POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Deep Learning Techniques for Side Channel Cipher Attack

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Massimiliano Roccamena**

Student ID: 10499005
Advisor: Prof. Matteo Matteucci
Co-advisors: Marco Cannici, Davide Zoni
Academic Year: 2021-2022

Abstract

Side-channel analysis of cryptographic implementations represents a critical issue for the design of secure hardware architectures. When observing physical properties of the hardware platform during executions, the side-channel attacker is able to leak informations about the secret used by the cryptosystem. Most recent line of works have showed how the machine learning revolution greatly enhance the attacking capabilities of the adversarial. In particular, artificial neural networks and deep learning models have been showed to play an important role for side-channel leakage modelling. Some of the most effective countermeasures to classical attacks relies on the introduction of process variability and desynchronizations in the computing platform. These protections have been proven to be very effective when applied for CMOS devices to counter most common attacks, like the well-known bayesian template attack. Deep learning models have instead showed to be very effective against a wide variety of hardware and software countermeasures. The objective of the thesis is to describe the role of machine learning and deep learning methods for modern side-channel analysis. We will present different techniques to exploit artificial neural networks for side-channel modelling, focusing on the capabilities of deep learning models to effectively counter protected implementations of the cryptosystem.

Keywords: side-channel attacks, FPGA, process variability, desynchronization, RDFVS, machine learning, deep learning, template attack, convolutional neural networks

Abstract in lingua italiana

L'analisi del canale laterale delle implementazioni crittografiche rappresenta un problema critico per la progettazione di architetture hardware sicure. Osservando le proprietà fisiche della piattaforma hardware durante varie esecuzioni, l'attaccante del canale laterale è in grado di estrarre informazioni sulla chiave segreta utilizzata dal crittosistema. Gli studi più recenti hanno mostrato come la rivoluzione dell'apprendimento automatico migliori notevolmente le capacità di attacco dell'avversario. In particolare, è stato mostrato come le reti neurali artificiali e i modelli di apprendimento profondo ricoprano un ruolo importante per la modellazione dell'informazione contenuta nel canale laterale. Alcune delle contromisure più efficaci agli attacchi classici si basano sull'introduzione di variabilità di processo e di desincronizzazioni nella piattaforma di calcolo. Queste protezioni si sono rivelate molto efficaci quando applicate nei dispositivi CMOS per contrastare gli attacchi più comuni, come il noto attacco bayesiano a template. I modelli di apprendimento profondo hanno invece dimostrato di essere molto efficaci contro un'ampia varietà di contromisure hardware e software. L'obiettivo della tesi è quello di mostrare il ruolo dei metodi di apprendimento automatico e apprendimento profondo nella moderna analisi del canale laterale. Presenteremo diverse tecniche per sfruttare le reti neurali artificiali in questo contesto, concentrando la nostra ricerca sulle capacità dei modelli di apprendimento profondo di contrastare efficacemente le implementazioni protette del crittosistema.

Parole chiave: attacchi a canale laterale, FPGA, variabilità di processo, desincronizzazione, RDFVS, apprendimento automatico, apprendimento profondo, attacco template, reti neurali convoluzionali

Contents

Introduction

In the context of computer security, *Implementation Attacks* (IA) are a family of cryptanalytic attacks which exploits practical realizations of cryptographic devices instead of logical descriptions of the cryptosystem. These type of attacks have gained much interest recently. For instance the *Internet-of-Things* (IoT) revolution highlighted the implementation attacks as a critical threat to the security of the embedded systems implementing and executing cryptographic primitives. In constrained devices such as chip–cards, naive implementations of cryptographic algorithms can be broken with minimal effort [49].

A family of implementation attacks are *Side-Channel Attacks* (SCA), which exploit measurements of some physical properties of the victim device while running the cryptosystem. Adversaries can obtain sensitive information from side channels such as timing of operations [29], power consumption [28], electromagnetic emanations [17] and others. *Continuous Side-Channel Attacks*, in which the adversary gets information at each invocation of the cryptosystem, are especially threatening. Many implementations of block ciphers have been practically broken by continuous side-channel analysis [34, 36].

One of the most studied side-channel in the literature is the power consumption of the computing platform, which has been showed to be a rich source of information for the side-channel attacker. Indeed, since the first pioneering work by Kocher et al. [28], a significant amount of literature has focused on the understanding of the principles and effectiveness of power-based side-channel attacks [6, 9, 30, 41, 47, 48] and the identification of sound and provably countermeasures [8, 11, 33, 34, 51].

The power consumption, and in general any side-channel signal, strongly depends on the physical properties and on the microarchitecture of the computing platform. For example, it is well known that for CMOS device the power consumption is linearly correlated with the number of bits set to 1, thus leaking informations about the value of some sensitive variable [26]. The implementation attacks can even breach into a mathematically secure cipher, therefore the security concerns for the system are moved from studying the theoretical weaknesses of the algorithm to designing secure physical and microarchitectural layers for the device executing the cryptographic primitives.

The classification of side-channel attacking methods is mainly driven by the degree of knowledge available for the adversarial about the attacked computing platform. Therefore, two main families of attacks can be distinguished:

- *Profiling SCA* are the most powerful attacks since they assume that the adversary may priorly use an open copy of the target device to tune all the parameters of the attack. First, the adversary uses a copy of the target device and uses it to model the physical leakage, and then he performs a key-recovery process on the victim device.
- *Non-profiling SCA* corresponds to a much weaker attacker who can only measure the physical leakage on the victim device. To recover the private key, he performs some statistical analyses to detect the dependency between the leakage measurements and the secret variable.

Within the research field of the implementation attacks, the profling attacks represents the class of side-channel attacks that demonstrated to be the most efficient at extracting the secret key. The most important representative of such techniques is the well known *Template Attack* [9], which assumes two main properties about the modelled leakage. First, each collected side-channel measurement is supposed to be affected by an additive gaussian noise. Second, each collected trace of the measured side-channel signal is modelled as a random multivariate gaussian distribution. A wide variety of variants for this method have been developed in the last years. In particular, in the context of side-channel analysis lot of results have been achieved by classical Machine Learning based methods which exploits decision tree based models and and maximum-margin classifiers [12, 30, 42].

However, a fundamental assumption for the success of most classical profiled attacks is that the engineered leakage model provides a good, ideally perfect, description about the behavior of the platform being attacked. In this scenario, the process variability between the two devices can severely affect the probability of success of attacks such as the Template Attack [7, 10]. Process variability, and in particular temporal misalignments of the side-channel, have a big impact on the probability of success of such attacks. This property is usually exploited to implement countermeasures.

The most recent line of works have investigated the usage of *Artificial Feed Forward Neural Networks* and *Deep Learning* techniques in the context of side-channel analysis [7, 16, 41]. These kind of models provides a very powerful leakage modelling framework to the side-channel attacker. Differently from the classical attacking methods, the neural based systems such as *Multi Layer Perceptrons* and *Convolutional Networks* are able to efficiently attack a wide variety of protected implementations of the cryptosystem.

Some works highlights the fact that the neural-based attacks achieve great results thanks to the the exploitation of some well-known Machine Learning and Deep Learning related concepts, such as *Data Augmentation* [5, 7], detection and localization based methods [32, 52] and *Domain Adaptation* [20, 40].

For our thesis, we will be focused on studying the role of Machine Learning and Deep Learning theory in the context of side-channel attacks, in particular in scenarios where the power consumption is the observable channel. Our main goal is to attack an unmasked AES implementation deployed on a FPGA based computing platform. Following the approach of Prouff et al. [41], our experiments will be performed on an extensive open-source database of power consumption traces generated by the mentioned device. The main goal of thesis is to show how different techniques behave when facing different architectural assumptions of the cryptosystem, focusing on the effectiveness of hardware countermeasures. Therefore, our work will not cover a comprehensive study of the hyper-parameterization of each kind of specific model employed for attacking the device. On the other end, we will present the main neural architectural patterns, learning and inference methods which could be exploited to effectively achieve such attacks. We will show that in this way the attacker can easily predict the secret used by one of the most theoretically secure cipher. The adversarial achieve the key guess by just observing the side-channel leakages of the platform during the encryptions, with the corresponding plain texts.

The rest of the manuscript is organized in 4 chapters. Chapter 1 presents the state of the art and the theoretical background needed for our thesis. The narrative of the overall chapter will be based on a probabilistic and bayesian characterization of the problem. Chapter 2 provides a full description of the method being used, with the formalizations of the main algorithms being used to realize the side-channel analysis. Chapter 3 provides an extensive experimental evaluation of the results obtained from the above mentioned attack settings. Finally, in chapter 4 conclusions will be inferred about the overall analysis, and future developments for the discussed methods will be identified.

1 | Background

Throughout this document, the symbols x and $\overrightarrow{x}$ will be used to denote respectively a scalar and a column vector, while bold capital letter $\boldsymbol{X}$ will be used to denote matrices or tensors. The i-th entry of an array-like variable is denoted by $\overrightarrow{x}[i]$ or $\boldsymbol{X}[i,j]$ respectively for a vector $\overrightarrow{x}$ and matrix $\boldsymbol{X}$. Calligraphic letters as $\mathcal{X}$ will be used to denote sets, the corresponding upper-case letter X is used to denote random variables (random vectors $\overrightarrow{X}$ if with an arrow) over $\mathcal{X}$, and the corresponding lower-case letter x_i (resp. $\overrightarrow{x_i}$ for vectors) to denote the i-th element of a finite set $\mathcal{X}$ or the i-th realization of random variable X (resp. $\overrightarrow{X}$). Given N observed realizations of the random vector $\overrightarrow{X}$ over $\mathcal{X}^M$, where M being the dimensionality of the vector, the realizations are related to the rows of the dataset matrix $\boldsymbol{X} \in \mathcal{X}^{N \times M}$ by the equivalence $\overrightarrow{x_i} = \boldsymbol{X}[i]^T$. The *probability mass function* (a.k.a. the *probability distribution function*, pdf for short) of a random variable X will be denoted by f_X. It is defined for any possible value $x \in \mathcal{X}$ by $f_X(x) = P(X = x)$. The symbol $\mathbb{E}[\dots]$ denotes the *expected value*, and might be subscripted by a random variable $\mathbb{E}_X[\dots]$, or by probability distribution $\mathbb{E}_{f_X}[\dots]$, to specify under which probability distribution it is computed. Sometimes categorical values $y_j \in \mathcal{Y}$ will be represented using the the *one-hot encoding* representation, which is the function $y_j \to \overrightarrow{y_j} \in \{0,1\}^{|\mathcal{Y}|}$ such that $\overrightarrow{y_j}[j] = 1 \wedge \overrightarrow{y_j}[i] = 0 \; \forall i \neq j$.

We will use directed probabilistic graphical model with plates notation (Figure 1.1) to describe in a bayesian way most of the data processing algorithms that will be discussed. Will usually denote as $\hat{f}$ the model which compute a given *generative* distribution of some features $\overrightarrow{X}$, and we will use the $\hat{g}$ symbol to identify the model which compute a given *predictive* distribution of some target variables $\overrightarrow{Y}$.

For our side-channel analysis we will be mainly focused on studying the power consumption leakages generated by the execution of *symmetric* cryptographic algorithms such as AES [7, 34] (Figure 1.2) running on FPGA based computing platform. Nevertheless the major concepts presented in the thesis can be used to tackle other families of cryptographic primitives and classes of hardware systems. Other notable symmetric cryptosystems studied in the literature are DES cipher [28] and RC4 stream cipher [9].

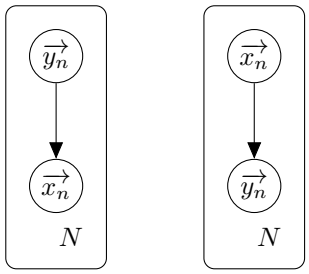

Figure 1.1: Bayesian representation of some generative (left) and predictive (right) distributions. We will usually adopt x and y to denote resp. the features and the target/hidden variable of a given dataset, which is composed of N independent and identically distributed (i.i.d.) data points

Also *asymmetric* encryption algortithms like D&H and RSA have been attacked successfully [29]. From what it concerns the hardware, another common class of computing platform usually tested in the SCA domain is the ATMega familiy of microcontrollers [7, 41]. We will now provide a basic description of AES-128 cipher and its main features.

The AES algorithm is a block cipher belonging to the family of *Substitution-Permutation Networks* which relies on the Shannon's *confusion and diffusion* property. Such systems takes a block of the plaintext and the key as inputs and applies several alternating rounds of substitution boxes (*S-Box*) with some diffusive permutation boxes (*P-Box*) to produce the ciphertext block. AES operates on a 4×4 column-major order array of bytes also called the state, and the relative calculations are done in the characteristic 2 finite Galois field with 256 elements, also referred to as $\mathsf{GF}(2^8)$. In particular, the AES-128 cipher is a 10-rounds algorithm which uses the following primitives for realizing the encryption:

- *SubBytes* is a non-linear substitution step where each byte is replaced with another according to a lookup table. The S-Box used is derived from the multiplicative inverse over $\mathsf{GF}(2^8)$.
- *ShiftRows* is a transposition step where each row of the state is shifted cyclically by an offset. Together with MixColumns, this step provides diffusion in the cipher.
- *MixColumns* is a linear mixing operation which operates on the columns of the state, combining the four bytes in each column. The columns of the state are combined using an invertible linear transformation, which consists in the modular multiplication of two four-term polynomials whose coefficients are elements of $\mathsf{GF}(2^8)$.
- *AddRoundKey* is the operator where the subkey of the next round is computed by the Rijndael key scheduling algorithm, and then further combined with the state.

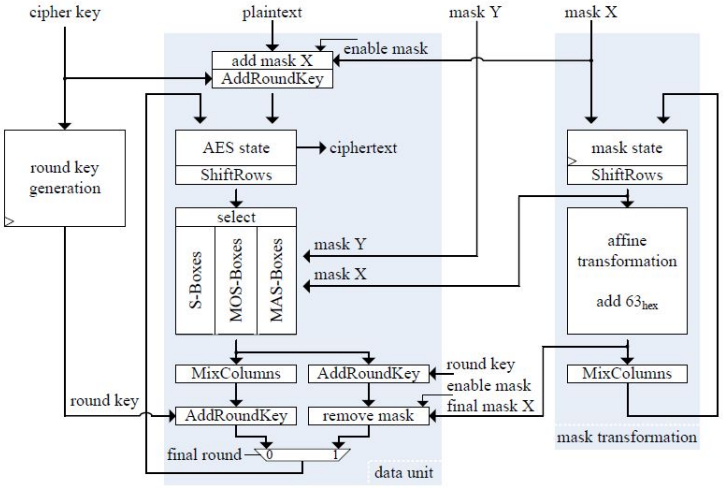

Figure 1.2: Architecture of an AES chip protected with masking method. Image taken from [34]

We will now define the basics assumptions that will be used for our analysis. We will assume that a side-channel leakage trace can be modelled as a realization of a random column vector $\overrightarrow{L}$ of real values. Each instantaneous measure will be denoted with L. During the acquisition of the leakages, a target sensitive variable $Z = \psi(P, K)$ is handled, where P denotes a known plaintext, and K the secret key the attacker aims to retrieve. Therefore the variables P, K, Z can be seen as categorical variables, which are a kind of random variables with a discrete set as support. We will also assume that P, K, represent only a part of the actual variables, typically just the first of the 16 bytes of the word (for AES-128). The techniques presented in the thesis can be easily extended to cover the entire key value using a extend-and-prune strategy [9] or divide-et-impera approaches.

The secret sensitive variable to be guessed is carefully selected by the attacker so that Z actually leaks its secret. This means that Z should target a variable which is K or a function of it, and possibly that Z is correlated with the side-channel leakages. The function $\psi(P, K)$ which generates the target sensitive variable $Z = \psi(P, K)$ is also called *leakage model* or *power model* in the case of power consumption analysis. The simplest form of prediction realizable by the attacker is to directly estimate the value of the key. However, cryptographic systems such as AES usually use the mentioned confusion subsystems such as the S-Box function (Figure 1.3) which can partially neglect the capability of the adversarial to directly target the key. If the attacker chooses to predict the key, the model realizing the malicious estimation could be forced to directly learn the inverse of the S-Box function, which can be a complex task [41].

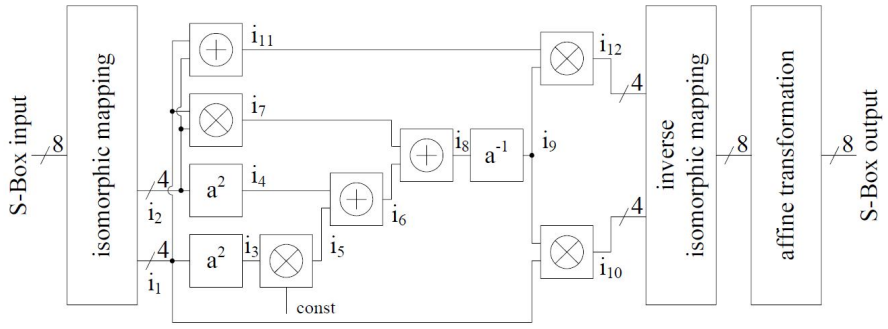

Figure 1.3: Architecture of an unmasked S-BOX implementation. Image courtesy of [34]

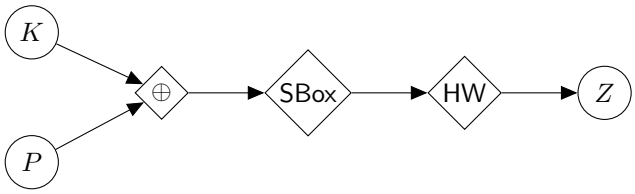

Figure 1.4: Bayesian overview of the selected sensitive variable model

Therefore, we will not use this type of prediction for guessing the secret. Instead, the model used to characterize the power consumption of the AES implementation will be based on the S-Box output (during the first round of the algorithm), so the target sensitive variable Z will be based on it. Moreover, it is possible to further characterize the value of Z by further applying the *Hamming Weight* (HW) function on that word, which is the function counting the number of bits set to 1.

The reason for this is that it can be shown that this quantity have positive correlation with the power consumption for CMOS devices [7, 26, 34, 41, 43]. The HW also reduces the number of values which can be attained by the sensitive variable, thus reducing the efforts needed by the discriminator $\hat{g}$ used for attacking. The selected sensitive variable $Z = \psi(P, K)$ is defined in equations (1.1). It is probabilistically depicted in Figure 1.4.

$$\overrightarrow{s} = \mathsf{SBox}(P \oplus K) \qquad \psi(P, K) = \mathsf{HW}(\overrightarrow{s}) = \sum_{i=1}^{N} \overrightarrow{s}[i] \tag{1.1}$$

Given our definition of ψ, we know that each plain text P and key K are uniform distributed on the first 256 integer values. This implies that each bit of the key is a Bernoulli realization with $p = 0.5$. We also assumed that the HW function is the sum of bits with value 1 of the S-BOX word $\overrightarrow{s}$. Therefore the HW function can be seen as the map from 8 independent Bernoulli realizations (with $p = 0.5$) to the number of positives ones, forcing

the categorically distributed Z to follow a binomial distribution. Therefore in equations (1.2) we define the distributions of the variables P, K and Z involved in the analysis.

$$P \sim \mathcal{U}niform(0, 255) \qquad K \sim \mathcal{U}niform(0, 255) \qquad Z \mid P, K \sim \mathcal{B}inomial(0.5, 8) \quad (1.2)$$

1.1. Non-profiled attacks

Non-profiled SCA extracts the secret key by only exploiting the observations of the attacked device. The main feature of this type of attack is that the adversarial does not need specific prior knowledge of the device used by the victim. Main non-profiled methods can be reported to a unique representation [13], which uses a leakage model M_k which in our case is related to the selected sensitive variable $Z = \psi(p, k)$.

This knowledge is then used to compute some univariate statistics about single measures L inside each random trace $\overrightarrow{L}$. During the attack, the adversarial uses a *distinguisher* function $\Delta_k = \Delta(L, M_k)$ in order to score each key hypothesis. Therefore the choice of the key is made by taking the key which maximizes the value of Δ_k.

The *Differential Power Analysis* (DPA) [28] method involves a distinguisher Δ defined as a Difference of Means (DoM) between two leakage partitions defined according to the image set $\mathsf{Im}(M_k)$. The *Correlation Power Analysis* (CPA) [6] method uses the *Pearson correlation* coefficient defined in equation (1.3) as the distinguishing function Δ of the keys. An example of a succesfull CPA can be visualized in Figure 1.5.

$$\rho(L, M_k) = \frac{\mathsf{Cov}[L, M_k]}{\sqrt{\mathsf{Var}[L]} \cdot \sqrt{\mathsf{Var}[M_k]}} \quad (1.3)$$

The *Mutual Information Analys* (MIA) [19] method uses the *mutual information* defined in equation (1.5) as the distinguisher of the hypothesis. We use $\mathbb{I}$ to denote the mutual information and $\mathbb{H}$ to denote the *Shannon entropy* which is defined in equation (1.4).

$$\mathbb{H}(X) = -\mathbb{E}[\mathsf{log}(X)] \quad (1.4)$$

$$\mathbb{I}(L, M_k) = \mathbb{H}(L) - \mathbb{H}(L \mid M_k) \quad (1.5)$$

All the methods described so far can be generalized to a *higher order* analysis [18, 26, 36] which exploits a multivariate characterization of the leakage to attack the crpytosystem.

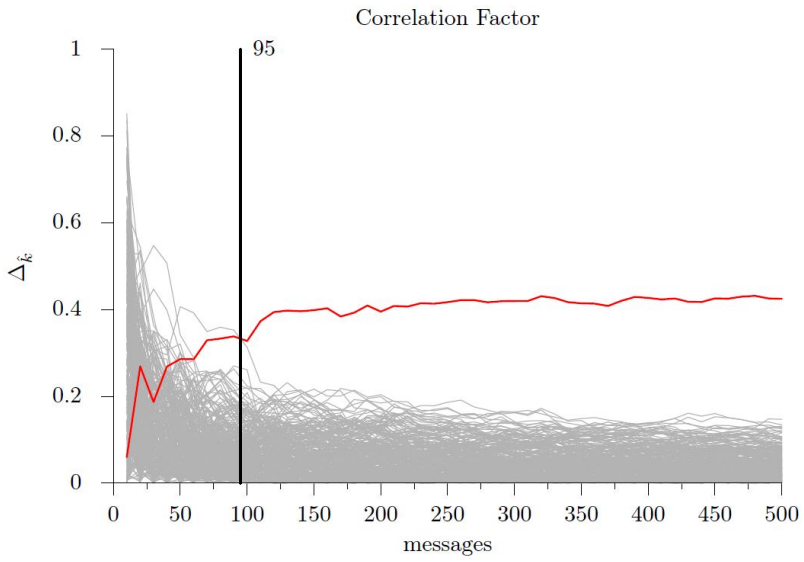

Figure 1.5: Evolution of the CPA distinguishing values for all key candidates. The curve of the real key used in the device is plotted in red. The attack is successfull after observing 95 power traces. Image taken from [13]

The main disadvantage of these kind of attacks is that the adversarial could be forced to collect a great number of power traces to succeed in retrieving the key, and this could be difficult for some real scenarios. This is the main reason which led to creation of more sophisticated attacks which exploit a prior knowledge of the attacked computing platform.

1.2. Classical profiled attacks

A particular class of side-channel attacks is called profiled and can be executed when the SCA adversary can use an open device identical to the attacked one, in which he can control all the inputs of the algorithm (including the key). Profiled attacks are able, in principle, to defeat any of the countermeasures devised for non-profiled attacks, such as masking and hiding, provided that enough measurements are collected [9].

Some of these attacks are based on univariate statistics and stochastic models [13, 43], while other tecnhniques use a combination of preprocessing followed by a classification of the secret variable. This is the case of the famous bayesian *Templates Attack* (TA) [9] which uses preprocessing procedures based on *Signal to Noise Ratio* (SNR) [49] or T-Test to select a small number of informative Points Of Interest (PoI) of a trace. The dimensional reduced data is then transformed by a *Quadratic Discriminant Analysis* (QDA) [15] classifier to infer the target sensitive variable.

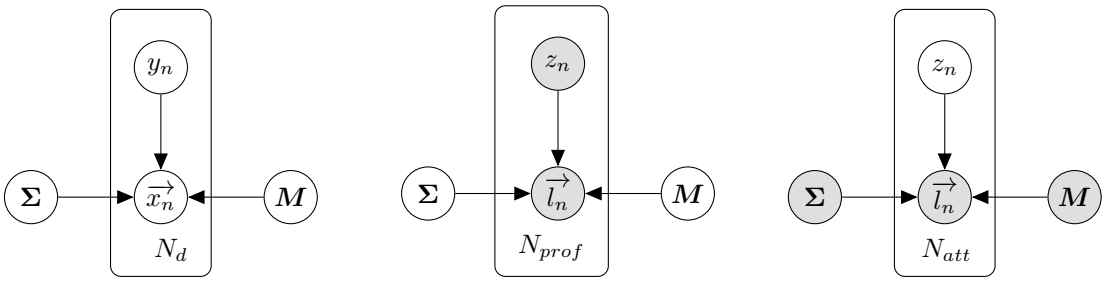

Figure 1.6: Bayesian network of QDA (left), estimation of templates on profiling traces (middle), using template to infer sensitive value from attack traces (right)

Other strategies used in the literature for dimensionality reduction in TA are based on *Principal Component Analysis* (PCA) and *Linear Discriminant Analysis* (LDA) [3, 15, 38, 47] or based on non-linear models such as *Self Organizing Maps* (SOM) [30]. Other types of classifier typically used for profiled SCA are based on *Kernel Discriminant Analysis* (KDA) [8] *Support Vector Machines* and *Random Forest* [4, 12, 30, 31, 42].

More recently, deep learning and artificial neural network have emerged in the SCA community as a powerful model for side-channel attacks. Lot of attacks have been successfully achieved by using feed-forward neural networks such as the *Multi-Layer Perceptron* (MLP) and *Convolutional Neural Networks* (CNN) [7, 16, 20, 33, 41].

1.2.1. Basyesian Template Attack

One of the most successful profiled attack is the well known gaussian *Template Attack* [9] a.k.a. QDA model. The main assumption of QDA is that the features $\overrightarrow{X}$ are a mixture, with Y as components, of some multivariate normal distributions. For our SCA problem (where $X = \overrightarrow{L}$ and $Y = Z$) the QDA model assumes that the generative distribution $\hat{f}$ of the trace $\overrightarrow{L}$ given the latent variable Z is a multivariate gaussian (Figure 1.6).

The variables $\boldsymbol{M}$ and Σ are $|\mathcal{Z}|$ mean vectors and covariances, one for each value of the hidden variable. We are not interested in the full distributions of each $\boldsymbol{\Sigma}[y]$ and $\boldsymbol{M}[y]$ because they will be estimated by *Maximum Likelihood Estimation* (MLE). As stated in equation (1.1), the sensitive variable can be seen as a binomial distributed categorical variable. Given Z the distribution of $\overrightarrow{L}$ is defined in equation (1.6).

$$\overrightarrow{L} \mid Z \sim \mathcal{N}ormal(\boldsymbol{M}[z], \boldsymbol{\Sigma}[z]) \tag{1.6}$$

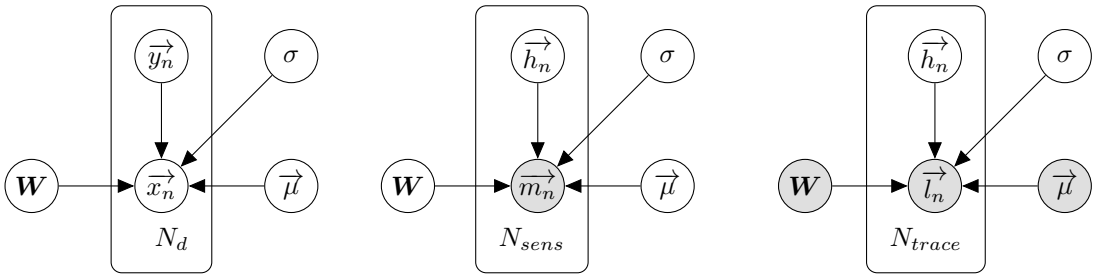

Figure 1.7: Bayesian network of probabilistic PCA (left), estimation of parameters for mean traces of each sensitive variable (middle), transform of some profiling/attack traces using fitted model (right)

The attacker can then observe some realizations $\overrightarrow{l_i}$ of the attacked device, and use Bayes theorem to compute the predictive distribution $\hat{g}$ with equation (1.7) by using generative distribution $\hat{f}$, which is given by equation (1.6).

$$P(Z = z_i|\overrightarrow{L} = \overrightarrow{l_i}) = \frac{P(\overrightarrow{L} = \overrightarrow{l_i}|Z = z_i)P(Z = z_i)}{\sum_{j=1}^{|\mathcal{Z}|} P(\overrightarrow{L} = \overrightarrow{l_i}|Z = z_j)P(Z = z_j)} \tag{1.7}$$

The information of the prediction is defined in equation (1.8), and it is proportional to the sum of the log likelihood of the trace $\overrightarrow{L}$ and the prior log probability of the sensitive variable Z.

$$\mathsf{log}(P(Z = z_i|\overrightarrow{L} = \overrightarrow{l_i})) \propto \mathsf{log}(P(\overrightarrow{L} = \overrightarrow{l_i}|Z = z_i)) + \mathsf{log}(P(Z = z_i)) \tag{1.8}$$

1.2.2. Principal Component Template Attack

High dimensionality of input data can be a problem for solving prediction tasks. In our context the dimensionality of the random side-channel trace is approx. the number of clock cycles needed to complete a full execution of the cryptosystem. For a typical cryptographic implementation $\overrightarrow{L}$ is usually a vector with dimensionality in the order of 10^5 time steps. In this situation classical classifiers can take long times to compute a solution, so it is useful to add a preprocessing step in the pipeline before feeding the data in the classification process. In this stage a given data point $\overrightarrow{x_i} \in \mathbb{N}^\alpha$ is transformed into a lower-dimensional vector $\overrightarrow{y_i} \in \mathbb{N}^\beta$ where $\beta < \alpha$. One of the most commonly used dimensionality reduction technique is the PCA method. It computes the orthonormal basis for the features which minimizes the explained variance of the projected dataset. For our SCA problem we have that $\overrightarrow{X} = \overrightarrow{L}$ and $\overrightarrow{Y} = \overrightarrow{H}$. The PCA model is probabilistically related to an *Indipendent Factor Analysis* (IFA) representation [5, 20] described in equations (1.9) and

(1.10). It is assumed that the latent vector $\boldsymbol{H}$ generates the original trace $\boldsymbol{L}$ by a linear transformation plus some random noise $\overrightarrow{N}$. The latent variable $\overrightarrow{H}$ is assumed to be a standardized multivariate gaussian. The $\overrightarrow{L}$ is assumed to be normally distributed with mean and covariance related to $\boldsymbol{W}$, $\overrightarrow{\mu}$ and σ, and which will be estimated by MLE.

$$\overrightarrow{L} = \boldsymbol{W}\overrightarrow{H} + \overrightarrow{\mu} + \overrightarrow{N} \qquad \overrightarrow{H} \sim \mathcal{N}ormal(0, \boldsymbol{I}) \qquad \overrightarrow{N} \sim \mathcal{N}ormal(0, \sigma^2\boldsymbol{I}) \tag{1.9}$$

$$\overrightarrow{L} \mid \overrightarrow{H} \sim \mathcal{N}ormal(\boldsymbol{W}\overrightarrow{H} + \overrightarrow{\mu}, \sigma^2\boldsymbol{I}) \tag{1.10}$$

The classical PCA transformation, related to the MLE solution of the probabilistic model, is defined in equation (1.11). As described in equation (1.12), the value of $\boldsymbol{W}$ is found by computing the eigendecomposition of the covariance matrix of the dataset $\boldsymbol{L}$.

$$\overrightarrow{H} = \boldsymbol{W}^T(\overrightarrow{L} - \overrightarrow{\mu}) \tag{1.11}$$

$$\boldsymbol{C} = \frac{1}{D-1}\boldsymbol{L}^T\boldsymbol{L} \qquad \boldsymbol{\Lambda} = \boldsymbol{W}^{-1}\boldsymbol{C}\boldsymbol{W} \tag{1.12}$$

We will use a variant of classic PCA [3, 47], in which we fit the model on the centroids of the clusters in the data (the mean trace $\overrightarrow{M}$ for each sensitive variable) instead on plain observed data points. This is done to efficiently compute the PCA fit method without loading all the data in memory, otherwise the high dimensionality of L would make the task infeasible. One may question the coherence of the method, because the solution found by PCA makes sense if the intrinsic dimension of the data manifold is much lower than number of observations. Fortunately, this is the case for power analysis SCA [3].

1.3. Countermeasures

Most effective ways to prevent classical attacks are based on implementing some software or hardware countermeasures. Most common software protections are based on introducing data randomization and masking [8, 11, 33, 34], which is basically adding a random value to intermediate variables. Typical hardware techniques are instead based on the introduction of *process variability* between the profiling and the attacked devices [10]. In this way, the assumption of the adversarial having full knowledge of the target environment is weakened, and the resulting attack is more difficult to be achieved. For example a static process variability can happen when the two devices use different versions of the same chip. Dynamic process variability can instead happen in *Random Dynamic Voltage and Frequency Scaling* (RDVFS) actuated systems [51].

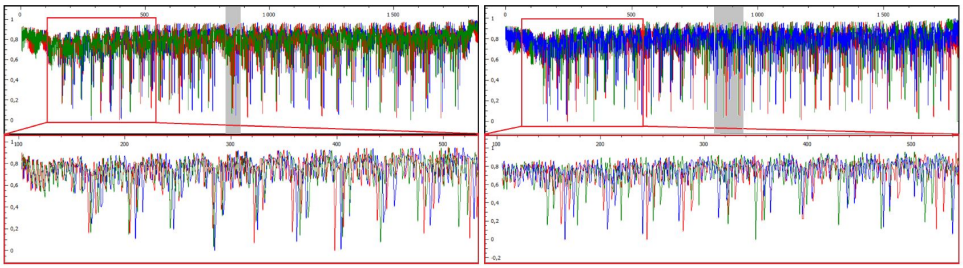

Figure 1.8: Left: some leakages with low clock jittering, a zoom of the part highlighted by the red rectangle is given in the bottom part. Right: some traces with high clock jittering. The interesting clock cycle is highlighted by the grey rectangular area. Image courtesy of [7].

The *temporal alignment* assumption ensures that the same operations across different measurements always happens at the same time. When this assumption is not satisfied we say that the side-channel traces are *desynchronized*. When this happens, most commons classical attacks such as DPA, CPA, and TA fails in successfully guessing the secret [41]. Also linear models for dimensionality reduction such as PCA and LDA needs the temporal alignment of traces in order to effectively do their job.

Desynchronizations and data augmentation

In the thesis we will be focused on process variability methods which exploits desynchronizations as a countermeasure. This property have been shown to be critical for a wide variety of side-channels and cryptographic implementations [7, 41, 51]. Some of the techniques used for this purpose are:

- *random delays*: it is obtained by randomly translating forward in time the trace. Usually it is implemented in software with the insertion of some random *nop* instructions in the program
- *clock jittering*: it is the random deviation from the true periodicity of the clock signal. While jitter may also induce electromagnetic interference and crosstalk issues in CMOS devices, we are specifically interested in its induced temporal misalignment effect.
- *frequency variations*: it is the process of dynamically scaling the operative frequency of the device at runtime. For every switch a frequency is picked from a fixed finite set and it is used by the RDFS module as the new operative clock frequency.

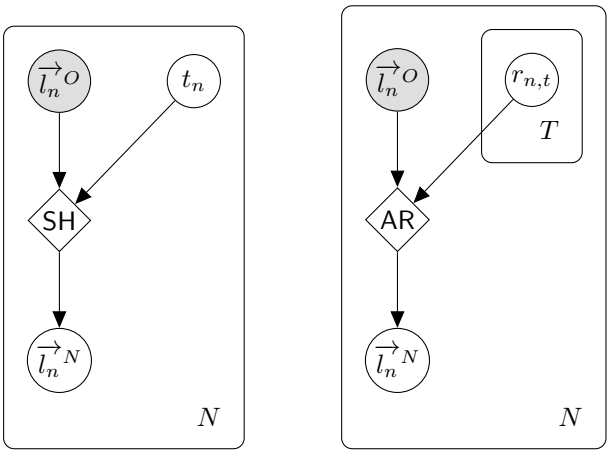

Figure 1.9: Probailistic overview of the generation of augmented power traces using: shifting deformation for random delays (left), add-remove deformation for clock jittering (right)

As described in [7, 41] there is a strict connection between desynchronized cryptographic implementations and *Data Augmentation* (DA) process, which is a well known ML method used to augment the number of data points in a dataset, which actually also showed to often improve generalization capabilities of the model [20].

We now provide a description of the generative process of the DA methods for process variability commonly used in the SCA literature, which are the random delays and clock jittering processes. These methods can be applied to generate desynchronized augmented side-channel traces from observations of unprotected implementations. For every leakage observation $\overrightarrow{l_n}^O : 1 \leq n \leq N$ we generate the augmented trace $\overrightarrow{l_n}^N$ by applying some deformations to $\overrightarrow{l_n}^O$:

- *random delays* are computed by sampling, with a given distribution, a time duration which is used to shift the trace for t_n time steps. The shifting operator is denoted with the SH function
- *clock jittering* is obtained by sampling a fixed amount of T random points $r_{n,t}$ of the trace. The points are then replaced by the add-remove AR function with some augmented points, which are typically an average of previous

The described operators can be composed, for example it is possible to compute an augmented delayed trace with clock jittering by using the function $\mathsf{AR} \circ \mathsf{SH}$. In chapter 2 we will define our method for the generation of leakage traces with frequency variations, following a similar approach of the previously discussed augmentation procedures.

1.4. Deep profiled attacks

Recent line of works have studied the role of *Artificial Neural Networks* (ANN) and *Deep Learning* (DL) based systems in the SCA context. Usually feed-forward ANN are used in profiled SCA settings, where they assume the role of the discriminator $\hat{g}$ which classifies the key value from the side-channel traces (Figure 1.10). We will refer to this setting as *fully deep* SCA, in order to distinguish it from another DL based technique which we will call *deep aligned* SCA. One of the most significant result showed by neural networks is the ability to easily handle protected implementations of the cryptosystem. Countermeasures such as masking and desynchronizations have been practically broken using *Multi Layer Perceptrons* (MLP) and *Convolutional Neural Networks* (CNN) [7, 33]. Further analysis have been developed on the optimization of learning and neural architectural hyper-parameters for SCA problems [16, 41]. Most recent studies have showed that the intrinsic characterization of profiled attack (i.e. the presence of a profiling and an attacked device) can be expressed in Deep Learning as a *Domain Adaptation* problem [20], leading to a state of the art framework for cross-device SCA [40].

1.4.1. Multi Layer Perceptron

Multi layer feed-forward networks are models which are proven to be *universal function approximators* if given enough resources [23]. They can learn a task using a hierarchical representation of data which minimize a given *loss* function $\mathcal{L}$ engineered for a given problem (e.g. regression or classification). They achieve this by learning some *weights* parameters $\boldsymbol{W}$, which are a simplified formalization of the interconnections of real neurons, and some *bias* $\overrightarrow{b}$. Each forward computation of the network (for a vector $\overrightarrow{x}$) is made by a sequential activation of n *Fully Connected* (FC) layers λ_i, where each neuron of each layer compute a weighted sum of its input plus the bias. On top of the weighted sum it is applied an activation function f_i, that is typically fixed for every neurons in a layer.

$$\lambda_i(\overrightarrow{x}) = f_i(\boldsymbol{W}_i \overrightarrow{x} + \overrightarrow{b}_i) \qquad \hat{g}(\overrightarrow{x}) = (\lambda_n \circ \lambda_{n-1} \dots \lambda_1 \circ \lambda_0)(\overrightarrow{x}) \tag{1.13}$$

Most commonly used activation functions are defined in equations (1.14). The last layer of the network plays a special role because define the output of the model, so its activation function is task dependent. For problem like regression, a linear layer (identity activation) or other surjective functions can be chosen as the final activation. Because fully deep SCA can be expressed as classification problems, the choice of the last activation f_n is the SoftMax function. For the hidden layers, common choices for activation functions

are the Logistic, Tanh and ReLU functions. There exists some extensions of the ReLU activation which are the Leaky ReLU and SeLU activations, however such functions will not be explored in much details for our thesis.

$$\begin{aligned} \mathsf{SoftMax}(\overrightarrow{x})[j] &= \frac{e^{\overrightarrow{x}[j]}}{\sum_i e^{\overrightarrow{x}[i]}} & \mathsf{Logistic}(\overrightarrow{x})[j] &= \frac{1}{1+e^{-\overrightarrow{x}[j]}} \\ \mathsf{ReLU}(\overrightarrow{x})[j] &= \max(0, \overrightarrow{x}[j]) & \mathsf{Tanh}(\overrightarrow{x})[j] &= \frac{e^{\overrightarrow{x}[j]} - e^{-\overrightarrow{x}[j]}}{e^{\overrightarrow{x}[j]} + e^{-\overrightarrow{x}[j]}} \end{aligned} \tag{1.14}$$

To learn the values of the weights, the *Maximum A Posteriori* (MAP) based learning of equation (1.15) is computed for each mini-batch $\boldsymbol{D}$ of data points. In the equation the last term is a *regularization* term, which basically defines a prior distribution for $\boldsymbol{W}$. Typical regularizations used are l^1 and l^2 regularization also called *lasso* and *ridge* regularizations, which are realized by adding to the loss an extra term made of (resp.) norm and squared norm of the weights. For our experiments we will primarily use other forms of regularization. The *batch normalization* [24] layer consists of normalizing batch-wise the output of a given layer, and it is often used in covolutional networks. The *dropout* [46] layer consists in randomly switching off some neurons during learning phase. Using dropout, neurons in the layer are forced to learn indipendent representations. It has been showed that the dropout method leads the model to learn an ensemble-like representation.

$$\underset{\boldsymbol{W}}{\mathsf{argmax}}\ \mathsf{log}(P(\boldsymbol{W}|\overrightarrow{X}, Y)) = \underset{\boldsymbol{W}}{\mathsf{argmax}}\ \mathsf{log}(P(Y|\overrightarrow{X}, \boldsymbol{W})) + \mathsf{log}(P(\boldsymbol{W})) \tag{1.15}$$

As visually depicted in Figure 1.10, for a *fully deep attack* we use a neural network as a classifier for our SCA problem, where we use the leakage trace $\overrightarrow{L}$ as the features $\overrightarrow{X}$ and the sensitive variable Z as the target variable Y of the classification problem.

With a softmax output layer, the model $\hat{g}$ can be seen as the estimation of the posterior distribution $P(Y|\overrightarrow{X}, \boldsymbol{W})$. In such a scenario, in order to learn from data the weigths $\boldsymbol{W}_i$ and biases $\overrightarrow{b}_i$ of each layer λ_i, the optimizer tries to iteratively minimize (over a mini-batch $\boldsymbol{D}$ of samples) the multi-label classification loss of equation (1.17), which is based on the Shannon *cross-entropy* function defined in equation (1.16).

$$\mathbb{H}(Y, \hat{g}) = -\mathbb{E}_Y[\mathsf{log}(\hat{g})] \tag{1.16}$$

$$\mathcal{L}_{\hat{g}}(\boldsymbol{D}) = \sum_{\overrightarrow{x_i}, y_i \in \boldsymbol{D}} \mathbb{H}(y_i, \hat{g}(\overrightarrow{x_i})) = -\sum_{i=0}^{|\boldsymbol{D}|} y_i\ \mathsf{log}(\hat{g}(\overrightarrow{x_i})) \tag{1.17}$$

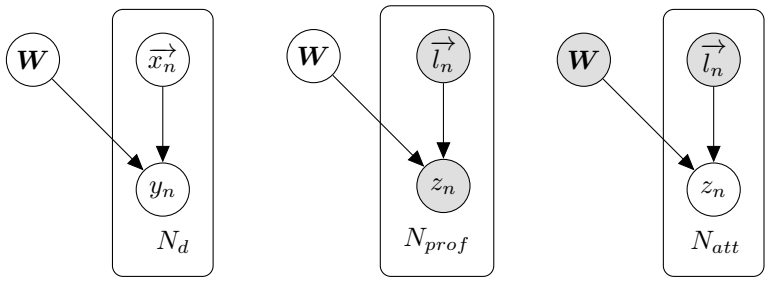

Figure 1.10: Bayesian network of the MLP classifier (left), fully deep profiling (middle) and respective sensitive variable prediction (right)

As defined in equations (1.18), minimizing the cross-entropy function is equivalent of minimizing the *Kullback-Leibler divergence* $\mathcal{KL}$ from the distribution of the data $\overrightarrow{X}$ to the distribution of the model $\hat{g}$, because the crossentropy is defined as the sum of the $\mathcal{KL}$ divergence itself plus the entropy of the data, which is not function of $\boldsymbol{W}$.

$$\mathbb{H}(Y, \hat{g}) = \mathbb{H}(Y) + \mathcal{KL}(Y \mid\mid \hat{g}) \qquad \underset{\hat{g}}{\operatorname{argmin}}\, \mathbb{H}(Y, \hat{g}) = \underset{\hat{g}}{\operatorname{argmin}}\, \mathcal{KL}(Y \mid\mid \hat{g}) \tag{1.18}$$

To realize the optimization procedure, algorithms based on Lagrange multipliers can be used, but usually the problem can be solved by *backpropagation* algorithm which uses numerical procedures based on gradient descent, such as Stochastic Gradient Descent (SGD) and Adam [27]. At each mini-batch iteration the gradient of the loss is computed by analyzing the architecture of the network. This is possible because ∇ can be evaluated backward from the last layer, by exploiting the chain rule of the derivative. Finally the *hebbian learning* update at each iteration t can be computed with equation (1.19).

$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^{t} - \eta \frac{\partial \mathcal{L}_{\hat{g}}}{\partial \boldsymbol{W}}\Big|_{\boldsymbol{W}^t} \tag{1.19}$$

where η is an amplification term for the step size of the gradient descent update which is called *learning rate*, and which must be hyper-optimized for each specific task.

1.4.2. Convolutional Networks

Convolutional Neural Newtorks (CNN) are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals (like SCA traces), and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications.

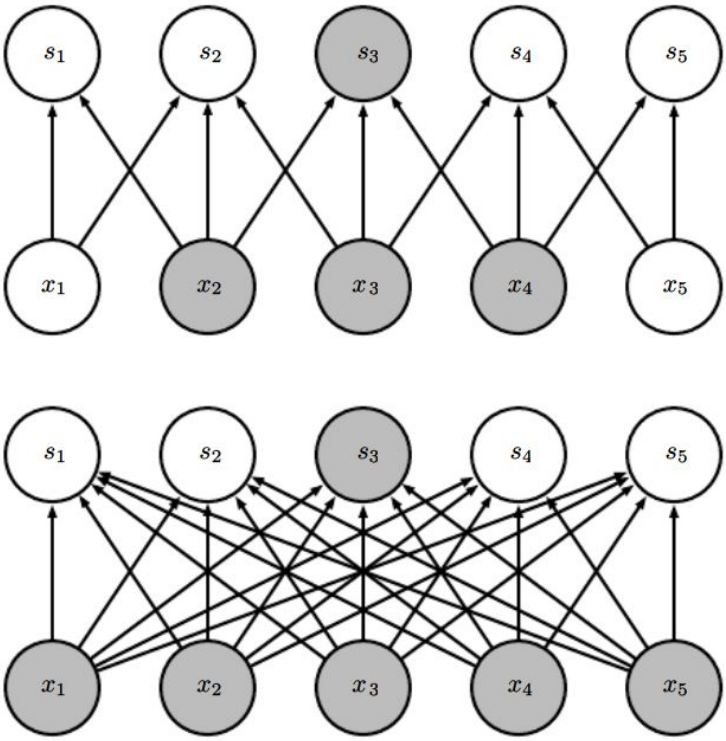

Figure 1.11: Visualization of the connectivity of CNNs (top) vs MLP (down), highlighting the perceptive field of one neuron in the final layer. Image taken from [20].

In particular, these models have been showed to be very effective in the context of computer vision by winning lot of image classification challenges such as ImageNet and ILSVRC competitions. The discrete convolution operator used by CNN, closely related to *cross-correlation*, is defined (for a 2D input) in equation (1.20), where $\boldsymbol{X}$ are the 2D features and $\boldsymbol{K}$ is the *kernel*, which is one of the 2D filters learned by the network..

$$\mathsf{Conv}(\boldsymbol{X}, \boldsymbol{K})[i, j] = \sum_m \sum_n \boldsymbol{X}[m, n] \cdot \boldsymbol{K}[i - m, j - n] \tag{1.20}$$

The same definition can be easily generalized to an input of any shape, for example in the SCA context where we will deal with 1D input power traces instead of 2D images. Convolutional networks can be simply formalized as neural networks that use convolution in place of general matrix multiplication in at least one of their layers. There are 3 main main advantages [20] of convolutional layers instead of fully connected:

- *sparse connectivity* is accomplished by making the kernel smaller than the input. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that

computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication used by FC layers requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime complexity. If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime complexity. For many practical applications, it is possible to obtain very good performance on the ML task while keeping k several orders of magnitude smaller than m

- *parameter sharing* refers to using the same parameter for more than one function in a model. In a traditional neural network each element of the weight matrix is used exactly once when computing the output of a layer. In a convolutional neural network each element of the kernel is instead used at every position of the input. This means that rather than learning a separate set of parameters for every location, we learn only one set used for the entire input.

- *equivariant representations* refers to the property for which if the input changes then the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g(x)$ if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input then the convolution function is equivariant to g. Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

A typical structured layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the ReLU function. This stage is sometimes called the *detector* stage.

In the third stage, we use a *pooling layer* to modify the output of the layer further. A pooling function replaces the output of one layer with a summary statistic of the nearby outputs. For example, the *max pooling* [53] is one of the most used operation which reports the maximum output within a rectangular neighborhood. In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

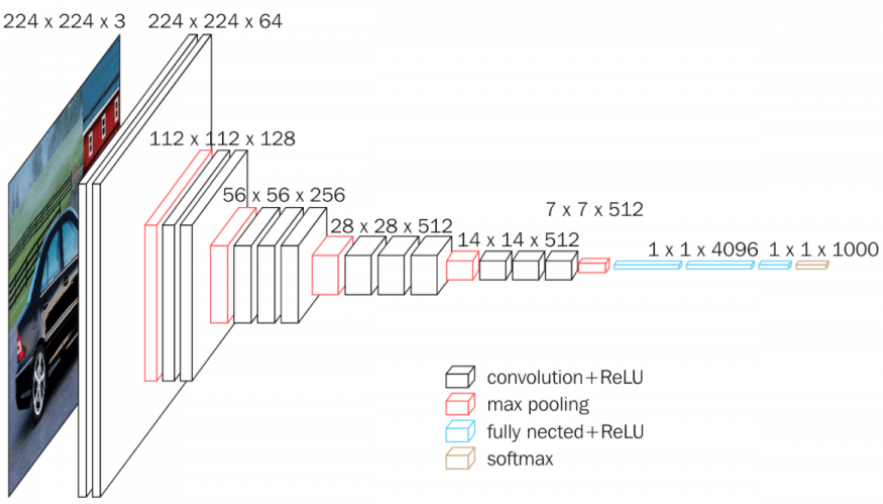

Figure 1.12: Network architecture of VGG-16 model for ImageNet competition. Image courtesy of [45].

Other popular pooling functions are the l^2 norm of a rectangular neighborhood and the *average pooling* layer. As we will see later the average pooling will play a central role for some localization techniques, when used at the end of the network encoder. In this setting the pooling layer will be called *Global Averaging Pooling* (GAP). This layer can also be a useful architectural choice for the model, because it gives the network the capability to process inputs of variable size.

VGG

One of the most important deep convolutional architecture is represented by the *VGG* [45] model. Historically this network achieved a second place in the ImageNet-2014 image classification competition, outperforming previous state of the art convolutional architectures. With respect to its predecessors, this architecture is deeper and have a smaller size of convolutive filters. The number of filters are recursively doubled with the increasing depth. The VGG network can thus learn a representation of the input which is increasingly smaller in space and with an increasingly higher number of channels.

The most commonly used architecture for VGG based model is depicted in figure 1.12. It is realized by stacking a repetition of basic blocks, which are made of 2 or 3 repetitions of convolution + ReLU with a final max pooling. First a given number of stacked 2-conv blocks are used, and then some 3-conv stacked blocks are appended to obtain the final representation. At the end of the convolutional encoder, which acts as a feature extractor, there is a GAP or flatten layer followed by a standard MLP classifier.

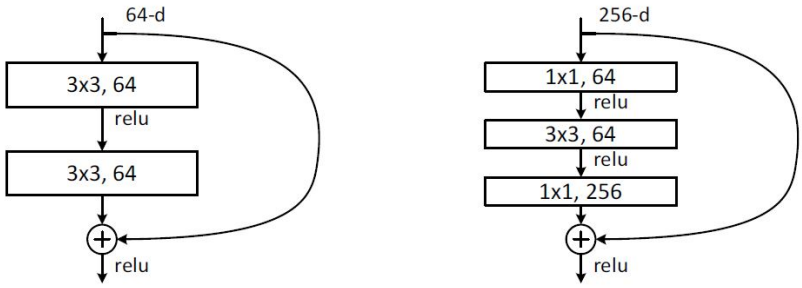

Figure 1.13: Deeper residual function for ImageNet. A building block (on 56×56 feature maps) for ResNet- 34 (left). A "bottleneck" building block for ResNet-50/101/152 (right). Image taken from [22]

ResNet

It have been observed that when deeper networks are able to start converging, a degradation problem shows up: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error. To circumvent this situation the neural network known as *ResNet* [22] was born. The main feature of this neural architecture is the introduction of the *residual block* $\mathcal{H}$, described in equation (1.21), which exploits some *skip connection* for computing a residual identity mapping of the input.

$$\mathcal{H}_{\boldsymbol{W}}(\boldsymbol{X}) = \mathcal{F}_{\boldsymbol{W}}(\boldsymbol{X}) + \boldsymbol{X} \tag{1.21}$$

This simple intution showed to have the capability of making the gradient to flow more efficiently backward in the network during learning phase. Residual based models outperformed previous architectures when applied to the CIFAR-10 and ImageNet-2012 image classification datasets.

The inner function $\mathcal{F}$ of the residual network is a convolutional based block, with usually 2 or 3 convolutions stacked (Figure 1.13) depending on some specific configurations. The full network is made by stacking multiple residual blocks while doubling the amount of filters with the depth of the network. Also max pooling layers are used between inner residual blocks. In order to better comprehend the role of residual models for our SCA problem, the hyper-parameters for this architecture will include the number of residual blocks needed to double the filters and to downsample the input via max pooling.

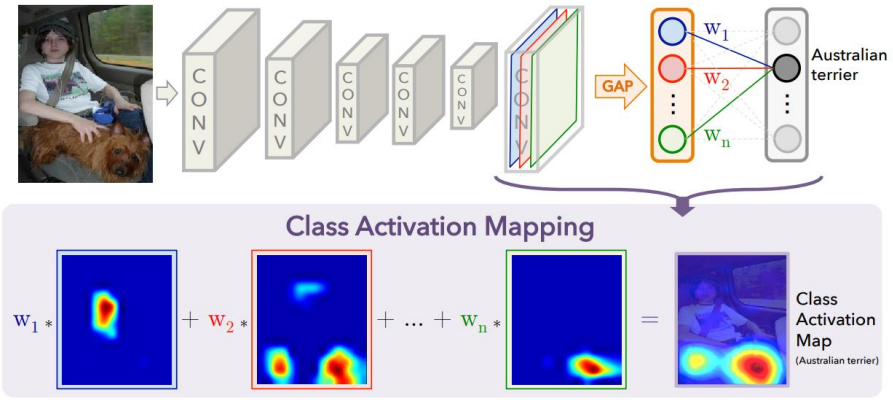

Figure 1.14: Visualization of CAM method for 2D input image, which exploits the weights of the softmax as an importance of each feature map. Image courtesy of [52]

1.4.3. Alignment through localization

Past works have studied the effects of detection based techniques of neural networks for SCA problems [16, 32]. We will now introduce the concepts related to *weakly supervised localization* that will be used for the *deep aligned* SCA method. We will use these techniques to build a localization component which detects where and how a given protection is showing up in some observations. Using this knowledge it is possible to bring the traces to an unprotected state, in order to attack them with other profiled methods (e.g. TA).

After the innovation of CNN, it have been noticed that neural units of convolutional layers actually behave as object detectors despite no supervision on the location of the object was provided. Despite having this remarkable ability to localize objects in the convolutional layers, this ability is lost when fully-connected layers are used for classification. One possible way to remove the FC problem is to use a *fully convolutional* network, which is basically replacing the FC layer with a 1×1 convolution. However there is an even more straighforward solution to this problem, that is to constrain the network to have a final GAP layer at the end of the encoder. In this way the classifier learns to predict the target variable basing on a spatial average of features described in the final maps of the encoder.

Class Activation Mapping

Using a network with final GAP layer, it is possible to build the final FC network with specific architectures. After being trained for classification, we can use the learned representation of the model to compute the internal activation of the network w.r.t. a particular class. This method is called *Class Activation Map* (CAM) [52], and it computes some importance spatial maps w.r.t some labels. The maps can be further used for segmentation.

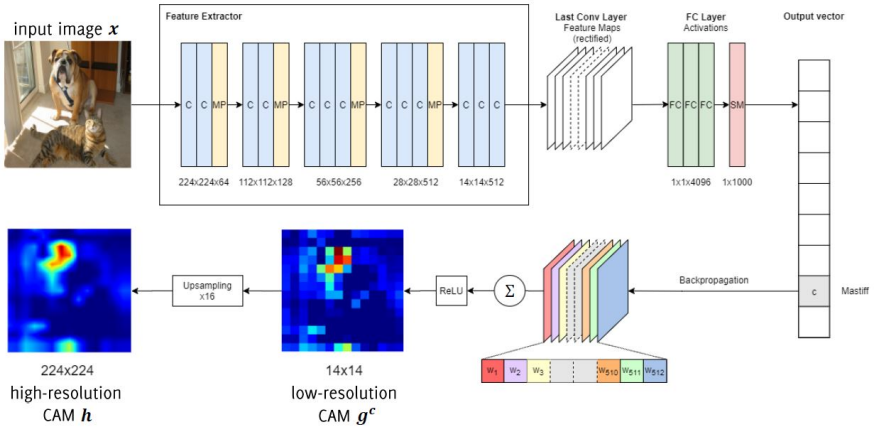

Figure 1.15: Visualization of Grad-CAM method for a 2D image, which uses the gradient as an importance of each feature map. Image taken from [44]

In this setting, the class activation maps are based on the feature maps encoded at the input of the final GAP layer (Figure 1.14). The method basically consists of using a classifier with no hidden layers, which allows to aggregate all the encoded maps by weighted sum. As described in equation (1.22), the activation maps $\boldsymbol{A}$ are computed for each class y, by summing each encoded map $\boldsymbol{F}[i]$ proportionally to its contribution $\boldsymbol{W}[y, i]$ w.r.t to label y.

$$\boldsymbol{A}[y] = \sum_i \boldsymbol{W}[y, i]\, \boldsymbol{F}[i] \tag{1.22}$$

However this method restricts the classifier network to be a simple softmax layer without any hidden layer. In general it could be the case that the final FC network actually needs more intermediate layers to effectively solve a generic problem. So we decided to extend our analysis to more general techniques of weakly supervised localization.

Gradient weighted Class Activation Mapping

As previously discussed, a drawback of basic CAM method is that it requires feature maps to directly precede softmax layer, so it is only applicable to a particular kind of CNN architectures performing GAP over convolutional maps immediately prior to prediction. Such architectures may achieve inferior accuracies compared to general networks, or may simply be inapplicable for other tasks (e.g. image captioning or visual question answering). We will now describe a further method called *Gradient based Class Activation Mapping* (Grad-CAM) [44] which combines feature maps using the gradient signal, and which does not require any modification for the network architecture (Figure 1.15).

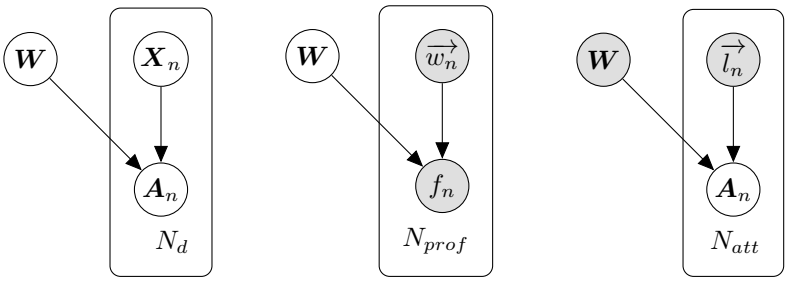

Figure 1.16: Bayesian network for CAM (left), fit of window frequency classifier on profiling data (middle) and CAM prediction on protected attack traces (right)

This allows the method to be applied to off-the-shelf CNN-based architectures. For a fully-convolutional architecture, CAM is a special case of Grad-CAM. Given a 2D input $\boldsymbol{X}$ (the same formalization can be extended for 1D) and corresponding class prediction $\hat{y} = \hat{g}(\boldsymbol{X})$, for each encoded map $\boldsymbol{F}[i]$ its importance $\overrightarrow{\alpha}[y, i]$ for class y is computed with equation (1.23) as proportional to the gradient of the predictor w.r.t average pooled map.

$$\overrightarrow{\alpha}[y, i] = \frac{1}{Z} \sum_{j} \sum_{k} \frac{\partial \hat{y}}{\partial \boldsymbol{F}[i, j, k]} \tag{1.23}$$

Where Z is a normalization constant, while the sums over j, k represent the GAP at the end of the encoder. The class activation map $\boldsymbol{A}[y]$ for input $\boldsymbol{X}$ and class y can then be computed with equation (1.24), by applying a ReLU activation on the importance-weighted sum of each feature map $\boldsymbol{F}[i]$.

$$\boldsymbol{A}[y] = \mathsf{ReLU}(\sum_{i} \overrightarrow{\alpha}[y, i]\, \boldsymbol{F}[i]) \tag{1.24}$$

The maps $\boldsymbol{A}$ so far computed have a smaller spatial size w.r.t to original input map $\boldsymbol{X}$, in fact they can be considered as coarse activation maps for the input. In order to align $\boldsymbol{A}$ with the size of $\boldsymbol{X}$, a linear interpolation is further used for upsampling purpose. Finally, by using all activation maps it is possible to compute a *weakly supervised segmentation* of the input w.r.t. class semantics. To achieve this we can further refine the maps $\boldsymbol{A}$ by using equation (1.25).

$$\boldsymbol{S}[j, k] = \underset{y_i}{\mathsf{argmax}}\, \boldsymbol{A}[y_i, j, k] \tag{1.25}$$

In Figure 1.16 the CAM method is probabilistically described, with the corresponding SCA model representing a frequency scaling detector. This subsystem will be used in our deep aligned SCA method for attacking RDFS protected implementations.

2 | Method

For our SCA experiments we will target the power consumption leakages generated by a given AES-128 cipher implementation for the Xilinx Artix-7 FPGA device. The target hardware architecture is the LAMP SoC [2], which is a general purpose computing platform conceived to perform side channel analysis on FPGA. The device includes a 32-bit in-order RISC-V CPU, a 32 bit Wishbone bus and a 64KB BRAM-based main memory. The attacked system has been built using SystemVerilog language, and the synthesis has been carried out using Xilinx Vivado 2020.2 hardware design suite. The FPGA is placed on the capture board which is the NewAE ChipWhisperer305 Artix-7 FPGA Target (CW305) [1]. Our attacking processing platform is a computing cluster of CPUs and GPUs located in the AIRLab department. The application is mainly written in Python, and we will also use NumPy [21], SciPy [50], scikit-learn [39] and PyTorch [37] libraries for handling and scaling the computation of large amounts of data, and for executing ML and DL tasks. Also JobLib software [25] will be used to implement some multiprocessed components, in order to increase the pipeline throughput when needed. Deep learning components of the application are realized using the PyTorch Lightning [14] software, which provides more abstractions to the deep learning computing platform w.r.t. PyTorch. The overall pipeline is composed of multiple stateless services, and we will use Docker [35] to virtualize our application on the available computing platform. We now describe the main functional interface provided by the data processing platform:

- The data processing pipeline will deal with multi-dimensional arrays (e.g. side-channel traces and datasets) which could represent vectors, matrices or tensors as described in chapter 1
- Arrays can be accessed by the shape function, which returns an array of dimensionalities, one for each of the array axis, while the functions len and size returns resp. the first and last elemet of the array shape
- It is possible to apply boolean operators to a given array (e.g. $\overrightarrow{x}_{where} \leftarrow \overrightarrow{x} < 3.14$) in order to obtain a boolean array of the same shape of the input, with each element equals to true if the element satisfy the logical expression

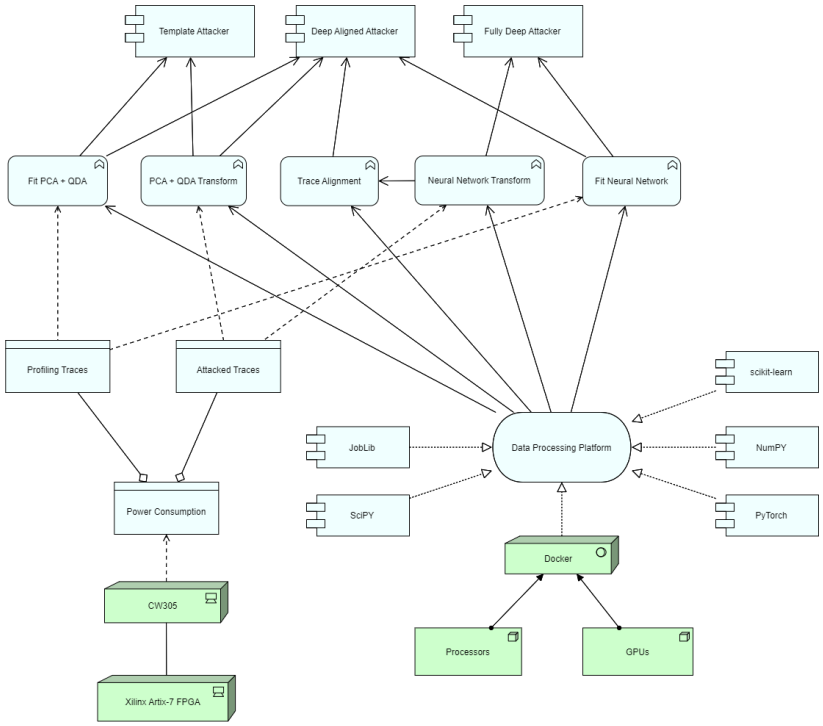

Figure 2.1: ArchimMate diagram of the applicative layers and computing resources involved in the side-channel analysis of our cryptosystem

- Boolean arrays can be used to retrieve, where the boolean value is true, some elements of a given array (e.g. $\overrightarrow{x} \leftarrow \boldsymbol{X}[\overrightarrow{x}_{where}]$). The function index returns an array of indices where a given boolean array is true
- The range function builds an array of int elements which go from 0 to a given number. The linspace function builds an array of float elements which go from 0 to a given number and spaced by a given quantity
- The zeros function builds an array with a given shape with all elements equal to 0. The sum function outputs the sum of each element of the passed array. The unique function returns an array of unique elements located inside a given array
- The dataframe function builds a relational table, which can be manipulated using the usual relational operators such as the insert, select or orderBy functions. The tuple function can be used to create a new tuple of a given relational table.

Figure 2.1 shows the ArchiMate diagram of the main application components and technological platform being used for our experiments. We will use the provided functional interface to describe, for the rest of the chapter, the method being used for the side-channels attacks of the given cryptosystem. As previously anticipated, we will be focused on profiled attacks, highlighting DL techniques in a couple of different SCA settings.

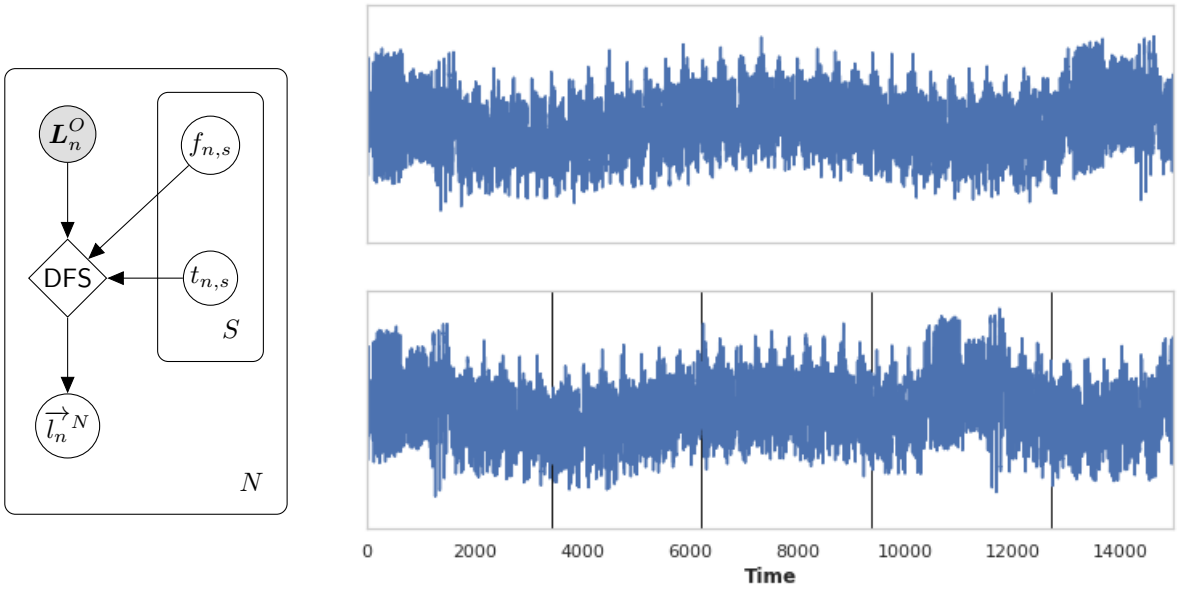

Figure 2.2: Bayesian overview of the generation of RDFS augmented power traces using static frequency traces (left). Example of a starting window of a 48 MHz static trace (top right) and the respective RDFS trace (bottom right) which alternates between 52,48,52,56,48 MHz

2.1. Leakage data

For our experiments we have 4 datasets composed of power compumption leakages of our cryptosystem. Each dataset is measured with the device running with different static operative frequencies. The set of available frequencies for the computing platform is $\mathcal{F} = \{48, 52, 54, 56\}$ [MHz]. The data have been sampled for each of the 256 values of the first byte of the key, and for 500 random values of the plain text. Because we are dealing with profiled SCA problems, the collected data is splitted into *profiling* and *attacking* data. To achieve such a split, we partitioned the data by splitting on the observed plain texts, so we have profiling set for 300 different plain texts and an attacking set with 200 different plain texts. The same type of splitting strategy is used in section 2.3 for partitionioning the data into training/validation sets.

The attacks will be also experimented for the ideal RDFS protected implementation. We now describe the probabilistic process used for generating ideal RDFS traces from static side-channel traces (Figure 2.2), in the same way as described in chapter 1 for random delays and clock jittering augmentations. We denote by $\boldsymbol{L}_n^O \in \mathbb{R}^{|\mathcal{F}| \times N}$ the trace realizations, one for each of the $|\mathcal{F}|$ static frequencies, and for the corresponding N distinct pairs p_n, k_n. For each frequency $f \in \mathcal{F}$ the corresponding static trace is $\overrightarrow{l_n}^O = \boldsymbol{L}_n^O[f]$.

For each pair of plain text p_n and key k_n, the corresponding augmented dynamic trace $\overrightarrow{l_n}^N$ is obtained by loading and positioning random static frequency windows inside the generated trace. For each of the S static windows $w_{n,s}$ showing in $\overrightarrow{l_n}^N$, the corresponding frequency $f_{n,s} \in \mathcal{F}$ is sampled with uniform distribution. Also the respective temporal duration $t_{n,s}$ is sampled with normal distribution. The function DFS then loads the corresponding trace slice, which is $w_{n,s} = \boldsymbol{L}_n^O[s, t : t_n]$. The starting time t of $w_{n,s}$ is computed by accumulating the elapsed time of $\overrightarrow{l_n}^N$ while sampling from time 0. The dynamic assembling function DFS eventually appends each $w_{n,s}$ at the end of $\overrightarrow{l_n}^N$.

Figure 2.2 shows a comparison of a static frequency leakage trace and the respective dynamic trace. In the figure it can be noticed a clear desynchronization in the leakages, by observing for example the relative temporal positions of the ending pattern. The same generative process can be extended to create RDVS traces, by sampling voltages $v_{n,s}$ instead of frequencies $f_{n,s}$. Because we are specifically interested in experimenting with desynchronizations, we will not cover DVS based countermeasures for our thesis.

2.2. Template Attack

In this section we will describe the procedures used to realize the bayesian TA method used to attack our cryptosystem. During the profiling phase, we fit the dimensionality reduction and classification subsystems on the profiling dataset. We will then use the computed models on the attacking dataset to transform the leakage traces into predictions of the key being used by the attacked device. The fitted templates will also be used later, in conjuction with a neural network, to perform a *deep aligned* template attack.

As described in chapter 1, we will first transform the side-channel traces $\boldsymbol{L}$ into the dataset $\boldsymbol{H}$, in which each data point have n_{dim} dimensions. The mentioned dimensionality reduction strategy is based on the PCA variant for TA [3]. During the profiling phase of PCA, the fitPca procedure (Algorithm 2.1) use the knowledge of the key values $\overrightarrow{k}$ and plain texts $\overrightarrow{p}$ of the traces $\boldsymbol{L}$ in order to compute the respective sensitive variables $\overrightarrow{z}$ of the traces. Then the average traces $\boldsymbol{M}$, one for each sensitive variable z_i, are computed in an online manner. Finally the fit procedure use a PCA implementation to compute the model parameters for the mean traces $\boldsymbol{M}$. This is accomplished using the PCA function, which is the implementation of the PCA method, available in the scikit-learn Python library. The PCA function is responsible of returning the values of $\mu, \boldsymbol{W}$ by computing the eigendecomposition of the data covariance matrix, as described in equation (1.12).

Algorithm 2.1 fit of PCA on profiling traces

procedure FITPCA($\overrightarrow{p}$, $\overrightarrow{k}$, $\boldsymbol{L}, n_{dim}$)
  $\boldsymbol{S} \leftarrow \mathsf{zeros}(|\mathcal{Z}|, \mathsf{size}(\boldsymbol{L}))$ ▷ accumulated traces
  $\boldsymbol{D} \leftarrow \mathsf{zeros}(|\mathcal{Z}|)$ ▷ traces count
  $\overrightarrow{s} \leftarrow \mathsf{SBox}(\overrightarrow{p} \oplus \overrightarrow{k})$ ▷ sbox output
  $\overrightarrow{z} \leftarrow \mathsf{HW}(\overrightarrow{s})$ ▷ sensitive variable
  for $z_j \in \mathcal{Z}$ **do**
    $\overrightarrow{z}_{where} \leftarrow \overrightarrow{z} = z_j$
    for $\overrightarrow{l}_n \in \boldsymbol{L}[\overrightarrow{z}_{where}]$ **do**
      $\boldsymbol{S}[z_j] \leftarrow \boldsymbol{S}[z_j] + \overrightarrow{l}_n$
    $\boldsymbol{D}[z_j] \leftarrow \boldsymbol{D}[z_j] + \mathsf{sum}(\overrightarrow{z}_{where})$
  $\boldsymbol{M} \leftarrow \boldsymbol{S}/\boldsymbol{D}$ ▷ mean traces
  $\boldsymbol{W}, \overrightarrow{\mu} \leftarrow \mathsf{PCA}(\boldsymbol{M}, n_{dim})$ ▷ PCA components and mean
  return $\boldsymbol{W}, \overrightarrow{\mu}$

Algorithm 2.2 PCA transform of profiling/attacking traces

procedure TRANSFORMPCA($\boldsymbol{L}, \boldsymbol{W}, \overrightarrow{\mu}$)
  for $i \in \mathsf{range}(\mathsf{len}(\boldsymbol{L}))$ **do**
    $\boldsymbol{L}[i] \leftarrow \boldsymbol{L}[i] - \overrightarrow{\mu}$
  $\boldsymbol{H} \leftarrow \boldsymbol{L}\boldsymbol{W}^T$ ▷ reduced traces
  return $\boldsymbol{H}$

After fitting the PCA model on the profiling data, the adversarial has computed the parameters $\boldsymbol{W}$ and $\overrightarrow{\mu}$. They are further used in the transformPca procedure (Algorithm 2.2) to transform some profiling or attacking traces $\boldsymbol{L}$ into the lower dimensional $\boldsymbol{H}$. Once fitted the PCA, it is possible to continue the profiling stage by computing the sensitive variable classifier $\hat{g}$. In the PCA based TA approach, the attacker fit a QDA classifier on the reduced side-channel traces $\boldsymbol{H}$. The fitQda procedure (Algorithm 2.3) estimates the gaussian generative distribution $\hat{f}$ of the dimensionality reduced traces. Because $\hat{f}$ is a mixture of gaussians, the distribution can be fully described by the estimation of the first two moments $\boldsymbol{M}, \boldsymbol{\Sigma}$ (called templates in SCA context) of each gaussian in the mixture. Using the knownledge of key values $\overrightarrow{k}$ and plain texts $\overrightarrow{p}$ of the traces, the algorithm incrementally compute the first two moments of the $|\mathcal{Z}| = 9$ normal distributions which completely describe the mixture of gaussians. The generative distribution $\hat{f}$ is then exploited for computing the predictive distribution $\hat{g}$ of the target sensitive variable.

Algorithm 2.3 fit of QDA on reduced profiling traces

procedure FITQDA($\overrightarrow{p}$, $\overrightarrow{k}$, $\boldsymbol{L}, \boldsymbol{W}, \overrightarrow{\mu}$)
  $n_{dim} \leftarrow \mathsf{len}(\boldsymbol{W})$
  $\boldsymbol{S_M} \leftarrow \mathsf{zeros}(|\mathcal{Z}|, n_{dim})$ ▷ accumulated means
  $\boldsymbol{S_C} \leftarrow \mathsf{zeros}(|\mathcal{Z}|, n_{dim}, n_{dim})$ ▷ accumulated covariances
  $\boldsymbol{D} \leftarrow \mathsf{zeros}(|\mathcal{Z}|)$ ▷ traces count
  $\boldsymbol{H} \leftarrow \mathsf{transformPca}(\boldsymbol{L}, \boldsymbol{W}, \overrightarrow{\mu})$
  $\overrightarrow{s} \leftarrow \mathsf{SBox}(\overrightarrow{p} \oplus \overrightarrow{k})$ ▷ sbox output
  $\overrightarrow{z} \leftarrow \mathsf{HW}(\overrightarrow{s})$ ▷ sensitive variable
  for $z_j \in \mathcal{Z}$ **do**
    $\overrightarrow{z}_{where} \leftarrow \overrightarrow{z} = z_j$
    $\boldsymbol{H}_{curr} \leftarrow \boldsymbol{H}[\overrightarrow{z}_{where}]$
    for $\overrightarrow{h}_n \in \boldsymbol{H}_i$ **do**
      $\boldsymbol{S_M}[z_j] \leftarrow \boldsymbol{S_M}[z_j] + \overrightarrow{h}_n$
    $\boldsymbol{S_C}[z_j] \leftarrow \boldsymbol{S_C}[z_j] + \boldsymbol{H}_{curr}^T \boldsymbol{H}_{curr}$
    $\boldsymbol{D}[z_j] \leftarrow \boldsymbol{D}[z_j] + \mathsf{sum}(\overrightarrow{z}_{where})$
  $\boldsymbol{M} \leftarrow \boldsymbol{S_M}/\boldsymbol{D}$ ▷ means
  $\boldsymbol{\Sigma} \leftarrow \boldsymbol{S_C}/(\boldsymbol{D}(\boldsymbol{D}-1))$ ▷ convariances
  return $\boldsymbol{M}, \boldsymbol{\Sigma}$

Algorithm 2.4 QDA transform of reduced profiling/attacking traces

procedure TRANSFORMQDA($\overrightarrow{p}, \boldsymbol{H}, \boldsymbol{W}, \overrightarrow{\mu}, \boldsymbol{M}, \boldsymbol{\Sigma}$)
  $\boldsymbol{P} \leftarrow \mathsf{zeros}(|\mathcal{K}|, \mathsf{len}(\overrightarrow{p}))$ ▷ accumulated predictions
  for $k_i \in \mathcal{K}$ **do**
    $\overrightarrow{s} \leftarrow \mathsf{SBox}(\overrightarrow{p} \oplus k_i)$ ▷ first byte of sbox output
    $\overrightarrow{z} \leftarrow \mathsf{HW}(\overrightarrow{s})$ ▷ sensitive variable for k_i
    $\hat{f} \leftarrow \mathcal{N}ormal(\boldsymbol{M}[\overrightarrow{z}], \boldsymbol{\Sigma}[\overrightarrow{z}])$ ▷ $\overrightarrow{H}$ generative distribution
    $\hat{z} \leftarrow \mathcal{B}inomial(8, 0.5)$ ▷ prior Z distribution
    $\boldsymbol{P}[\overrightarrow{z}] \leftarrow \mathsf{log}(\hat{f}(\boldsymbol{H})) + \mathsf{log}(\hat{z}(\overrightarrow{z}))$
  return $\boldsymbol{P}$

Therefore, the value of Z is inferred via bayesian inference, which is implemented in the transformQda procedure (algorithm 2.4). By knowing their plain texts $\overrightarrow{p}$, the algorithm computes the probabilities of each possible key hypothesis k_i. The prediction entropies are accumulated using equation (1.8), in order to exploit multiple invocation of the same cryptosystem for different values of the plain text (but the same key value).

Algorithm 2.5 fit of a NN using backpropagation and early stopping

procedure FITNEURALNETWORK($\hat{g}, \boldsymbol{X}, \overrightarrow{y}, \mathcal{L}, n_{epoch}, \eta, \delta, \gamma, limit, patience$)
  $\boldsymbol{X}_{train}, \overrightarrow{y}_{train}, \boldsymbol{X}_{valid}, \overrightarrow{y}_{valid} \leftarrow$ splitData($\boldsymbol{X}, \overrightarrow{y}$) ▷ train/valid split
  $stop_{epoch} \leftarrow 0$
  $stop_{\hat{g}} \leftarrow \hat{g}$
  $stop_{\mathcal{L}} \leftarrow \infty$
  for $epoch \in$ range(n_{epoch}) **do**
    $\boldsymbol{X}_{epoch}, \overrightarrow{y}_{epoch} \leftarrow$ sampleData($\boldsymbol{X}_{train}, \overrightarrow{y}_{train}, limit.train$) ▷ epoch train data
    $\eta \leftarrow$ schedulerStep(η, δ, γ) ▷ LR scheduling
    for $\boldsymbol{X}_{batch}, \overrightarrow{y}_{batch} \in \boldsymbol{X}_{epoch}, \overrightarrow{y}_{epoch}$ **do**
      $\mathcal{L}_{batch} \leftarrow \mathcal{L}(\overrightarrow{y}_{batch}, \hat{g}(\boldsymbol{X}_{batch}))$ ▷ train batch loss
      $\nabla_{\boldsymbol{W}} \leftarrow$ computeGrad($\hat{g}, \mathcal{L}_{batch}$)
      optimizerStep($\hat{g}, \nabla_{\boldsymbol{W}}, \eta$)
    $\boldsymbol{X}_{epoch}, \overrightarrow{y}_{epoch} \leftarrow$ sampleData($\boldsymbol{X}_{valid}, \overrightarrow{y}_{valid}, limit.valid$) ▷ epoch valid data
    $\mathcal{L}_{valid} \leftarrow \mathcal{L}(y_{valid}, \hat{g}(\boldsymbol{X}_{valid}))$ ▷ valid loss
    if $\mathcal{L}_{valid} < stop_{\mathcal{L}}$ **then**
      $stop_{epoch} \leftarrow epoch$
      $stop_{\hat{g}} \leftarrow \hat{g}$
      $stop_{\mathcal{L}} \leftarrow \mathcal{L}_{valid}$
    else if $epoch - stop_{epoch} > patience$ **then**
      return $stop_{\hat{g}}$
  return $stop_{\hat{g}}$

2.3. Deep Learning Attacks

As described in chapter 1, some of the most effective ways to tackle the SCA problem is to exploit the expressive power of DL based models. We will now present the fitNeuralNetwork procedure (Algorithm 2.5) that will be used for training a given neural network $\hat{g}$ to map some features of the generic dataset $\boldsymbol{X}$ to the corresponding target variables y.

The dataset is first splitted into training and validation set by partititioning on the plain texts. The weights $\boldsymbol{W}$ of the network are optimized by minimizing the classification loss $\mathcal{L}_{\hat{g}}$ (equation 1.16) of the training set. The data is recursevely processed by the network for n_{epoch} times. During each iteration, a random subset of *limit* data points is used, in order to not use the same points every time and thus improving the learning quality of the network. During each epoch the network uses the training data for mini-batch optimization of $\boldsymbol{W}$, by exploiting the gradient computed with computeGrad function.

The optimization is performed with the learning rate η, by using the optimizerStep function, which uses a given algorithm (e.g. SGD or ADAM). Furthermore, in order to incrementally refine the optimization convergence through epoches, we additionally adapt the learning rate η every δ epoches. The only tested learning rate scheduler for our SCA task, identified by schedulerStep procedure, updates the value of the learning rates η by multipliing it by a given fixed constant γ.

The validation data is instead used for the *early stopping* algorithm, a regularization technique which evaluate the model on unseen data at the end of each epoch. This allows us to perform a *model selection* before the network starts to overfit. If the loss on validation data stops decreasing for *patience* epoches, the training procedure is stopped and the selected weights are the ones obtained at the end of the last decreasing epoch.

To achieve a *fully deep* attack we will build an end-to-end model for SCA, which directly computes the distribution of the secret Z from a leakage $\overrightarrow{L}$. Therefore, in this context the algorithm 2.5 will be used with $\boldsymbol{X} = \boldsymbol{L}$ and $\overrightarrow{y} = \overrightarrow{z}$ for obtaining the attacking predictor.

2.3.1. Deep aligned Template Attack

Algorithm 2.5 will be also used for *deep aligned* attack. However, in this scenario the neural network $\hat{g}$ will not play the role of the estimator of the secret Z distribution. The deep model will instead be used to localize some desynchronizations introduced to counter most commons attacks. These protections will be removed in order to apply an already known and optimal classifer such as the PCA+QDA model, which is typically not able to recover the key from misaligned traces. The method also allows to use templates previously computed for unprotected device for attacking protected implementations.

In the thesis we will be experimenting with RDFS based desynchronizations. We will engineer our frequency scaling detector using the weakly supervised localization techniques described in chapter 1. Therefore we build a network with frequency scaling detection capabilities by just training it to classify the frequencies. In this way, we avoid building some segmentation or object detection architectures, which are typically more complex and more difficult to train. To detect when and how this type of protection occurs, a network is fitted to predict the frequencies $f_i \in \mathcal{F}$ (the ones used by the RDFS) of some fixed-sized windows w_i located in the static traces. The slicing of the windows is performed starting from random positions of the traces. For this frequency detection task, the neural networks that will be used will always have a final GAP layer at the end of the encoder.

Algorithm 2.6 segment one trace with frequency semantic

procedure SEGMENTTRACE($\overrightarrow{l}, \hat{g}$)
  $\boldsymbol{F} \leftarrow \mathsf{computeFeatMaps}(\hat{g}, \overrightarrow{l})$ ▷ feature maps
  $\overrightarrow{\alpha} \leftarrow \mathsf{zeros}(|\mathcal{Z}|, \mathsf{len}(\boldsymbol{F}))$ ▷ maps importances
  for $z_i \in \mathcal{Z}$ **do**
    for $i = 1 : \mathsf{len}(\boldsymbol{F})$ **do**
      $\overrightarrow{\alpha}[z_i, i] = \sum_j \frac{\partial \hat{g}(\overrightarrow{l})[z_i]}{\partial \boldsymbol{F}[i,j]}$
  $\boldsymbol{A} \leftarrow \mathsf{zeros}(|\mathcal{Z}|, \mathsf{size}(\boldsymbol{F}))$ ▷ class activation maps
  for $z_i \in \mathcal{Z}$ **do**
    $\boldsymbol{A}[z_i] \leftarrow \sum_{j=1}^{\mathsf{len}(\boldsymbol{F})} \overrightarrow{\alpha}[z_i, i]\, \boldsymbol{F}[z_i, j]$
    $\boldsymbol{A}[z_i] \leftarrow \mathsf{ReLU}(\boldsymbol{A}[z_i])$
  $\overrightarrow{s} \leftarrow \mathsf{zeros}(\mathsf{len}(\overrightarrow{l}))$ ▷ frequency segmented trace
  for $i = 1 : \mathsf{len}(\overrightarrow{s})$ **do**
    $\overrightarrow{s}[i] \leftarrow \mathsf{argmax}_{z_j}\, \boldsymbol{A}[z_j, i]$
  return $\overrightarrow{s}$

Algorithm 2.7 localize static windows inside some dynamic traces

procedure LOCALIZERDFS($\overrightarrow{s}$)
  $lclz \leftarrow \mathsf{dataframe}('f' : \mathsf{int}, 'start' : \mathsf{int}, 'end' : \mathsf{int})$ ▷ RDFS localization KB
  for $f_i \in \mathcal{F}$ **do**
    $\overrightarrow{f}_{where} \leftarrow \overrightarrow{s} = f_i$
    $\overrightarrow{s}_{curr} \leftarrow \overrightarrow{s}[\overrightarrow{f}_{where}]$
    $\overrightarrow{s}_{objs} \leftarrow \mathsf{labelFeatures}(\overrightarrow{s}_{curr})$ ▷ clustered trace for f_i
    $\overrightarrow{w}_{ids} \leftarrow \mathsf{unique}(\overrightarrow{s}_{objs})$ ▷ IDs of the windows
    for $w_j \in \overrightarrow{w}_{ids}$ **do**
      $\overrightarrow{w}_{where} \leftarrow \overrightarrow{s}_{objs} = w_j$
      $time_j \leftarrow \mathsf{index}(\overrightarrow{w}_{where})$
      $start_j \leftarrow \mathsf{argmin}(time_j)$
      $end_j \leftarrow \mathsf{argmax}(time_{id})$
      $knw_j \leftarrow \mathsf{tuple}(f_i, start_j, end_j)$ ▷ knowledge of the window
      $\mathsf{insert}(lclz,\, knw_j)$
  $lclz \leftarrow \mathsf{orderBy}(lclz, 'start')$
  return $lclz$

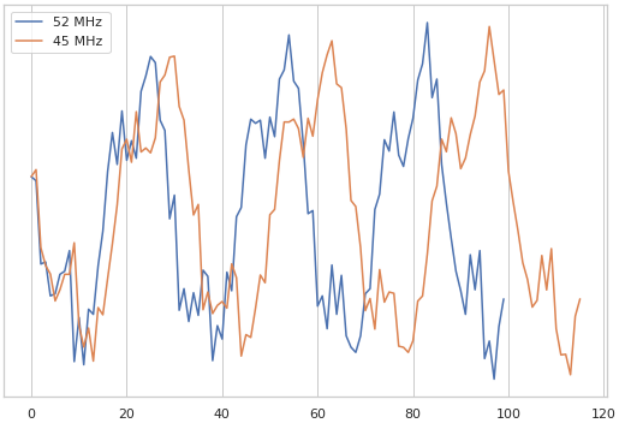

Figure 2.3: Using interpolation to scale a random leakage window from 52 to 48 MHz

Algorithm 2.8 change the frequency of a window using interpolation

procedure SHIFTFREQUENCY($\overrightarrow{w}, f_{in}, f_{out}$)
  $time_{in} \leftarrow \mathsf{range}(\mathsf{size}(\overrightarrow{w}))$
  $func_{interp} \leftarrow \mathsf{interp1d}(time_{in}, \overrightarrow{w})$ ▷ interpolating function
  $f_{ratio} \leftarrow f_{out} \,/\, f_{in}$
  $n_{steps} \leftarrow \mathsf{floor}(\mathsf{size}(\overrightarrow{w}) \,/\, f_{ratio}) + 1$
  $time_{out} \leftarrow \mathsf{linspace}(\mathsf{size}(\overrightarrow{w}) - 1, n_{steps})$
  $\overrightarrow{w_s} \leftarrow func_{interp}(time_{out})$ ▷ scaled window
  return $\overrightarrow{w_s}$

Using the GAP implies that we can then feed an input of any shape into the network. We then exploit this last assumption and the Grad-CAM method to obtain the frequency detection system. The model $\hat{g}$ is trained by classification to find the window to frequency representation of the traces. Then full dynamic traces are processed in the model, and the frequencies inside the signal are detected inside the whole inputs. The procedure segmentTrace (Algorithm 2.6) is used by the attacking system to perceive the frequency used by the device during each time step of a given trace $\overrightarrow{l}$.

The function computeFeatMaps returns the feature maps $\boldsymbol{F}$ of the frequency classifier $\hat{g}$ for the input trace $\overrightarrow{l}$. The number of feature maps extracted depends of the model being used. For each features map the corresponding importance is computed with equation (1.23). Then the class activation mapping is computed using equation (1.24) by a weighted sum of each encoded map of the input. The segmented trace $\overrightarrow{s}$ is then computed with equation (1.25). The segmentation contains a frequency label for each time step of $\overrightarrow{l}$, and it is further used to compute a frequency localization task.

Algorithm 2.9 align a dynamic trace to a static frequency

```
1: procedure ALIGNTRACE(l⃗, ĝ, f_target)
2:     s⃗ ← segmentTrace(l⃗, ĝ)                          ▷ frequency segmented trace
3:     lclz ← localizeRdfs(s⃗)                           ▷ RDFS localization KB
4:     l⃗_align ← zeros(size(l⃗))                        ▷ aligned trace
5:     for knw_i ∈ lclz do
6:         f_i, start_i, end_i ← kwn_i
7:         w⃗_i ← l⃗[start_i : end_i]
8:         if f_i ! = f_target then
9:             w⃗_s ← shiftFrequency(w⃗_i, f_i, f_target)   ▷ scaled window
10:            end_i ← start_i + size(w⃗_s)
11:        l⃗_align[start_i : end_i] ← w⃗_i
12:    return l⃗_align
```

The procedure localizeRdfs (Algorithm 2.7) finds each static frequency window inside $\overrightarrow{l}$ by exploiting its frequency segmented map $\overrightarrow{s}$. The frequency labels at each time step are transformed into the structured knowledge of each static window instance perceived in the trace. This is accomplished by using function labelFeatures on $\overrightarrow{s}$, which clusters all the adjacent time steps with same labels into the same window instance. Therefore each window object is logically represented by its frequency and time boundaries.

After the localization of each static window $\overrightarrow{w}$ of the dynamic trace $\overrightarrow{l}$, the attacking system use this knowledge in the alignTrace procedure (algorithm 2.9) to bring the trace to a state where there is no dynamic frequency scaling. This is done by shifting the frequency of each detected window to a fixed frequency f_{target}, which is also the frequency of the corresponding templates that will be used to attack after the alignment process.

To change the frequency of a time series, the procedure shiftFrequency (algorithm 2.8) use the 1D interpolation method interp1d to realize a temporal scaling (Figure 2.3). This method basically joins each pairs of adjacent points with a polynomial curve of a given order. We will only deal with linear interpolation in our work. To compute the rescaled window $\overrightarrow{w_s}$, the linear interpolating function is called for the same time steps rescaled by f_{out}/f_{in} where $f_{out} = f_{target}$. Other methods could be investigated to realize the frequency shift operator, but this is out out of scope for our thesis.

3 | Experimental evaluation

In this section we will apply the methods described in chapter 2 to attack the AES cipher deployed for our FPGA platform. Our analysis will be focused on comparing the performance of TA, fully deep SCA and deep aligned TA when attacking both unprotected and RDFS protected implementations of the cryptosystem. We will now describe the evaluation metrics that will be used to validate the performance of the each method.

3.1. Metrics

In order to guess the secret, the attacker must build a predictive model $\hat{g}$ to decide which is the true key k^*. During profiling stage the main metrics used are the typical ML classification metrics described in equations (3.1) and (3.2), which are the *Precision*, *Recall*, *F1* and *Accuracy* metrics. The respective *Confusion Matrix* (CM) will be also reported, which represent in a grid the real vs predicted classes of each data point passed to the model.

$$\mathsf{Prec} = \frac{\mathsf{TP}}{\mathsf{TP} + \mathsf{FP}} \qquad \mathsf{Rec} = \frac{\mathsf{TP}}{\mathsf{TP} + \mathsf{FN}} \tag{3.1}$$

$$\mathsf{F1} = 2 \cdot \frac{\mathsf{Prec} \cdot \mathsf{Rec}}{\mathsf{Prec} + \mathsf{Rec}} \qquad \mathsf{Acc} = \frac{\mathsf{TP} + \mathsf{TN}}{\mathsf{TP} + \mathsf{FP} + \mathsf{TN} + \mathsf{FN}} \tag{3.2}$$

In equations (3.1) and(3.2) we denote with $\mathsf{TP}, \mathsf{FP}, \mathsf{TN}, \mathsf{FN}$ the true positives, false positives, true negatives and false negatives of the predictions made by $\hat{g}$. For SCA problems the metrics so far described can be used to evaluate a *simple attack*, which is the attack where the inference is made by observing one leakage trace. However, the attacker could need to make more observations to guess the key with more certainty. Therefore, during attacking phase, the adversarial performs the key decision process by maximizing the most probable (according to $\hat{g}$) key given all the observed leakages of the attacked device.

For a dataset $\boldsymbol{L}$ made of N traces $\overrightarrow{l_i}$, each one computed for a known plain text p_i and an unknown true key k^*, the attacker computes the probability of equation (3.3) for each key hypothesis k. The logarithm is further applied in order to obtain information-like

quantities instead of probabilities. In this way each prediction can be accumulated by adding each $\mathsf{log}\ d_{\hat{g}}$ leaked from each $\overrightarrow{l_i}$, also avoiding numerical computational issues such as underflows which can arise from multiplying real values near zero.

$$d_{\hat{g}}(\boldsymbol{L}, N, k) = \prod_{i=1}^{N} \hat{g}(\overrightarrow{l_i})[\psi(p_i, k)] \qquad \mathsf{log}\ d_{\hat{g}}(\boldsymbol{L}, N, k) = \sum_{i=1}^{N} \mathsf{log}\ \hat{g}(\overrightarrow{l_i})[\psi(p_i, k)] \tag{3.3}$$

Using the above formulation, the attacker builds a set of $|\mathcal{K}|$ information curves, one for each key hypothesis. After a given number of observations, the attacker chooses the key k which have the lowest $\mathsf{log}\ d_{\hat{g}}$. We aggregate these curves for each attack by further computation of a couple of SCA specific metrics [48]. In this way each attack can be described by one curve, and different attacking techniques can be compared more easily. The *certainty* of the attack, described by equation (3.4), is the ratio between the probability of the true key k^* being picked divided by the probability of selecting the second guess k'. This probability ratio as well can be computed more easily by computing the log certainty with equation (3.5).

$$\mathsf{Cert}_{\hat{g}}(\boldsymbol{L}, N, k^*) = \frac{d_{\hat{g}}(\boldsymbol{L}, N, k^*)}{\mathsf{max}_{k' \in \mathcal{K} \setminus k^*}\ d_{\hat{g}}(\boldsymbol{L}, N, k')} \tag{3.4}$$

$$\mathsf{log}\ \mathsf{Cert}_{\hat{g}}(\boldsymbol{L}, N, k^*) = \mathsf{log}\ d_{\hat{g}}(\boldsymbol{L}, N, k^*) - \max_{k' \in \mathcal{K} \setminus k^*} \mathsf{log}\ d_{\hat{g}}(\boldsymbol{L}, N, k') \tag{3.5}$$

The *guessing entropy* is the measure of how many keys are more likely, after N observations, than the true key. This quantity is what is called in the SCA context as the *rank* of the key, which is defined in equation (3.6).

$$\mathsf{Rank}_{\hat{g}}(\boldsymbol{L}, N, k^*) = |\{k' \in \mathcal{K} : d_{\hat{g}}(\boldsymbol{L}, N, k') > d_{\hat{g}}(\boldsymbol{L}, N, k^*)\}| \tag{3.6}$$

The *efficiency* is the minimum size of the attack set needed to ensure that the correct key is picked. It is defined in equation (3.7), where a and b are function of k^*.

$$\mathsf{Eff}_{\hat{g}}(\boldsymbol{L}, k^*) = N\ :\ \mathsf{Rank}_{\hat{g}}(\boldsymbol{L}, a, k^*) > 0\ \wedge\ \mathsf{Rank}_{\hat{g}}(\boldsymbol{L}, b, k^*) = 0 \qquad \forall\ a < N < b \tag{3.7}$$

The above defined metrics are computed for each true key value k^*. We are mainly interested in estimating the performance of the attack independently of the $|\mathcal{K}|$ values that k^* could assume. Therefore the metrics will be aggregated on the true key, resulting in considering the average metric case over the uniformly random values of k^*

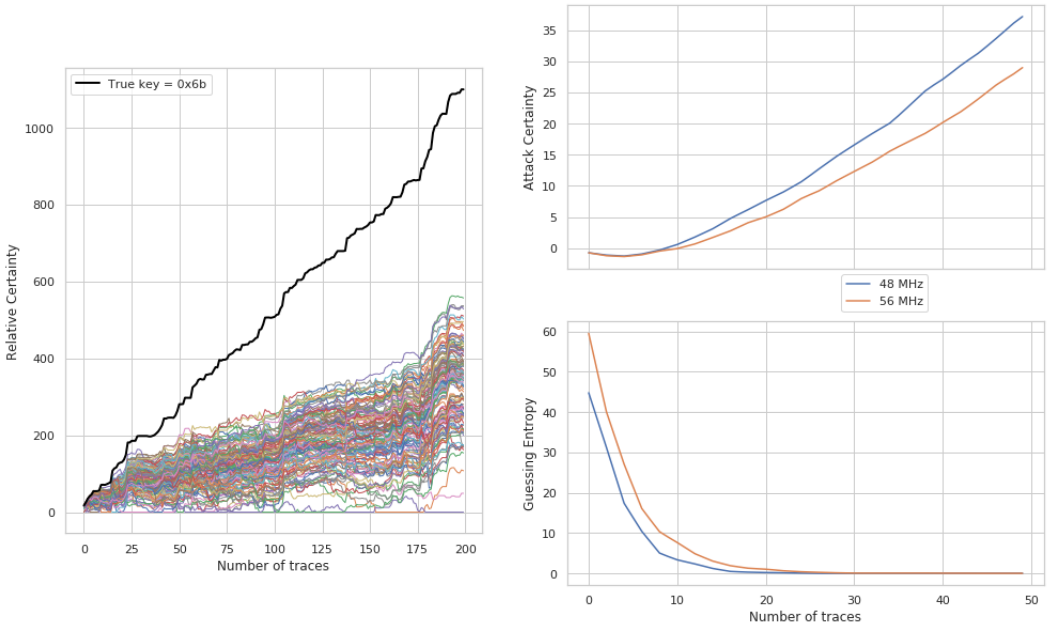

Figure 3.1: TA attack on static 48 MHz device using 48 MHz templates (left, right-blue) and using the same templates on static 56 MHz device (right-red). The plot on the left is the prediction for a given k^*, the one on the right is the average case.

Lastly, for the evaluation of segmentation tasks we will use the *Intersection over Union* (IoU) score of equation (3.8), which is basically the percentage of the area correctly labelled by a given map.

$$\mathsf{IoU}(\boldsymbol{A}, \boldsymbol{B}) = \frac{|\boldsymbol{A} \cap \boldsymbol{B}|}{|\boldsymbol{A} \cup \boldsymbol{B}|} \tag{3.8}$$

3.2. Unprotected device

The starting point for the experimentations is to fit and test each method on the static leakage datasets. The baseline model in this context is the gaussian TA with PCA dimensionality reduction. The dimensionality of the hidden PCA representation must be smaller than the number of data points used during the fitting process. Therefore, because we want to process $|\mathcal{Z}| = 9$ mean traces (one for each z_i), the PCA latent representation is chosen to be 8-dimensional.

In Figure 3.3 the matrix $\boldsymbol{W}$ of the PCA transformation is plotted. In the picture the PCA features (rows) can be analyzed by observing the importances of each temporal measurement (columns). Once preprocessed with PCA, the data is fed to the sensitive variable QDA classifier. In Figure 3.3 the 9 multivariate gaussian means $\boldsymbol{M}$ are plotted, with the variance-aggregation of the corresponding 9 covariance matrices $\boldsymbol{\Sigma}$.

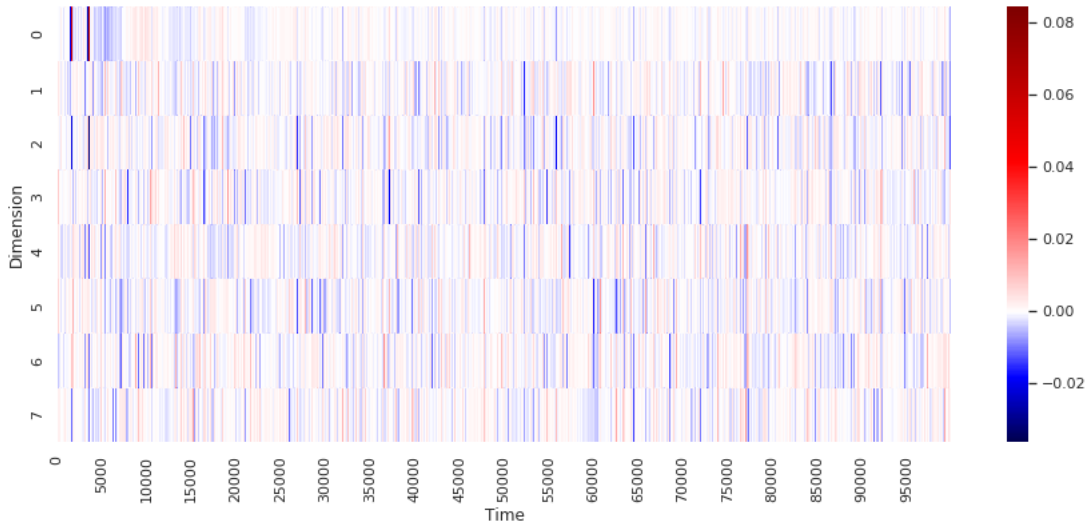

Figure 3.2: Visualization of matrix $\boldsymbol{W}$ for a fitted PCA on 54 MHz static dataset

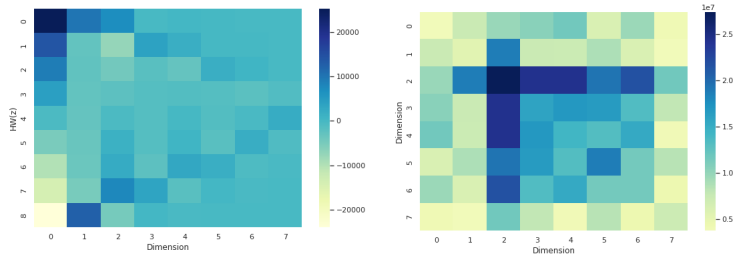

Figure 3.3: Visualization of the means $\boldsymbol{M}$ of each multivariate gaussian (left) and the variance $\mathsf{Var}_Z[\boldsymbol{\Sigma}]$ of the corresponding 9 covariance matrices (right) for a fitted QDA on 54 MHz static traces

From the last plot we can assert that the QDA model is more suited w.r.t LDA model for this problem, because in LDA each covariance matrix is the same while it can been noticed that there are differences in the covariances of each mixture distribution.

In Figure 3.1 the resulting TA attack for a 48 MHz static platform can be visualized. Table 3.1 shows the performance of the attack, which after an average of 20 leakage observations can be considered successful. The same templates are then further used to estimate the performance of the model when the attacked device has a different frequency w.r.t the profiling device exploited by the attacker. As it can be clearly seen in the picture, when the power traces are not synchronized in frequency there is a drop of performance. By increasing the frequency of 6 MHz, the resulting certainty of the attack is about $\frac{4}{5}$ of the synchronized setting. As we will see later, the misalignment is even more effective when there is a dynamic desynchronization in the cryptosystem.

experiment	efficiency
static - TA	20
static - TA with 6 MHz variability	26

Table 3.1: Attack results for TA on 48 MHz static device, and using 54 MHz templates

HW	0	1	2	3	4	5	6	7	8
Prior	0.0039	0.0313	0.1094	0.2188	0.2734	0.2188	0.1094	0.0313	0.0039
Weight	36.536	4.5714	1.3061	0.6531	0.5224	0.6531	1.3061	4.5714	36.536

Table 3.2: Selected weights used for the weighted classification loss employed for training the fully deep predictors

After computing the TA baseline, we will start testing deep learning models for our SCA problem. From now on, each experiment involving the fit of a neural network (Algorithm 2.5) will be performed with a data split of 80% for the training set and 20% for the validation set. The optimizers that will be tested are the SGD and Adam algorithms. When training VGG models on full traces, we are usually assuming a batch size of 128. When ResNet models are instead being used, we use a smaller batch size of 32 because the residual architectures require higher memory resources at inference time. For both architectures some batch normalization layers are always used. These layers are used for regularization purpose, and are used in both architectures inside the encoder after the i-th pooling layer. Also a final GAP layer is always used in our convolutional networks.

For VGG models the hyper-parameters of the newtork are the number of base filters, the kernel size, the convolution stride, the number of 2-conv blocks, the number of successive 3-conv blocks. For ResNet models the hyper-parameters of the newtork are the number of base filters, the kernel size, the convolution stride, the increasing filter / downsample gap and the number of residual blocks. For each training performed, we found that the power of convolution is very effective to solve each task needed for the thesis. The model is able to efficiently fit each problem without needing deep FC layers for classification. So we decided to keep fixed the structure of the network after the encoded features map layer, keeping just one linear FC hidden layer with dropout before the softmax classifier.

We first fit a deep classifier for a fully deep attack on all unprotected device data, and then we will compare it against the previous PCA+QDA attack. During our first experiments, we found that small networks hardly find a good representation for border values of Z, because the HW function forces them to happen with low probability.

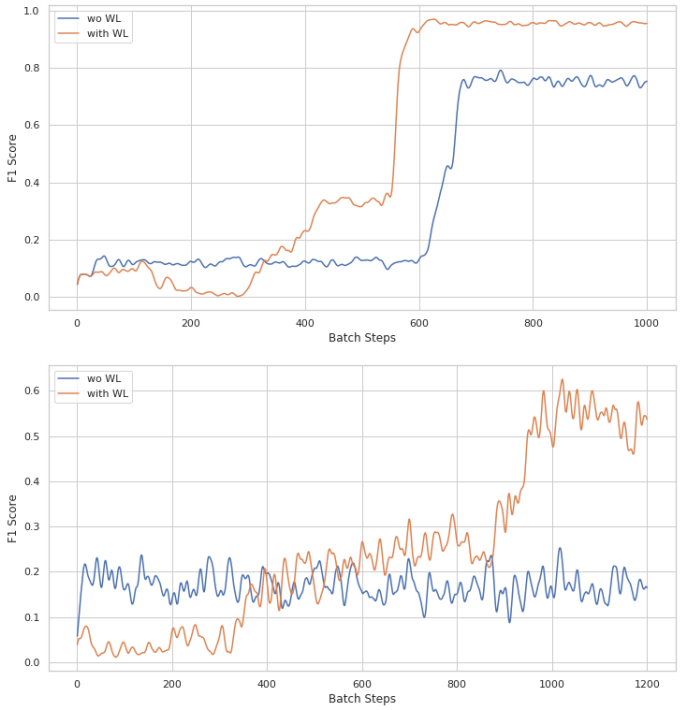

Figure 3.4: Training F1 score for VGG1 (top) and ResNet1 (bottom) models fitted on unprotected device data, with (red) and without (blue) weighted loss

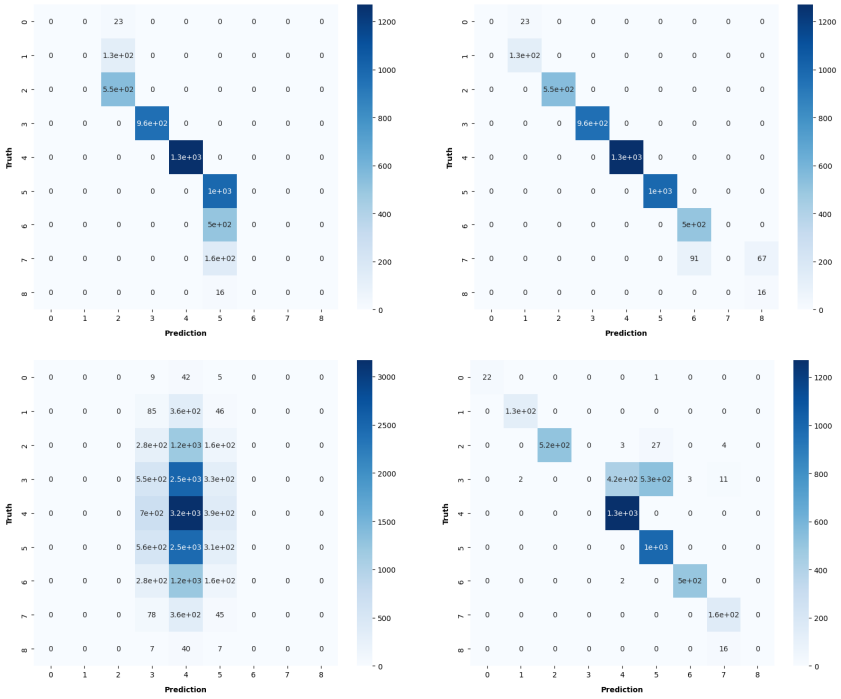

Figure 3.5: Validation confusion matrices for VGG1 (top) and ResNet1 (bottom) models fitted on unprotected device data, with (right) and without (left) weighted loss

experiment	loss	accuracy	F1	efficiency
static - VGG1	1.435	99.3%	99.2%	12

Table 3.3: Attack results for VGG1 on 52 MHz static device

experiment	loss	accuracy	F1	efficiency
static - ResNet1	1.604	98.2%	97.8%	15

Table 3.4: Attack results for ResNet1 on 52 MHz static device

One possible solution to this situation is to subsample likely instances (with HW near 5) or to oversample the unlikely ones (with HW near 0 or 9). However, there is still a solution which can help the network to learn a better representation without altering the dataset composition, which is weighting the loss of a data point proportionally to its class prior probability. We engineered the weights of Table 3.2 to be inversely proportional to the a priori distribution of Z defined in equation (1.2). The weights are also designed to be equals to 1 on the average case, in order to effectively compare the same learning rates when not using the weighted loss. This technique is first tested for the smallest and shallowest architectures which manifests a converging behavior for the problem. We do a first comparison with TA representation by using a single hidden FC layer of 8 neurons in the classifier network, and without using dropout. Therefore, the tested models are:

- *VGG1* (4.6M parameters): 48 base filters, kernel size 40, conv stride 4, one 2-conv starting block, two 3-conv final blocks, $\eta = 10^{-4}$, $\delta = 0.9$, $\gamma = 1$ and SGD
- *ResNet1* (2.1M parameters): 48 base filters, kernel size 30, conv stride 3, increase filter / downsample gap equals to 2 and 6 residual blocks, $\eta = 10^{-4}$, $\delta = 0.9$, $\gamma = 1$ and SGD

As can be seen from Figure 3.4 and 3.5, the learned representation with weighted loss is much better w.r.t to simple loss, both for VGG and ResNet architectures. Also, there is no real need of using lot of FC layers (which can cause overfitting) or dropout regularization to reach a good quality solution. Tables 3.3 and 3.4 summarizes the performance of the optimal convolutional models, which after an average of 12 and 15 leakage observations both achieve a successful attack. The main differences of hyper-parameters for these models w.r.t. previous tested models is the 40 sized FC layer followed by 0.2 dropout, with one more 2-conv block for the VGG and 2 more residual blocks for the ResNet. Also Adam is used instead of SGD. Figure 3.6 shows the profiling and attacking results for the optimal VGG (7.3M parameters) and ResNet (6.1M parameters) models.

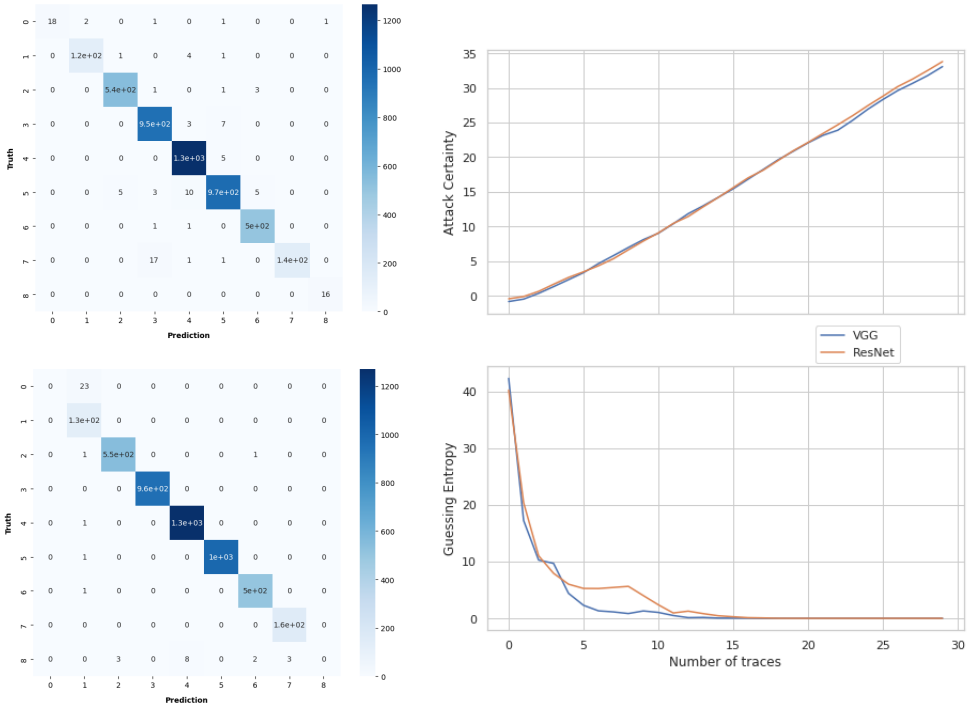

Figure 3.6: Comparison of fully deep attack on static 52 MHz device, using VGG1 (top-left, blue) and ResNet1 (bottom-left, red) models. Left shows CM for validation set.

3.3. Protected device

After evaluating each method for the static leakage datasets, we will now see how each technique is performing when attacking an RDFS based device. As already discussed, the desynchronization introduced by the module is very effective against the gaussian TA, which is impacted even more than in the case of static process variability. However, even if the dynamic frequency process invalidate the QDA assumptions, it is still possible to remove the RDFS protection and then exploit the templates computed previously for the static case. But because the frequency switches are actuated in a random way, the attacker has no knowledge on where and how the alignment process must be performed. Therefore the QDA attacking system must improved with a RDFS perceiving component, which performs a static frequency window localization task. The frequency detection network is obtained with the Grad-CAM method. First the neural network is trained to classify static frequency windows, which are taken from random positions of all the static traces $\forall f \in \mathcal{F}$. We passed to this network windows with length equals to 500 time steps. The dynamic environment that will be tested is realized by switching frequency with durations around 10 times longer than the static windows passed to the localization model.

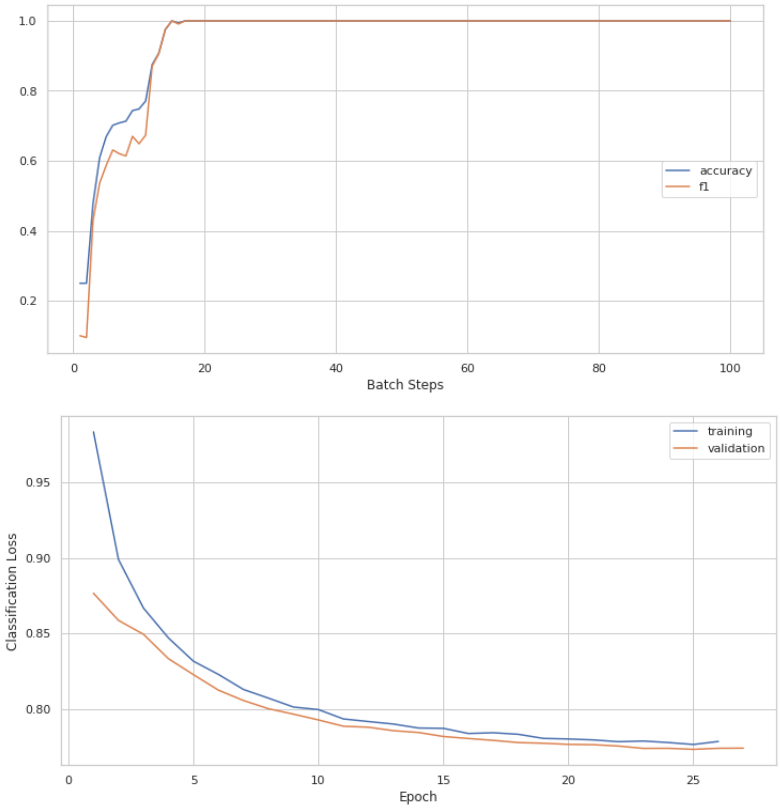

Figure 3.7: Comparison of training accuracy and F1 step scores (top) with relative training and validation epoch loss (right) for ResNet3 discriminator of 48, 52, 54 and 56 MHZ frequencies used by the RDFS actuator

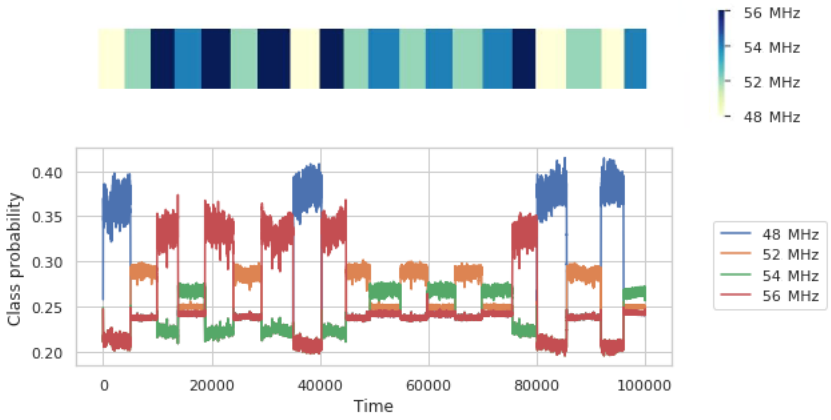

Figure 3.8: Visualization of the class activation map $\boldsymbol{A}$ of a RDFS trace (bottom) and corresponding segmented map $\overrightarrow{s}$ (top), computed using ResNet3 frequency classifier and the Grad-CAM method. A further SoftMax is applied along time steps in order to obtain a probability distribution instead of a score mapping.

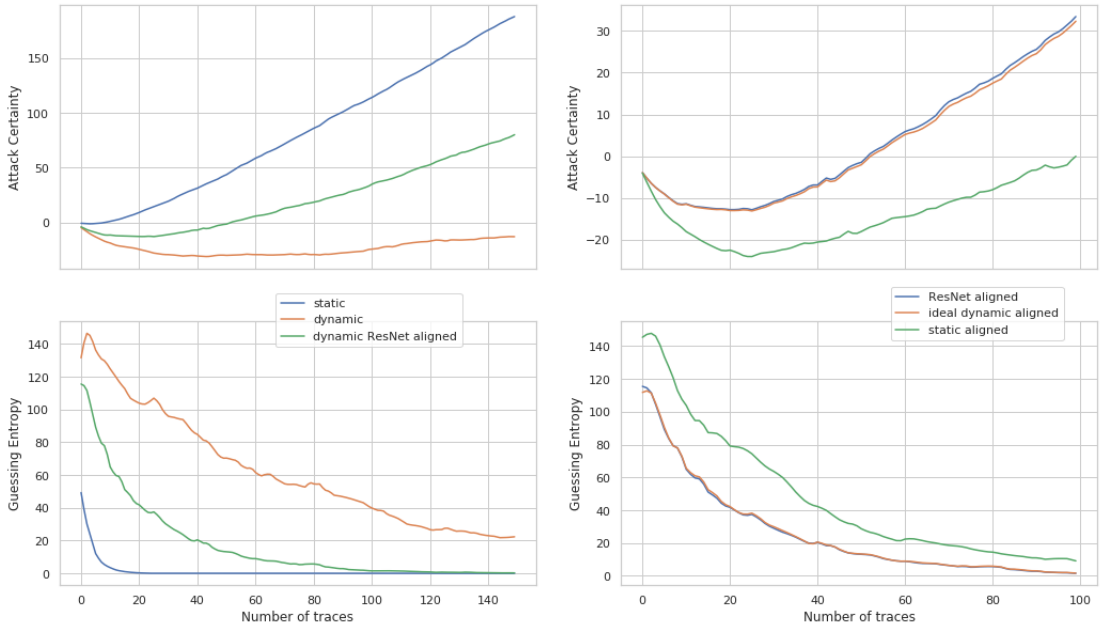

Figure 3.9: Right, a comparison of template attacks on different settings: when attacking on a static 52MHz device (blue), when attacking with the same templates but the device uses RDFS (red) and when attacking the protected implementation with the same templates + ResNet based alignment (green). Left, evaluation of the noise introduced by the interpolation.

Therefore the selected RDFS component randomly changes the 4 available static frequencies in $\mathcal{F} = \{48, 52, 54, 56\}$ [MHz] by sampling a gaussian distributed duration of $T \sim \mathcal{N}ormal(5k, 600)$. The selected detection model for the described RDFS setting is:

- *ResNet3* (42k parameters): 16 filters, kernel size 10, conv stride 1, increase filter gap equals to 1, two residual blocks with max pooling followed by two final residual blocks, $\eta = 10^{-4}$, $\delta = 0.9$, $\gamma = 1$ and Adam

Once trained, the detection network is used in Algorithm 2.6 for the computation of the class activation maps $\boldsymbol{A}$ (Figure 3.8), which are then aggregated into a frequency segmented map $\overrightarrow{s}$. By using Algorithm 2.7, the attacking system exploit the segmentation to obtains the knowledge of where and how the protection is showing up in each leakage. It then shift the frequency of each static window by using Algorithm 2.8 for each detected static frequency window. The overall trace alignment process, which encapsulates perceptive and frequency shift operators, is performed using Algorithm 2.9. The network learned to classifiy the frequency of random windows very easily, in fact it quickly approached F1 scores near 1, while slowly improving through epoches the learned representation without overfitting (Figure 3.7).

experiment	efficiency
dynamic - TA	>200

Table 3.5: Attack results for TA on 48,52,54,46 MHz dynamic device

experiment	IoU
ResNet3 segmentation	98.7

experiment	efficiency
dynamic - ResNet3 aligned TA	120

Table 3.6: Attack results for ResNet3 aligned TA on 48,52,54,46 MHz dynamic device

Because the frequency shift is highly sensitive of the gap between input and output values, the alignment process is performed by bringing back the traces to the middle frequency (52 MHz) available in $\mathcal{F} = \{48, 52, 54, 56\}\,[MHz]$. Once the attacking process reaches this point, the system have removed the RDFS countermeasure, and the data have been reported to a 52 MHZ static device case. Finally the QDA previously fitted for the 52 MHz unprotected device can be apply in order to classify the refined leakage trace. Table 3.5 shows the performance of the plain TA applied to the dynamic case, while table 3.6 presents the performances of the corresponding deep aligned TA. Figure 3.9 provides a full attack comparison of the described TA methods in the given RDFS setting. As it can be clearly seen in the picture, the RDFS component greatly inpact the performances of the TA, practically neglecting the capability of the method to recover the key. However, when the induced desynchronization is removed with the described protection removal method, the bayesian template attack can be considered successful.

A further analysis of the system showed that the mentioned detection network achieve an IoU segmentation score of 98.7, so the resulting localization is very effective, while the interpolation alignment induces a modest amount of noise into the QDA classifier. This is depicted in the right part of Figure 3.9, where the deep aligned TA is compared with (a) the same attack with a priori knowledge of the RDFS manifestation (b) a TA on static 56 MHz device attacked with 52 MHz templates with prior global alignment. As it can be clearly seen, the localization component is very effective, while increasing the amount of interpolation greatly impacts on the attack performance. In final chapter 4 we will outline some possible solutions which can be considered to improve the performance of this type of attack.

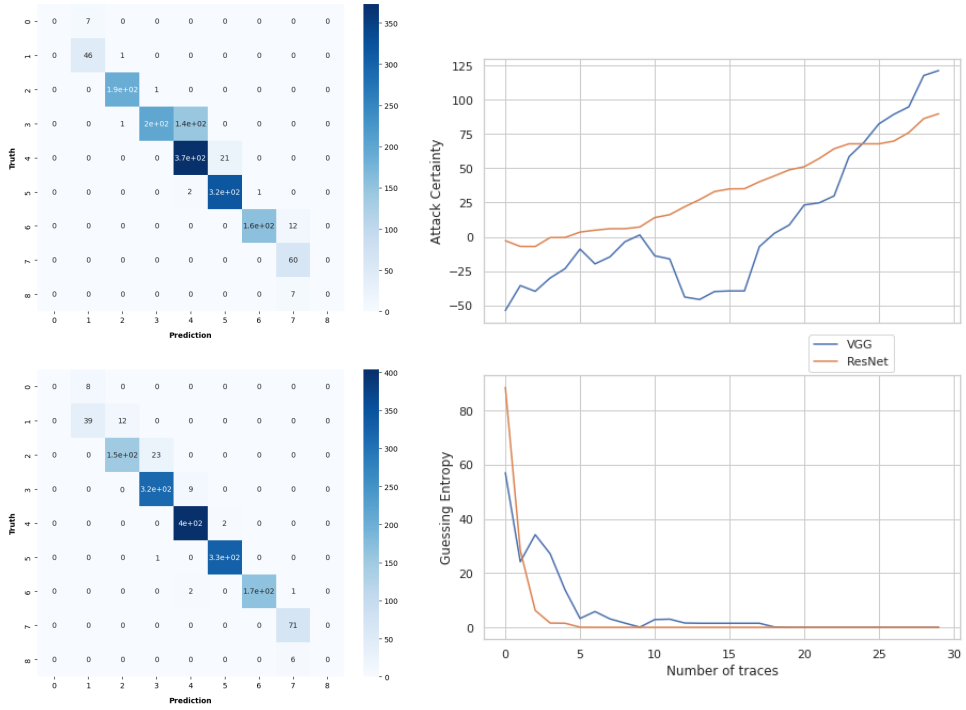

Figure 3.10: Comparison of fully deep attacks on RDFS protected device, using VGG2 (top-left, blue) and ResNet2 (bottom-left, red) models. Left shows CM for validation set.

The final step for attacking the RDFS protected device is to use CNN models in a fully deep setting as in the case of the attack to the unprotected cryptosystem. For this settings, we starts from the same optimal models VGG1 and ResNet1 found from the unprotected device case and progressively change hyper-parameters when needed. The training is performed by feeding RDFS traces instead of static ones. The selected optimal deep models for this task are:

- *VGG2* (9.4M parameters): with 64 base filters, kernel size 30, conv stride 3, two 2-conv starting block, two 3-conv final blocks, $\eta = 5 \cdot 10^{-5}$, $\delta = 0.9$, $\gamma = 2$ and Adam
- *ResNet2* (6.1M parameters): with 32 filters, kernel size 30, conv stride 3, increase filter / downsample gap equals to 2 and 8 residual blocks, $\eta = 5 \cdot 10^{-5}$, $\delta = 0.9$, $\gamma = 2$ and Adam

As showed in [7, 41] for the case of random delays and clock jittering, the RDFS protection ampilfy the generalization capabilities for the model, thus resulting in obtaining a very robust model. The performances of the optimal models are described in tables 3.7 and 3.8. The corresponding full comparison for the attacks involving the two models is showed in Figure 3.10.

experiment	loss	accuracy	F1	efficiency
dynamic - VGG2	1.702	76.8%	76.3%	18

Table 3.7: Attack results for VGG2 on 48,52,54,46 MHz dynamic device

experiment	loss	accuracy	F1	efficiency
dynamic ResNet2	1.664	90.3%	90.1%	5

Table 3.8: Attack results for ResNet2 on 48,52,54,46 MHz dynamic device

The performed evaluation shows that both convolutional models, unlike the TA, are able to efficiently detect the leakages even with RDFS protection. The ResNet is very effective, and achieve a successful attack after 5 observations. The VGG is also very good in founding the correct key, however it seems to be a bit more unstable w.r.t. ResNet, in fact it achieves the attack after 17 observations. From the results obtained, it is clear that convolutional models outperforms classical profiled methods such as the bayesian template attack. This performance gap is greatly amplified in situations where there are frequency scaling based protections. Deep learning based models provide a very efficient leakage modelling framework w.r.t to classical techniques, at least in the context of power consumption analysis.

4 | Conclusions and future developments

Side-channel attacks can be achieved with minimal efforts by using convolutional neural networks. In particular, these models have been showed to be extremely effective when attacking platforms with hardware countermeasures. In this context, the power of convolutional networks greatly enhance attacking capabilities of the adversarial w.r.t classical models such as the gaussian Template Attack. Moreover, other Deep Learning techniques can also be used in hybrid attacking settings, where localization based systems are employed to enhance classical side-channel attacking models, which fail when performing the power analysis in the presence of desynchronizations.

With respect to the fully deep attack, the deep aligned strategy can provide some practical benefits to the adversarial. It allows for a previously optimized and working method used for unprotected implementations to still be able to attack protected systems. Because neural power is only used for its protection removal capabilities, the resulting network is much smaller than the end-to-end sensitive variable classifier. Because the templates as well weight much less than the fully deep predictor, the overall hybrid deep aligned attacking system uses far less space, but requires observing much more traces w.r.t the fully deep approach. Therefore, an attacker could exploit this computational trade-off between space and time, for example in hypothetical scenarios where few computing resources are available.

For what it concerns future developments, the fully deep attacking system can be greatly improved by introducing some Domain Adaptation in the leakage modelling, by adapting the unprotected representation with the protected RDFS environment. The deep aligned attacking system can instead be improved by replacing the linear interpolation method for side-channel trace alignment. Some techniques which can be investigated for this purpose are based on shifting the signal in the frequency domain using the Fast Fourier Transform algorithm. The same task can also be accomplished by using models based on regression or other Deep Learning models such as the ones based on denoising autoencoders.

Bibliography

[1] NewAE CW305 Artix FPGA Target. URL `https://rtfm.newae.com/Targets/CW3052021`.

[2] Lamp-platform, 2019. URL `http://www.lamp-platform.org/`.

[3] C. Archambeau, E. Peeters, F. X. Standaert, and J. J. Quisquater. *Template Attacks in Principal Subspaces*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-46561-4.

[4] T. Bartkewitz and K. Lemke-Rust. *Efficient Template Attacks Based on Probabilistic Multi-class Support Vector Machines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-37288-9.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] E. Brier, C. Clavier, and F. Olivier. *Correlation Power Analysis with a Leakage Model*, volume 3156. Springer Berlin Heidelberg. ISBN 978-3-540-22666-6 978-3-540-28632-5. doi: 10.1007/978-3-540-28632-5_2.

[7] E. Cagli, C. Dumas, and E. Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures – profiling attacks without pre-processing –. (740). URL `http://eprint.iacr.org/2017/740`.

[8] E. Cagli, C. Dumas, and E. Prouff. *Kernel Discriminant Analysis for Information Extraction in the Presence of Masking*, volume 10146 of *Lecture Notes in Computer Science*. Springer, Cannes, France, Nov. 2016. doi: 10.1007/978-3-319-54669-8_1. URL `https://hal.archives-ouvertes.fr/hal-01532186`.

[9] S. Chari, J. R. Rao, and P. Rohatgi. *Template Attacks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-36400-9.

[10] O. Choudary and M. G. Kuhn. Template attacks on different devices. In E. Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, volume 8622. Springer International Publishing. ISBN 978-3-319-10174-3 978-3-319-10175-0.

doi: 10.1007/978-3-319-10175-0_13. URL http://link.springer.com/10.1007/978-3-319-10175-0_13. Series Title: Lecture Notes in Computer Science.

[11] J. Coron, C. Paar, C. Hardware, J.-s. Coron, and L. Goubin. On boolean and arithmetic masking against differential power analysis.

[12] V. V. Cortes Corinna. *Support-vector networks*. Springer. URL https://doi.org/10.1007/BF00994018.

[13] J. Doget, E. Prouff, M. Rivain, and F.-X. Standaert. Univariate side channel attacks and leakage modeling. (302). URL http://eprint.iacr.org/2011/302.

[14] W. Falcon. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning Cited by*, 3, 2019.

[15] R. A. Fisher. The use of multiple measurements in taxonomic problems. 1936.

[16] A. H. A. V. Gabriel Zaid, Lilian Bossuet. Methodology for efficient cnn architectures in profiling attacks. doi: 10.13154/tches.v2020.i1.1-36.

[17] K. Gandolfi, C. Mourtel, and F. Olivier. *Electromagnetic Analysis: Concrete Results*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44709-2.

[18] B. Gierlichs, L. Batina, B. Preneel, and I. Verbauwhede. Revisiting higher-order DPA attacks: Multivariate mutual information analysis. (228). URL http://eprint.iacr.org/2009/228.

[19] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. Mutual information analysis a generic side-channel distinguisher. *Lecture Notes in Computer Science*, 5154:426–442, 2008.

[20] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[21] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[22] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. URL http://arxiv.org/abs/1512.03385.

[23] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. 2(5):359–366. ISSN 08936080. doi: 10.1016/0893-6080(89)90020-8. URL https://linkinghub.elsevier.com/retrieve/pii/0893608089900208.

[24] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. URL http://arxiv.org/abs/1502.03167.

[25] Joblib Development Team. Joblib: running python functions as pipeline jobs, 2020. URL https://joblib.readthedocs.io/.

[26] M. Joye, P. Paillier, and B. Schoenmakers. On second-order differential power analysis. pages 293–308.

[27] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. URL http://arxiv.org/abs/1412.6980.

[28] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. pages 388–397, 1999.

[29] P. C. Kocher. Timing attacks on implementations of Diffiee-Hellman, RSA, DSS, and other systems. 1996.

[30] L. Lerman, G. Bontempi, and O. Markowitch. Power analysis attack: an approach based on machine learning. 3, . ISSN 1753-0563, 1753-0571. doi: 10.1504/IJACT.2014.062722. URL http://www.inderscience.com/link.php?id=62722.

[31] L. Lerman, R. Poussier, O. Markowitch, and F.-X. Standaert. Template attacks vs. machine learning revisited. .

[32] C. D. Loïc Masure and E. Prouff. *Gradient visualization for general characterization in profiling attacks*, volume 11421 of *Lecture Notes in Computer Science*. Springer, 2019.

[33] H. Maghrebi, T. Portigliatti, and E. Prouff. *Breaking Cryptographic Implementations Using Deep Learning Techniques*, volume 10076 of *Lecture Notes in Computer Science*. Springer, 2016. ISBN 978-3-319-49444-9. doi: 10.1007/978-3-319-49445-6_1. URL http://dx.doi.org/10.1007/978-3-319-49445-6_1.

[34] S. Mangard, N. Pramstaller, and E. Oswald. *Successfully Attacking Masked AES Hardware Implementations*, volume 3659. Springer Berlin Heidelberg. ISBN 978-3-540-28474-1 978-3-540-31940-5. doi: 10.1007/11545262_12. URL http://link.springer.com/10.1007/11545262_12.

[35] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[36] T. S. Messerges. *Using Second-Order Power Analysis to Attack DPA Resistant Software*, volume 1965 of *Lecture Notes in Computer Science*. Springer, 2000. doi: 10.1007/3-540-44499-8_19.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[38] K. Pearson. On lines and planes of closest fit to systems of points in space. 1901.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] X. L. Pei Cao, Chi Zhang and D. Gu. Cross-device profiled side-channel attack with unsupervised domain adaptation. doi: 10.46586/tches.v2021.i4.27-56.

[41] 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. URL `http://eprint.iacr.org/2018/053`.

[42] L. Rokach and O. Maimon. *Data mining with decision trees: theory and applications*. World Scientific, second edition edition. ISBN 978-981-4590-07-5.

[43] W. Schindler. Advanced stochastic methods in side channel analysis on block ciphers in the presence of masking. *Journal of Mathematical Cryptology*, 2(3):291–310, 2008. doi: doi:10.1515/JMC.2008.013. URL `https://doi.org/10.1515/JMC.2008.013`.

[44] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. 128 (2):336–359. ISSN 0920-5691, 1573-1405. doi: 10.1007/s11263-019-01228-7. URL `http://arxiv.org/abs/1610.02391`.

[45] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. URL http://arxiv.org/abs/1409.1556.

[46] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. page 30, 2014.

[47] F.-X. Standaert and C. Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154, pages 411–425. Springer Berlin Heidelberg. ISBN 978-3-540-85052-6 978-3-540-85053-3. doi: 10.1007/978-3-540-85053-3_26. URL http://link.springer.com/10.1007/978-3-540-85053-3_26. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science.

[48] F.-X. Standaert, T. Malkin, and M. Yung. A formal practice-oriented model for the analysis of side-channel attacks. 07 2006.

[49] E. O. Stefan Mangard and T. Popp. Power analysis attacks - revealing the secrets of smart cards. 2007.

[50] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17: 261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[51] S. Yang, W. Wolf, N. Vijaykrishnan, D. N. Serpanos, and Y. Xie. Power attack resistant cryptosystem design: A dynamic voltage and frequency switching approach. 3.

[52] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization. URL http://arxiv.org/abs/1512.04150.

[53] Y. T. Zhou and R. Chellappa. Computation of optical flow using a neural network. *IEEE 1988 International Conference on Neural Networks*, pages 71–78 vol.2, 1988.

List of Figures

List of Tables

List of Symbols

Variable	Description
$\hat{f}$	generative distribution
$\hat{g}$	predictive distribution
X	feature variable
Y	target or hidden variable
$\boldsymbol{W}$	model weights
P	plain text
K	secret key
Z	sensitive variable
ψ	leakage model function
HW	Hamming weight function
$\overrightarrow{L}$	leakage trace
$\overrightarrow{M}$	mean leakage trace
$\overrightarrow{H}$	dimensional reduced leakage trace
$\overrightarrow{W}$	leakage trace window
$\boldsymbol{A}$	class activation maps
$\overrightarrow{s}$	segmented trace
$\mathcal{L}$	optimization loss
$\mathbb{I}, \mathbb{H}$	Shannon information and entropy functions
$\mathcal{KL}$	Kullback-Leibler divergence
η	learning rate
δ	frequency of learning rate scheduling
γ	learning rate step scheduling coefficient