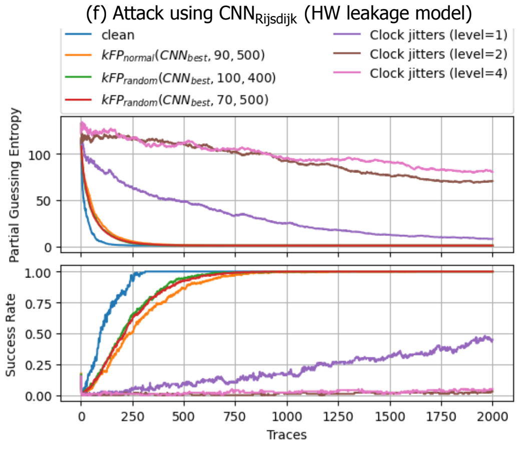

(h) Attack using MLP$_{Wu}$ (HW leakage model)

Figure 5.4: Comparison of our proposal with Clock-Jitters

Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
Clock-jitters ($range = 1$)	ID	MLP$_{\text{best}}$	>2000	114.15	0.01	104.59	0.00	86.91	0.01
		CNN$_{\text{best}}$	>2000	83.41	0.02	64.77	0.02	38.93	0.07
		CNN$_{\text{Rijsdijk}}$	237	1.16	0.91	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	82.54	0.00	50.13	0.06	24.72	0.13
	HW	MLP$_{\text{best}}$	>2000	79.20	0.02	26.52	0.14	5.42	0.39
		CNN$_{\text{best}}$	>2000	24.63	0.19	1.52	0.85	1.01	0.99
		CNN$_{\text{Rijsdijk}}$	1348	19.57	0.22	1.11	0.98	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	68.79	0.02	24.95	0.17	8.26	0.45
Clock-jitters ($range = 2$)	ID	MLP$_{\text{best}}$	>2000	108.60	0.00	85.48	0.01	60.95	0.03
		CNN$_{\text{best}}$	>2000	116.34	0.01	97.41	0.00	80.07	0.02
		CNN$_{\text{Rijsdijk}}$	492	2.00	0.73	1.00	1.00	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	95.44	0.00	64.30	0.04	32.43	0.10
	HW	MLP$_{\text{best}}$	>2000	118.00	0.00	89.98	0.03	80.37	0.03
		CNN$_{\text{best}}$	>2000	73.30	0.01	28.54	0.10	15.15	0.32
		CNN$_{\text{Rijsdijk}}$	>2000	49.04	0.09	7.12	0.40	1.76	0.70
		MLP$_{\text{Wu}}$	>2000	118.66	0.00	92.41	0.01	70.47	0.03
Clock-jitters ($range = 4$)	ID	MLP$_{\text{best}}$	>2000	125.06	0.00	131.38	0.01	127.95	0.01
		CNN$_{\text{best}}$	>2000	127.43	0.00	137.75	0.00	139.76	0.00
		CNN$_{\text{Rijsdijk}}$	1531	14.44	0.31	1.08	0.97	1.00	1.00
		MLP$_{\text{Wu}}$	>2000	118.09	0.00	100.15	0.01	81.40	0.02
	HW	MLP$_{\text{best}}$	>2000	126.68	0.00	126.52	0.00	114.08	0.00
		CNN$_{\text{best}}$	>2000	115.69	0.00	90.85	0.01	81.49	0.01
		CNN$_{\text{Rijsdijk}}$	>2000	96.20	0.00	69.25	0.04	53.95	0.04
		MLP$_{\text{Wu}}$	>2000	118.60	0.01	95.53	0.02	80.33	0.05

Table 5.6: SCA metrics for attacks against datasets protected using clock-jitters

consumption. A possible benefit of our proposed countermeasure is the fact that clock jitters require specific hardware. In practice, the application of this hardware countermeasure requires certain features from the device's microprocessor, such as clock-scaling and phase-shifted clock signals, which are usually available on FPGAs [31]. Hence, our countermeasure could be a viable alternative when these features are not available.

5.3.3. Masking countermeasures

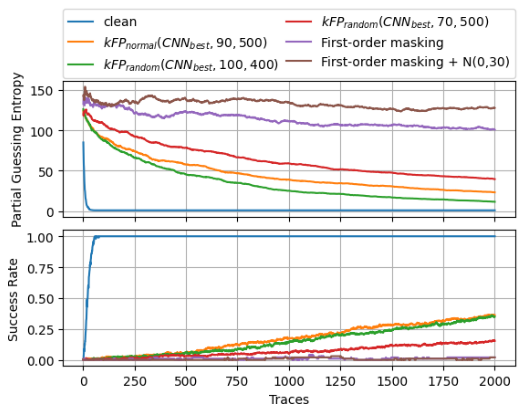
As explained in section 5.2.2, it is not possible to simulate the application of . The difficulty occurs because this mask is applied to the secret key on the software level and requires modifying the encryption source code. Thus, our only alternative is to use an existing dataset protected with a masking countermeasure. Thus, for this simulation, we use the random key ASCAD. We can safely assume that a first-order masked version of our clean dataset would be equally protected to ASCAD, as these two datasets are extracted from the same power traces but focus on a different subkey. In particular, ASCAD contains tracepoints correlated with the third subkey, while our clean dataset contains points for the first unprotected subkey.

Recall that in section 4.2.1, we added artificial noise on our traces using the normal distribution $N(0, 30)$. The motivation of that choice was to increase the difficulty of the attack and, consequently, the resolution of our results since the attacks on the original dataset were successful after processing only 2 traces. To justify the existence of this noise, we assumed that our measurements include noise caused by external factors and not because of the application of the Gaussian noise countermeasure. Hence, to perform a fair comparison between our adversarial-based countermeasure and the masking countermeasure, we have to assume that this external noise was present during the acquisition of the ASCAD dataset. Thus, we add normal noise of equal magnitude to the original ASCAD. In our experiments, we will use both of these datasets.

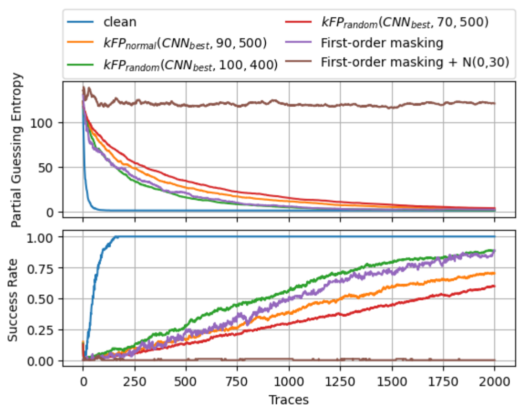
Table 5.7 presents the used SCA metrics to track the performance of the attacks against the original and modified ASCAD. Similar to previous sections, we also provide the progress of GE and SR during these attacks in comparison to the attacks on the clean and our generated datasets.

Based on these results, we can see that adding the external noise on the ASCAD makes the attack infeasible in all of the attacking scenarios. Regarding the attacks on the original ASCAD, we see that the performance is similar to the attacks against our methods when the attack uses the MLP$_{\text{best}}$ model. However, original ASCAD seems to be better protected against our CNN$_{\text{best}}$ and the state-of-the-art models.

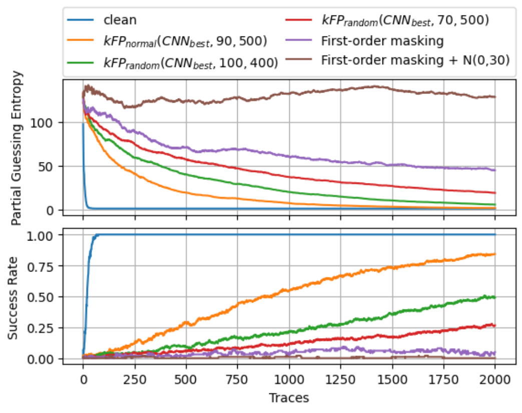
Overall, it seems that the first-order masking can provide a better level of protection than our



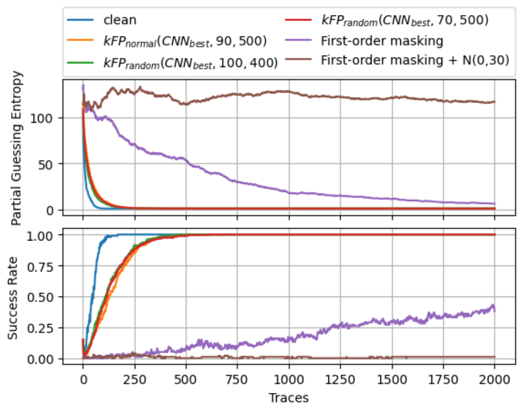
(a) Attack using MLP_{best} (ID leakage model)



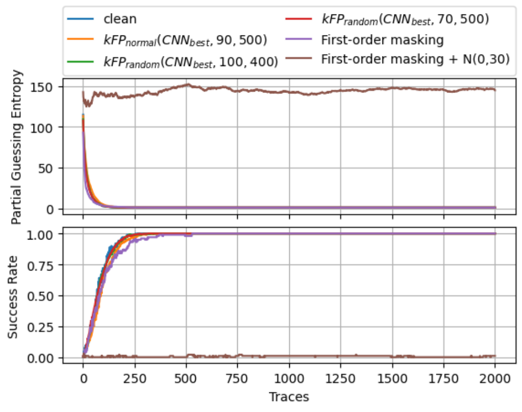
(b) Attack using MLP_{best} (HW leakage model)



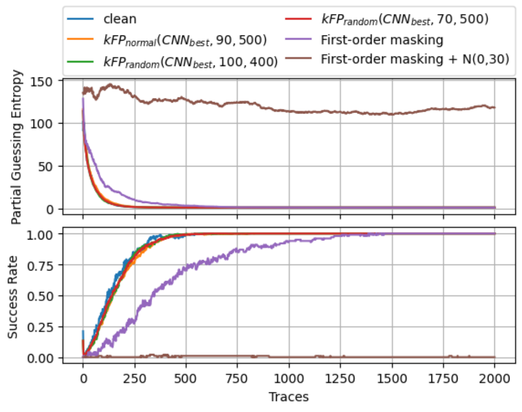
(c) Attack using CNN_{best} (ID leakage model)



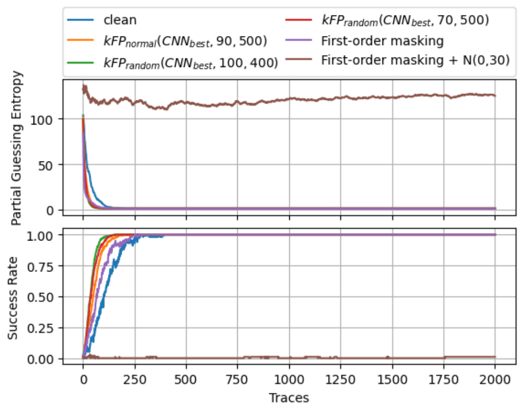
(d) Attack using CNN_{best} (HW leakage model)



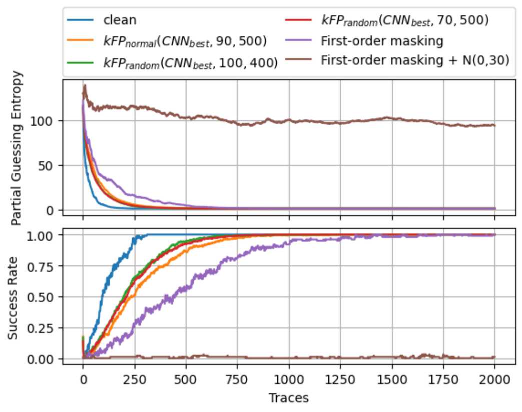
(e) Attack using $\text{CNN}_{\text{Rijsdijk}}$ (ID leakage model)



(f) Attack using $\text{CNN}_{\text{Rijsdijk}}$ (HW leakage model)



(g) Attack using MLP_{Wu} (ID leakage model)



(h) Attack using MLP_{Wu} (HW leakage model)

Figure 5.5: Comparison of our proposal with first-order masking

Method	Leakage	Att. model	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
First-order masking	ID	MLP_{best}	>2000	128.15	0.00	111.48	0.01	100.83	0.02
		CNN_{best}	>2000	93.72	0.02	60.25	0.03	44.84	0.05
		$\text{CNN}_{\text{Rijsdijk}}$	530	1.57	0.86	1.00	1.00	1.00	1.00
		MLP_{Wu}	252	1.11	0.94	1.00	1.00	1.00	1.00
	HW	MLP_{best}	>2000	47.20	0.04	5.51	0.51	2.07	0.89
		CNN_{best}	>2000	77.13	0.02	17.99	0.14	6.25	0.38
		$\text{CNN}_{\text{Rijsdijk}}$	1386	14.89	0.20	1.16	0.93	1.00	1.00
		MLP_{Wu}	1326	19.77	0.17	1.10	0.92	1.00	1.00
First-order masking + Gaussian noise ($N(0,30)$)	ID	MLP_{best}	>2000	136.56	0.00	134.45	0.01	127.65	0.02
		CNN_{best}	>2000	116.20	0.00	132.68	0.01	128.69	0.00
		$\text{CNN}_{\text{Rijsdijk}}$	>2000	136.31	0.01	142.03	0.01	144.63	0.01
		MLP_{Wu}	>2000	122.27	0.00	121.58	0.00	125.36	0.01
	HW	MLP_{best}	>2000	120.44	0.00	118.93	0.01	120.53	0.00
		CNN_{best}	>2000	127.24	0.02	128.01	0.01	116.80	0.01
		$\text{CNN}_{\text{Rijsdijk}}$	>2000	136.58	0.00	113.26	0.00	117.88	0.00
		MLP_{Wu}	>2000	114.27	0.01	100.12	0.00	93.86	0.01

Table 5.7: SCA metrics for attacks against datasets protected using first-order masking

adversarial-based countermeasure. However, a drawback of masking countermeasures is that they are implemented at the software level and require altering the encryption algorithm's source code properly. On the other hand, our proposed countermeasure is designed to work as a separate noise generator. Thus, we can conclude that in certain use cases that modifying the source code is not possible, our proposal can be an effective alternative option for protection against SCA.

5.4. Conclusions

Based on the observations of the previous sections, we conclude that our adversarial-based countermeasure needs to be improved so it can offer a clear added value in terms of protection compared to other existing SCA countermeasures.

First of all, one could believe that our proposed countermeasure should work better than the Gaussian noise countermeasure, as they both add noise on the y-axis of the input. Still, our method generated adversarial instead of random noise. According to our results, this seems to be partially true. In particular, this hypothesis holds only for the attacking scenario that our adversarial noise is constructed, namely for the specific attacking and leakage models. On the other hand, Gaussian noise seemed to provide more general protection.

Moreover, our proposed countermeasure failed to compete with the performance of the countermeasure that adds randomness to the time domain. Especially, RDIs and clock jitters outperformed our methods. We believe that the main reason for the superiority of this countermeasure compared to ours is the randomness that it is introduced in the time domain. In particular, our methods aim to hide the true amplitude of the power consumption that leaks information regarding the sensitive encryption operations in a certain time sample. On the other hand, RDIs and clock jitters introduce local randomness on the time domain, which breaks the alignment of the traces and reduces the vertical correlation with the true label. That triggers a new approach to designing an adversarial-based countermeasure that uses adversarial techniques to introduce such misalignment more ”cleverly”. This direction is left for future research.

Compared with masking defensive techniques, we conclude that our methods cannot provide any added value. The result indicate that the protection offered by the first-order masking technique is superior and the only possible benefits of our approach are not method-specific but lie on the general comparison of masking and hiding countermeasures.

Finally, the enriched attacking scenarios of this chapter revealed another weakness of our defensive techniques. On the one hand, our methods failed to protect state-of-the-art models. However, these models presented an amazing performance against most of the examined countermeasures, except for the RDIs and the clock jitters. On the other hand, the performance of our method was decreased significantly when the HW leakage model was used for the attack.

Overall, we see that our adversarial-based countermeasure is weak. We believe that it weakness is because it adds noise on the y-axis of the measurements. Our results show that existing time-based and masking countermeasures are outperforming the noised-based countermeasures. However, we believe that an adversarial-based approach on the SCA countermeasure design still has room for improvement. Conventional countermeasure relies their security on pure randomness, and they are more or less limited to certain parameters, such as the added noise level, the size of the delay interrupts, etc. On the other side, our method introduces a new way of designing protected traces with unlimited capabilities. Some of these capabilities are discussed in section 7.3.

6 Performance against non-profiled side-channel attacks

This chapter aims to answer the third research question of this thesis, namely to investigate the effectiveness of the proposed adversarial-based countermeasures for non-profiled side-channel attacks. First, we present our motivation for this research direction and why we believe that this question is important. Then, we present the experimental process for such an investigation, and we present our results. Last, we execute a comparison of existing countermeasures for non-profiled side-channel attacks, similar to what we did in chapter 5.

6.1. Motivation

As explained in section 2.4, profiled side-channel attacks require the attacker to train the attacking models on using power traces retrieved by a clone device, i.e., a device that has the same technical specifications as the target device. That allows the attacker to remove any hardware-specific biases introduced by the target device and build a strong profiler that guesses the device's secret key after processing a small amount of traces from the target device. However, it is not always possible for the attacker to know the exact specifications of the target device or acquire such a clone device. In these cases, a non-profiled attack is the only workaround for the attacker. These attacks exploit the leaked side-channel information from the target device without prior training. Thus, they usually demand a larger amount of attacking traces for a reliable result.

Considering that the non-profiled attacks are the attacker's only choice in the described situations, it is interesting to investigate how our adversarial-based countermeasure can perform against this family of side-channel attacks. The ultimate goal of the present chapter is to compare our adversarial-based countermeasure with other existing countermeasures regarding the provided protection against non-profiled attacks. This comparison is important as it provides a good indication of our countermeasure's protection compared to their "competitors". At the same time, it reveals its possible added value regarding the specific family of side-channel attacks.

As the threat model is changed, it is necessary to test the effectiveness of our adversarial-based countermeasure against non-profile attacks before performing the comparison described above. Even though our proposed defensive technique designs adversarial examples based on a trained deep neural network and designed to protect against DNN-based SCAs, we expect that the protection should be transferable against non-profiled SCAs. We believe that this transferability is a hard requirement for a modern SCA countermeasure. A lack of this property would render our adversarial-based countermeasure pointless as, despite the sufficient protection against the former type of attacks, the attacker could perform an attack from the later type to breach the countermeasure's protection.

The rest of this chapter aims to explore the above research directions. First, we measure the level of protection that our implemented countermeasure can provide against non-profiled side-channel attacks. Then, we evaluate this performance compared to other existing countermeasures, similar to what we did in chapter 5.

6.2. Protection against non-profiled SCA

The goal of this section is to explore the effectiveness of the designed adversarial-based countermeasures against non-profiled attacks, and in particular, against CPA attacks (see section 2.4.2). In particular, we are interested in finding how the different method's variants and parameters presented in section 4.3.1 can affect the degree of protection. In the next subsections, we explain in detail the experimental process and present the results. Last, we derive our conclusions based on these results about how the method's parameters affect the performance of our different variants.

6.2.1. Experimental process

For our experiments, we use the datasets generated in section 4.3. Half of these datasets are generated using the MLP$_{\text{best}}$ model for the generation of the adversarial examples, while the CNN$_{\text{best}}$ model was used for the rest of them. Each half consists of a simulation with different combinations of the added noise bounds and affected features. In detail, the lower bound is set to zero, while the upper bound takes values from 100 to 500 with a step of 100. Regarding the modified features, we set the corresponding parameter from 10 to 100 with a step of 10. Finally, to eliminate any bias introduced by the stochastic behavior of our method, we generate 5 instances for each combination, and we present the average for each of the used metrics. Each generated dataset consists of 50,000 profiling traces, where the key is random, and 10,000 attacking traces, where the key is fixed.

The attacking process follows the usual threat model for non-profiled attacks. As the attacker can manipulate the input of the target device, the input plaintext is considered known. Furthermore, the attacker has physical access to the device and can measure the power consumption during the encryption of his chosen plaintext. Thus, the attacker can capture the side-channel observations and perform a CPA attack for the retrieval of the secret key, using the methodology described in section 2.4.2. In our experiments, we use all the 10,000 attacking traces generated during the simulation of our adversarial-based countermeasure. Last, the attacks of the present chapter use both the ID and HW leakage models.

Regarding the metrics that we use for the evaluation, we use the same set of metrics as in chapters 4 and 5. In particular, we define the metric $n_{success}$, which describes the number of traces that the attacker needs to acquire $GE = 1$ and $SR = 100\%$. Moreover, we define the metrics GE_x and SR_x that represent the achieved guessing entropy and success rate when $x\%$ of the total attacking traces is processed by the attacking model. In our experiments, we measure for $x = \{10, 50, 100\}$. Since the non-profiled attacks require more attacking traces than the profiled ones, we increase the total attacking traces from 2,000 to 10,000.

6.2.2. Results

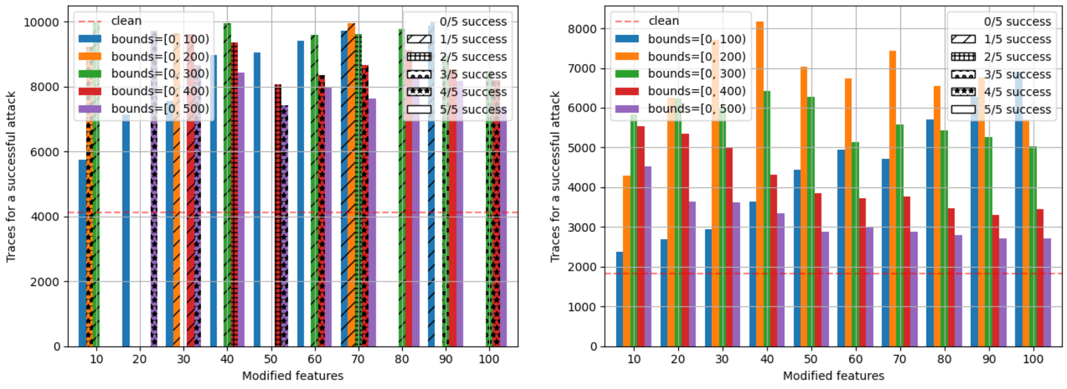
In the following sections, we present the results of the above experimental process divided into three subsections, one for each variant of our adversarial-based countermeasure.

Normal k-Point Protection

Figure 6.1 presents the average $n_{success}$ metric for the CPA attacks against the datasets that are generated by the normal k-Point Protection method while using the MLP$_{\text{best}}$ model for the generation of the adversarial examples.

In figure 6.1a, we see that our method can offer significant protection against attacks using the ID leakage model. In particular, when the added noise is bounded in the range $[0, 100)$, our method needs to change at least 70 tracepoints to make the attack infeasible in four out of five executions. In contrast, modification of 80 features makes the attack infeasible for all of the executions. When the upper bound of the noise is changed to 200, the attack is rendered infeasible for almost all the executions that change at least 20 features. Similarly, we notice that an upper bound of 300 can offer significant protection. However, in this scenario, we notice the optimal performance is noticed when the method alters 20-30 features. At the same time, the attack succeeds using just under 10,000 traces in some scenarios where the amount of affected features is higher. Last, we notice that with an upper bound of 400 or 500, we only need to modify 10 features to make the attack always infeasible. At the same time, increasing the number of altered tracepoints decreases the performance of our method.

On the other hand, our method doesn't offer equivalent protection when using the HW leakage model. If we recall that our methods are using the ID model to protect the dataset, this drop in the performance is expected. figure 6.1b shows that the performance of the attack against datasets generated with $bounds = [0, 500)$ is similar to the one against the clean dataset for small k. At the same time, it is increased gradually when the method modifies more tracepoints. If the upper bound is set to 200, we notice that the peak of the method's performance is reached when it changes 40 features, where the attack is about 4 times harder than the attack on the clean dataset. For an upper bound of 300, we notice that the method's performance is more or less the same for all the different values of k. Last, when the upper bound is set to 400 or 500, the performance of our method drops as we increase the number of modified features.



(a) CPA attack using the ID leakage model (b) CPA attack using the HW leakage model

Figure 6.1: Performance of normal k-Point Protection using the MLP$_{\text{best}}$ model

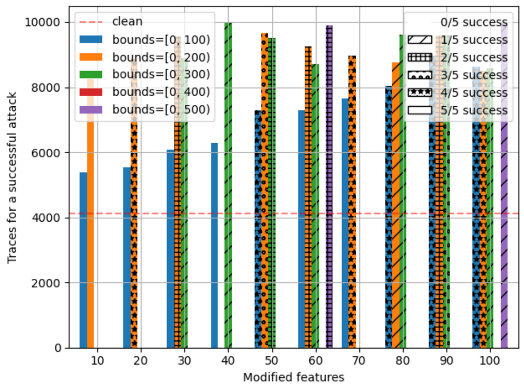
Continuing, figure 6.2 shows the performance of the CPA attacks against datasets that are generated by our $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, \cdot, \cdot)$. We see that the usage of the CNN$_{\text{best}}$ model for the generation of the adversarial examples changes majorly the observations we did above.

Regarding the attacks that use the ID leakage model (see figure 6.2a), we notice that a small range for the $bounds$ parameter does not offer a significant level of protection. In detail, almost all the executions with $bounds = [0, 100)$ do not protect the secret subkey within 10,000 traces, while similar results we observe when setting the upper bound to 200. When this range is changed to $[0, 300)$, the attack becomes infeasible within 10,000 traces for $k < 30$, while more and more attacks become feasible for a higher k. Last, we see that setting the upper bound to 400 or 500 makes the attack almost always infeasible, even when we change only 10 features.

Similarly to what we observed earlier in this section, we see that the performance of our method against attacks that use the Hamming weight leakage model drops, as presented in figure 6.2b. In particular, for the smaller examined range of added noise, the attacker needs around 50% of our traces to retrieve the secret subkey in most of the cases. In detail, for $bounds = [0, 100)$, the performance of the attack remains the same when our method modifies a small number of tracepoints, while the $n_{success}$ metric is only doubled when the method alters 100 features. If the $bounds$ parameter is set to $[0, 200)$, we notice that the optimal performance of our method is met in the scenario that we alter 50 to 70 features, where the attack needs around 5,000 traces to retrieve the secret subkey. When the upper bound is set to 300, our method performs better when it modifies 20 to 40 traces where the $n_{success}$ metric is close to 6,000.

When the upper bound of the added noise is bigger, we notice that the performance of our method is improved significantly, as the $n_{success}$ metric exceeds the 8,000 traces, but all of the attacks are still feasible. In detail, the method $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 70, 400)$ increases this metric to 8,000 traces, while the $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 60, 500)$ succeeds the best protection among the examined scenarios, where the attacker needs around 9,000 traces to guess the secret subkey correctly.

Based on the above observations, we select implementations that combine a good performance against both leakage models. More specifically, we aim to pick implementations that render the ID-based attacks infeasible within 10,000 features and achieve a high $n_{success}$ metric against HW-based attacks. Moreover, we include the $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 90, 500)$, which was selected as the best implementation for the normal k-Point Protection in section 4.3. In table 6.1, we present the SCA metrics of attacks on the datasets generated by these configurations.


(a) CPA attack using the ID leakage model

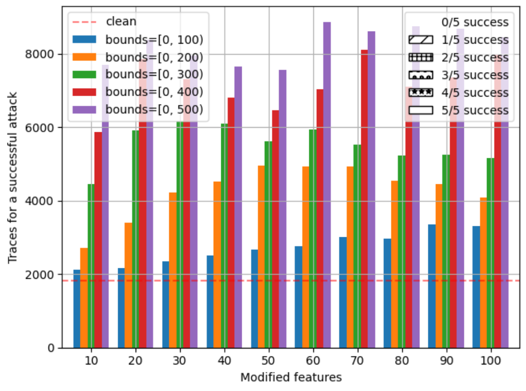

(b) CPA attack using the HW leakage model

Figure 6.2: Performance of normal k-Point Protection using the CNN$_{\text{best}}$ model

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 70, 400)$	ID	>10000	124.18	0.01	55.75	0.05	12.67	0.25
	HW	8110 (5/5)	110.88	0.01	1.88	0.69	1.00	1.00
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$	ID	>10000	130.71	0.00	91.23	0.01	46.41	0.00
	HW	>10000	117.34	0.00	49.52	0.02	4.54	0.28
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 300)$	ID	>10000	125.57	0.01	100.98	0.01	63.60	0.00
	HW	>10000	123.60	0.00	25.35	0.08	2.05	0.52

Table 6.1: SCA metrics for attacks against datasets generated by our normal k-Point Protection. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">10000" denotes that all five attacks were unsuccessful.

A noteworthy point is that the $kPP_{\text{normal}}(\text{MLP}_{\text{best}}, 90, 500)$, which was among the top implementations for DNN-based side-channel attack, presents the top performance among the selected implementations. As we discussed in section 6.1, this transferability between non-profiled and profiled attacks is an important factor for the evaluation of our adversarial-based method.

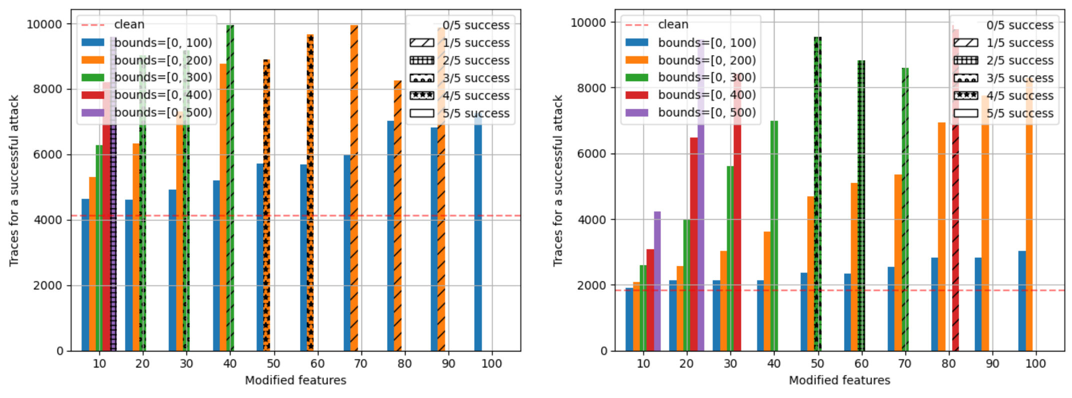
However, taking into consideration the minimality of the added noise, there are configurations with similar performance, such as the $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 70, 400)$ that has a smaller l_2 distance metric, as we see in tables B.1 and B.2. Thus, for the rest of this chapter, we consider that the best configuration for the normal k-Point Protection method is the $kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 70, 400)$.

Random k-Point Protection

Figure 6.3 presents the average $n_{success}$ metrics for the CPA attacks against the datasets that are generated by the random k-Point Protection method with the usage of the MLP$_{\text{best}}$ model for the generation of the adversarial examples.

Regarding the CPA attacks that use that ID leakage model, we see in figure 6.3a that our method presents remarkable results in the extreme majority of the experiments, except for the case that the bounds of the added noise are set to $[0, 100)$. In these cases, all the attacks on the protected datasets managed to guess the secret I correctly using 4500 traces for $k = 10$ to 7000 traces for $k = 100$, when the attack on the unprotected dataset needs a bit more than 4000 traces. However, for bigger ranges, the performance of our methods improves. Starting with the upper bound is set to 200, we notice that only one-fifth of the attacks are successful for $k > 60$, while all the attacks fail within 10,000 traces for $k = 100$. Continuing, for $bounds = [0, 300)$, only one of the attacks succeed with the metric $n_{success}$ being close to 10,000 for $k = 40$. At the same time, modifying 50 or more features results in 0% successful attacks. Last, we notice that our methods need to add noise only on 20 features when the upper bound is set to 400 or 500 to protect completely the secret key against that attack that uses up to 10,000 traces.

On top of the aforementioned observations, we notice that the usage of the MLP$_{\text{best}}$ for the generation of the adversarial examples can offer significant protection against CPA attacks that use the HW leakage model when the range of the added noise is large enough. In particular, for an upper bound of 400 or 500, our method needs to perturb 50 or 40 tracepoints to achieve a 0% feasibility of CPA attacks within 10,000 traces. At the same time, an upper bound of 300 requires at least 70 traces to get modified for the same outcome. Again, this approach doesn't provide sufficient protection for smaller ranges since the secret subkey is retrieved successfully in all the performed attacks.

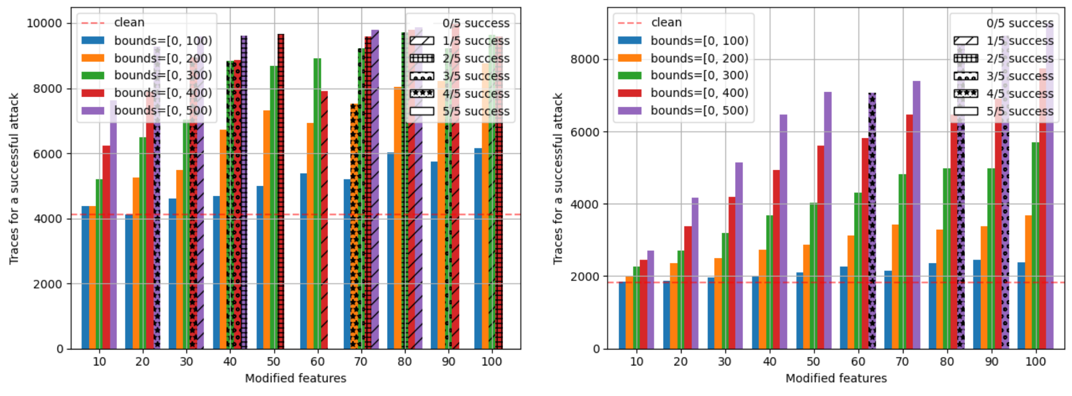


(a) CPA attack using the ID leakage model

(b) CPA attack using the HW leakage model

Figure 6.3: Performance of random k-Point Protection using the MLP$_{\text{best}}$ model

Moving to apply this method with the usage of the CNN$_{\text{best}}$ for the generation of the adversarial examples, the results presented in figure 6.4 are surprisingly different from what we would expect.



(a) CPA attack using the ID leakage model

(b) CPA attack using the HW leakage model

Figure 6.4: Performance of random k-Point Protection using the CNN$_{\text{best}}$ model

On the one hand, we see that the vast majority of the ID-based CPA attacks result in successfully retrieving the secret key in 10,000 traces. In detail, for an upper bound of 100, the performance of the attack barely changes for $k = 0$ compared to the attack on the clean dataset, while it becomes only half times harder for $k = 100$. When the upper bound is increased to 200, the results are similar; there is not any observable protection when the method changes ten tracepoints, and it needs to change 90 tracepoints so that the $n_{success}$ metric approaches 9,000. In the case that the range of the added noise is $[0, 300)$, we observe that some of the attacks become infeasible for $k > 40$ and only one-fifth of attacks is feasible, while the feasible attack has $n_{success} \approx 10,000$, when $k = 100$. The method's performance is improved for larger ranges, as it needs to modify 80 features to protect 4/5 of the generated datasets when the upper bound is 400 and 50 features to render all the performed attack infeasible when the upper bound is 500.

In table 6.2, we present the SCA metrics of the CPA attacks against the above noteworthy implementations in addition to the $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$ and $kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 500)$, which were selected as the top configurations for the random k-Point Protection in section 4.3. Despite the fact the some of the attacks against these two last implementations are feasible within 10,000 traces, we

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 100, 200)$	ID	>10000	121.57	0.01	49.59	0.04	4.92	0.08
	HW	8292 (5/5)	109.30	0.01	3.04	0.54	1.00	1.00
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 300)$	ID	>10000	124.41	0.00	69.44	0.01	32.04	0.00
	HW	9535 (4/5)	107.27	0.00	11.26	0.25	1.30	0.87
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$	ID	>10000	130.71	0.00	91.23	0.01	46.41	0.00
	HW	>10000	117.34	0.00	49.52	0.02	4.54	0.28
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 40, 500)$	ID	>10000	123.92	0.00	94.66	0.01	53.91	0.17
	HW	>10000	121.64	0.00	57.66	0.03	13.47	0.19
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 70, 300)$	ID	9237 (3/5)	114.31	0.01	14.34	0.22	1.31	0.79
	HW	4812 (5/5)	88.65	0.01	1.01	0.99	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 80, 400)$	ID	9800 (1/5)	115.78	0.01	40.09	0.09	5.54	0.66
	HW	6474 (5/5)	111.59	0.02	1.28	0.84	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 100, 400)$	ID	9590 (2/5)	123.37	0.00	46.49	0.07	4.19	0.48
	HW	7748 (5/5)	117.19	0.00	2.57	0.64	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 50, 500)$	ID	>10000	120.91	0.01	39.09	0.09	3.66	0.33
	HW	7092 (5/5)	109.88	0.01	1.63	0.72	1.00	1.00
$kPP_{\text{random}}(\text{CNN}_{\text{best}}, 80, 500)$	ID	9990 (1/5)	127.32	0.00	65.34	0.06	13.92	0.28
	HW	8455 (4/5)	119.83	0.00	9.61	0.28	1.04	0.97

Table 6.2: SCA metrics for attacks against datasets generated by our random k-Point Protection. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">10000" denotes that all five attacks were unsuccessful.

notice that the average $n_{success}$ and GE_{100} are quite high, and the SR_{100} sufficiently low –especially for attacks that use the ID leakage model. However, the best performance for the random k-Point Protection is achieved by the implementations $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$ and $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 40, 500)$. According to table B.3, the former achieves a lower l_2 distance metric, and, thus, we select it as the best implementation of the random variant.

Correlation k-Point Protection

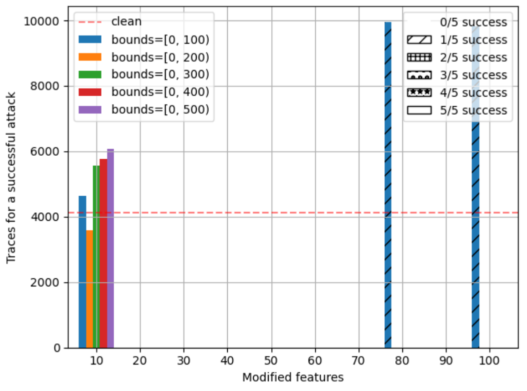
Figure 6.5 presents the average $n_{success}$ metrics for the CPA attacks against the datasets that are generated by the correlation k-Point Protection method with the usage of the MLP_{best} model for the generation of the adversarial examples.

Regarding the attacks that use the ID leakage model, our method presents astonishing results. In particular, our method needs to modify only 20 features to make all the attacks infeasible regardless of the range of the added noise. The only exception is for the implementations $kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 80, 100)$ and $kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 100)$, where one attack in each scenario is successful, but with $n_{success}$ very close to 10,000.

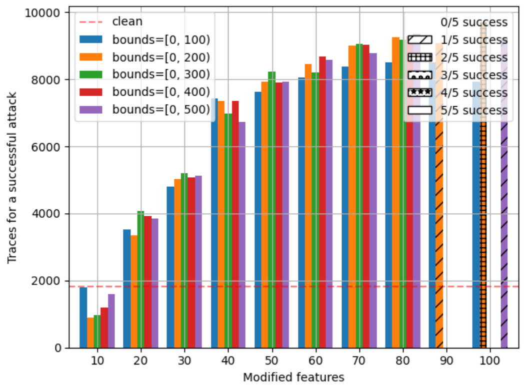
On the other hand, this observation doesn't hold for the Hamming weight model's attacks. In particular, the CPA attacks against datasets that generated with $bounds = [0, 100)$ are always feasible and the highest protection is observer for $k = 90$ where $n_{success} \approx 8500$. If the upper bound is set to 200, all the attacks are feasible for $k < 90$, while most of the attacks are infeasible for higher k. For upper bound equal greater than 300, all the attacks are infeasible when $k \geq 90$, with a single exception for the $kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 500)$ where one-fifth of the attacks is feasible within 10,000 traces.

The results regarding the correlation k-Point Protection when the CNN_{best} is used to generate adversarial examples figure 6.6. We see that, when the CPA attacks use the ID leakage model, this approach can sufficiently protect the secret subkey for larger ranges of added noise. In particular, for the upper range equal to or greater than 300, all the attacks with $k \geq 20$ are infeasible, while the same holds for the upper range equal to 200 and k from 20 to 50. On the other hand, all configurations with $bounds = [0, 100)$ fail to provide sufficient protection.

However, the performance of this approach against CPA attacks that use the HW leakage model is different. For $bounds = [0, 100)$, our method is barely decreasing the performance of the CPA attack, while for upper range equal to 200 or 300 the offered protection is small ($n_{success} \approx 7,000$ for $k = 60$ and $n_{success} \approx 8,000$ for $k \geq 80$). The best performance is achieved with the configurations



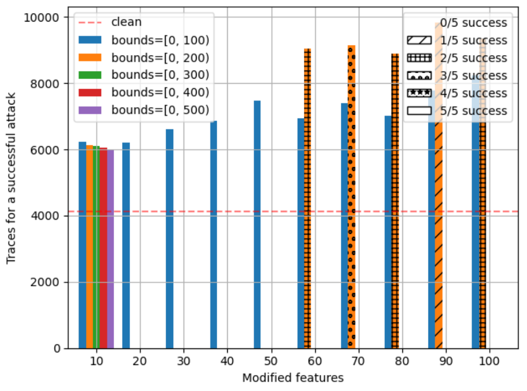
(a) CPA attack using the ID leakage model



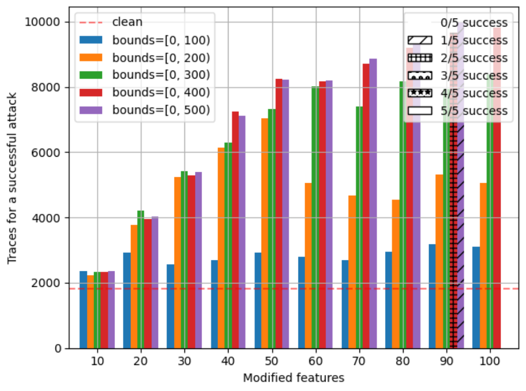
(b) CPA attack using the HW leakage model

Figure 6.5: Performance of correlation k-Point Protection using the MLP_{best} model

$kPP_{\text{corr}}(\text{CNN}_{\text{best}}, 90, 400)$ and $kPP_{\text{corr}}(\text{CNN}_{\text{best}}, 100, 500)$. The former configuration protects completely three out of five instances within 10,000 traces, while the latter makes protect all five instances.



(a) CPA attack using the ID leakage model



(b) CPA attack using the HW leakage model

Figure 6.6: Performance of correlation k-Point Protection using the CNN_{best} model

Based on the above observations, we select the implementations that perform better against both leakage models. Table 6.3 presents the SCA metrics for the CPA attacks against the corresponding generated datasets. This table shows that all the selected methods succeed in fully protecting the secret subkey from CPA attacks that use the ID model, with the attack's success rate being 0%. At the same time, we see that the $kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 300)$ and $kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 400)$ provide more protection than the rest of them against attacks with $SR_{100} = 0.52$ and $SR_{100} = 0.43$ respectively. The Euclidean distance of these two implementations is 1730.25 and 2306.84, respectively, according to table B.5. Thus, we conclude that the former implementation succeeds similar protection with lower added noise.

6.2.3. Conclusions

Based on the observations of the section 6.2.2, we can derive some useful outcomes regarding the performance of the three variants of our adversarial-based countermeasure.

First of all, we notice that the correlation k-Point Protection approach can significantly protect the secret key against CPA attacks, unlike our conclusion regarding its performance against DNN-based side-channel attacks. The explanation for the effectiveness of this particular method against the CPA attacks is straightforward. Recall that this variant picks the k traces points that are most correlated with the secret subkey and adds adversarial noise. At the same time, CPA attacks exploit the same correlation to retrieve the encryption key. Hence, our method directly cloaks this correlation, and the performance of the CPA attack decreases naturally.

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 90, 300)$	ID	>10000	129.85	0.00	107.09	0.00	80.57	0.00
	HW	>10000	113.51	0.00	18.80	0.11	1.12	0.91
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 300)$	ID	>10000	125.57	0.01	100.98	0.01	63.60	0.00
	HW	>10000	123.60	0.00	25.35	0.08	2.05	0.52
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 90, 400)$	ID	>10000	122.61	0.00	105.06	0.01	81.12	0.00
	HW	>10000	124.79	0.00	18.82	0.11	1.11	0.91
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 400)$	ID	>10000	127.99	0.00	112.44	0.00	87.20	0.00
	HW	>10000	120.49	0.01	33.63	0.05	2.18	0.43
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 90, 500)$	ID	>10000	131.92	0.00	108.01	0.00	80.15	0.00
	HW	>10000	117.95	0.00	24.20	0.07	1.07	0.94
$kPP_{\text{corr}}(\text{CNN}_{\text{best}}, 90, 400)$	ID	>10000	125.49	0.00	86.72	0.01	58.92	0.00
	HW	9665 (2/5)	112.99	0.01	15.18	0.14	1.10	0.91
$kPP_{\text{corr}}(\text{CNN}_{\text{best}}, 100, 500)$	ID	>10000	125.06	0.01	103.49	0.01	57.75	0.00
	HW	>10000	122.79	0.01	33.66	0.05	1.72	0.59

Table 6.3: SCA metrics for attacks against datasets generated by our correlation k-Point Protection. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">10000" denotes that all five attacks were unsuccessful.

Moreover, we observe that the usage of the MLP$_{\text{best}}$ model can build stronger adversarial examples and, consequently, provide stronger protection than CNN$_{\text{best}}$ model when we apply the random or the correlation variant of our adversarial-based countermeasure. This contradicts the conclusion from section 4.3, where CNN$_{\text{best}}$ could build stronger adversarial examples that trick both models. Considering the general superiority of the CNN models against the MLP models, this observation is almost unexpected, and it is hard to explain the reasons behind this effect.

Furthermore, another important outcome is the transferability of the protection that our adversarial-based countermeasure offers between non-profiled and profiled attacks. As we saw in the previous subsection, the methods that were performed better against DNN-based SCA can still protect sufficiently the secret key. As explained in section 6.1, we believe that this transferability is a hard requirement for an SCA countermeasure as it ensures that the attacker cannot disable the offer protection by choosing a non-profiled attack instead of a profiled attack.

Last, it is important to compare and conclude which of the three variants presented above performs better. In table 6.4, we summarize the measured SCA metrics for attacks against these methods. Based on this table, we see that the $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$ provides better protection for the two different leakage models. Regarding the l_2 distance metric, using tables B.2, B.3 and B.5, we see that this implementation has $l_2 = 1635.43$, which is the lowest of among these three implementations that we see for the configuration. Hence, we conclude that the random variant provides better protection than the other two approaches. This outcome aligns with what we concluded for DNN-based side-channel attacks.

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
$kPP_{\text{normal}}(\text{CNN}_{\text{best}}, 70, 400)$	ID	>10000	124.18	0.01	55.75	0.05	12.67	0.25
	HW	8110 (5/5)	110.88	0.01	1.88	0.69	1.00	1.00
$kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$	ID	>10000	130.71	0.00	91.23	0.01	46.41	0.00
	HW	>10000	117.34	0.00	49.52	0.02	4.54	0.28
$kPP_{\text{corr}}(\text{MLP}_{\text{best}}, 100, 300)$	ID	>10000	125.57	0.01	100.98	0.01	63.60	0.00
	HW	>10000	123.60	0.00	25.35	0.08	2.05	0.52

Table 6.4: SCA metrics for attacks against datasets generated by the best configurations for our adversarial-based countermeasure. Values are the average of attacks on 5 different instances per configuration. The parenthesis in column $n_{success}$ represents the number of successful attacks out of the 5 instances. The ">10000" denotes that all five attacks were unsuccessful.

6.3. Comparison with existing countermeasures

In this section, we explore how the effectiveness of our adversarial-based countermeasure compares with the conventional countermeasures proposed by the research community for the protection against CPA attacks. As explained in section 6.1, this comparison is very important, as it reveals the added value of our adversarial-based countermeasure. In the rest of this section, we first present the experimental process, the results of our experiments, and our derived conclusions.

6.3.1. Experimental Process

Our experimental process is similar to one conducted in chapter 5. More specifically, we will use the generated datasets of this chapter that simulate the hiding and masking countermeasures under investigation. Regarding the datasets generated by our methods, we will use the top-performing datasets as presented in section 6.2, in addition to the datasets that we found that perform better against the DNN-based CPA attacks.

For the attack, we execute a CPA attack script that follows the process described in section 2.4.2. As profiled attacks require more traces than the profiled attacks, we use all the 10,000 attacking traces generated by our adversarial-based methods.

Last, we use the same evaluation metrics as the previous section. More specifically, we use the $n_{success}$ metric that represents how many traces the attack needs to process to achieve $GE = 1$ and $SR = 100\%$. Moreover, we use the GE_x and SR_x that describe the GE and SR when x% of the 10,000 traces is processed. In our experiments, we measure for $x = \{10, 50, 100\}$.

6.3.2. Results

In this section, we present the results of our experiments related to each countermeasure under examination separately. Then, using the provided tables and figures, we point out our observations.

Gaussian Noise

In table 6.5, we present the performance of the CPA attacks against the protected dataset with the Gaussian Noise countermeasure. In the case that we attack using the ID leakage model, we see that the Gaussian noise countermeasure renders the attack infeasible, with $GE_{100} = 2.7$ and $SR_{100} = 0.11$ after processing 10,000 traces. However, using the HW leakage model for the attack, the attack can retrieve the secret subkey after processing 6,040 traces.

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
Gaussian noise ($N(0, 80)$)	ID	>10000	121.30	0.00	34.06	0.05	2.70	0.11
	HW	6040	94.63	0.00	1.28	0.84	1.00	1.00

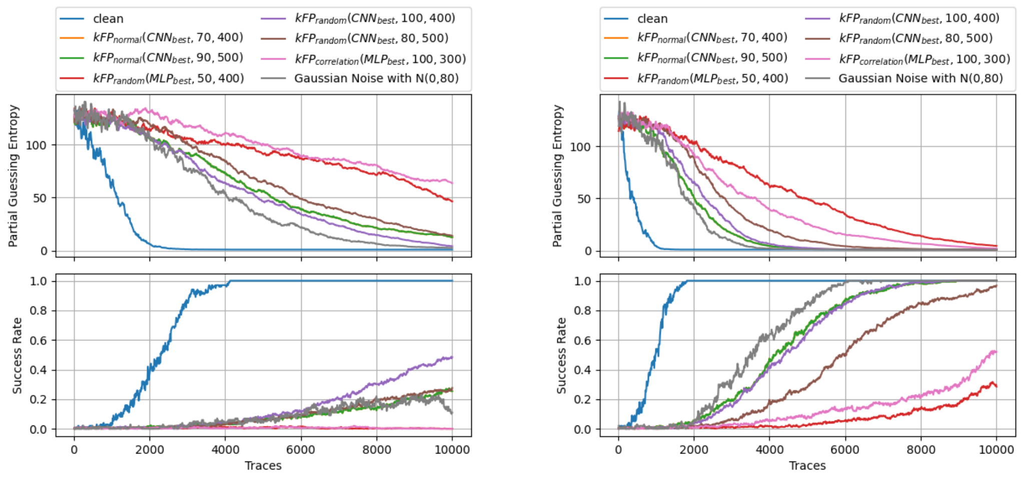
Table 6.5: SCA metrics for CPA attacks against datasets protected using Gaussian noise

As shown in figure 6.7, the Gaussian noise countermeasure performs worse than all of our implementations. In particular, we see that the gray line, which corresponds to the attack against this particular countermeasure, converges to $GE = 1$ faster than any other line. Thus, we can safely conclude that our implementations provide stronger protection against CPA attacks than the Gaussian Noise countermeasure.

Desynchronization

In table 6.5, we present the performance of the CPA attacks against the protected dataset with the desynchronization countermeasure. As we can see, even for the smaller value for the level of desynchronization, the performance of the attack drops dramatically. This effect explains that CPA attacks exploit the horizontal correlation of the dataset's feature with the secret key. As the desynchronization introduces global randomness horizontally, this correlation is majorly decreased, and the CPA attack fails.

In figure 6.8, we see the progress of the PGE and SR metrics of the CPA attacks against the simulated datasets by this particular countermeasure alongside the corresponding attacks against our methods. The aforementioned effects of desynchronization are clearly visible. Since our methods modify only the amplitude of k tracepoints, the correlation of the rest features remains untouched and



(a) CPA attack using the ID leakage model (b) CPA attack using the HW leakage model

Figure 6.7: Comparison of Gaussian noise countermeasure with our top implementations for CPA attacks

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
Desync ($level = 50$)	ID	>10000	113.38	0.01	87.67	0.01	16.72	0.00
	HW	>10000	125.85	0.00	74.21	0.02	9.31	0.03
Desync ($level = 100$)	ID	>10000	132.98	0.00	110.17	0.00	61.09	0.00
	HW	>10000	103.38	0.01	43.76	0.07	3.13	0.19
Desync ($level = 150$)	ID	>10000	128.36	0.01	133.83	0.01	197.30	0.00
	HW	>10000	137.32	0.00	101.44	0.00	66.78	0.00

Table 6.6: SCA metrics for CPA attacks against datasets protected using desynchronization

exploitable by the CPA attack. Hence, we conclude that our adversarial-based countermeasure cannot perform better than the desynchronization countermeasure against CPA attacks by design.

Since our methods present sufficient protection, we could claim that it is preferred to be used instead of another countermeasure if they present another benefit, such as introducing smaller power consumption. However, desynchronization does not require extra power consumption for its application, as it only introduces minor delays in the execution of each encryption. Thus, we cannot find any benefit of our adversarial-based method against the desynchronization countermeasure.

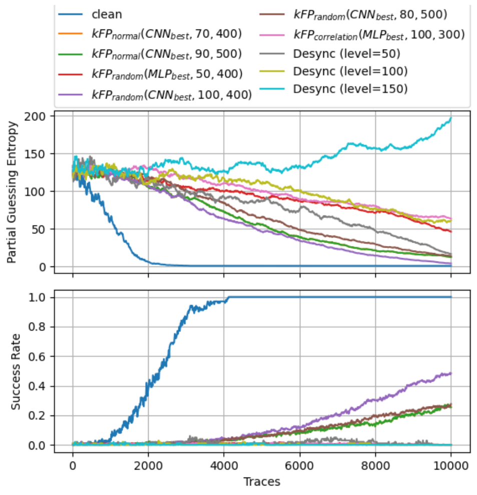
RDIs

RDIs follow a similar approach with the desynchronization countermeasure, as they introduce local randomness to every feature on the time domain of the traces. Thus, we expect that the results will be similar to what we observed in the previous section. Indeed, the performance of the CPA attacks drops majorly with the application of this countermeasure, as displayed in table 6.7.

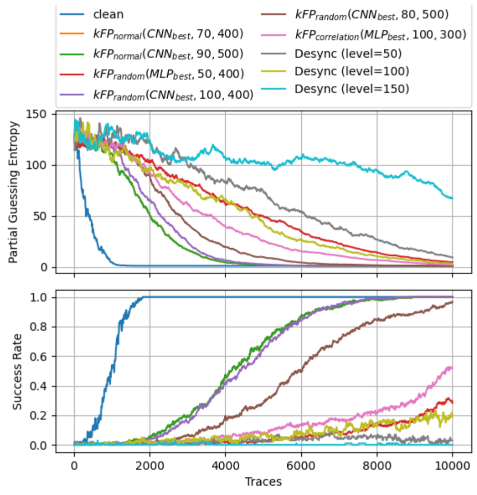
Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
RDI ($a = 3, b = 1, rdi\_amplitude = 10$)	ID	>10000	128.23	0.00	111.60	0.00	59.40	0.00
	HW	>10000	129.74	0.02	53.92	0.01	9.90	0.03
RDI ($a = 5, b = 2, rdi\_amplitude = 10$)	ID	>10000	117.30	0.00	142.02	0.00	207.91	0.00
	HW	>10000	133.05	0.01	90.91	0.02	37.53	0.00
RDI ($a = 8, b = 5, rdi\_amplitude = 10$)	ID	>10000	121.93	0.00	114.02	0.01	65.94	0.00
	HW	>10000	126.64	0.00	67.12	0.03	9.14	0.00

Table 6.7: SCA metrics for CPA attacks against datasets protected using RDIs

The superiority of this countermeasure compared to our proposal is clearly visible in figure 6.9. The reason behind this behavior is the same as what we presented in the previous section, namely, a partial



(a) CPA attack using the ID leakage model

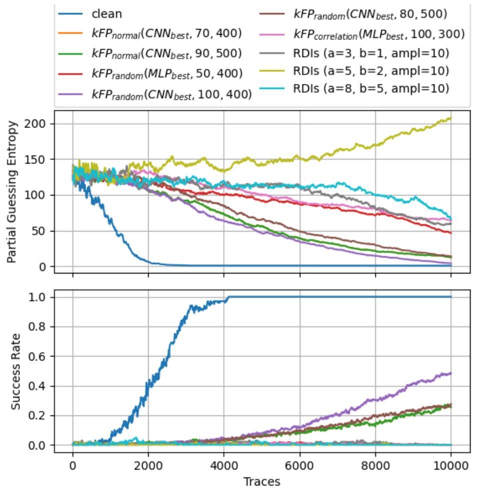


(b) CPA attack using the HW leakage model

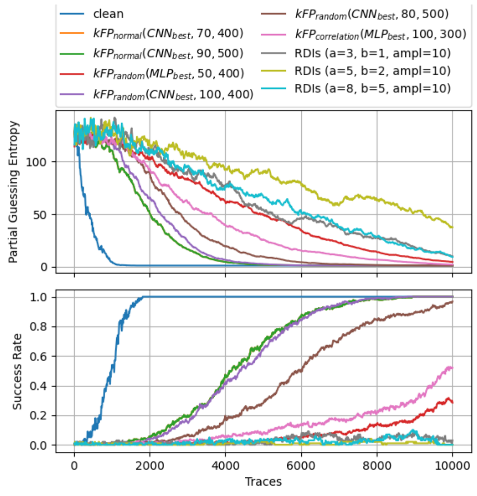
Figure 6.8: Comparison of desynchronization countermeasure with our top implementations for CPA attacks

decrease of the Pearson correlation between the features and the label of the dataset.

However, as we mentioned in section 5.3.2, this particular countermeasure increases significantly the execution time and, consequently, the power consumption. In detail, the simulations with parameters $(a, b, rdi\_amplitude) = \{(3, 1, 10), (5, 2, 10), (8, 5, 10)\}$ increase the input size from $1,000$ to $1,764$, $3,141$, and $5,358$, respectively, and generate datasets with euclidean distance l_2 of 2816.86, 3867.59, and 5023.14, respectively. Hence, we can fairly claim that our $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$ with $l_2 = 1635.43$ is more preferable, as it requires less power consumption and offer significant protection against ID-based and HW-based CPA attacks.



(a) CPA attack using the ID leakage model



(b) CPA attack using the HW leakage model

Figure 6.9: Comparison of RDIs countermeasure with our top implementations for CPA attacks

Clock-jitters

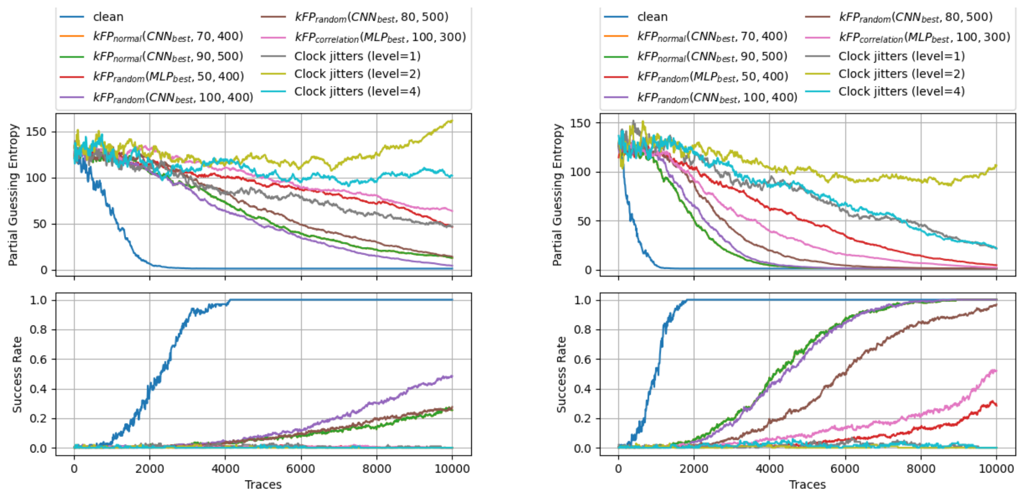
Clock-jitters operate similarly to the RDIs. Thus, the massive performance of this particular countermeasure, as presented in table 6.8, and its superiority compared to our adversarial-based countermeasure, as presented in figure 6.10, is expected. This behavior is explained in the same way as what we described in the two previous sections.

Regarding the required power consumption, we see that the three simulations present slightly higher l_2 metric than our top-performing implementation $kPP_{\text{random}}(\text{MLP}_{\text{best}}, 50, 400)$ $(1829.79\ 1948.61$, and

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
Clock-jitters ($range = 1$)	ID	>10000	116.19	0.00	82.32	0.02	47.01	0.00
	HW	>10000	121.84	0.02	77.97	0.02	21.64	0.00
Clock-jitters ($range = 2$)	ID	>10000	128.74	0.00	122.31	0.01	161.97	0.00
	HW	>10000	135.89	0.01	102.58	0.01	106.25	0.00
Clock-jitters ($range = 4$)	ID	>10000	127.00	0.00	103.13	0.00	102.05	0.00
	HW	>10000	129.54	0.01	80.57	0.01	21.50	0.00

Table 6.8: SCA metrics for CPA attacks against datasets protected using clock-jitters

$l_2 = 2014.18$ for $range = \{1, 2, 4\}$, respectively). Thus, our method could be preferred if power consumption is critical, as this method provides sufficient protection against CPA attacks.



(a) CPA attack using the ID leakage model

(b) CPA attack using the HW leakage model

Figure 6.10: Comparison of clock-jitters countermeasure with our top implementations for CPA attacks

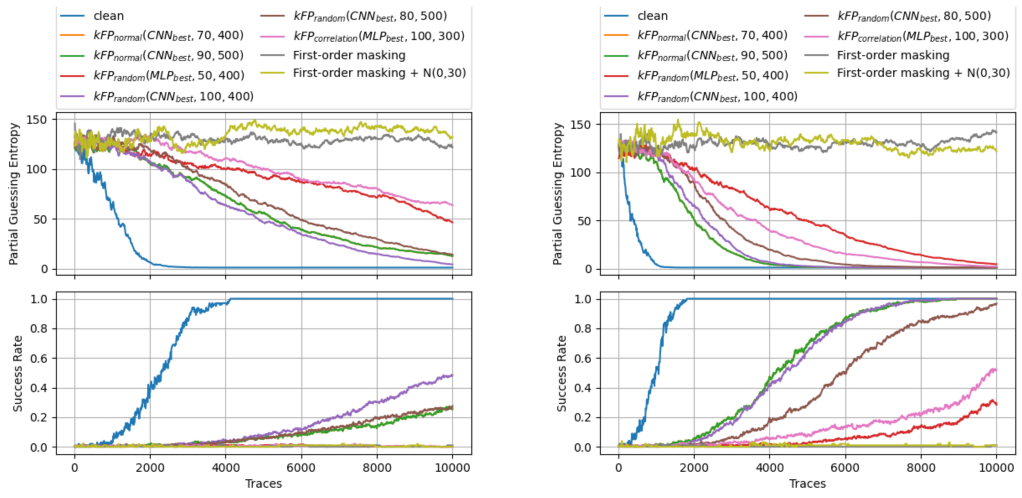
First-order masking

In table 6.9, we present the performance of the CPA attacks against the protected dataset with the first-order masking countermeasure. As we can see, the performance of this countermeasure is similar to what we observed in the cases of desynchronization, RDIs, and clock jitters. Even though that these techniques operate completely differently from the first-order masking, the outcome is the same. First-order masking uses a piece of additional secret information, the mask, that breaks the direct correlation of each tracepoint with the secret subkey. Thus, a CPA attack can only overcome this countermeasure by increasing the attack's order and guess both the mask and the secret key.

Method	Leakage	$n_{success}$	GE_{10}	SR_{10}	GE_{50}	SR_{50}	GE_{100}	SR_{100}
First-order masking	ID	>10000	138.36	0.00	131.36	0.00	121.93	0.01
	HW	>10000	124.81	0.02	126.62	0.00	141.43	0.01
First-order masking + N(0,30)	ID	>10000	128.12	0.00	142.07	0.00	131.85	0.00
	HW	>10000	127.68	0.01	132.38	0.01	121.75	0.01

Table 6.9: SCA metrics for CPA attacks against datasets protected using first-order masking

Again, we conclude that our adversarial-based countermeasure fails to provide better protection than the first-order masking because of its design. In figure 6.11, we see clearly this superiority, as the gray and yellow lines barely change during the a. In contrast, while the lines that correspond to our methods converge slowly to $PGE = 1$, the attack's success rate increases while the attacker processes more power traces. Since the first-order masking does not introduce extra power consumption and does not increase the algorithm's execution time, it is hard to find a solid advantage of using our adversarial-based method instead of first-order masking.



(a) CPA attack using the ID leakage model

(b) CPA attack using the HW leakage model

Figure 6.11: Comparison of first-order masking countermeasure with our top implementations for CPA attacks

6.3.3. Conclusions

Based on the aforementioned observations, we notice that our adversarial-based countermeasure can perform better than the Gaussian countermeasure. We believe that this is a noteworthy point, considering that these two defensive techniques operate in a similar way –they both introduce noise on the y-axis of the input. At the same time, we see that our adversarial-based countermeasure fails to provide a clear added value compared to the existing countermeasures.

we confirm the conclusions presented in section 5.4 The only exception is the comparison with the Gaussian noise countermeasure, where our methods provide clearly stronger protection.

The main conclusion of this section that the misalignment of the tracepoints is crucial for the success of the CPA attacks. This misalignment breaks the correlation of the tracepoints with the secret key and, in most cases, results in an attack with $SR = 0\%$. Hence, the idea of an adversarial-based countermeasure that introduces adversarial misalignment of the tracepoints comes again to the surface.

7 Conclusion

The goal of this chapter is to summarize the findings and contributions of this particular research project. Furthermore, we present the limitations of our conducted research. Last, we propose how these findings could behave as a foundation for future researches.

7.1. Answering our Research Questions

To summarize our conducted research, we present below the outline of answers for our research questions, as presented in section 3.3.

> ***RQ1:*** *Can an adversarial-based countermeasure be designed so that DNN-based side-channel attacks become impractical regarding the observations needed for the secret key retrieval?*

In our research, we show that the construction of an adversarial-based countermeasure is possible. Based the one-pixel attack method [76]. We build such a countermeasure that generates adversarial noise and decreases sufficiently the performance of a DNN-based attack. During our research, we investigated what are the properties of this adversarial noise, in terms of how many tracepoints have to be modified and what should be the level of the added noise. Furthermore, our results indicate that an important factor is the type of the DNN model that is used for the generation for the adversarial examples. Using a CNN model for this generation, our methods appear to build stronger adversarial examples. Moreover, we tested three different approaches for the selection of the modified tracepoints; using an optimization algorithm, using random selection, using correlation analysis to pick features that leak more information. Our experiments showed that the random approach builds stronger adversarial examples, while the correlation approach fails completely to provide any protection. Last, we presented that our method produces transferable protection among different attacking models.

> ***RQ2:*** *What is the added value of using our proposed countermeasure compared to existing SCA countermeasures?*

Unfortunately, despite the confirmed effectiveness of our adversarial-based countermeasure, we concluded that there is not a clear added value of using it instead of other existing SCA countermeasures, at least in its current state. Our experiments showed that our proposed countermeasure can perform similarly, or even better in some cases, to existing noise-based countermeasures, such as Gaussian noise. However, it performs worse than first order masking and time-based countermeasures, such as desynchronization, RDIs, and clock jitters. Thus, any possible added value does not lie on how much the performance of the DNN-based SCA is decreased but on other benefits, such as the introduced power consumption or the capability of the defender to apply a certain countermeasure. For example, one might prefer using our proposed countermeasure instead of RDIs in cases that keeping the device's power consumption low is critical, or instead of clock jitters when the device cannot support clock scaling.

RQ3: *Can this adversarial-based countermeasure provide security against non-profiling side-channel attacks too?*

In our research, we showed that the designed adversarial-based countermeasure is effective against CPA attacks, which is the most common non-profiled attack against the AES encryption scheme. In particular, we tested how the number of perturbed tracepoints, the bounds of the added noise, and the type of the trained model used for the adversarial noise generation can effect the performance of the countermeasure. Furthermore, we showed that the configurations that provide strong protection against DNN-based SCA perform well for CPA attacks as well. This is an important requirement for a countermeasure, since a countermeasure that protects only against a specific type of attack allows the attack to choose a different type to overcome its protection. When comparing with the performance of existing countermeasures on CPA attacks, our conclusions are similar to the previous research question; our approach can work better than Gaussian noise worse than time-based and masking countermeasures.

As a tailpiece of our research, we could say that the results partially satisfy our initial motivation. On the one hand, we succeed in building a countermeasure that uses adversarial noise to decrease significantly the performance of DNN-based side-channel attacks, when other researches concluded that this approach is not viable [29]. On the other hand, our method fails to outperform most of the existing defensive techniques against SCAs. However, we believe that this is still room for improvement for our method to increase its performance. Some ideas to that direction are presented in section 7.3.

7.2. Scientific Contributions

The main scientific contributions of this research are the following:

- We built an adversarial-based countermeasure that is applied on both phases of the attack and decreases the performance of the DNN-based side-channel attacks. Previous attempts were applied only on the attacking phase [63] or were unsuccessful [29]. This approach opens new roads apart from the conventional defending techniques that rely on pure randomness to provide protection.
- To the best of our knowledge, it is the first time that the used state-of-the-art models are tested against this large range of SCA countermeasures.

7.3. Future Work

This novel approach of designing countermeasures for side-channel attacks opens a new undiscovered road for research. As the adversarial noise is calculated at the software level, the capabilities of improving its calculation are limitless. Some interesting ideas for future research are the following:

- The results of chapters 5 and 6 indicate that the horizontal alignment is crucial for the performance of the attack. Thus, hiding countermeasures, such as desynchronization, RDIs, and clock jitters, that break this alignment succeed in decreasing the performance of the attack dramatically. Thus, a possible improvement on our adversarial-based countermeasure would be to use adversarial techniques to introduce randomness on the time domain of the observations instead of modifying the amplitude of the power consumption. For example, it could produce adversarial delay interrupts that create a local randomness on the x-axis.
- The optimization problem that we define for the generation of the adversarial examples is to minimize the confidence of the true key. However, defining a different objective function could possibly improve the robustness of the generated adversarial examples. For example, the adversarial method could try to balance the predictions and decrease their variance. As a result, the confidence of the true key would be equal to every other key candidate.
- In our research, we do not explore how different values of the differential evolution parameters can affect the robustness of the generated adversarial examples. Tweaking this parameters could improve the performance of our adversarial-based countermeasure.

- In our adversarial-based method, we use untargeted adversarial attacks, which means that the goal of the adversarial technique is to minimize the prediction of the true class. A possible improvement is to use a targeted approach in the designing of the adversarial examples. As in a dataset, the classes are not equally represented, the DNN classifier is trained well to recognize patterns that belong to the highly represented class. Using a targeted approach to adversarial noise crafting, the method could look for less represented classes in the dataset and aim to craft adversarial noise that maximizes their prediction. In that way, the method can reverse the distribution of the classes in the dataset. Thus, the attack will build a profiler that recognizes well patterns on fake labels and is less trained on highly represented classes in the original dataset.
- For the generation of the adversarial noise, we use only one leakage model, particularly the identity model. That results in a good performance of our method against attacks that use this leakage model, but a poor performance on attacks that use the Hamming weight leakage model. Our method could use both leakage models for the generation of the adversarial noise, trying to find a possible perturbation that minimizes the prediction of both true labels simultaneously.
- Our research is purely conducted under the assumption that this adversarial noise can be crafted by an ideal noise generator placed inside the microprocessor chip. This idea generator is assumed that can create noise of any kind by increasing the energy consumption of a specific point to a certain amount. Of course, these capabilities are not realistic, and the final result will differ in practice. It would be exciting to research how an adversarial noise generator could work in a real-world scenario and how far would be the actual results compared to our theoretical approach.
- Another interesting research direction would be to investigate how the application of multiple countermeasures at the same time could improve the overall protection. Our experiments showed that our countermeasure performed better under a specific attacking scenario than another one but worse under a different attacking scenario. Thus, a simultaneous application of these two countermeasures could increase the protection level, as when one fails, the other succeed.

Bibliography

[1] T. Ahmed, B. Oreshkin, and M. Coates. Machine learning approaches to network anomaly detection. In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, SYSML'07, pages 1–6, USA, Apr. 2007. USENIX Association.

[2] ANSSI. ASCAD, June 2021. URL https://github.com/ANSSI-FR/ASCAD. original-date: 2018-01-19T10:04:55Z.

[3] S. Bhasin, N. Bruneau, J.-L. Danger, S. Guilley, and Z. Najm. Analysis and Improvements of the DPA Contest v4 Implementation. In R. S. Chakraborty, V. Matyas, and P. Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering*, Lecture Notes in Computer Science, pages 201–218, Cham, 2014. Springer International Publishing. ISBN 978-3-319-12060-7. doi: 10.1007/978-3-319-12060-7_14.

[4] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique Cryptanalysis of the Full AES. In *Advances in Cryptology – ASIACRYPT 2011*, volume 7073, pages 344–371. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-25384-3 978-3-642-25385-0. doi: 10.1007/978-3-642-25385-0_19. URL http://link.springer.com/10.1007/978-3-642-25385-0_19. Series Title: Lecture Notes in Computer Science.

[5] E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, Lecture Notes in Computer Science, pages 16–29, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-28632-5. doi: 10.1007/978-3-540-28632-5_2.

[6] E. Cagli, C. Dumas, and E. Prouff. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, Lecture Notes in Computer Science, pages 45–68, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66787-4. doi: 10.1007/978-3-319-66787-4_3.

[7] D. Canright and L. Batina. A Very Compact "Perfectly Masked" S-Box for AES. In S. M. Bellovin, R. Gennaro, A. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, pages 446–459, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-68914-0. doi: 10.1007/978-3-540-68914-0_27.

[8] N. Carlini and D. Wagner. Towards Evaluating the Robustness of Neural Networks. *arXiv:1608.04644 [cs]*, Mar. 2017. URL http://arxiv.org/abs/1608.04644. arXiv: 1608.04644.

[9] N. Carlini and D. Wagner. Audio Adversarial Examples: Targeted Attacks on Speech-to-Text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7, May 2018. doi: 10.1109/SPW.2018.00009.

[10] S. Chari, J. R. Rao, and P. Rohatgi. Template Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, Lecture Notes in Computer Science, pages 13–28, Berlin, Heidelberg, 2003. Springer. ISBN 978-3-540-36400-9. doi: 10.1007/3-540-36400-5_3.

[11] CommonLounge. The Advanced Encryption Standard (AES) Algorithm, 2019. URL https://www.commonlounge.com/discussion/e32fdd267aaa4240a4464723bc74d0a5.

[12] J.-S. Coron and I. Kizhvatov. An Efficient Method for Random Delay Generation in Embedded Software. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, Lecture Notes in Computer Science, pages 156–170, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-04138-9. doi: 10.1007/978-3-642-04138-9_12.

[13] N. T. Courtois and L. Goubin. An Algebraic Masking Method to Protect AES Against Power Attacks. In D. H. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005*, Lecture Notes in Computer Science, pages 199–209, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-33355-5. doi: 10.1007/11734727_18.

[14] E. G. Dada, J. S. Bassi, H. Chiroma, S. M. Abdulhamid, A. O. Adetunmbi, and O. E. Ajibuwa. Machine learning for email spam filtering: review, approaches and open research problems. *Heliyon*, 5(6):e01802, June 2019. ISSN 2405-8440. doi: 10.1016/j.heliyon.2019.e01802. URL https://www.sciencedirect.com/science/article/pii/S2405844018353404.

[15] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In J.-J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications*, Lecture Notes in Computer Science, pages 277–284, Berlin, Heidelberg, 2000. Springer. ISBN 978-3-540-44534-0. doi: 10.1007/10721064_26.

[16] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 99–108, New York, NY, USA, Aug. 2004. Association for Computing Machinery. ISBN 978-1-58113-888-7. doi: 10.1145/1014052.1014066. URL http://doi.org/10.1145/1014052.1014066.

[17] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen. ASNI: Attenuated Signature Noise Injection for Low-Overhead Power Side-Channel Attack Immunity. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(10):3300–3311, Oct. 2018. ISSN 1558-0806. doi: 10.1109/TCSI.2018.2819499. Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers.

[18] L. Deng and X. Li. Machine Learning Paradigms for Speech Recognition: An Overview. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(5):1060–1089, May 2013. ISSN 1558-7924. doi: 10.1109/TASL.2013.2244083. Conference Name: IEEE Transactions on Audio, Speech, and Language Processing.

[19] J. F. Dooley. *History of Cryptography and Cryptanalysis: Codes, Ciphers, and Their Algorithms*. History of Computing. Springer International Publishing, 2018. ISBN 978-3-319-90442-9. doi: 10.1007/978-3-319-90443-6. URL http://www.springer.com/gp/book/9783319904429.

[20] P. N. Druzhkov and V. D. Kustikova. A survey of deep learning methods and software tools for image classification and object detection. *Pattern Recognition and Image Analysis*, 26(1):9–15, Jan. 2016. ISSN 1555-6212. doi: 10.1134/S1054661816010065. URL https://doi.org/10.1134/S1054661816010065.

[21] W. Du, Z. Wang, and D. Chen. *Optimizing of Convolutional Neural Network Accelerator*. IntechOpen, Mar. 2018. ISBN 978-1-78923-305-6. doi: 10.5772/intechopen.75796. URL https://www.intechopen.com/books/green-electronics/optimizing-of-convolutional-neural-network-accelerator. Publication Title: Green Electronics.

[22] F. Durvaux and M. Durvaux. SCA-Pitaya: A Practical and Affordable Side-Channel Attack Setup for Power Leakage–Based Evaluations. *Digital Threats: Research and Practice*, 1(1):3:1–3:16, Mar. 2020. ISSN 2692-1626. doi: 10.1145/3371393. URL http://doi.org/10.1145/3371393.

[23] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. D. Jr. Advanced Encryption Standard (AES). Nov. 2001. URL https://www.nist.gov/publications/advanced-encryption-standard-aes. Last Modified: 2021-03-01T01:03-05:00.

[24] G. Emadi, A. M. Rahmani, and H. Shahhoseini. Task Scheduling Algorithm Using Covariance Matrix Adaptation Evolution Strategy (CMA-ES) in Cloud Computing. *Journal of Advances in Computer Engineering and Technology*, 3(3):135–144, Aug. 2017. ISSN 2423-4192. URL https://jacet.srbiau.ac.ir/article_10641.html. Publisher: Science and Research Branch,Islamic Azad University.

[25] H. Fujiyoshi, T. Hirakawa, and T. Yamashita. Deep learning-based image recognition for autonomous driving. *IATSS Research*, 43(4):244–252, Dec. 2019. ISSN 0386-1112. doi: 10.1016/j.iatssr.2019.11.008. URL https://www.sciencedirect.com/science/article/pii/S0386111219301566.

[26] R. Gilmore, N. Hanley, and M. O'Neill. Neural network based attack on a masked implementation of AES. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111, May 2015. doi: 10.1109/HST.2015.7140247.

[27] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[28] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat]*, Mar. 2015. URL http://arxiv.org/abs/1412.6572. arXiv: 1412.6572.

[29] R. Gu, P. Wang, M. Zheng, H. Hu, and N. Yu. Adversarial Attack Based Countermeasures against Deep Learning Side-Channel Attacks. *arXiv:2009.10568 [cs]*, Sept. 2020. URL http://arxiv.org/abs/2009.10568. arXiv: 2009.10568.

[30] T. S. Guzella and W. M. Caminhas. A review of machine learning approaches to Spam filtering. *Expert Systems with Applications*, 36(7):10206–10222, Sept. 2009. ISSN 0957-4174. doi: 10.1016/j.eswa.2009.02.037. URL https://www.sciencedirect.com/science/article/pii/S095741740900181X.

[31] T. Güneysu and A. Moradi. Generic Side-Channel Countermeasures for Reconfigurable Devices. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, B. Preneel, and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917, pages 33–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-23950-2 978-3-642-23951-9. doi: 10.1007/978-3-642-23951-9_3. URL http://link.springer.com/10.1007/978-3-642-23951-9_3. Series Title: Lecture Notes in Computer Science.

[32] S. S. Haykin. *Neural Networks and Learning Machines*. Prentice Hall, 2009. ISBN 978-0-13-147139-9. Google-Books-ID: K7P36lKzI_QC.

[33] B. Hettwer, S. Gehrer, and T. Güneysu. Applications of machine learning techniques in side-channel attacks: a survey. *Journal of Cryptographic Engineering*, 10(2):135–162, June 2020. ISSN 2190-8516. doi: 10.1007/s13389-019-00212-8. URL https://doi.org/10.1007/s13389-019-00212-8.

[34] A. Heuser and M. Zohner. Intelligent Machine Homicide. In W. Schindler and S. A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design*, Lecture Notes in Computer Science, pages 249–264, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-29912-4. doi: 10.1007/978-3-642-29912-4_18.

[35] A. Hiassat, A. Diabat, and I. Rahwan. A genetic algorithm approach for location-inventory-routing problem with perishable products. *Journal of Manufacturing Systems*, 42:93–103, Jan. 2017. ISSN 0278-6125. doi: 10.1016/j.jmsy.2016.10.004. URL https://www.sciencedirect.com/science/article/pii/S0278612516300693.

[36] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293–302, Dec. 2011. ISSN 2190-8508, 2190-8516. doi: 10.1007/s13389-011-0023-x. URL http://link.springer.com/10.1007/s13389-011-0023-x.

[37] M. S. Inci, T. Eisenbarth, and B. Sunar. DeepCloak: Adversarial Crafting As a Defensive Measure to Cloak Processes. *arXiv:1808.01352 [cs]*, Apr. 2020. URL http://arxiv.org/abs/1808.01352. arXiv: 1808.01352.

[38] A. Jancic and M. Warren. PKI - advantages and obstacles. *Proceedings of the 2nd Australian Information Security Management Conference: securing the future*, pages 1–9, Jan. 2004. URL https://dro.deakin.edu.au/view/DU:30005427. Conference Name: Australian Information Security Management. Conference (2nd : 2004 : Perth, Western Australia) ISBN: 9780729805728 Publisher: Edith Cowan University.

[39] X. Jia, X. Wei, X. Cao, and H. Foroosh. ComDefend: An Efficient Image Compression Model to Defend Adversarial Examples. pages 6084–6092, 2019. URL https://openaccess.thecvf.com/content_CVPR_2019/html/Jia_ComDefend_An_Efficient_Image_Compression_Model_to_Defend_Adversarial_Examples_CVPR_2019_paper.html.

[40] H. Kim, S. Hong, and J. Lim. A Fast and Provably Secure Higher-Order Masking of AES S-Box. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, Lecture Notes in Computer Science, pages 95–107, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-23951-9. doi: 10.1007/978-3-642-23951-9_7.

[41] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic. Make Some Noise. Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 148–179, May 2019. ISSN 2569-2925. doi: 10.13154/tches.v2019.i3.148-179. URL https://tches.iacr.org/index.php/TCHES/article/view/8292.

[42] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, Lecture Notes in Computer Science, pages 388–397, Berlin, Heidelberg, 1999. Springer. ISBN 978-3-540-48405-9. doi: 10.1007/3-540-48405-1_25.

[43] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Lecture Notes in Computer Science, pages 104–113, Berlin, Heidelberg, 1996. Springer. ISBN 978-3-540-68697-2. doi: 10.1007/3-540-68697-5_9.

[44] J. Kos, I. Fischer, and D. Song. Adversarial Examples for Generative Models. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 36–42, May 2018. doi: 10.1109/SPW.2018.00014.

[45] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, 13:8–17, Jan. 2015. ISSN 2001-0370. doi: 10.1016/j.csbj.2014.11.005. URL https://www.sciencedirect.com/science/article/pii/S2001037014000464.

[46] V. Kuleshov, S. Thakoor, T. Lau, and S. Ermon. Adversarial Examples for Natural Language Classification Problems. Feb. 2018. URL https://openreview.net/forum?id=r1QZ3zbAZ.

[47] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv:1607.02533 [cs, stat]*, Feb. 2017. URL http://arxiv.org/abs/1607.02533. arXiv: 1607.02533.

[48] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp. In G. Montavon, G. B. Orr, and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade: Second Edition*, Lecture Notes in Computer Science, pages 9–48. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_3. URL https://doi.org/10.1007/978-3-642-35289-8_3.

[49] L. Lerman, G. Bontempi, and O. Markowitch. Power analysis attack: an approach based on machine learning. *International Journal of Applied Cryptography*, 3(2):97, 2014. ISSN 1753-0563, 1753-0571. doi: 10.1504/IJACT.2014.062722. URL http://www.inderscience.com/link.php?id=62722.

[50] L. Lerman, S. F. Medeiros, G. Bontempi, and O. Markowitch. A Machine Learning Approach Against a Masked AES. In A. Francillon and P. Rohatgi, editors, *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 61–75, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08302-5. doi: 10.1007/978-3-319-08302-5_5.

[51] L. Lerman, R. Poussier, O. Markowitch, and F.-X. Standaert. Template attacks versus machine learning revisited and the curse of dimensionality in side-channel analysis: extended version. *Journal of Cryptographic Engineering*, 8(4):301–313, Apr. 2017. ISSN 2190-8508, 2190-8516. doi: 10.1007/s13389-017-0162-9. URL http://link.springer.com/10.1007/s13389-017-0162-9.

[52] J. Liu, W. Zhang, Y. Zhang, D. Hou, Y. Liu, H. Zha, and N. Yu. Detection Based Defense Against Adversarial Examples From the Steganalysis Point of View. pages 4825–4834, 2019. URL https://openaccess.thecvf.com/content_CVPR_2019/html/Liu_Detection_Based_Defense_Against_Adversarial_Examples_From_the_Steganalysis_Point_CVPR_2019_paper.html.

[53] M. M. Lopez and J. Kalita. Deep Learning applied to NLP. *arXiv:1703.03091 [cs]*, Mar. 2017. URL http://arxiv.org/abs/1703.03091. arXiv: 1703.03091.

[54] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang. Traffic Flow Prediction With Big Data: A Deep Learning Approach. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):865–873, Apr. 2015. ISSN 1558-0016. doi: 10.1109/TITS.2014.2345663. Conference Name: IEEE Transactions on Intelligent Transportation Systems.

[55] H. Maghrebi, T. Portigliatti, and E. Prouff. Breaking Cryptographic Implementations Using Deep Learning Techniques. In C. Carlet, M. A. Hasan, and V. Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering*, Lecture Notes in Computer Science, pages 3–26, Cham, 2016. Springer International Publishing. ISBN 978-3-319-49445-6. doi: 10.1007/978-3-319-49445-6_1.

[56] S. Mirjalili. Ant Colony Optimisation. In S. Mirjalili, editor, *Evolutionary Algorithms and Neural Networks: Theory and Applications*, Studies in Computational Intelligence, pages 33–42. Springer International Publishing, Cham, 2019. ISBN 978-3-319-93025-1. doi: 10.1007/978-3-319-93025-1_3. URL https://doi.org/10.1007/978-3-319-93025-1_3.

[57] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997. ISBN 978-0-07-042807-2.

[58] J. Padmanabhan and M. J. J. Premkumar. Machine Learning in Automatic Speech Recognition: A Survey. *IETE Technical Review*, 32(4):240–251, July 2015. ISSN 0256-4602. doi: 10.1080/02564602.2015.1010611. URL https://doi.org/10.1080/02564602.2015.1010611. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/02564602.2015.1010611.

[59] S. Picek, A. Heuser, and S. Guilley. Template attack versus Bayes classifier. *Journal of Cryptographic Engineering*, 7(4):343–351, Nov. 2017. ISSN 2190-8516. doi: 10.1007/s13389-017-0172-7. URL https://doi.org/10.1007/s13389-017-0172-7.

[60] S. Picek, A. Heuser, A. Jovic, S. A. Ludwig, S. Guilley, D. Jakobovic, and N. Mentens. Side-channel analysis and machine learning: A practical perspective. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4095–4102, May 2017. doi: 10.1109/IJCNN.2017.7966373. ISSN: 2161-4407.

[61] S. Picek, I. P. Samiotis, J. Kim, A. Heuser, S. Bhasin, and A. Legay. On the Performance of Convolutional Neural Networks for Side-Channel Analysis. In A. Chattopadhyay, C. Rebeiro, and Y. Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering*, Lecture Notes in Computer Science, pages 157–176, Cham, 2018. Springer International Publishing. ISBN 978-3-030-05072-6. doi: 10.1007/978-3-030-05072-6_10.

[62] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni. The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 209–237, 2019. ISSN 2569-2925. doi: 10.13154/tches.v2019.i1.209-237. URL https://tches.iacr.org/index.php/TCHES/article/view/7339.

[63] S. Picek, D. Jap, and S. Bhasin. Poster: When Adversary Becomes the Guardian – Towards Side-channel Security With Adversarial Attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2673–2675, New York, NY, USA, Nov. 2019. Association for Computing Machinery. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3363284. URL https://doi.org/10.1145/3319535.3363284.

[64] E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Dumas. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database. Technical Report 053, Cryptology ePrint Archive, 2018. URL http://eprint.iacr.org/2018/053.

[65] A. Pumsirirat and L. Yan. Credit Card Fraud Detection using Deep Learning based on Auto-Encoder and Restricted Boltzmann Machine. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 9(1), 2018. ISSN 2156-5570. doi: 10.14569/IJACSA.2018.090103. URL https://thesai.org/Publications/ViewPaper?Volume=9&Issue=1&Code=IJACSA&SerialNo=3. Number: 1 Publisher: The Science and Information (SAI) Organization Limited.

[66] H. Richter and S. Yang. Dynamic Optimization Using Analytic and Evolutionary Approaches: A Comparative Review. In I. Zelinka, V. Snášel, and A. Abraham, editors, *Handbook of Optimization: From Classical to Modern Approach*, Intelligent Systems Reference Library, pages 1–28. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-30504-7. doi: 10.1007/978-3-642-30504-7_1. URL https://doi.org/10.1007/978-3-642-30504-7_1.

[67] J. Rijsdijk, L. Wu, G. Perin, and S. Picek. Reinforcement Learning for Hyperparameter Tuning in Deep Learning-based Side-channel Analysis. Technical Report 071, 2021. URL https://eprint.iacr.org/2021/071.

[68] M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, Lecture Notes in Computer Science, pages 413–427, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-15031-9. doi: 10.1007/978-3-642-15031-9_28.

[69] K. Schramm and C. Paar. Higher Order Masking of the AES. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, Lecture Notes in Computer Science, pages 208–225, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-32648-9. doi: 10.1007/11605805_14.

[70] B. Shi, X. Bai, and C. Yao. An End-to-End Trainable Neural Network for Image-Based Sequence Recognition and Its Application to Scene Text Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(11):2298–2304, Nov. 2017. ISSN 1939-3539. doi: 10.1109/TPAMI.2016.2646371. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[71] N. P. Smart. *Cryprography Made Simple*. Information Security and Cryptography. Springer International Publishing, Cham, 2016. ISBN 978-3-319-21936-3. doi: 10.1007/978-3-319-21936-3_7. URL https://doi.org/10.1007/978-3-319-21936-3_7.

[72] F. Standaert, E. Peeters, and J. Quisquater. On the masking countermeasure and higher-order power analysis attacks. In *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, volume 1, pages 562–567 Vol. 1, Apr. 2005. doi: 10.1109/ITCC.2005.213.

[73] F.-X. Standaert. Introduction to Side-Channel Attacks. In I. M. Verbauwhede, editor, *Secure Integrated Circuits and Systems*, Integrated Circuits and Systems, pages 27–42. Springer US, Boston, MA, 2010. ISBN 978-0-387-71829-3. doi: 10.1007/978-0-387-71829-3_2. URL https://doi.org/10.1007/978-0-387-71829-3_2.

[74] F.-X. Standaert, T. G. Malkin, and M. Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, Lecture Notes in Computer Science, pages 443–461, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-01001-9. doi: 10.1007/978-3-642-01001-9_26.

[75] R. Storn and K. Price. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, Dec. 1997. ISSN 1573-2916. doi: 10.1023/A:1008202821328. URL https://doi.org/10.1023/A:1008202821328.

[76] J. Su, D. V. Vargas, and K. Sakurai. One Pixel Attack for Fooling Deep Neural Networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, Oct. 2019. ISSN 1089-778X, 1089-778X, 1941-0026. doi: 10.1109/TEVC.2019.2890858. URL https://ieeexplore.ieee.org/document/8601309/.

[77] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv:1312.6199 [cs]*, Feb. 2014. URL http://arxiv.org/abs/1312.6199. arXiv: 1312.6199.

[78] B. Tao and H. Wu. Improving the Biclique Cryptanalysis of AES. In *Information Security and Privacy*, Lecture Notes in Computer Science, pages 39–56, Cham, 2015. Springer International Publishing. ISBN 978-3-319-19962-7. doi: 10.1007/978-3-319-19962-7_3. URL https://link.springer.com/chapter/10.1007%2F978-3-319-19962-7_3.

[79] L. A. Tawalbeh, H. Houssain, and T. F. Al-Somani. Review of Side Channel Attacks and Countermeasures on ECC, RSA, and AES Cryptosystems. 5:11, 2016.

[80] H. Wang, N. Wang, and D.-Y. Yeung. Collaborative Deep Learning for Recommender Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1235–1244. Association for Computing Machinery, New York, NY, USA, Aug. 2015. ISBN 978-1-4503-3664-2. URL http://doi.org/10.1145/2783258.2783273.

[81] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng. End-to-end text recognition with convolutional neural networks. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 3304–3308, Nov. 2012. ISSN: 1051-4651.

[82] Y. Wang and W. Xu. Leveraging deep learning with LDA-based text analytics to detect automobile insurance fraud. *Decision Support Systems*, 105:87–95, Jan. 2018. ISSN 0167-9236. doi: 10.1016/j.dss.2017.11.001. URL https://www.sciencedirect.com/science/article/pii/S0167923617302130.

[83] L. Wu and S. Picek. Remove Some Noise: On Pre-processing of Side-channel Measurements with Autoencoders. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 389–415, Aug. 2020. ISSN 2569-2925. doi: 10.13154/tches.v2020.i4.389-415. URL https://tches.iacr.org/index.php/TCHES/article/view/8688.

[84] L. Wu, G. Perin, and S. Picek. I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis. Technical Report 1293, 2020. URL https://eprint.iacr.org/2020/1293.

[85] T. Wu, S. Liu, J. Zhang, and Y. Xiang. Twitter spam detection based on deep learning. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '17, pages 1–8, New York, NY, USA, Jan. 2017. Association for Computing Machinery. ISBN 978-1-4503-4768-6. doi: 10.1145/3014812.3014815. URL http://doi.org/10.1145/3014812.3014815.

[86] H. Yao, X. Tang, H. Wei, G. Zheng, and Z. Li. Revisiting Spatial-Temporal Similarity: A Deep Learning Framework for Traffic Prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):5668–5675, July 2019. ISSN 2374-3468. doi: 10.1609/aaai.v33i01.33015668. URL https://ojs.aaai.org/index.php/AAAI/article/view/4511. Number: 01.

[87] L. Yin. A Summary of Neural Network Layers, July 2018. URL https://medium.com/machine-learning-for-li/different-convolutional-layers-43dc146f4d0e.

[88] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial Examples: Attacks and Defenses for Deep Learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, Sept. 2019. ISSN 2162-2388. doi: 10.1109/TNNLS.2018.2886017. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

A
Fine-tuning

In this appendix, we present in detail the results of the hyperparameter tuning process, as described in section 4.2.2. In each plot, we present the achieve guessing entropy and success rate, alongside to the required number of traces for a successful attack (GE=1 and SR=100%). A missing bar indicates that the attack was not successful with 2,000 traces.

A.1. MLP architecture

A.1.1. Architecture hyperparameters

Number of fully connected layers

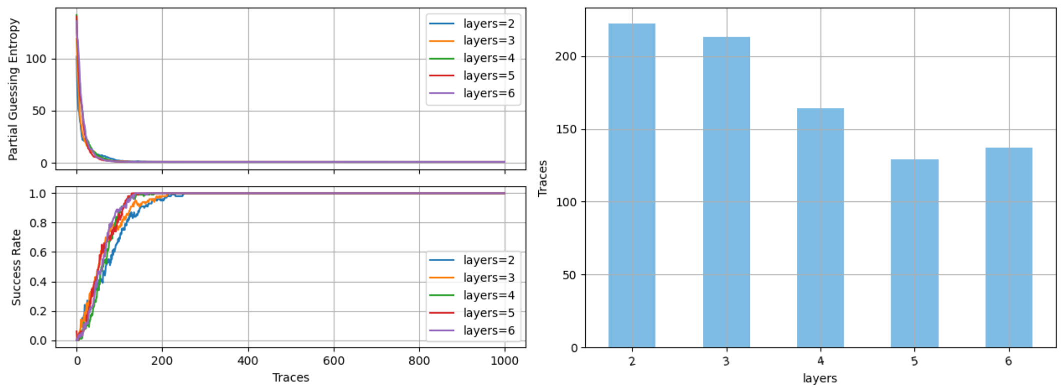


Figure A.1: Performance of MLP model for different number of fully connected layers

Units per layer

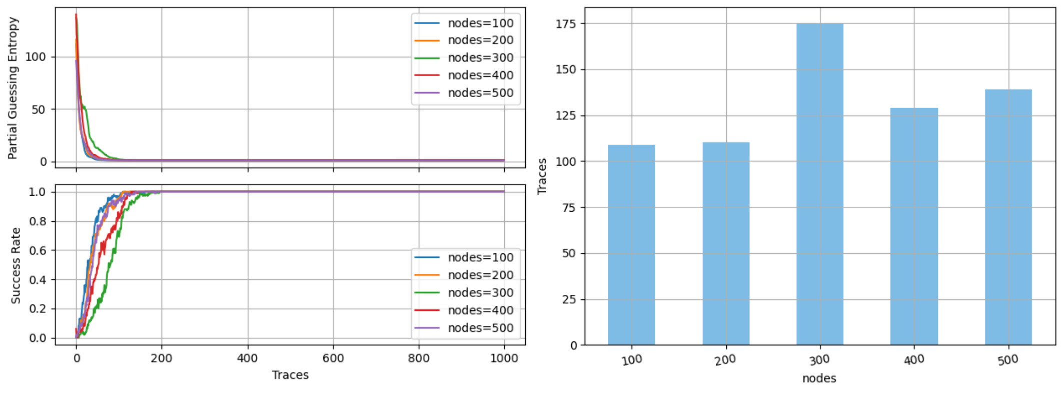


Figure A.2: Performance of MLP model for different number of nodes per layer

Activation function

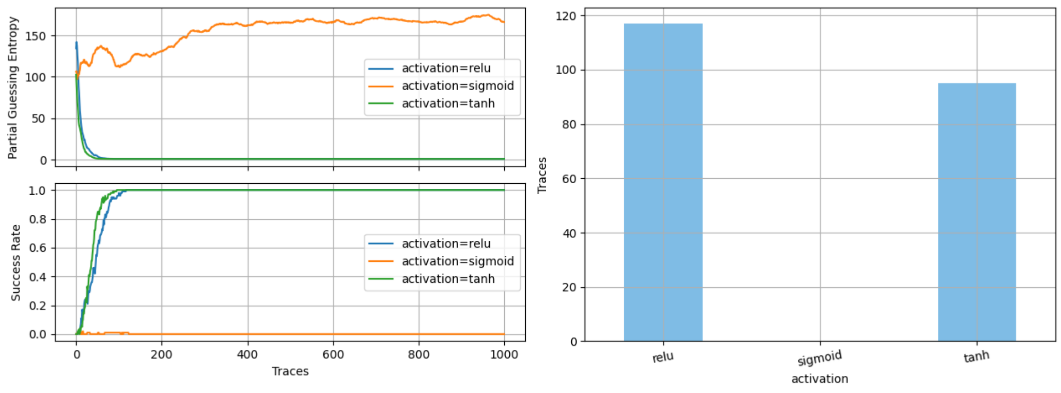


Figure A.3: Performance of MLP model for different activation functions

A.1.2. Training hyperparamaters

Epochs

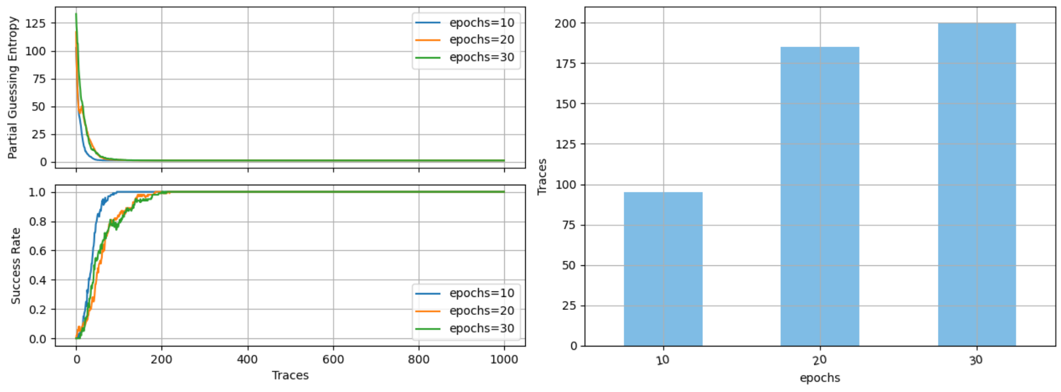


Figure A.4: Performance of MLP model for different number of training epochs

Batch Size

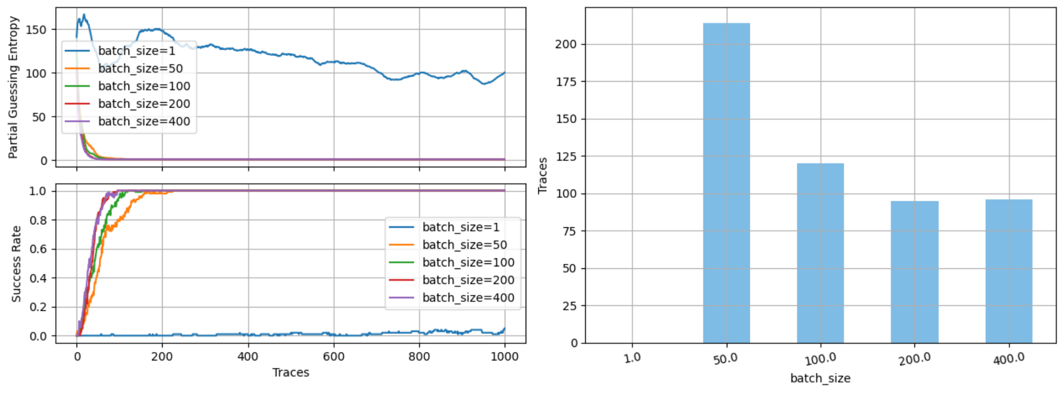


Figure A.5: Performance of MLP model for different batch size during training

Optimizer

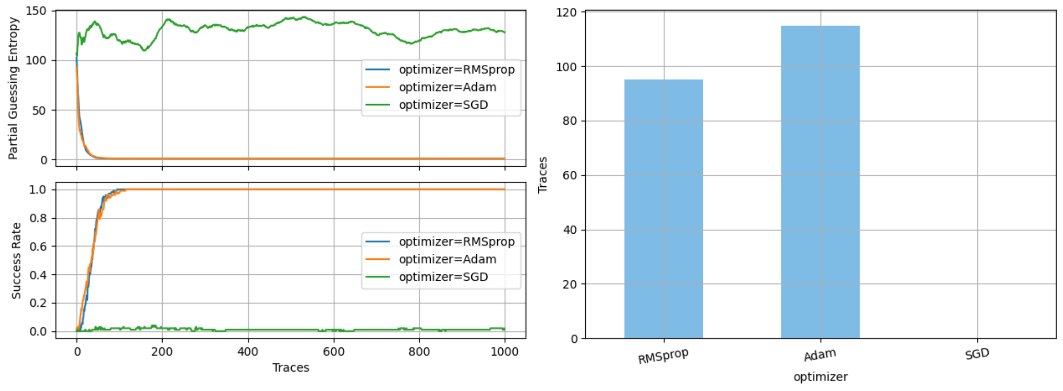


Figure A.6: Performance of MLP model for different optimizer functions

Learning rate

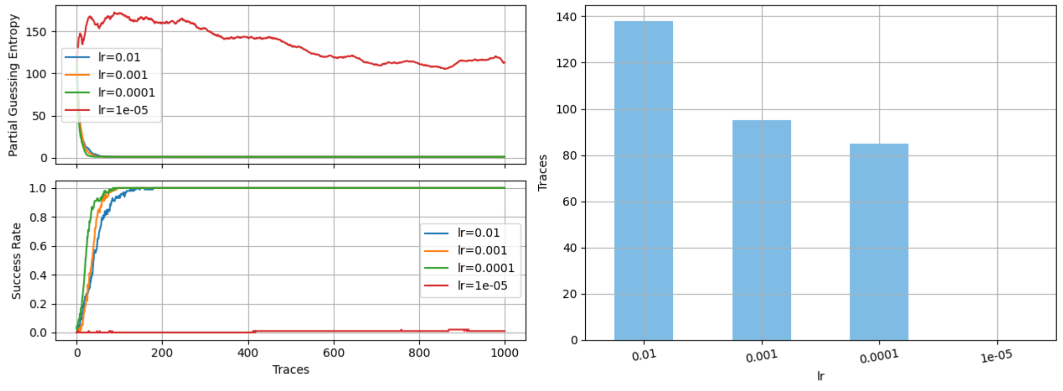


Figure A.7: Performance of MLP model for different learning rates

A.2. CNN architecture

A.2.1. Architecture hyperparameters

Number of convolution blocks

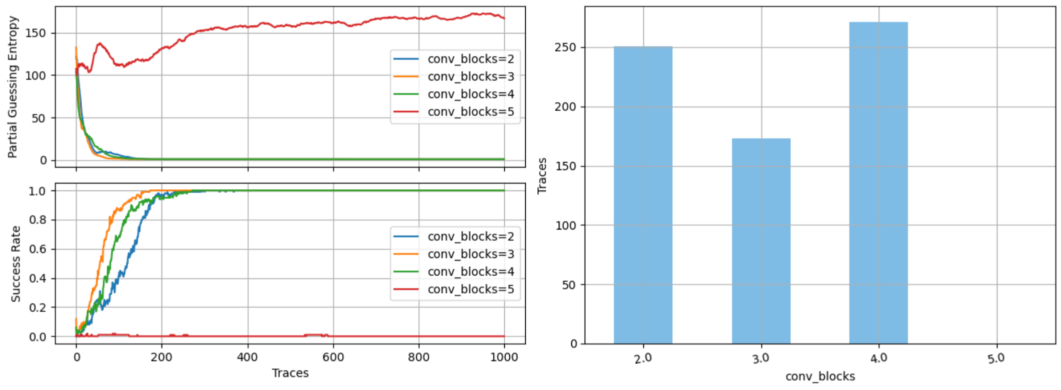


Figure A.8: Performance of CNN model for different number of convolution blocks

Output size of the first convolution layer

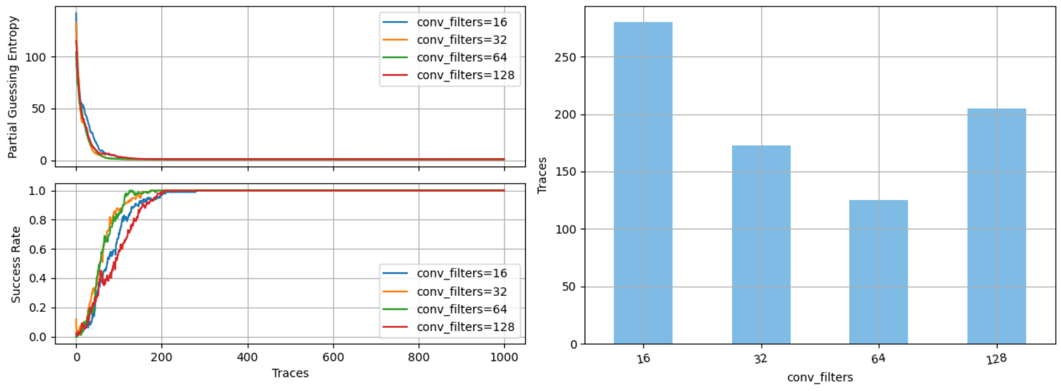


Figure A.9: Performance of CNN model for different number of filters

Length of the 1D convolution window

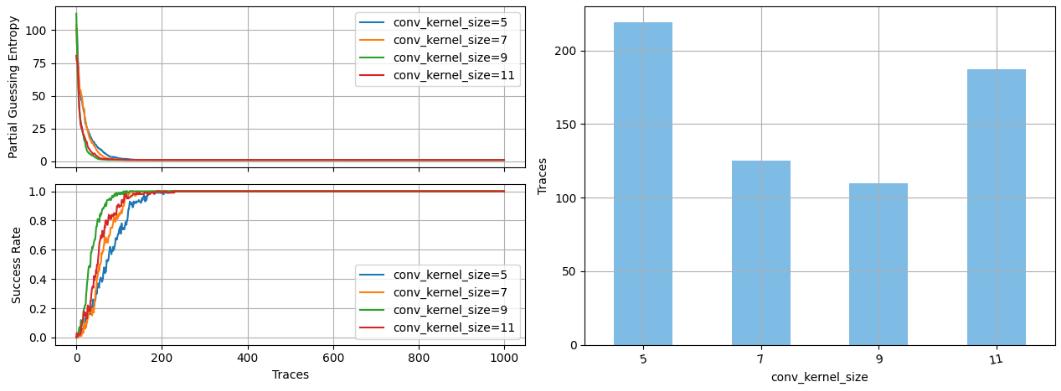


Figure A.10: Performance of CNN model for different sizes of convolution window

Type of pooling layer

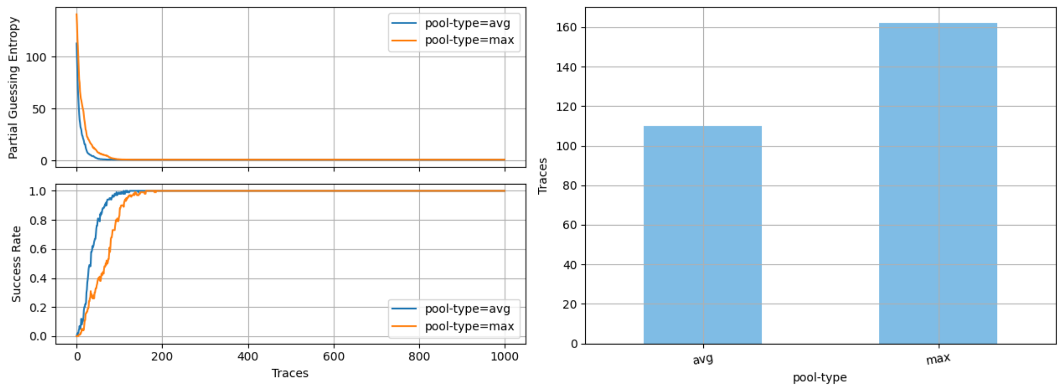


Figure A.11: Performance of CNN model for pooling layer types

Number of fully connected layers

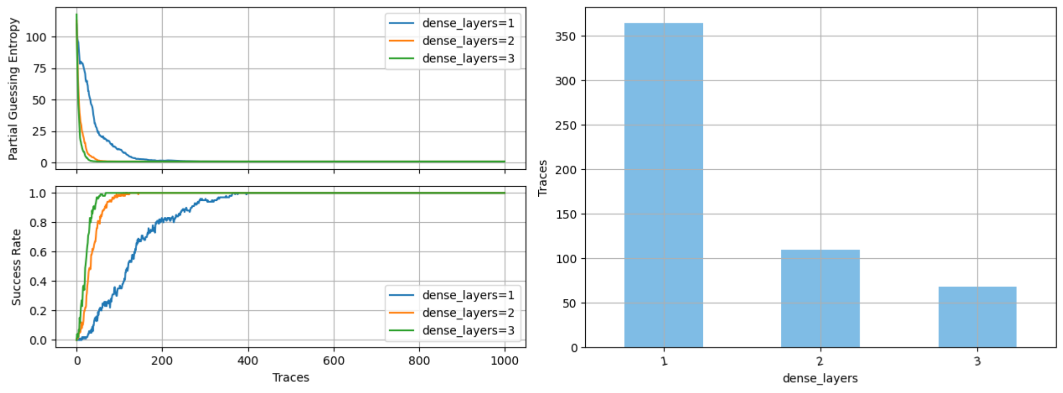


Figure A.12: Performance of CNN model for different number of FC layers

Units per FC layer

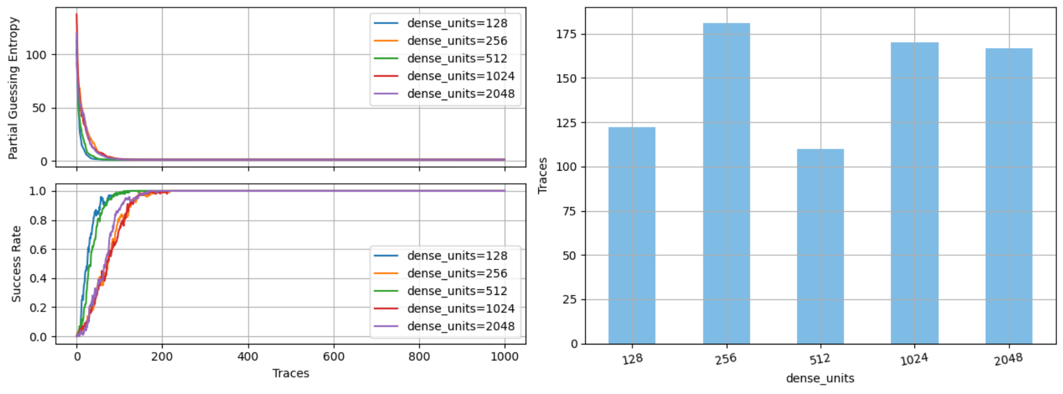


Figure A.13: Performance of CNN model for different number of units per FC layer

Activation function

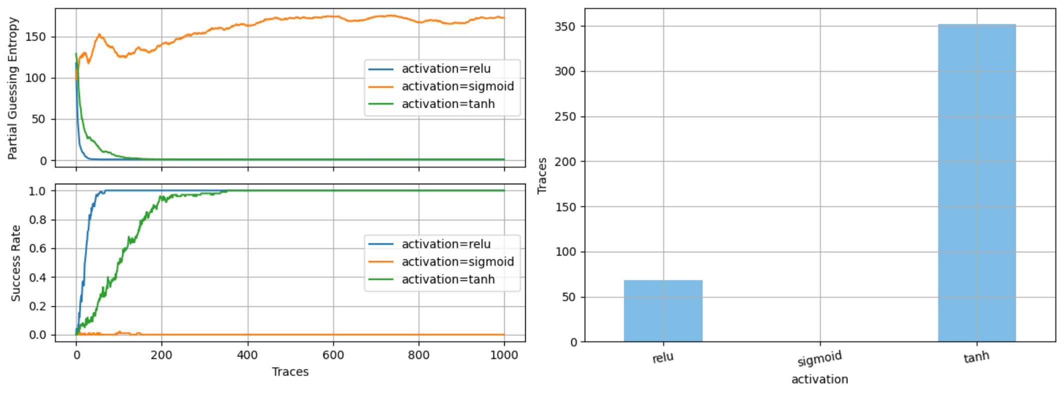


Figure A.14: Performance of CNN model for different activation functions

A.2.2. Training hyperparamaters

Epochs

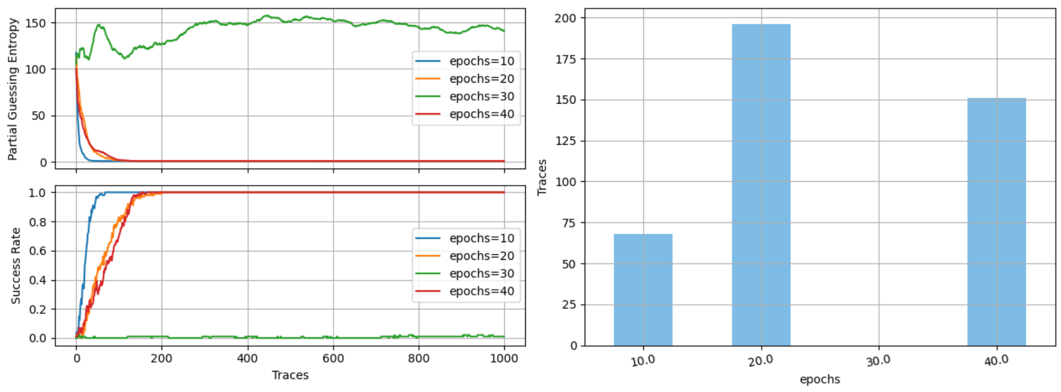


Figure A.15: Performance of CNN model for different number of training epochs

Batch Size

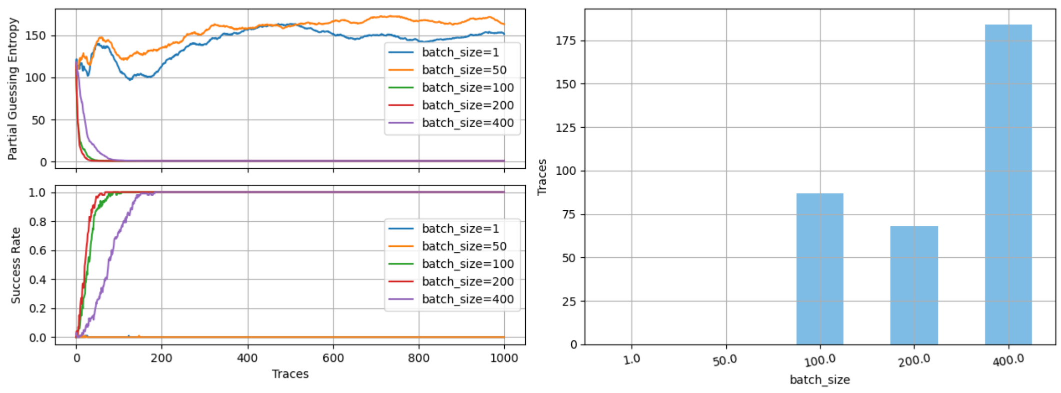


Figure A.16: Performance of CNN model for different batch size during training

Optimizer

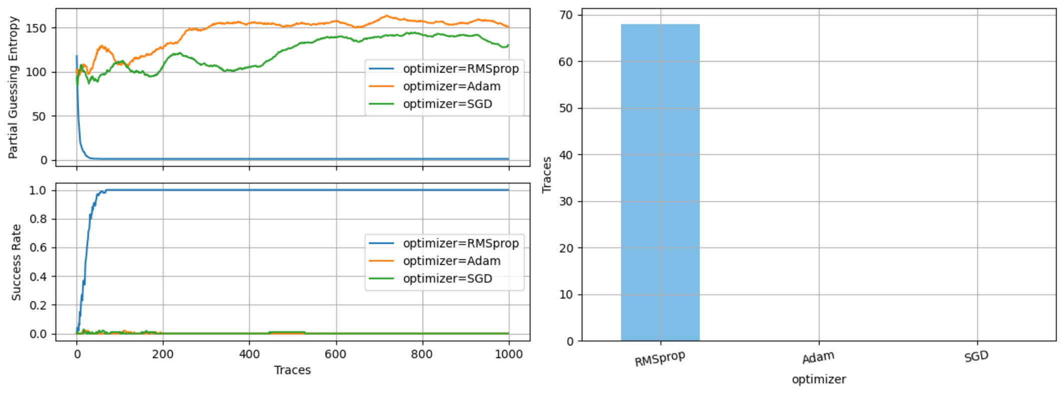


Figure A.17: Performance of CNN model for different optimizer functions

Learning rate

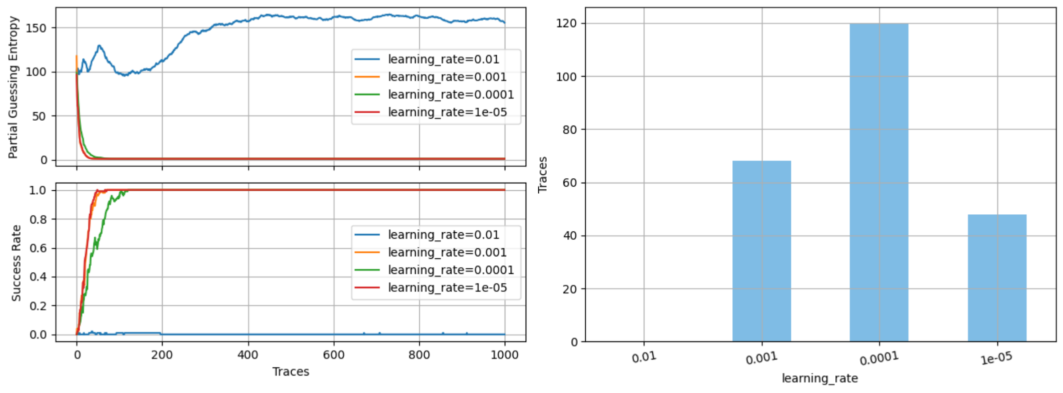


Figure A.18: Performance of CNN model for different learning rates

B

L2 norms of generated datasets

	bounds=[0, 100)	bounds=[0, 200)	bounds=[0, 300)	bounds=[0, 400)	bounds=[0, 500)
10 features	199.50	394.42	588.37	780.93	973.36
20 features	271.19	539.17	805.97	1071.71	1335.10
30 features	328.08	653.51	977.69	1300.23	1620.11
40 features	377.57	752.58	1125.90	1496.96	1866.45
50 features	421.73	840.95	1258.33	1672.72	2085.65
60 features	462.07	921.70	1378.82	1833.65	2287.08
70 features	499.46	996.57	1490.76	1982.89	2474.29
80 features	535.25	1067.88	1596.89	2125.15	2652.13
90 features	569.25	1135.50	1698.52	2260.08	2821.03
100 features	601.65	1199.96	1795.34	2389.12	2982.13

Table B.1: Average l_2 distance metric of datasets generated by k-Point Protection using the MLP_{best} model

	bounds=[0, 100)	bounds=[0, 200)	bounds=[0, 300)	bounds=[0, 400)	bounds=[0, 500)
10 features	185.05	369.93	554.58	738.56	921.35
20 features	261.03	522.76	784.43	1044.17	1303.27
30 features	320.31	641.79	962.20	1281.54	1599.36
40 features	371.03	743.64	1114.31	1483.63	1852.70
50 features	416.24	834.05	1249.79	1663.66	2077.51
60 features	457.64	916.79	1373.24	1828.62	2283.49
70 features	496.11	993.03	1487.55	1981.12	2475.25
80 features	532.32	1064.90	1594.87	2124.67	2654.86
90 features	566.83	1133.32	1696.99	2261.34	2826.57
100 features	599.68	1198.34	1794.80	2392.51	2989.60

Table B.2: Average l_2 distance metric of datasets generated by k-Point Protection using the CNN_{best} model

	bounds=[0, 100)	bounds=[0, 200)	bounds=[0, 300)	bounds=[0, 400)	bounds=[0, 500)
10 features	188.03	373.05	557.13	740.19	922.31
20 features	260.80	519.68	777.26	1032.99	1287.60
30 features	317.15	632.74	947.06	1259.31	1569.31
40 features	365.10	728.93	1090.60	1449.92	1806.84
50 features	407.40	813.35	1216.76	1618.02	2016.53
60 features	445.64	889.66	1331.36	1770.09	2206.73
70 features	480.92	960.09	1436.48	1910.25	2382.38
80 features	513.84	1025.74	1534.40	2040.16	2545.42
90 features	544.67	1087.43	1626.51	2163.12	2699.63
100 features	573.82	1145.48	1713.51	2278.89	2844.70

Table B.3: Average l_2 distance metric of datasets generated by random k-Point Protection using the MLP$_{\text{best}}$ model

	bounds=[0, 100)	bounds=[0, 200)	bounds=[0, 300)	bounds=[0, 400)	bounds=[0, 500)
10 features	179.32	359.64	542.07	726.38	911.79
20 features	254.83	511.59	771.94	1032.86	1294.25
30 features	312.76	629.17	948.57	1267.36	1584.94
40 features	361.81	728.47	1097.32	1463.73	1828.47
50 features	405.21	816.14	1227.34	1635.43	2042.48
60 features	444.51	895.37	1344.58	1790.41	2236.35
70 features	480.67	967.85	1451.68	1932.63	2413.72
80 features	514.51	1034.93	1551.14	2065.91	2580.40
90 features	546.41	1097.72	1644.13	2190.00	2736.71
100 features	576.34	1156.90	1732.32	2307.70	2884.09

Table B.4: Average l_2 distance metric of datasets generated by random k-Point Protection using the CNN$_{\text{best}}$ model

	bounds=[0, 100)	bounds=[0, 200)	bounds=[0, 300)	bounds=[0, 400)	bounds=[0, 500)
10 features	196.85	384.52	557.92	720.31	878.53
20 features	268.27	514.95	757.60	1000.00	1240.53
30 features	322.69	625.17	920.18	1225.51	1542.13
40 features	368.04	715.96	1061.69	1423.84	1788.53
50 features	409.33	799.46	1189.95	1599.21	2003.85
60 features	446.65	876.50	1315.39	1755.37	2178.17
70 features	481.08	947.82	1420.57	1891.67	2363.00
80 features	512.70	1013.03	1522.35	2028.65	2520.70
90 features	543.24	1078.26	1619.08	2147.07	2670.63
100 features	571.64	1137.42	1704.60	2264.12	2815.33

Table B.5: Average l_2 distance metric of datasets generated by correlation k-Point Protection using the MLP$_{\text{best}}$ model

	bounds=[0, 100)	bounds=[0, 200)	bounds=[0, 300)	bounds=[0, 400)	bounds=[0, 500)
10 features	185.22	367.75	546.34	724.55	902.23
20 features	261.29	515.11	770.09	1024.67	1287.26
30 features	316.88	629.94	943.28	1263.52	1576.18
40 features	365.29	727.59	1095.04	1456.80	1821.34
50 features	408.22	813.05	1223.25	1629.56	2037.21
60 features	446.94	893.89	1339.23	1785.65	2231.98
70 features	482.59	966.28	1447.50	1929.05	2411.67
80 features	515.62	1032.77	1547.50	2062.41	2579.04
90 features	546.90	1094.69	1641.36	2188.39	2735.73
100 features	576.42	1154.13	1730.25	2306.84	2884.24

Table B.6: Average l_2 distance metric of datasets generated by correlation k-Point Protection using the CNN$_{\text{best}}$ model

C
Attacking models

In this appendix, we present the architecture of the models used in our expirements. The names of the layers and hyperparameters follow the Tensorflow2 framework's notations.

C.1. MLP$_{\text{best}}$

Layer	Hyperparameters
Input layer	
Input	`nodes=1000`
Hidden layers	
Dense	`nodes=100, activation='tanh'`
Dense	`nodes=100, activation='tanh'`
Dense	`nodes=100, activation='tanh'`
Dense	`nodes=100, activation='tanh'`
Dense	`nodes=100, activation='tanh'`
Output layer	
Dense	`nodes=256(ID)/9(HW), activation='softmax'`

Table C.1

C.2. CNN$_{\text{best}}$

Layer	Hyperparameters
Input layer	
Input	`nodes=1000`
Hidden layers	
Conv1D	`filters=64, kernel_size=9, activation='relu', padding='same'`
AveragePooling1D	`pool_size=2, strides=2`
Conv1D	`filters=128, kernel_size=9, activation='relu', padding='same'`
AveragePooling1D	`pool_size=2, strides=2`
Conv1D	`filters=256, kernel_size=9, activation='relu', padding='same'`
AveragePooling1D	`pool_size=2, strides=2`
Flatten	`N/A`
Dense	`nodes=512, activation='relu'`
Dense	`nodes=512, activation='relu'`
Dense	`nodes=512, activation='relu'`
Output layer	
Dense	`nodes=256(ID)/9(HW), activation='softmax'`

Table C.2

C.3. CNN$_{\text{Rijsdijk}}$

C.3.1. ID model

Layer	Hyperparameters
Input layer	
Input	`nodes=1000`
Hidden layers	
Conv1D	`filters=128, kernel_size=3, activation='selu', padding='same'`
AveragePooling1D	`pool_size=75, strides=75`
Flatten	`N/A`
Dense	`nodes=30, activation='selu'`
Dense	`nodes=2, activation='selu'`
Output layer	
Dense	`nodes=256, activation='softmax'`

Table C.3

C.3.2. HW model

Layer	Hyperparameters
Input layer	
Input	`nodes=1000`
Hidden layers	
Conv1D	`filters=8, kernel_size=3, activation='selu', padding='same'`
AveragePooling1D	`pool_size=25, strides=25`
Flatten	`N/A`
Dense	`nodes=30, activation='selu'activatio`
Dense	`nodes=30, activation='selu'activatio`
Dense	`nodes=2, activation='selu'activatio`
Output layer	
Dense	`nodes=9, activation='softmax'`

Table C.4

C.4. MLP$_{\text{Wu}}$

C.4.1. ID model

Layer	Hyperparameters
Input layer	
Input	`nodes=1000`
Hidden layers	
Dense	`nodes=256, activation='elu'`
Dense	`nodes=256, activation='elu'`
Dense	`nodes=296, activation='elu'`
Dense	`nodes=840, activation='elu'`
Dense	`nodes=280, activation='elu'`
Dense	`nodes=568, activation='elu'`
Dense	`nodes=672, activation='elu'`
Output layer	
Dense	`nodes=256, activation='softmax'`

Table C.5

C.4.2. HW model

Layer	**Hyperparameters**
	Input layer
Input	`nodes=1000`
	Hidden layers
Dense	`nodes=328, activation='elu'`
Dense	`nodes=328, activation='elu'`
	Output layer
Dense	`nodes=9, activation='softmax'`

Table C.6