POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Hardware Design of a Homomorphic-like Encryption Scheme for Spiking Neural Networks

Supervisor

Prof. Guido MASERA

Prof. Maurizio MARTINA

Project Ass. Alberto MARCHISIO

Candidate

Gianluca MENDITTO

07 2022

Summary

The last decades have seen a significant development of Artificial Neural Networks (ANNs). These advancements have enabled the application of ANNs to various domains, such as handwriting and speech recognition, or classification and computer vision tasks. The basic idea is to emulate the human brain in analyzing and processing information to solve several tasks, especially if they are hardly described formally.

Depending on the use cases, different kinds of neural networks are employed. For instance, the Convolutional Neural Networks (CNNs) are widely used in classification tasks. However, the high power effort required by these NNs has pushed the research toward the low power domain. A new paradigm of NNs, named the Spiking Neural Networks (SNNs) have demonstrated promising results in terms of performance and efficiency. The advantage is the different way in which the neurons communicate. In fact, since in SNNs the information is propagated through spikes that can be represented with a single bit, the computation is lighter. Moreover, these networks can closely mimic the human brain, becoming the most biologically-plausible NNs model.

The success achieved by NNs certainly lies in the great availability of data. On the contrary, these datasets may contain sensitive information which make it challenging from the privacy perspective to maintain the confidentiality of data. An efficient solution is represented by the Homomorphic Encryption (HE), a cryptographic method that allows performing computations over encrypted data instead of its raw version. The data owner encrypts the data and sends them to an SNN to obtain an encrypted prediction. The application of HE to NNs leads to ensure privacy both on the data and on the prediction since only the data owner can access their actual value. A HE scheme includes four algorithms: the key generation, the encryption,

the decryption and the evaluation. The last three processes are extremely complex and require a high computational effort.
In this thesis, a Somewhat Homomorphic Encryption(SHE) scheme is presented. Despite allowing only a limited number of computations over encrypted data, it is sufficient and convenient since this scheme has a lower overhead than a complete Fully Homomorphic Encryption (FHE) scheme. In this thesis the design of a SHE system for SNNs using the Brakerski-Fan-Vercauteren (BFV) scheme is conducted. In particular, the implementation of the algorithm is carried out partially in Python and partially in VHDL. The key and the other parameters generations are implemented in Python; these functions require a random sampling in a specific distribution to obtain a set of variables which are put as inputs to the different blocks. The encryption, the decryption and addition between ciphertexts blocks, are implemented in VHDL. The three above-discussed units includes different operations between polynomials, such as addition, multiplication and division. While the first two are explicitly involved in homomorphic equations, the division is useful to scale the size of the polynomial and its coefficients after the other two operations. The polynomial multiplication results in the most complex operation both in terms of resources and clock cycles. For this reason, the encryption unit which performs the multiplication over two input data is the most computationally-heavy block. Instead, the decryption unit is simpler since it performs operations only over one data, namely the ciphertext. The addition between ciphertexts is the computationally lightest operation, since it is very close to the standard addition between polynomials, except for an additional scaling operation.
The whole design is simulated with *Modelsim* to test its correct behaviour.
The last step is the logical synthesis with *Synopsys Design Compiler* to evaluate the performance of the design in terms of timing, area and power; moreover, the optimizations are applied to improve the above-discussed performance. As expected, the multiplication operation shows high complexity since it requires a high number of clock cycles, and the encryption unit consumes higher resources than the other blocks.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

artificial intelligence

ML

machine learning

ANN

artificial neural network

DL

deep learning

CNN

convolutional neural network

SNN

spiking neural network

IF

Integrate and Fire

LIF

Leaky-integrate and fire

STDP

Spike timing dependent plasticity

LTP

Long term potentiation

LTD

Long term depression

MIA

Member inference attack

DP

Differential privacy

HE

Homomorphic Encryption

SMC

Secure Multi-Party Computation

DPFE

Deep Private Feature Extraction

FL

Federated Learning

TEE

Trusted Execution Environment

FHE

Fully Homomorphic Encryption

PHE

Partially Homomorphic Encryption

SHE

Somewhat Homomorphic Encryption

LWE

Learning With Errors

RLWE

Ring Learning With Errors

MAC

Multiply-Accumulate

NTT

Number theoretic transform

Chapter 1

Introduction

Nowadays the terms *Artificial intelligence* (**AI**), *Machine learning* (**ML**), *Artificial neural networks* (**ANN**) and *Deep learning* (**DL**) are often improperly used to mean the same thing, but actually, they have different meanings even though they are related to each other to some extent.

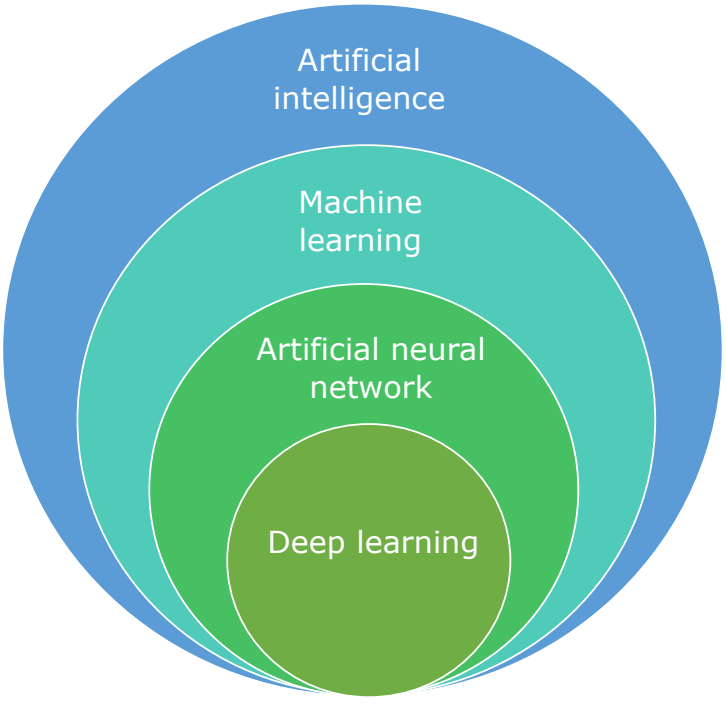


Figure 1.1: Relation among AI, ML, ANN, DL

Figure 1.1 shows the relation among the above-discussed terms. Starting from the top, AI is a field of computer science that enables machines to solve tasks as humans do. All the others can be considered components of the prior term.
Initially, the artificial intelligence aimed at solving problems that are intellectually difficult for humans but relatively simple for computers; these problems can be described using a mathematical approach. The true challenge to AI became solving the tasks that are easily tackled by the human brain but hardly described formally. The machine learning proposes to overcome this limitation [1]. ML seeks to make intelligent systems that can automatically learn through experience, without being explicitly programmed or requiring any human intervention.
A branch of ML is the ANN which aims to emulate the human brain in analyzing and processing information to solve several tasks. At the same time, ANNs are still different from a biological point of view compared to the actual brain. In this sense, the *Spiking neural networks* (**SNNs**) represent the most biologically-plausible ML models based on neurons and synapse interaction.
A subset of ANN is the DL which aims to make self-teaching system. The starting point for realising DL is a complex ANN.

This chapter provides a more detailed overview of ANNs and SNNs.

1.1 Artificial neural network - ANN

Artificial neural networks, inspired by the human brain, are characterized by interconnected artificial neurons that employ some mathematical rules to generate the output which in turn can propagate along with the network. One of the earliest and most basic ANN models is the perceptron [2]. Perceptrons were developed by Frank Rosenblat in 1957 and were considered the most notable invention of AI.
A perceptron emulates the behaviour of a neuron which takes many binary inputs $x_1, x_2, ...x_n$ and produces only one binary output. Figure 1.2 compares the structure of a typical biological neuron and an artificial one.

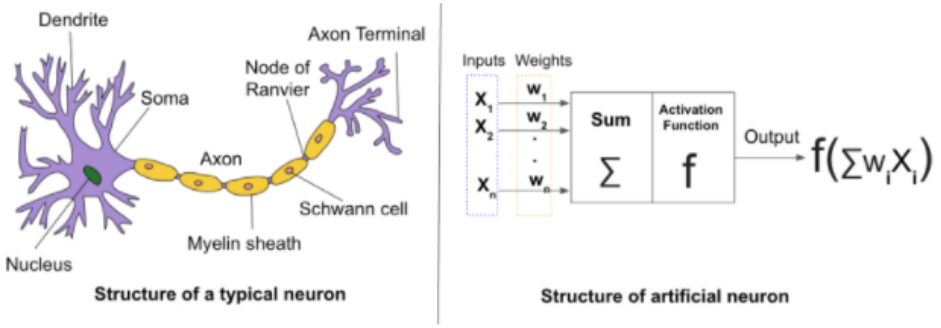


Figure 1.2: Biological neuron vs artificial neuron

The following list defines the main components of the neuron:

- **dendrites**: or inputs; they permit the receipt of signals from other neurons. These inputs are multiplied by that dendrite's *weight*.
- **soma** or body: it behaves like the sum function of the signals coming from dendrites. They can be positive or negative (excitatory or inhibitory, respectively), increasing or decreasing the membrane potential.
- **axon**: it represents the output of the neuron. Each neuron has an axon that can have many terminals to connect it to the dendrites of other neurons through a synapse.

The neuron's output is determined according to a nonlinear function, named the *activation function*. The simplest case is a step function:

$$output = \begin{cases} 0 \text{ when } \sum_j w_j x_j \leq threshold \\ 1 \text{ when } \sum_j w_j x_j > threshold \end{cases} \tag{1.1}$$

The term x_j represents the input, while w_j is the *weight*, which is a real number defining the relevance of the respective input to the output. When the output is equal to one, it means the membrane potential exceeds the threshold and so the neuron fires and produces an output signal called *action potential* while the membrane is reset to a rest value.

The weight is specific for each synapse, so it leads to increasing or decreasing the sum according to its value. Hence, some inputs have a more significant effect on the output. Therefore, by varying the weights (and eventually the threshold), it is possible to get different models of decision-making. This property of strengthening or weakening the synapse is known as *plasticity*.

The model described so far represents the starting point for the building of the simplest ANN.

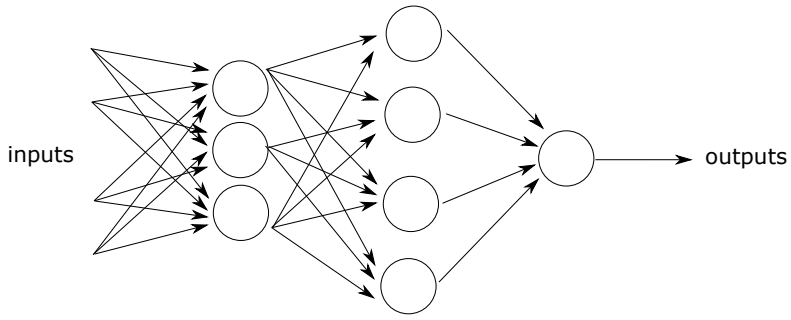


Figure 1.3: Generic ANN

It is organized in a maximum of three *layers* and in a so-called *fully-connected graph*: in other words, each node of a layer is connected to all the nodes of the previous and following layers. More in detail, they are distinguished in:

- **first layer**: it is composed of nodes that represent the inputs of the network;
- **last layer**: it is the last column of neurons where they are not connected to other nodes. They provide the result of the network;
- **hidden layer**: it is between the first and output layer. In more complex ANNs, there can be multiple hidden layers.

There exists another mathematical way to describe the perceptrons that simplifies the writing; the first modification is to write $\sum_j w_j x_j$, as a dot product $w \cdot x$, where w and x are vectors whose components are the weights and inputs, respectively. Moreover, the threshold is moved to the left-hand side of the inequality (1.1) and now it is indicated as the perceptron's bias, $b = -threshold$. The consequent

expression becomes:

$$output = \begin{cases} 0 \text{ if } w \cdot x + b \leq 0 \\ 1 \text{ if } w \cdot x + b > 0 \end{cases} \tag{1.2}$$

The bias can be thought to how easy it is to get the output equal to 1, or rather how easy it is to get the neuron to fire. In fact, if the bias is large, it is very easy to obtain 1 as output; vice-versa if the bias is small or even negative.

1.1.1 Convolutional neural network - CNN

There are several kinds of artificial neural networks, which are used for different use cases and data types [3]. For instance, the *recurrent NNs* where the neurons can be connected with loops and generally are used for handwriting or speech recognition; whereas *convolutional neural networks* have a connectivity pattern of the neurons analogous to the human brain, exploiting the temporal and spatial correlation among input data; the basic idea is to offer a more scalable approach to classification and computer vision tasks.

A *ConvNet* has three main types of layers:

- **convolutional** layer
- **pooling** layer
- **fully-connected** (*FC*) layer

There can be one or more convolutional layers but of course, the *FC* is the last layer of the network. Earlier layers are in charge of detecting the simplest features of an image, and as going through the layers the complex features are detected. Let's describe each layer more in detail.

The convolutional layer is the most important component of the network because it is responsible for the majority of the computation. The key operation is the application of a specific filter, to a sub-portion of the input data. The filter is characterized by a size, called *kernel*, which defines the dimensions of the

subwindow of the image, named *receptive field*. The application of the filter means to multiply the kernel weights with the corresponding pixel of the image, and then sum them. The result, known as *feature map*, is so obtained. An example of the calculation is shown in Figure 1.4.

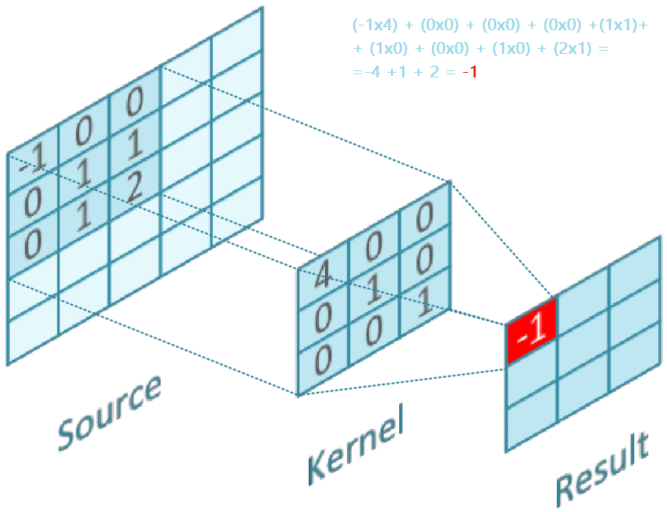


Figure 1.4: Numerical example of a convolutional filter application with a 3x3 kernel

The filter is shifted step by step in both x and y directions as far as the kernel has covered the entire image with a pitch, named *stride*, that defines the new receptive field progressively.
Moreover, an image can have multiple channels: in this case, the filter must have the same number of channels of the input data. Each channel is characterized by a different set of weights and the feature map is computed summing up the result from each channel. Sometimes there is also another contribution to the sum, known as *bias*. Figure 1.5 provides a better understanding of the process.
In that case, in addition to the convolution, also an operation of *zero-padding* is applied; in general, it is useful to modify the size of the output. There are three different kinds of zero-padding:

- **valid padding**: the last convolution is dropped if the dimensions do not align.
- **same padding**: this padding ensures that the output layer has the same size as the input layer.

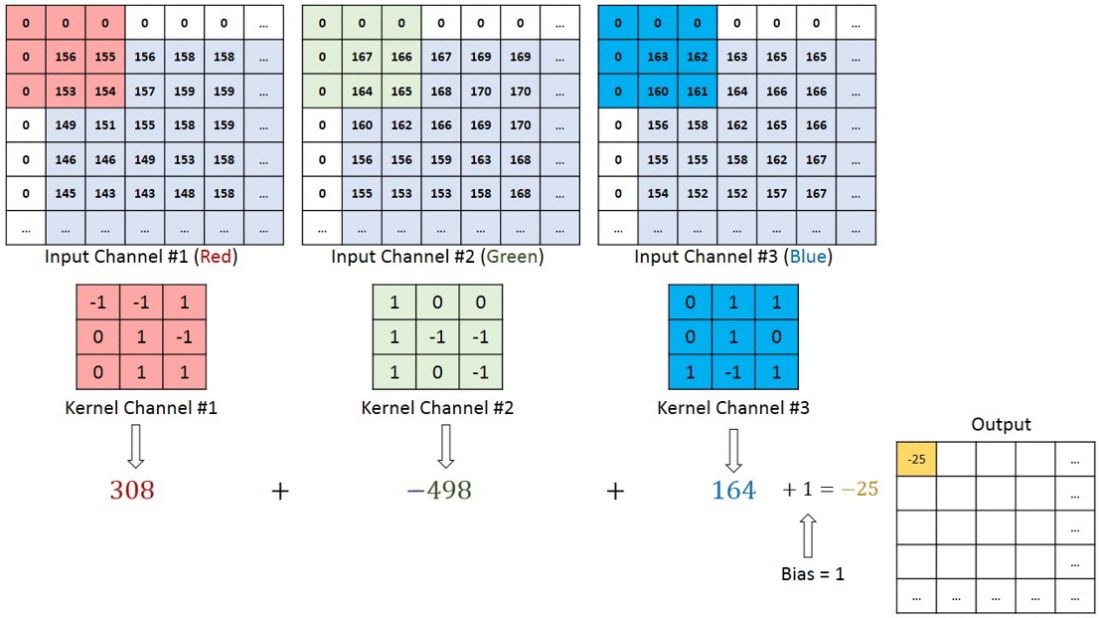


Figure 1.5: Numerical example of a convolutional filter application with multi channel and an addition of a bias value [Source: [4]]

- **full padding**: this kind of padding increases the size of the output by adding zeros to the border of the input.

The example in Figure 1.5 shows a full padding application.

A convolutional layer can be followed by another convolutional layer or a pooling layer. The latter aims to reduce the size of the feature map. Similarly to the convolutional layer, a filter is swept across the whole image, but this time it has no weights. Rather an aggregation function is applied to the value within the receptive field to get the output. As shown in Figure 1.6, there are two main types of pooling:

- **max pooling**: it returns the maximum value of a pixel from the subwindow of the image swept by the filter.
- **average pooling**: it computes the average value of the receptive field.

The last layer of a ConvNet is a fully-connected layer. Unlike the previous layers, where the outputs are not connected directly to each pixel in the input image,

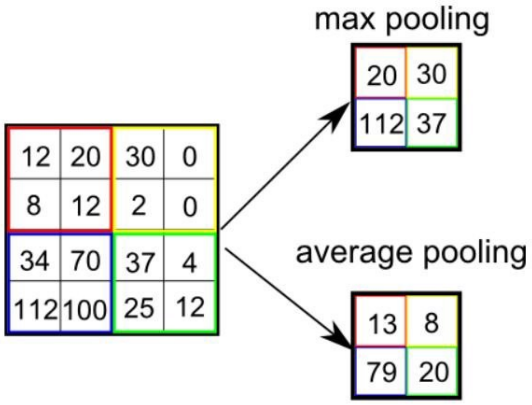


Figure 1.6: Types of pooling [Source: [4]]

instead here this happens, as the name itself describes. The image is flattened in a column vector.
This layer classifies the features extracted through the previous layers and their different filters.

1.1.2 Spiking neural network - SNN

Spiking neural networks, referred as the *third generation of NNs*, closely mimic the human brain. The main difference with ANN is the information propagation approach: the neurons communicate via spikes, discrete events which happen at defined times. The general idea is that the SNN takes as input a train of spikes and generates a train of spikes as output.

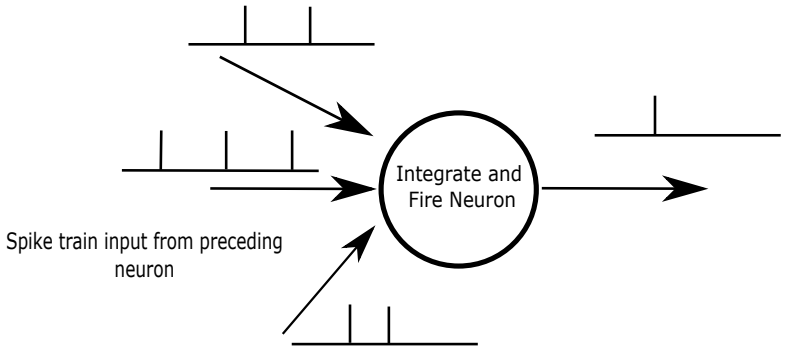


Figure 1.7: Schematic depiction of a spiking neuron [Source: [5]]

As shown in Figure 1.7, the neuron receives several input spikes from the previous layers' neurons. These spikes are modulated with weights and accumulated as the neuron membrane potential. As soon as the membrane potential crosses a certain threshold, the neuron emits a spike to the following layer and the membrane potential is reset.
The SNN requires data are coded into spikes trains [5]. To perform this conversion, the concept of *time* is introduced in the model. In fact, there are different neural coding strategies based on spike timing. The main three ones are:

- **time to first spike**: the information is encoded as the time between the stimulus and correspondent output.
- **rank-order coding**: the information is encoded by the order of spikes from a population of neurons.
- **latency code**: the information is coded as the delay between two subsequent spikes.

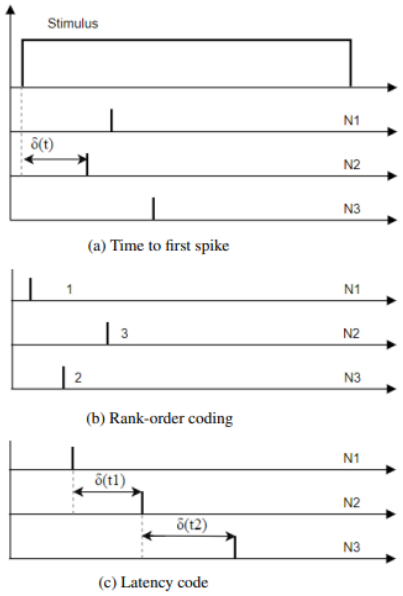


Figure 1.8: Different encoding strategies of information in spikes [Source: [5]]

Chapter 2

Mathematical models of the membrane potential

There are a lot of mathematical models to describe more or less faithfully the behaviour of the neuron. These models can be classified into different categories: the most interesting is the ″Electrical input-output membrane voltage"; the models in this category illustrate the link between neuronal membrane currents, evaluated at the input stage, and membrane voltage, evaluated at the output stage. This category includes the biophysical models, such as the **Hodgkin-Huxley** model, which depict the membrane voltage as a function of the input current and the activation of ion channels, and the simpler mathematical models, such as the **Integrate and Fire** model, that represent the membrane voltage as a function of the input current and predict spike times without biophysical descriptions.

2.1 Hodgkin-Huxley (H&H) model

The Hodgkin–Huxley model [6] depicts the relationship between the flow of ionic currents across the neuronal cell membrane and the membrane voltage of the cell. It can be described through the equivalent circuit in figure 2.1. The current can be brought through the membrane either by charging the membrane capacity or by the flow of ions through the resistances placed in parallel with the capacitance.

Mathematically [6], it can be formulated through Equation 2.1:

$$I = C_M \frac{dV}{dt} + I_i \tag{2.1}$$

where C_M represents the capacitive contribution and the I_i the ionic current that can cross the membrane.

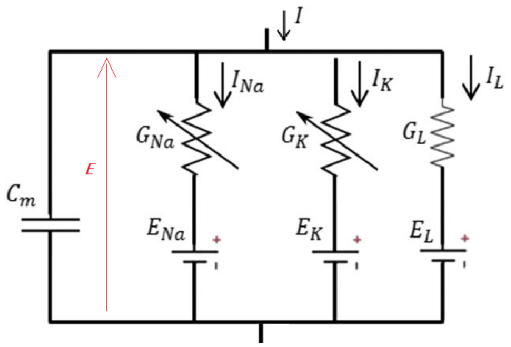


Figure 2.1: Equivalent circuit diagram of the Hodgkin-Huxley neuron [Source: [7]]

The ionic current consists of different components, the sodium and potassium ions, respectively I_{Na} and I_K, and one small leakage current I_L, made by chloride and other ions. Each component can be computed as the correspondent conductance multiplied by the electrical potential difference, given by the membrane potential (E) and the equilibrium potential (E_{Na}, E_K or E_L). For practical applications it is convenient to rewrite the above relation in a different way assuming that:

$$V = E - E_r \tag{2.2}$$

$$V_{Na} = E_{Na} - E_r \tag{2.3}$$

$$V_K = E_K - E_r \tag{2.4}$$

$$V_L = E_L - E_r \tag{2.5}$$

where E_r is the absolute value of resting potential. The ionic current can be described through Equation 2.6:

$$I_i = I_{Na} + I_K + I_L = g_i(V - V_i) \tag{2.6}$$

Where the term g_i is the conductance. The components related to potassium and sodium are variable and depend on membrane potential; they can be expressed as their maximum conductance $\bar{g}$ and the activation and inactivation fractions m and h. The total current through the membrane is given by Equation 2.7:

$$I = C_M \frac{dV}{dt} + \bar{g}_K n^4 (V - V_K) + \bar{g}_{Na} m^3 h (V - V_{Na}) + \bar{g}_L (V - V_L) \tag{2.7}$$

N, m and h can assume a value between 0 and 1 and can be computed through differential equations, which combined to Equation 2.7, complete the set of the equations of the model.
The Hodgkin-Huxley neuron represents the closest model to our current understanding of the actual biological neurons but its mathematical complexity makes it unsuitable to be used in practical applications.

2.2 Integrate and Fire (IF) model

The Integrate and Fire is one of the earliest models of neurons, and can be traced back to Louis Lapicque in 1907 [8].
The *IF* model can be sketched with a simple electric circuit, shown in Figure 2.2:

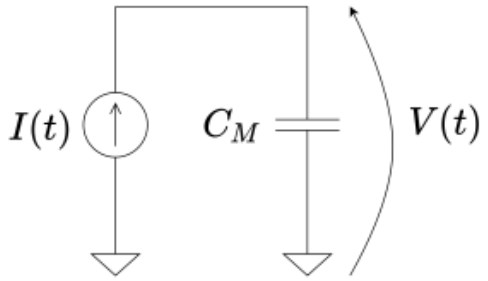


Figure 2.2: Equivalent circuit diagram of the IF model

Modeled as in Equation 2.8, the neuron's membrane behaves as a capacitor ideally isolated, in which the voltage represents the membrane potential, and the source

current represents the input from all the synapses.

$$I(t) = C_M \frac{dV(t)}{dt} \tag{2.8}$$

When the input current is applied, the voltage increases up to reach a specific value, called threshold voltage. At that point an output current spike is produced and the voltage is reset to its resting value.
Then the membrane starts to increase again as new spikes arrive at the input.
This model, whose behavior is shown in Figure 2.3, results the simplest one and does not have any bio-plausible reference.

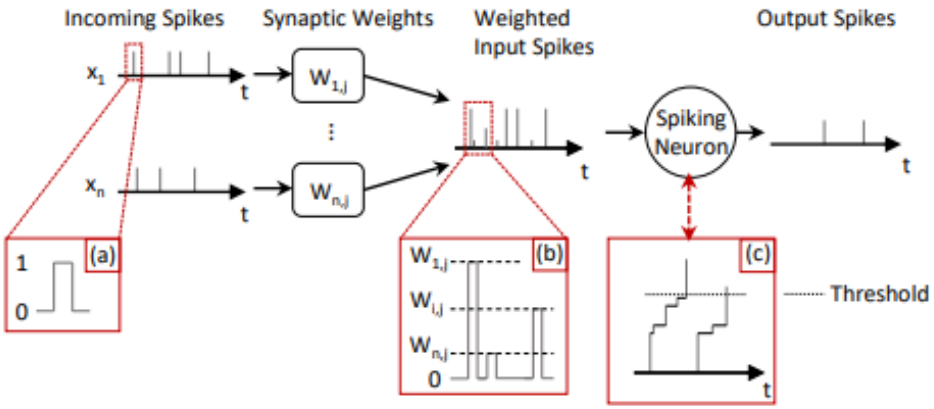


Figure 2.3: Schematic representation of the IF model [Source: [9]]

2.2.1 Leaky-integrate and fire (LIF) model

The Leaky-integrate and fire (LIF) neuron represents an improvement over the IF model, by introducing two new concepts, the *leakage* and *refractory period*. The first concept derives from the removal of the ideal isolation condition: the membrane, if it is not stimulated with any input, exhibits a leakage which tends to discharge the capacitance along the time. This behaviour is shaped by introducing a resistor in parallel to the capacitor, as shown in Figure 2.4: Hence, the LIF model can be described with the Equation 2.9, where R_M is the membrane resistance:

$$I(t) = C_M \frac{dV(t)}{dt} + \frac{V(t)}{R_M} \tag{2.9}$$

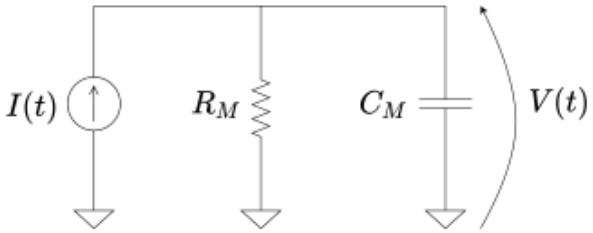


Figure 2.4: Equivalent circuit diagram of the LIF model

The other new feature relates to the fact that as the membrane reaches the threshold is reset. After this reset, there is an amount of time, called refractory period (T_R), during which the neuron remains in a sort of quiet state: all the spikes received within this time window are ineffective, even though the membrane potential is beyond the threshold. Only when the T_R is elapsed, a new spike can be emitted. Figure 2.5 depicts such a behaviour. In summary, the essential parameters of a

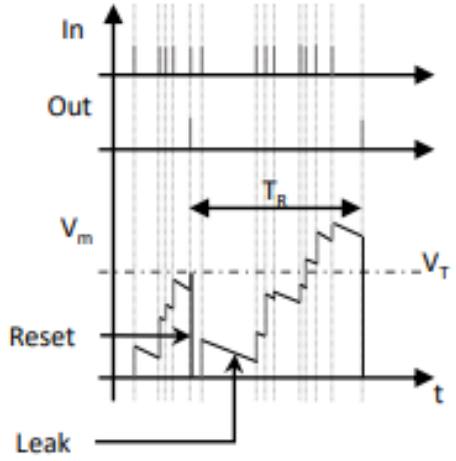


Figure 2.5: Schematic representation of the LIF model [Source: [9]]

LIF neuron are the membrane threshold voltage, the reset potential, the refractory period and the leak rate. At each time step t, the membrane potential can be described by Equation 2.10, where $x_{i,j}(t-1)$ is the input spike from the neuron i

in the previous layer to neuron j in the current layer and the refractory period[9].

$$V(t) = V(t-1) + \sum_i w_{i,j} x_{i,j}(t-1) - \lambda \tag{2.10}$$

The parameter λ corresponds to the leak, and $w_{i,j}$ is the weight. The output spike of a neuron can be expressed by Equation 2.11:

$$x(t) = \begin{cases} 1 \text{ if } V(t) > threshold \text{ and } t - t_{spike} > T_R \\ 0 \text{ otherwise} \end{cases} \tag{2.11}$$

where t_{spike} is the time step at which the neuron fired.

2.3 Izhikevich model

The Izhikevich model, designed by Eugene M. Izhikevich in 2003, combines the biological plausibility of the Hodgkin–Huxley-type dynamics and the computational efficiency of integrate-and-fire neurons [10]. It is described through a 2-D system of differential equations (Equations 2.12 and 2.13), which can be considered as a reduction of the more accurate H&H model:

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{2.12}$$

$$\frac{du}{dt} = a(bv - u) \tag{2.13}$$

With the after-spike resetting formula in Equation 2.14.

$$\text{if } v \geq 30\ mV, \text{ then} \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{2.14}$$

Where all the terms are dimensionless:

- v stands for the membrane potential of the neuron
- u symbolizes the recovery variable, which takes into consideration for the activation of K+ ionic currents and inactivation of Na+ ionic currents, and it produces negative feedback to v

- I symbolizes the synaptic currents and injected dc-currents
- a describes the time scale of the recovery variable u. The smaller the values, the slower the recovery
- b describes the sensitivity of the recovery variable u to the subthreshold swing of the membrane potential v. Greater values couple v and u more strongly providing possible subthreshold fluctuations and low-threshold spiking dynamics
- c represents the after-spike reset value of the membrane potential v
- d represents the after-spike reset of the recovery variable u

The part corresponding to $0.04v^2 + 5v + 140$ in Equation 2.12 derives by fitting the spike initiation dynamics of real neurons, in such a way the membrane potential v has mV scale and the time t has ms scale.
As the spike amounts to its peak $(+30mV)$, the membrane voltage and the recovery variable are reset according to Equation 2.14.

Chapter 3

Learning process

One of the most important problems which affects the performance of the network is to make it able to recognize the input data and classify them: in other words, a **training** has to be performed.
In order to train a NN, a **dataset** is necessary; a dataset is a collection of data of different kinds as images, sounds, texts and so on. They normally are divided into two groups: the *train set*, which contains data actually used for training and, the *test set*, which contains data not provided to the network during training, but they are used to evaluate the network. Let's see how the training process happens.

3.1 Supervised and unsupervised learning

The training process can be of two main types:

- **supervised**: each input is associated with a label that identifies which class it belongs to. The training is performed knowing in advance the expected output.

- **unsupervised**: the network independently determines the features and becomes capable to classify the input data.

In the former method, to compare the actual predicted output with the expected one, a *cost function* is used. It is also called loss function and can be expressed

through Equation 3.1:

$$C(w,b) = \frac{1}{2n}\sum_{x}\|y(x) - a\|^2 \tag{3.1}$$

As usually, w and b represent the weights and the bias respectively, while n is the total number of input, x is the input, $y(x)$ is the desired output, and a is the output of the network. The notation $\|v\|$ indicates the length function for a vector. The goal of the training process is to find the weights and biases of the network which minimize the cost function. To achieve this task, the **gradient descent algorithm** is used. It is composed of various steps that are repeated until the function reaches its minimum:

1. forward phase during which the input is given to the network and the output is generated.
2. the output is compared with the expected one and the cost function is therefore computed.
3. the derivative of the loss function with respect to each weight of the network is calculated, starting from the output and moving towards the input. This step is normally known as **backpropagation**.
4. the weights are modified proportionally to the result of the derivative.

The backpropagation is the most used method for training ANNs, even though it is not ensured as the way in which the human brain learns. Instead, in the case of SNNs, this method cannot be applied due to the non-differentiable nature of spike events [11]. For this reason, two novel techniques have been proposed to overcome the limitation and produce supervised learning for SNNs:

- training the SNNs directly using spike-based backpropagation [12].
- converting the trained ANNs to SNNs [13]

For what concerns unsupervised learnings, the most biologically plausible method is represented by the *Spike timing dependent plasticity (**STDP**)*. This method leads to the adjustment of the strength (or of the weights) of connections between neurons

in the brain [14]. The process is based on the timing correlation between the arrival time of the input spike and the instant in which a new spike is generated; in other terms, it is related to the so-called *pre* and *post neuronal spikes*. Moreover, the STDP process includes the *long term potentiation* (**LTP**) and *long term depression* (**LTD**):

- LTP: when an input spike arrives before the production of a new spike, the synapse weight is incremented. This increase is proportional to the timing difference between the two events: the shorter difference, the higher the increment. Hence, the synapse is responsible to make the neuron fire.
- LTD: when an input spike arrives after the emission of an output spike, the synapse weight is decreased. In other words, the synapse is irrelevant in making the neuron fire.

The learning rule can be expressed mathematically with the Equation 3.2:

$$w_i \leftarrow w_i + \Delta w_i,\ \Delta w_i = \begin{cases} +a^+ w_i(1 - w_i), \text{ if } \Delta t > 0 \\ -a^- w_i(1 - w_i), \text{ if } \Delta t \leq 0 \end{cases} \tag{3.2}$$

$$\Delta t = t_{out} - t_{in} \tag{3.3}$$

Where t_{in} and t_{out} indicate the time instant of the presynaptic and postsynaptic spikes, respectively. The synaptic weight w_i is modulated by a quantity Δw_i, according to LTP and LTD. The quantity by which the weight is incremented or reduced is proportional to the learning rates a^+ and a and to the weight itself.
The figure 3.1 shows the representation of the STDP equation assuming that the long term potentiation and long term depression have the same intensity. Graphically it implies a symmetrical characteristic; however, it is not always true in reality.

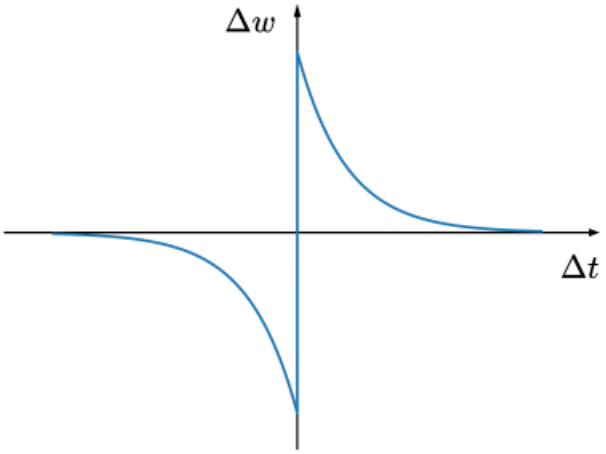


Figure 3.1: Plot of the STDP equation

Chapter 4

Privacy in deep learning

The algorithms based on ANNs have achieved significant success and have been widely applied in various fields including image classification, autonomous driving, natural language processing and so on; this has led to their adoption in many production systems. The main contribution to these advances is certainly the great availability of data; at the same time, the resulting datasets may contain sensitive information. Thus privacy and security issues have been arisen.

This chapter describes the most common attacks and privacy concerns against deep learning. It also depicts the privacy-preserving countermeasures in recent years.

4.1 Existing threats

Generally, the threats against DL and ML are classified into two main categories [15]: *direct* and *indirect* information exposure as shown in Figure 4.1. In the case of direct threats, the attacker has the possibility to access the information via direct exposure; while in indirect ones, the attacker aims at extracting the information while not having direct access to it.

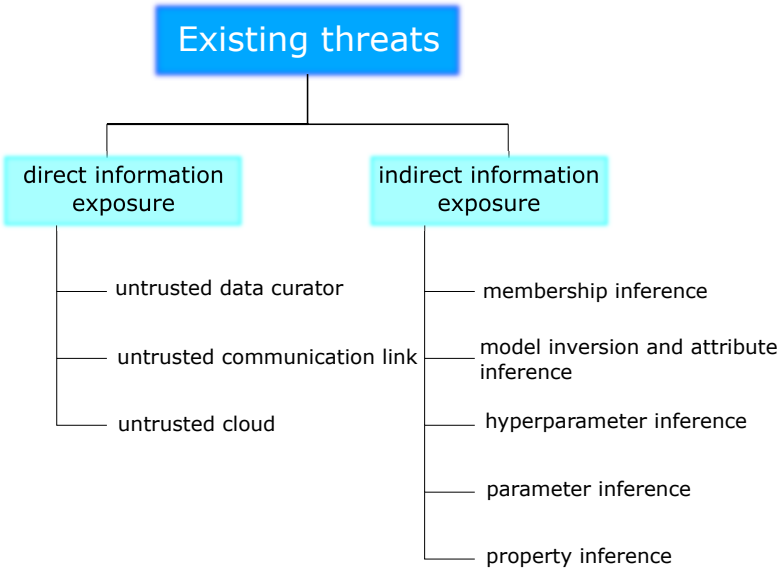


Figure 4.1: Categorization of existing threats [Source: [15]]

4.1.1 Direct information exposure

As shown in figure 4.1, the direct information exposure may happen in different settings and not only limited to ML. For instance, the *untrusted data curator* is related to dataset security: this breach is often caused unintentionally by virus, malware, or by carefreely sharing sensitive data with attackers. In 2016 Intel security published a study [16] which demonstrates that about 21% of data leakage is provoked by employees unintentionally.
Another possible breach is due to the leakage through communication links. It may occur because of the absence of proper data encryption. Similarly, in an *untrusted cloud*, the services don't tell exactly what occurs to the data after they have been processed.

4.1.2 Indirect information exposure

The indirect information exposure can be divided into 5 main groups, as shown in figure 4.1. Before going into details about each of them, it is worth defining the two possible ways that the adversary has to access the ML model [17]:

- **white-box**: the attacker has complete knowledge of the target ML model, its

architectures and parameters, and training data;

- **black-box**: the adversary knows neither the target model, consisting of its architecture and parameters nor the training data. The attacker identifies the ML model's exposure by exploiting the knowledge about output and then sends a series of queries to retrieve the ML model.

Membership Inference

Assuming white or black-box access to the model, a member inference attack (**MIA**) establishes the contribution of a given data instance to the training step. Shokri *et al.* [18] proposed the first MIA in which the so-called *shadow training* is used and it generates multiple shadow models to simulate the target model. However, this attack model requires making several assumptions, like using multiple shadow models, or the knowledge of the target model; for this reason, successive studies [19] have relaxed these hypotheses, demonstrating the effectiveness of the attack also at lower cost and without the access to confidence score [20].

Model Inversion and Attribute Inference

In a model inversion attack, the adversary is able to infer some training data; he knows non delicate attributes related to instances in the training set and tries to deduce the value of a delicate attribute [21]. For instance, Fredrikson *et al.* [22] proposed an attack inverting the ML model of a medicine dose prediction task. Besides this main information, they could recover the genomic information about the patient, or other non-sensitive attributes such as height, age, and weight. Another formulation by Fredrikson [23] shows that the adversary could infer instances of training data exploiting oracle access to a ML classification model. For example, Figure 4.2 shows a face image recovered from a training set of images, by utilizing this kind of attack.

Hyperparameter and Parameter Inference

As described in the previous chapters, the DNNs store the information acquired during the training step; the training data can be considered properties of their owners, therefore the inference of the model results in a privacy breach. There is a

Figure 4.2: On the left an image recovered thanks to a model inversion attack and, on the right the image from the dataset

model stealing mechanism that tries to infer the model parameters via black-box access to the target model. Tramer *et al.* [24] proposed an attack which seeks parameters of a model given its confidence values solving equations based on pairs of input-outputs. Instead, the hyperparameter stealing attack occurs during the training.

Property inference

During the learning process, there could be a possibility that the model learns attributes that are not relevant with its main task. The property inference takes advantage of this and tries to deduct these secondary properties [25].

4.2 Privacy-Preserving Mechanism

As the threats, also the defense mechanisms can be classified into groups as shown in Figure 4.3.

4.2.1 Data aggregation

The data aggregation methods [26], whose purpose is the collection of data and the creation of datasets preserving the privacy of the contributors, can be divided into two groups:

- context-free privacy: the solutions are unaware of how the data will be used;

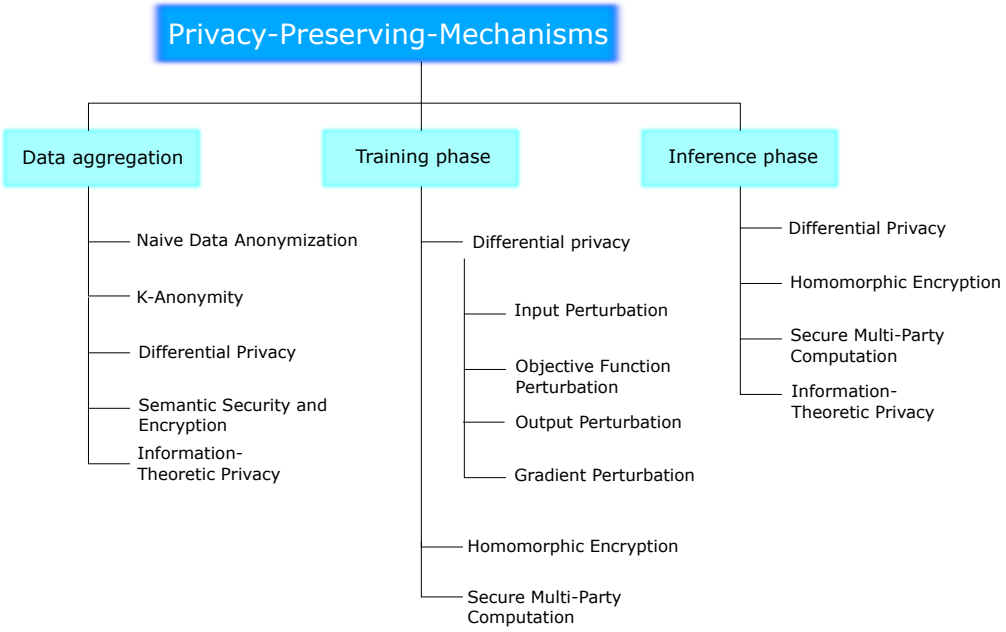


Figure 4.3: Classification of Privacy-Preserving-Mechanisms for deep learning [Source :[15]]

- context-aware: the solutions are aware of the aim whereby the data will be used for.

Naive Data Anonymization

Naive anonymization means the suppression of identifiers from data, like the names, and other information of the participants in order to protect privacy. For example, this method was used for protecting patients in medical fields; however, it has been demonstrated to fail in several scenarios [27].

k-Anonymity

"A dataset has k-anonymity property if each participant's information cannot be distinguished from at least $k-1$ *other participants whose information is in the dataset"* [26]. In other words, the K-anonymity implies the presence of at least k rows with the same set of attributes for any combination of *quasi-identifiers*, which are attributes available to the adversary. Again this method is limited to small datasets [28].

Differential Privacy

Differential Privacy (**DP**) was introduced by Dwork *et al.* [29]. Their goal was to ensure an algorithm to learn statistical information about the population without leakage of sensitive information. ϵ-DP is verified for a randomized mechanism M if, for any pair of datasets D and D' that differ in only one element, satisfy the Equation 4.1:

$$Pr[M(D) \in S] \leq exp(\epsilon)Pr[M(D') \in S] \tag{4.1}$$

where S are all the subsets of M and ϵ is the privacy budget parameter which defines the privacy level. M can be formulated as in Equation 4.2:

$$M(D) = f(D) + n \tag{4.2}$$

where n is the additional noise to the function *f(D)* provoked by *M*. Generally, n is drawn from a Laplace or Gaussian distribution. Therefore, in other terms, even if an adversary is aware of the whole dataset D except for a single element, he is not able to infer additional information about that record by accessing the output[17]. The function is also characterized by a *sensitivity* Δf which defines how much the output of the function varies by changing any element in D, the output of the function varies, as expressed in 4.3:

$$\Delta f = max\|f(D) - f(D')\| \tag{4.3}$$

In order to loosen the definition of ϵ-DP which is very strict, the (ϵ, δ)-DP were introduced, where the parameter δ is added to the right-hand side of Equation 4.1:

$$Pr[M(D) \in S] \leq exp(\epsilon)Pr[M(D') \in S] + \delta \tag{4.4}$$

Semantic Security and Encryption

The semantic security, expressed as λ [30] is a typical privacy specification of encryption schemes. It establishes that the *advantage* (a measure of an adversarial attack success versus a cryptographic algorithm) of an adversary with background information should be negligible or small.

Information-Theoretic Privacy

The information-theoretic privacy is a context-aware privacy solution, which clearly models the dataset statistics. Some of these solutions try to achieve privacy and fairness through information degradation. Huang *et al.* [31] introduced a solution known as generative adversarial privacy, which leads to privatized datasets.

4.2.2 Training phase

The private training of deep learning models is extremely effective. It can be either based on differential privacy or semantic security and encryption.

Differential Privacy

During the training phase, the application of DP can occur in different places of the whole deep learning framework as shown in Figure 4.4. The DP can be applied to the input, to the gradient updates, to the loss function, to the labels and to the output.

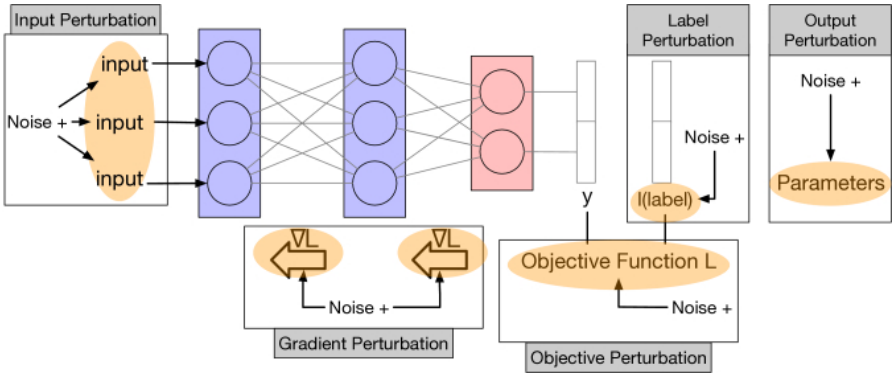


Figure 4.4: Overview of how differential privacy can be applied during training [Source: [15]]

The **input perturbation** is similar to what discussed in Section 4.2.1 to describe *sanitized dataset*. The **objective** and **output perturbation** are performed with convex objective functions. However, in Deep Learning, because the non-convexity of the objective function, the calculation of the sensitivity of the function (which is needed to compute the intensity of the additional noise) is non-trivial. A possible solution is to use an approximate convex polynomial function in place of the

non-convex function [32] but this leads to restricting the capability of conventional DNNs. For this reason, the **gradient perturbation** approach is widely used in DL. The basic idea is to inject noise into the gradient at each step of the stochastic gradient descent algorithm.
The **label perturbation**, is performed by injecting noise into the label. More in detail, Papernot *et al.* [33] proposed a new approach, *Private Aggregation of Teacher Ensembles*: an ensemble of "teacher" models learns only unrelated subsets of the sensitive data, and after a "student" model is trained. But the student model is not allowed access the sensitive data and due to the noise added, privacy is ensured.

Homomorphic Encryption

Homomorphic Encryption (**HE**) enables the computation on the sensitive data guaranteeing the privacy of data. A client encrypts and sends the data; the server performs the computation without decrypting, and finally, it is sent to the client again who can decrypt it. This privacy mechanism will be discussed more in detail in Chapter 5.

Secure Multi-Party Computation

Secure Multi-Party Computation (**SMC**) offered two main approaches in DL [17]. The first approach is that a data owner does not want to exhibit all the training data to a server to train the network. Hence, the idea is to distribute the training/testing data to different servers in a way that each server does not know the training/testing data of other ones. The second approach is that multiple data owners want to train a shared network with all the available data, while ensuring the privacy of their ones.

4.2.3 Inference Phase

As shown in Figure 4.3, the categorization of inference privacy is almost equal to the training phase except for a new field, named *Information-Theoretic*. The basic idea of inference privacy is to assume the ML model is already trained and its parameters are not further updated. The solutions are context-aware and the goal

is to provide a reduced information content (or better, the much as needed) to the service for the inference.
Generally, in literature, it is possible to highlight a diffused trend: the use of DP for the training phase and encryption and SMC methods for the inference phase. A possible reason lies in the computational cost [34]: in fact, the encryption results particularly slow during the training phase and, as told before, also degrades the performance. Instead, the application of HE during the inference phase seems more trivial, due to the fact the model is already trained. Similarly, the differential privacy and hence the addition of noise becomes simpler in this stage.

Information-Theoretic Privacy

The information-theoretic approaches usually assume that the service tries to degrade any unnecessary information in the input data that is not required for the primary task [35]. There exists another solution, the *Deep Private Feature Extraction* (**DPFE**) [36] which aims at degrading the input images by modifying the network topology and architecture. In particular, the network is divided into two partitions, one related to the edge and one to the cloud. Moreover, an encoder is inserted in order to resize the input data, reducing the communication cost and the amount of data sent, thus enhancing the privacy.
DPFE algorithm forecasts the retraining of the given neural network and of the encoder with its loss function. The training attempts to generate clustered representations of data: inputs with the same private labels join different clusters, and inputs with different labels go in the same cluster. In this way, the inference of the private labels is compromised by the adversary. After training, the noise is added to the intermediate results but it is not related to differential privacy.
Instead, the Shredder solution [37] performs a similar approach by cutting the NN and executing only a part of it; hence a benefit is the decreasing of execution time with respect to the whole execution time of the DNN inference. Therefore, both DPFE and Shredder show a reduction of information in the sent intermediate representation versus the original data. The main difference is that the DPFE becomes effective only if the user knows what to protect, whereas Shredder is a more general approach which tries to delete any information unrelated to the main task. At the same time, DPFE is more efficient in terms of inferring private labels,

since the misclassification rate is higher thanks to its access to the labels during the training phase.

4.2.4 Privacy-Enhancing Execution Models and Environments

Besides the method described so far, there are also other privacy-preserving approaches which exploit specific execution models and environments. The most known are the **federated learning**, **split learning** and **trusted execution environments**.

Federated Learning

Federated learning (**FL**) is a machine learning technique by which multiple clients train a model on decentralized devices or servers while keeping the training data local. The concept is totally opposite to the centralized ML techniques and tends to reduce the privacy risk [38].
The flow of operations can be divided into 6 stages [15]:

1. Problem identification: definition of the problem to solve with FL.
2. Client instrumentation: the clients are equipped with the data needed for training.
3. Simulation prototyping (optional): prototyping of several architectures and testing several hyperparameters in a federated learning simulation.
4. Federated model training: multiple federated training tasks are launched which train different variations of the model or use different optimization hyperparameters.
5. Model evaluation: after the training, which can last also days, the models are analyzed and evaluated, considering both the standard centralized datasets and the local client data.
6. Deployment: a model is selected and the launch process is started. This process consists of different tests and it does not depend on how the model is trained.

Split Learning

Split-learning is an execution model in which the neural network is split between the client and the server [39]. The approach is very similar to the partitioning described in Section 4.2.3. There exist two main split learning configurations, the *Vanilla split learning* and the *boomerang split learning*. In the former, each client performs the computation until a specific layer, called the cut layer. The outputs of this layer, known as smashed data, are sent from the edge device to another entity, such as a server which concludes the remaining operation. In this scheme, the raw data is not shared between client and server.
The boomerang scheme (or U-shaped) denies not only the sharing of raw data but also of the labels between the two entities. Figure 4.5 depicts the two configurations.

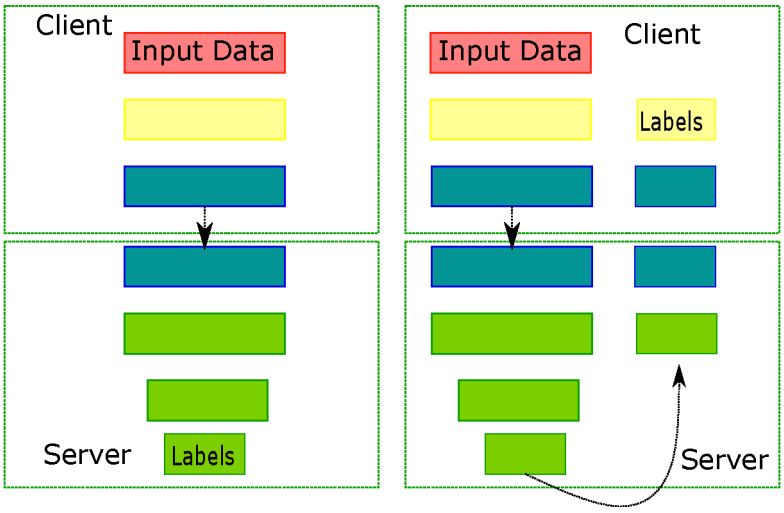


Figure 4.5: On the left the Vanilla configuration. On the right the U-shaped one [Source: [15]]

Trusted execution environments

The Trusted Execution Environment (**TEE**) is an operating environment of the main processor that ensures a safe execution environment for trusted applications. Moreover, the TEE guarantees integrity and confidentiality during execution. There are few works that utilize TEE to provide privacy in DL [17].

Chapter 5

Homomorphic encryption

Homomorphic encryption (HE) is a cryptographic method that allows performing computations over encrypted data instead of its raw version. The implementation of this defense mechanism has long been studied since the possibility of applying it in several fields, mainly to computation on sensitive data ensuring privacy [40].
This chapter provides an accurate description of the HE and its evolution in time. First, to properly understand it, some basic notions are discussed in the following preliminary sections.

5.1 Cryptographic Background

Cryptography is a method to make secure the communication from the sender to the receiver allowing only them to read and process the information. First of all, it is worth defining the main actors of any cryptographic scheme.
The **plaintext** is the term used to indicate the information in plain language. The plaintext is sometimes also referred to as cleartext. Instead, the **ciphertext** is the term used to refer to the information in encrypted form. In other words, they can be seen as the input and output of the **encryption** algorithm which is the process that transforms the plaintext into the ciphertext. On contrary, the **decryption** algorithm performs the opposite transformation, from the ciphertext to the plaintext.
These processes need specific parameters, named **keys**. They are like codes used

to encrypt and decrypt the information.

Now that these terms are known, it is also important to understand how to evaluate a cryptographic scheme. In particular, there are two main features to characterize the effectiveness of a cryptographic scheme:

- The semantic security expressed as λ [30] is a typical privacy specification of encryption schemes which establishes that the *advantage* (a measure of an adversarial attack success versus a cryptographic algorithm) of an adversary with background information should be negligible or small. The semantic security notion is also referred to as *ciphertext indistinguishability under chosen-plaintext attack*: the encryption scheme is secure if the attacker cannot distinguish two ciphertexts encrypting the same message [30].

- Correctness: a homomorphic scheme is correct if the decryption function always returns the original plaintext [41]. Notice that the HE is probabilistic and this implies the addition of noise during encryption. In these cases, the decryption acts as a filter that removes the noise under a certain noise threshold. Otherwise, if the noise is large and overcomes this limit, the ciphertext cannot be decrypted anymore.

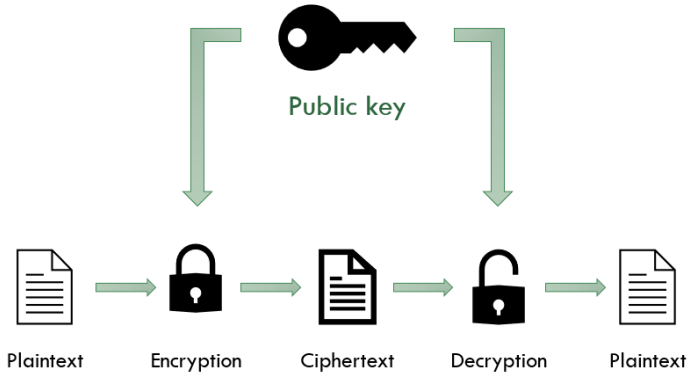
The problem of noise together with the complexity of defining schemes makes the *Fully Homomorphic Encryption* (**FHE**) schemes too inefficient for practical use. Hence, there exist intermediate solutions easier to build which allow to achieve better efficiency.
Partially Homomorphic Encryption (**PHE**) schemes are characterized by the verification of homomorphism properties only for a subset of functions belonging to plaintext space's algebraic operations, for instance assuming that the decryption function is a homomorphism of the multiplicative group. Therefore, only functions composed by multiplications have an equivalent in the encrypted space [41].
Somewhat Homomorphic Encryption (**SHE**) schemes are characterized by the verification of homomorphism properties for two kinds of operations, addition and multiplication. Nevertheless, it cannot be a complete FHE schemes because of the limit of its evaluation capability, namely a limited number of computations over encrypted data.

5.1.1 Symmetric and asymmetric encryption

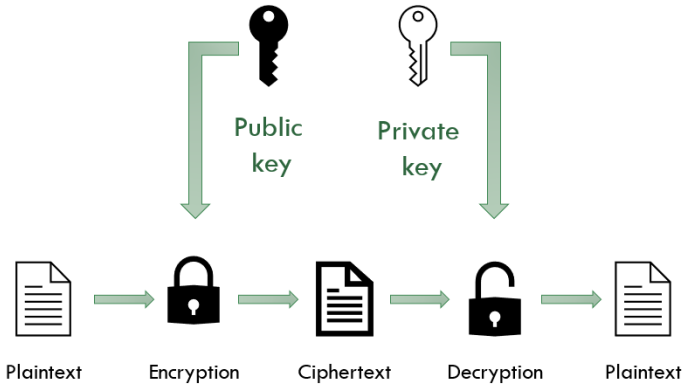
There exist two main forms of encryption, the **symmetric encryption** and **asymmetric encryption**, as shown in Figure 5.1.
The former technique uses the same key both to encrypt and decrypt the information; for this reason, it is also known as a *single-key algorithm* or *private key encryption* [42]. Instead, the latter technique uses two different keys, the *public key* for encryption, and the *secure key* for decryption. The algorithm is also known as *public key encryption*.
The symmetric scheme is less secure than the asymmetric scheme because anyone who has the public key can both encrypt and decrypt the message, while in the asymmetric scheme, any person who gets the public key can encrypt the information, but he cannot decrypt it without the secure key. At the same time, symmetric encryption is characterized by a higher operating speed than asymmetric one.

5.1.2 Algorithms in an Encryption scheme

With a standard encryption scheme, the only possible operations are storage and retrieval. To perform any kind of computation, it is needed to decrypt the data first. Instead, the HE mechanism allows to perform computations directly upon encrypted data.
Craig Gentry, who can be considered the author of the FHE, provides a very effective and fancy example to explain HE [43] modelling it as a transparent glove box, as depicted in Figure 5.2. It assumes that a jewellery store owner has a collection of expensive raw materials from which a costly final product can be obtained. However, since the owner does not trust his workers, he would find a strategy to allow the workers to deal the materials without having direct access to them. Hence, he devises to insert the material inside a transparent glow box and lock it with a personal and secure key; in this way, the workers can insert their hands in the gloves and process the materials to obtain a piece of finished jewellery. Finally, the workers give back the box to the owner who can open it and get the jewellery.
Let's try to understand the above example with the correct formalism. The whole process models the encryption procedure. In particular, the raw materials inserted



(a) *Symmetric encryption scheme*



(b) *Asymmetric encryption scheme*

Figure 5.1: Overview of two encryption schemes

in the box, represent the plaintext $\{m_1, m_2, ...m_n\}$; the handling of the materials symbolizes the processing of data to encrypt them, Enc_k, where k is the key used by the owner to lock the box. Finally, the obtained product represents the cyphertext:

$$c = Enc_k(m_1, m_2, ...m_n) \tag{5.1}$$

To conclude, a Homomorphic encryption scheme includes four algorithms:

- **KeyGen**: takes as input some security parameters and then outputs a pair of

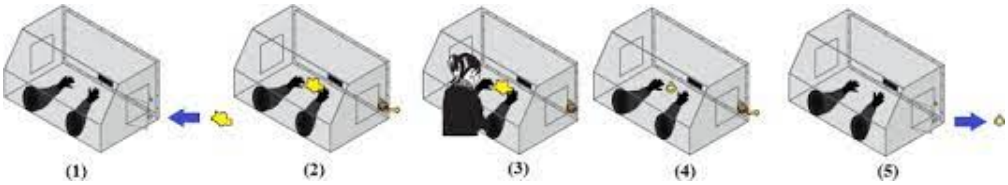


Figure 5.2: The raw materials are locked inside the box and the worker can only manipulate them without having direct access[Source: [44]]

key, the public key p_k and the private key s_k:

$$(p_k, s_k) \leftarrow KeyGen(parameters) \tag{5.2}$$

- **Enc**: takes as inputs the plaintext m $\in \{0, 1\}$ and the public key. Then it outputs the ciphertext c. The process can be indicates as:

$$c \leftarrow Enc_{p_k}(m) \tag{5.3}$$

- **Dec**: takes as inputs the ciphertext and the private key. It returns the plaintext message if decryption is successful. This process is denotes as:

$$m = Dec_{s_k}(c) \tag{5.4}$$

- **Eval**: takes as inputs n ciphertexts $c_1, c_2, ..c_n$, a defined function f and the public key. It returns $f(c_1, c_2, ..c_n)$. The operation is succesful if:

$$Dec(Eval(f, c_i, p_k), s_k) = f(m_i) \tag{5.5}$$

where c_i is the set of ciphertexts and m_i the corresponding decrypted messages.

Notice that, the above algorithms are defined in the case of asymmetric encryption. However, it is possible to transform an asymmetric encryption into a symmetric one, simply considering that the homomorphic evaluation algorithm needs the encryption key. To counteract this trouble, the encryption key can be attached to the end of each ciphertext [45]. Instead, the transformation from a symmetric scheme to an asymmetric one is more difficult.

5.1.3 History towards Fully Homomorphic Encryption

The origin of homomorphism lies in another cryptosystem, the *RSA*. It was introduced by Rivest, Shamir and Adleman [46] in 1974 and it is nowadays one of the most widely used for secure data transmission. The work got immediate interest after its publication which leads to a quick improvement: the question was the possibility to compute upon encrypted data. Hence, with Dertouzos, they proposed a method of solving the question with a process called **homomorphism** [47]. The idea was to create a cryptosystem with a large defined set of operations which permit computations on encrypted data without requiring the decryption of the operands: they named this encryption function "*privacy homomorphism*" [48]. The term was borrowed from mathematics: *homomorphism* is an application between two algebraic structures of the same type which preserves the operations defined in such structure.
However, they cannot pursue the task in their work because of the current state of research in cryptography.
Afterward, there were long periods of silence in the research community alternating with a few works related to privacy homomorphism. In 1999 Tomas Sander *et al* [49] launched the classical problem:

> *"Alice has an input x and Bob has function C. Alice should learn the value of C(x) but nothing else substantial about C. Bob should learn nothing else substantial about Alice's input x."*

They proposed a new protocol that involved one party sending encrypted data to a second party, the computing one, which evaluates the data securely, and provides the output to the sender without engaging in additional communications. Some years later, Dan Boneh published a paper [50] which described a homomorphic encryption scheme, that allowed the evaluation of quadratic function on the ciphertext. This formulation was a keystone toward the first plausible implementation of HE. Then, in fact, a graduate student named Craig Gentry expanded Boneh's work, and in 2009 published a seminal work [51] which makes it possible to perform arbitrary computations on encrypted data, without first decrypting it and keeping the data secure. Notwithstanding his initial implementation was unfeasible, Gentry's work is generally considered as the FHE breakthrough.

5.1.4 Four generations of FHE

Homomorphic encryption schemes have been developed using different approaches. Generally, FHE schemes are grouped into four generations.

First generation: bootstrapping

The Gentry's implementation is based on the so called *bootstrapping* procedure and demonstrates that, from a "bootstrappable" SHE scheme, it is possible to obtain an FHE scheme. As described in Section 5.1, SHE scheme allows only a limited number of operations. The task of bootstrapping is to reduce the noise over the ciphertext; it applies a sort of periodic "refreshing" to the ciphertext whenever the noise becomes large, leading to an arbitrary number of computations without increasing the noise much.
However, the first implementations of bootstrapping on FHE schemes turned out to be unfeasible [52] due to both the noise growth and the complexity of the architecture.

Second generation: Leveled-FHE

The schemes belonging to the second generation are not strictly FHE because the complexity is not arbitrary but defined in advance. For instance, the complexity is measured in terms of multiplicative depth. This is because the increase of noise due to a multiplication is higher than due to an addition. The reason can be explained with the following example. In an HE scheme, the underlying equations are verified:

- homomorphic addition:

$$Encrypt(a) + Encrypt(b) = Encrypt(a + b) \tag{5.6}$$

- homomorphic multiplication:

$$Encrypt(a) * Encrypt(b) = Encrypt(a * b) \tag{5.7}$$

Whenever an operation is performed, some noise is added to ciphertext. For the addition, the noise can be considered negligible, in fact:

$$Noise(Encrypt(a) + Encrypt(b)) = Noise(Encrypt(a)) + Noise(Encrypt(b)) \tag{5.8}$$

However, the noise introduce by a multiplication becomes very large:

$$Noise(Encrypt(a) * Encrypt(b)) = Noise(Encrypt(a)) * Noise(Encrypt(b)) \tag{5.9}$$

These schemes are commonly known as **Leveled-FHE**. The technique used is called *modulus switching* and constitutes the basic concept for designing the well known BGV scheme [53]. The goal of these moduli is to control the noise growth; nevertheless, they cause variation in the size of the ciphertext.
Brakerski [54] proposed a new technique, named *scale-invariant* which let to control the noise without the moduli. The security of these schemes is based on the problem of *Learning With Errors* (**LWE**) or its variant, the *Ring-LWE*(**RLWE**) like the BFV scheme [55], another well accepted scheme to implement FHE.
Some years later, Cheon, Kim, Kim and Song released their CKKS scheme [56] which allowed homomorphic computations on real numbers.
All these three schemes follow Gentry's original construction, namely, they first construct a SHE scheme and then convert it to an FHE scheme through the combination of some techniques in order to reduce the noise.

Third generation: simplified Leveled-FHE

The third generation reviews the previous one avoiding the insertion of other manipulation in order to reduce the noise, like the *relinearization* step. Some popular schemes belonging to this generation are the GSW [57] and SHIELD [58]. Both implementations are based on Brakerski's scheme [54]. They lead to more specialized constructions with a highly reduced noise growth.

Fourth generation: gate bootstrapping

The fourth generation revisits the bootstrapping technique: the refreshing of the ciphertext occurs after every single operation. In this way, the noise is not cumulative anymore. This approach is known as *gate bootstrapping* [59].
Before the FHEW and then the TFHE [60] scheme adopt this new approach. In particular, the TFHE can perform the bootstrapped gate in the order of tenth of milliseconds. However, due to its extreme complexity, it requires dedicated hardware implementation.

5.1.5 Bootstrapping

All existing HE schemes are affected by the noise growth during the encryption. Every time a computation is performed on ciphertext, the noise increases. If it overcomes a specific threshold, the decryption process fails [55].
Reminding the example of the glove box, the issue of noise can be illustrated by thinking the box is defective: namely, after a certain number of operations or of time, the gloves lock. A possible solution is to provide several glove boxes with a one-way insertion slot to the workers. Each one of these boxes contains a key to open the previous box: for example, *box 2* contains the *key 1*, *box 3* contains the *key 2* and so on. On the other hand, the key of the last box is held only by the owner.
As soon as the time or operations elapses, the whole box is inserted in another one. Therefore, the worker takes the key and retrieves the raw materials in order to continue the manipulation. This operation is repeated every time the lock of gloves occurs, until the final product is obtained. At that time only the jewellery owner who has the key of the last box can access the final product.
Bootstrapping is a technique used to reduce the noise growth which occurs during the encryption and keep it under a certain limit depending on the complexity of the circuit. Originally the technique was named by Gentry [43] as *recrypt* operation. The effect is to "refresh" the chipertext at each operation; removing this noise, some other additional noise is added in the evaluation function. Nevertheless, as long as the new noise is less than the eliminated one from the encrypted data, the

scheme should operate properly.
The problem is that the bootstrapping is extremely difficult: its complexity is at least the complexity of the decryption times the bit-length of the single ciphertexts that are used to encrypt the bits of the public key [53].

5.1.6 Modulus Switching

Due to the high complexity of bootstrapping, alternative methods to handle the noise have been proposed. One of these is the *modulus switching* explained by Gentry, Brakerski and Vaikuntanathan [53]. The main idea of this technique is to use an evaluator, which has information about the length of the private key s_k, but not about s_k itself; it transforms a ciphertext $c \ mod \ q$ into a new ciphertext $c' \ mod \ p$ without compromising the accuracy of the scheme. The procedure consists of scaling by a factor the chipertext after each multiplication; this factor is gradually decreased for each level of the multiplication. The result of the scaling is the reduction of the noise without applying costly bootstrapping.

5.1.7 LWE and RLWE

The Learning With Error problem was introduced by Regev in 2005 [61] and soon became an adaptable basis for cryptographic constructions. Precisely the LWE problem is defined as follows:

> *Let Z_q be the ring of integers modulo q and let Z_q^n the set of n-vectors over Z_q. For a vector $s \in Z_q^n$, called secret, the LWE distribution $A_{s,\chi}$ over $Z_q^n x Z_q$, is sampled by choosing $a \in Z_q^n$ uniformly at random, choosing $e \leftarrow \chi$, and outputting:*

$$(a, b = \langle s, a \rangle + e \ mod \ q) \tag{5.10}$$

There exist two main versions of the problem, the *search* problem, in which the task is to find the secret vector from the LWE samples; instead in the other version, named *decision*, the goal is to distinguish between LWE samples and random ones. The LWE problem is demonstrated to be hard to solve as several worst-case lattice problems and for this reason, it results useful in cryptography.
In 2010, Lyubashevsky, Peikert, and Regev proposed the RLWE [62]. It can be

seen as a larger version of the LWE which handles polynomial rings of the form $R = Z[X]/(X^n + 1)$ for power-of-two n. The definition is:

> *For a vector $s \in R_q$, called secret, the RLWE distribution $A_{s,\chi}$ over $R_q \times R_q$, is sampled by choosing $a \in R_q$ uniformly at random, choosing $e \leftarrow \chi$, and outputting:*

$$(a, b = s \cdot a + e \; mod \; q) \tag{5.11}$$

Also, the RLWE has two different versions, *search* and *decision*, which are similar to the simpler LWE. The higher complexity to solve the problem even on a quantum computer represents a hardness assumption to build cryptosystems.

5.1.8 Models of homomorphic computation

Regardless of the generations to which they belong, the FHE schemes can be divided into three classes [63] depending on their computation model. Moreover, this classification represents a good starting point to choose the right approach to implement an FHE scheme.

- boolean circuits
- modular arithmetic
- approximate number arithmetic

Table 5.1 offers a comparison overview among the three models, focusing on how the plaintext data are represented and on the type of circuit used for the computations. According to their features, each model fits better for different cases. The first one is characterized by a very fast bootstrapping and it results highly effective for number comparison. The schemes used are the GSW, FHEW and TFHE. The modular design of the model allows high precision integer arithmetic and fast scalar multiplication. The selected schemes for this model are the Leveled-FHE as the BGV or BFV. Finally, the third class deals with floating-point numbers. It allows fast polynomial approximations and generally is implemented with Leveled-FHE, such as the CKKS scheme.

Models	**Plaintext data**	**Circuit type**
boolean circuit	bits	boolean
modular arithmetic	integers	integer
approximate number arithmetic	real	floating-point

Table 5.1: Comparison among the three computation models

Chapter 6

Model Design and Implementation of the HE Scheme

Somewhat Homomorphic Encryption (**SHE**) schemes are an extension of the PHE schemes. They are characterized by the verification of homomorphism properties for two kinds of operations, addition and multiplication. However, it cannot be considered a complete FHE scheme due to the limit of its evaluation capability. In other words, this means that only a limited number of computations over encrypted data are feasible.

Despite these limitations, the SHE scheme is the chosen structure in this thesis. The reason lies in the fact there are occurrences in which the SHE may be sufficient: for instance, when only a simple function is required to be computed, it is demonstrated that the SHE scheme has an overhead (the ratio between the time to compute operations on ciphertext over the time to compute operations on the plaintext) lower than the FHE scheme [52].

The HE scheme deals with addition and multiplication of polynomials. To better understand the implemented architecture, it is worth making a quick example of how to work with it.

At first, the *modulus* operation provides as result the remainder of the division.

It is very simple to apply to real numbers but let's analyze how it works with polynomials. For instance, considering the polynomial $p(x) = x^5 + 3x^4 + 2x^3 + x + 5$ and $q = 4$, computing $p(x)\ mod\ q$ the result is:

$$p(x)\ mod\ q = x^5 + 3x^4 + 2x^3 + x + 1 \tag{6.1}$$

Instead, considering the function $f = x^4 + 1$ and the same polynomial $p(x)$, let's try to calculate $p(x)\ mod\ f$. At first, notice that for example x^5 can be written as $x(x^4 + 1) - x$ and so computing the $x^5\ mod\ f$ the result is $-x$. Similarly:

$$p(x)\ mod\ f = (x^5 + 3x^4 + 2x^3 + x + 5)\ mod\ f = \tag{6.2}$$

$$= -x - 3 + 2x^3 + x + 5 =$$

$$= 2x^3 + 2$$

6.1 BFV: Brakerski-Fan-Vercauteren

As described in Chapter 5, different generations of FHE exist, and each of them includes several schemes. The chosen one in this work is the BFV [55].
In 2012, Fan and Vercauteren revisited Brakerski's scheme which is based on LWE, and adopt the assumption of RLWE to build a new scheme. The BFV includes both the relinearization and the bootstrapping, which are implemented with a more simple procedure called *modulus switching*.
The plaintext space in the chosen scheme is $R_t = Z_t[x]/(x^n + 1)$ where:

- t is the *plaintext modulus*
- n is power of 2 and represents the polynomial modulus degree

The encryption process in the BFV generates a ciphertext which is expressed as two polynomials with the same polynomial modulus but with a different coefficient modulus $q \gg t$: so the ciphertext space is $R_q = Z_q[x]/(x^n + 1)$.
The BFV scheme is characterized by 8 functions:

1. **Private key generation**: it takes as input the security parameter λ and, from a uniform distribution R_2 samples s, so the private key s_k will be a

binary polynomial:

$$s_k = PrivateKeyGen(\lambda) \tag{6.3}$$

2. **Public key generation**: it takes as input the private key, indicated as s and samples the value a uniformly over R_q, and a small error e from a discrete Gaussian distribution χ over R_q:

$$p_k = PublicKeyGen(s_k) = \tag{6.4}$$

$$= ([-(a \cdot s + e)]_q, a)$$

Notice that, notwithstanding the multiplication between two polynomial, which implies the addition of the exponent, the maximum degree remains the same thanks to the *mod* operation by $(x^n + 1)$.

3. **Evaluation key generation**: it takes as input the private key s_k and T which is a positive integer base. Let l be $\lfloor log_T(q) \rfloor$, assume $s_k = s$ and sample, as before, a_i and e_i for $i = 0, ..., l$, so:

$$evk = EvaluationKeyGen(s_k, T) = \tag{6.5}$$

$$= ([-(a_i \cdot s + e_i) + T^i \cdot s^2]_q, a_i)$$

for $i = 0, ..., l$.

4. **Encrypt**: it takes as input the public key p_k and a message $m \in R_t$. Assuming $p_0 = p_k[0]$ and $p_1 = p_k[1]$, and randomly sampling u, e_1, e_2 from χ and let be $\Delta = \lfloor q/t \rfloor$:

$$ct = Encrypt(p_k, m) = \tag{6.6}$$

$$= ([p_0 \cdot u + e_1 + \Delta \cdot m]_q, [p_1 \cdot u + e2]_q)$$

5. **Decrypt**: it takes as input the ciphertext ct and the private key s_k. Setting $s_k = s$, $c_0 = ct[0]$ and $c_1 = ct[1]$:

$$m' = Decrypt(s_k, ct) = \tag{6.7}$$

$$= \left[\left\lfloor \frac{t \cdot [ct[0] + ct[1] \cdot s]_q}{q} \right\rceil\right]_t$$

Let's understand better how the decryption function acts. Since $p_0 + p_1 \cdot s$ is small, $c_0 + c_1 \cdot s$ is close to $\Delta \cdot m$. Hence to recover the original message m it is needed to divide by Δ, that is q/t; finally the rounding is applied.
Going more into detail,

$$[ct(s)]_q = [c_0 + c_1 \cdot s]_q = [(p_0 \cdot u + e_1 + \Delta \cdot m) + (p_1 \cdot u + e2) \cdot s]_q = \quad (6.8)$$

$$= [-(a \cdot s + e) \cdot u + e_1 + \Delta \cdot m + a \cdot u \cdot s + e_2 \cdot s]_q =$$

$$= \Delta \cdot m - e \cdot u + e_1 + e_2 \cdot s =$$

$$= \Delta \cdot m + v$$

or rather the scaled message with a noise "v". Now, since ($\Delta \cdot m + v$ is lower than q, the $mod\, q$ operation has no effect, this means until $v < \Delta/2$, the plaintext m is always recovered correctly.

6. **Addition between ciphertexts**: it takes as inputs two ciphertexts ct_0 and ct_1:

$$(ct_0 + ct_1) = (ct_0[0] + ct_1[0], ct_0[1] + ct_1[1]) \quad (6.9)$$

From a practical point of view, adding ciphertexts is like adding two polynomials in R_q.

7. **Multiplication between ciphertexts**: it takes as inputs two ciphertexts ct_0 and ct_1 and returns a 3-tuple:

$$c_0 = \left[\left\lfloor \frac{t \cdot ct_0[0] \cdot ct_1[0]}{q} \right\rceil\right]_q \quad (6.10)$$

$$c_1 = \left[\left\lfloor \frac{t \cdot (ct_0[0] \cdot ct_1[0] + ct_0[1] \cdot ct_1[0])}{q} \right\rceil\right]_q \quad (6.11)$$

$$c_2 = \left[\left\lfloor \frac{t \cdot ct_0[1] \cdot ct_1[1]}{q} \right\rceil\right]_q \quad (6.12)$$

But this has 3 coefficients, therefore is not the final result. Moreover, this

size can grow linearly as much as the further multiplication performed on the ciphertexts. To get the correct final result, it is needed the following operation.

8. **Relinearization**: the task is to restore the size of the ciphertext back to 2-degree. This operation requires the result of the Evaluation key generation function:

$$c_0' = [c_0 + evk[0]c_2]_q \tag{6.13}$$

$$c_1' = [c_1 + evk[1]c_2]_q \tag{6.14}$$

The issue is that c_2 can have coefficients up to size q and this could lead the decryption process to fail. So, c_2 needs to change to a smaller base before being used in the above equations. Namely:

$$c_2 = \sum_{i=0}^{l} c_2^i T^i \tag{6.15}$$

Figure 6.1 depicts a high level overview of the BFV scheme, considering two plaintext messages m and m', which are *encrypted*, getting ct and ct'. Hence, they are *evaluated* and finally *decrypted*.

6.2 Implementation

The implementation regards some functions of the BFV scheme described in the previous section. The basic idea is to separate these functions into software implementation and hardware one as shown in Table 6.1.

Software implementation	**Hardware implementation**
Private Key Generation	Encryption
Public Key Generation	Decryption
	Addition between ciphertexts

Table 6.1: Design structure

More in detail, the Key Generation functions are implemented in *Python* and all

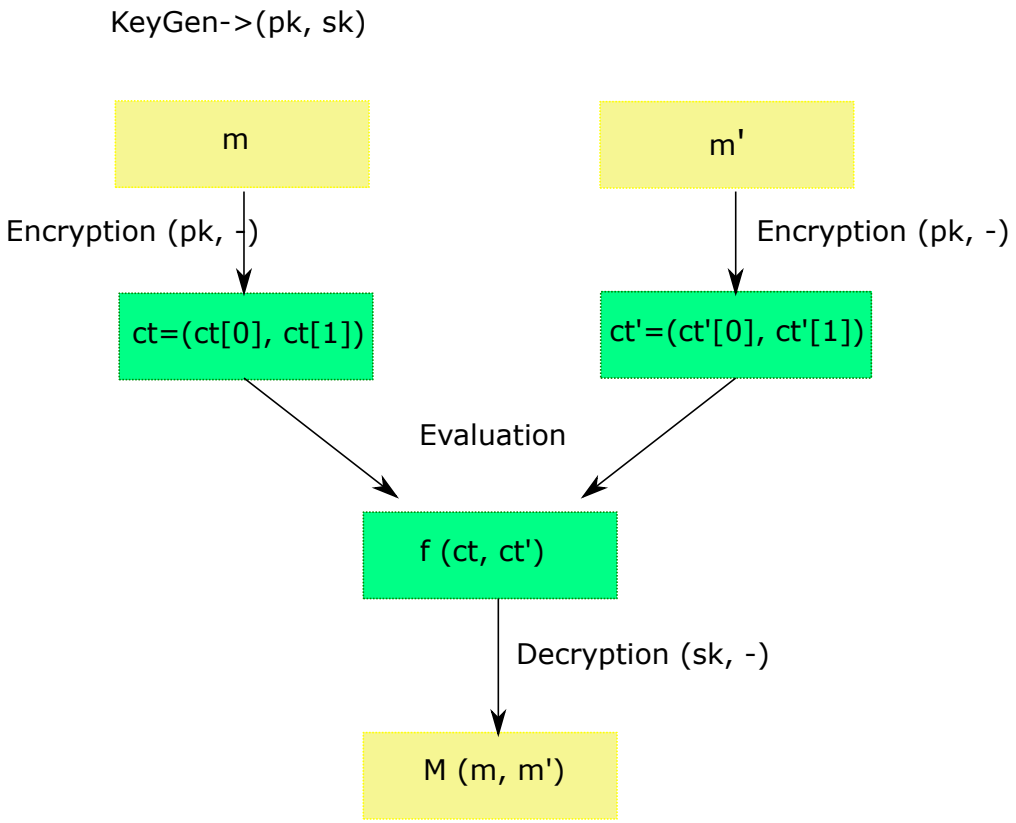


Figure 6.1: High level overview of the BFV scheme

the HW implementation is carried out using *VHDL*.

6.2.1 Key Generation functions

The reason why the Key Generation functions are implemented in software lies in their definition; in fact, they are defined by drawing samples from different kinds of distribution. A simple way to model these distributions is using Python language. Python is an interpreted programming language and provides several programming paradigms, like structured, functional and object-oriented. Nowadays Python is one of the most popular programming languages thanks to its high level of abstraction which makes it simpler, and also to the possibility to use it in several application domains. In fact, Python's library offers a lot of tools suited for any task. For instance, there exist more than 300 000 packages which provide functionality in automation, databases, GUI, image processing, machine learning, mobile apps, web

framework and so on.
The library used for the implementation of the key Generation functions is the *NumPy*. It is a mathematical library, in fact, the name is an abbreviation of *Numerical Python*.
In the Algorithm 1 is shown the function to compute the private and public key as described in [64]:

Algorithm 1 Key Generation function s_k from a uniform distribution and $p_k = ([-(a \cdot s + e)]_q, a)$.

```
1: function KEYGEN(size, modulus, poly_mod, mean, std)
2:     ▷ size is the size of the polynomials
3:     ▷ modulus is the ciphertext modulus q
4:     ▷ polymod is polynomial modulus
5:     ▷ mean is the mean of distribution
6:     ▷ std is the standard deviation
7:     s_k ← gen_binary_poly(size)
8:     a ← gen_uniform_poly(size, modulus)
9:     e ← gen_normal_poly(size, mean, std)
10:    b ←
   poly_add(poly_mul(−a, s_k, modulus, poly_mod), −e, modulus, poly_mod)
11:    return (b, a), s_k
12: end function
```

Appendix A reports the details about the functions *gen_binary_poly*, *gen_uniform-poly* and *gen_normal_poly*. The functions *poly_mod* and *poly_mul*, perform addition and multiplication between polynomials, respectively. As explained in the previous section, these operations require an additional computation to scale the consequent polynomial. This operation is explained more in detail in Section 6.2.2. The results of this function are put as inputs of the Encryption block or Decryption one in according to Equations 6.6 and 6.7.

6.2.2 Encryption

Looking at Equation 6.6, other parameters are missing to execute the encryption process. u, e_1 and e_2 are generated exploiting the same function discussed in the Key Generation algorithm, as described in Algorithm 2.

Algorithm 2 Parameters Generation function u, e_1 and e_2.

1: **function** PARAM_GEN(size, mean, std)
2: ▷ size is the size of the polynomials
3: ▷ *mean* is the mean of distribution
4: ▷ *std* is the standard deviation
5: $u \leftarrow gen\_binary\_poly(size)$
6: $e_1 \leftarrow gen\_normal\_poly(size, mean, std)$
7: $e_2 \leftarrow gen\_normal\_poly(size, mean, std)$
8: **return** u, e_1, e_2
9: **end function**

At this point, the plaintext m which has to be encrypted is chosen and the encryption process can start. The Figure 6.2 shows the datapath implemented for this function.

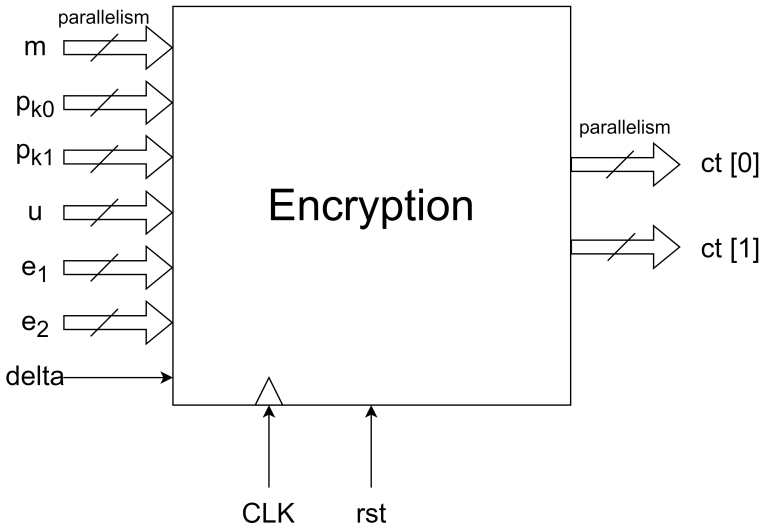


Figure 6.2: Schematic representation of the encryption block

Let's analyze the internal structure, shown in Figure 6.3, focusing step by step on the operations performed. Recalling the Equation 6.6:

$$ct = Encrypt(p_k, m) =$$

$$= ([p_0 \cdot u + e_1 + \Delta \cdot m]_q, [p_1 \cdot u + e2]_q)$$

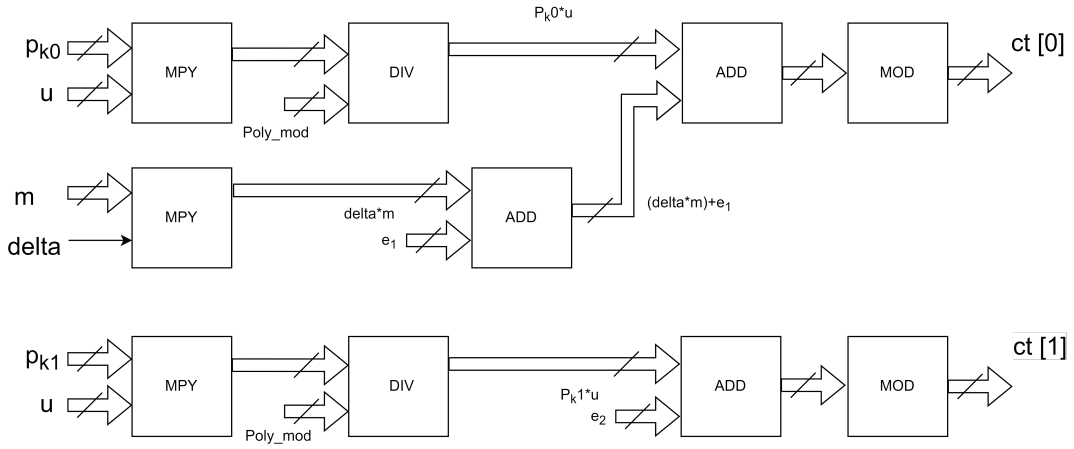


Figure 6.3: Schematic internal representation of the encryption block

the public keys are multiplied by u. The key is a polynomial with a coefficient in the range [0,q[, while u is a binary polynomial; both have the same size. The product provides a polynomial with double size. The operation is very expensive, because it requires a number of multiplications equal to $size^2$, and then the sum of the coefficients having the same degree. For example:

$$(x^4 + 2x^3 + x^2 + 3x + 1) * (2x^4 + x^3 + 3x^2 + x + 2) = \tag{6.16}$$

$$= 2x^8 + x^7 + 3x^6 + x^5 + 2x^4 + 4x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^3 + 2x^6 + x^5 + 3x^4 + x^3 + 2x^2 +$$

$$+ 6x^5 + 3x^4 + 9x^3 + 3x^2 + 6x + 2x^4 + x^3 + 3x^2 + x + 2 =$$

$$= 2x^8 + 5x^7 + 7x^6 + 14x^5 + 12x^4 + 15x^3 + 8x^2 + 7x + 2$$

The best approach to implement this operation is to use a *Multiply-Accumulate* (**MAC**). It is a well known unit in computing and allows to perform a multiplication between two numbers, and add the result to an accumulator. The Figure 6.4 depicts the external block and its internal components. The whole operation is described in VHDL using three different *processes*:

1. *product_process*: it handles the multiplication between two coefficients of the two polynomials.

2. *reg_update*: it adds the result to the previous value stored in the register

file; moreover it evaluates a flag, named *end_mpy* which indicates that all multiplications are computed and hence the results can go forward in the path.

3. *index_update*: it deals with the indexes of the two polynomials in order to swap all coefficients; also, these indexes are used to address the register file. Finally, when all coefficients are elapsed, the flag *end_mpy* is raised.

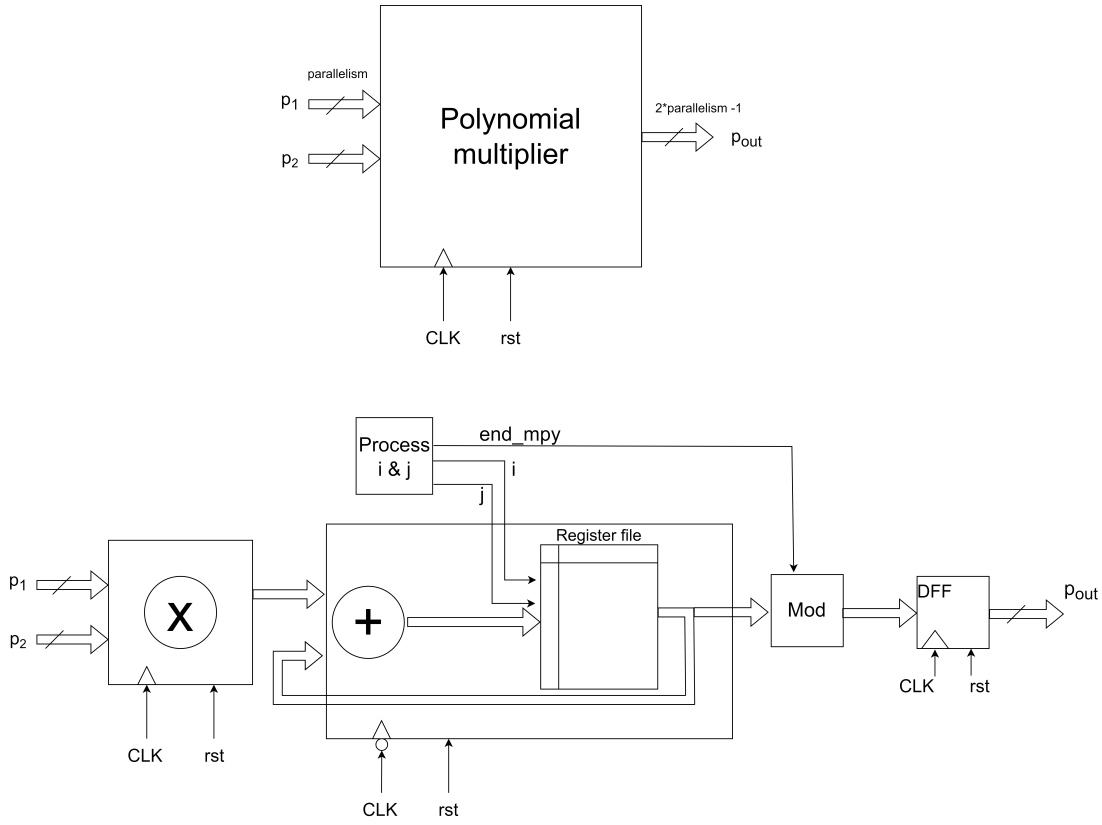


Figure 6.4: Schematic representation of the polynomial multiplication block

For the purpose of reducing the latency, the second process works on the falling edge of the clock, whereas the other two are on the rising edge. In this way, the total latency is halved.
The raising of the flag *end_mpy* enables the *mod q* operation and in the next clock cycle these $size^2$ coefficients are stored in a register.
Now a scaling operation is needed, both to restore the size of the polynomial and to scale the coefficient. This operation is implemented with the datapath shown in

Figure 6.5 which performs a division of the polynomial by the *polynomial modulus*. It is equal to $x^n + 1$. Generally, a division provides a quotient and a reminder, but the wanted scaling operation needs the reminder only. At the end of computation, as usual, the *mod q* operation is performed, and in the following clock cycle the result is stored in a register.

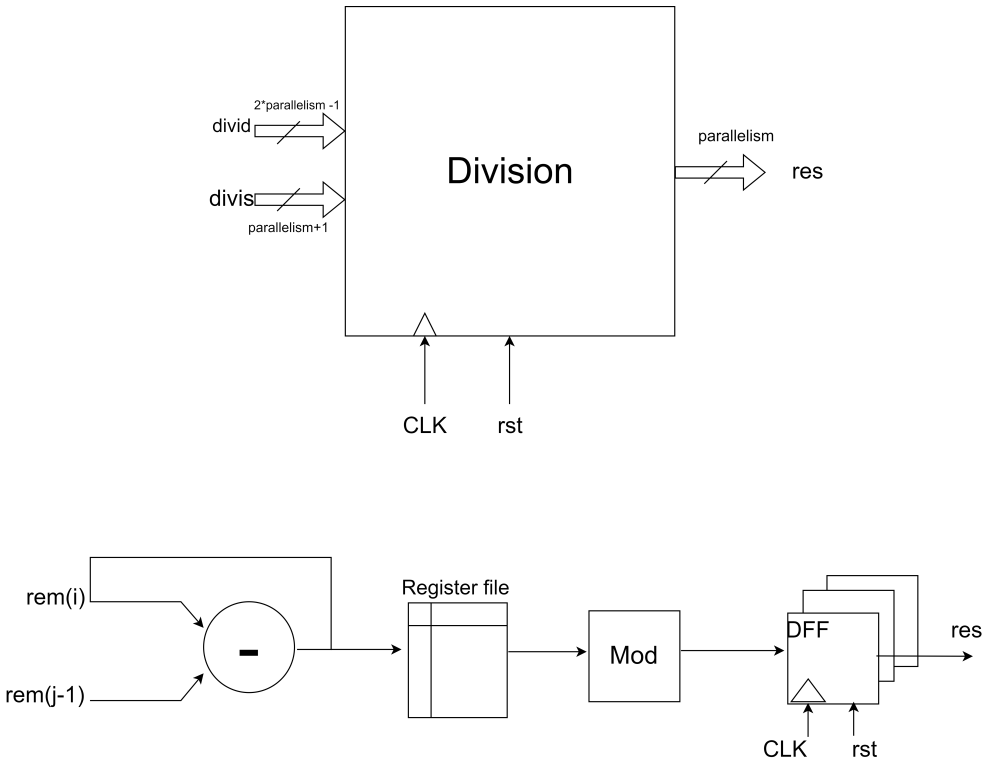


Figure 6.5: Schematic representation of the division block

Moving on in the encryption formula 6.6, the following operation is an addition. Of course, it is the simplest operation in terms both of complexity and area. The unit is depicted in Figure 6.6. The idea is very simple: to sum the coefficients having the same degree in the two polynomials. Now all the operations are complete, and the last thing to do to get the two components of the ciphertexts is the *mod q* operation.

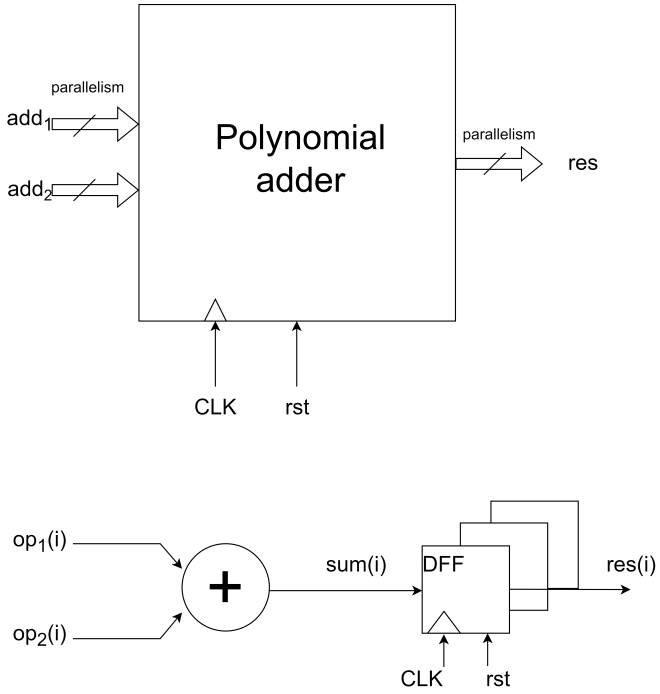


Figure 6.6: Schematic representation of the polynomial adder

6.2.3 Decryption

Similarly to the approach conducted for the encryption, let's recall the decryption equation 6.7 and evaluate the datapath and its internal composition shown in Figure 6.7 and Figure 6.8, respectively.

$$m' = Decrypt(s_k, ct) =$$

$$= \left[\left\lfloor \frac{t \cdot [ct[0] + ct[1] \cdot s]_q}{q} \right\rceil \right]_t$$

The decryption block provides only one result which is the decrypted message. All operations are computed with the same blocks seen before: at first, there is a polynomial multiplication between $ct[1]$ and s, followed by a sum with $ct[0]$ and a simple integer multiplication t/q. Moving on there is a *round to nearest integer* operation indicated as $\lfloor \cdot \rceil$. This rounding is implemented in the following way. Considering, for example, the division 27/7, the algebraic result is 3.86. The result of the integer division is 3, but the rounding to the nearest integer should provide

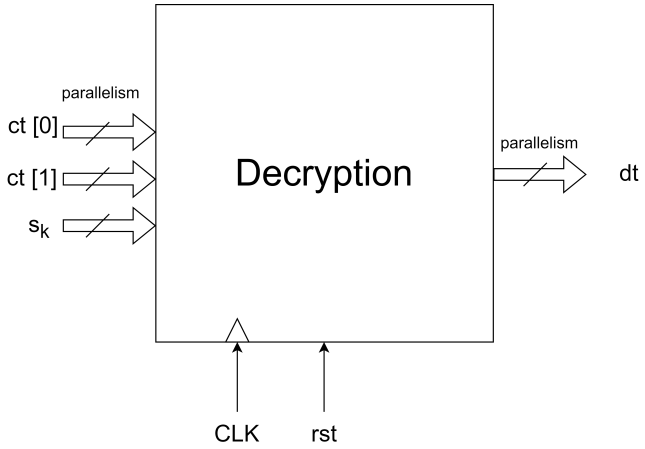


Figure 6.7: Schematic representation of the decryption block

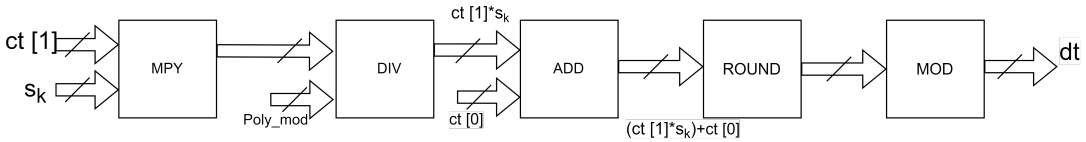


Figure 6.8: Schematic internal representation of the decryption block

4. Let's move to the binary domain:

$$27_2 = 11011 \; and \; 7_2 = 00111$$

Assuming p =number of shift= 3. Let's multiply $27 * 2^p$, that is $27 * 8 = 216$.

$$216_2 = 11011000$$

Now let's perform the division 216/7:the result is 30 plus reminder.

$$30_2 = 00011110$$

Add $0.5 * 2^p = 0.5 * 2^3 = 4$, so the number becomes 34.

$$34_2 = 00100010$$

Finally a number of bits equal to p are truncated and the result is:

$$00100 = 4$$

which is $\lfloor 27/7 \rfloor = \lfloor 3.86 \rfloor = 4$.
So generalizing: let be $c = a/b$ and p =number of shift

1. compute $a' = a * 2^p$
2. compute $c' = a'/b$
3. compute $c'' = c' + 0.5 * 2^p = c' + 2^{p-1}$
4. the result is $c = c''/(2^p)$

6.2.4 Addition between ciphertexts

The addition is one of the two possible operations computable on ciphertexts. It is particularly simple because is similar to a standard polynomial addition. The only difference is the additional scaling operation to reduce the coefficient of the result. The Figure 6.9 shows the block scheme of the function. After the addition of each component of the two ciphertexts, there is the division block in order to scale the result, which finally is stored in a register.

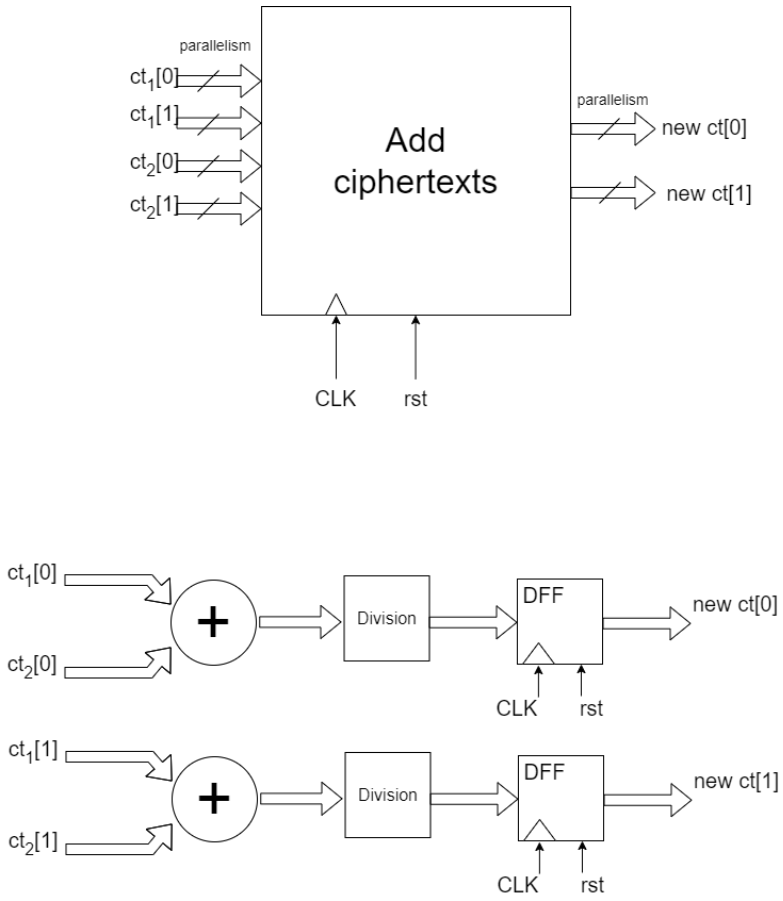


Figure 6.9: Schematic representation of the addition between ciphertexts block

Chapter 7

Evaluation of the implemented design

Now that the design is complete, it can be simulated to verify the proper execution and to evaluate its performance. Appendix B contains the main parts of the VHDL code of the design.

7.1 Simulations

The simulations are carried on with *Modelsim*. While several tests have been conducted, in the following paragraph a specific example is discussed. At first, the Python code is launched and the results provided by the *Key Generation* and *Parameters Generation* functions are set as inputs to the corresponding block. As well as the plaintext message is set ($m = 1011$), and hence the simulation is launched. In Figure 7.1 is shown the start of the simulation of the encryption process. In particular, the simulation regards the computation of $ct[0]$, in fact, the input parameters are $p_k[0], u, e_1$ and m, as displayed on the upper let corner of the image. After the first 3 clock cycles, where the yellow line is placed, the temporary product (*product_temp*) is updated: in fact, a first complete cycle of *index_j* is elapsed. This loop is repeated until all coefficients have been multiplied and this requires 16 clock cycles since the parallelism is set to 4. Then, there are other 3 clock cycles to terminate the computation due to the division, the addition and

the storing in the output register. Hence the whole process takes 20 clock cycles. The Figure 7.2 shows the correct result. The outputs provided by the encryption block, ct[0] and ct[1], are set as inputs to decryption unit to check its behaviour. In this way, its correctness can be easily verified by comparing its results with the original plaintext. The Figure 7.3 depicts the end of the simulation of the decryption process. The red circle indicates the result, confirming the accuracy of execution.

Also in this case, as expected, the whole process lasts 20 clock cycles: the kind and the number of operations are the same compared to the encryption process, except for the integer multiplication and rounding operations that are performed combinatorially.

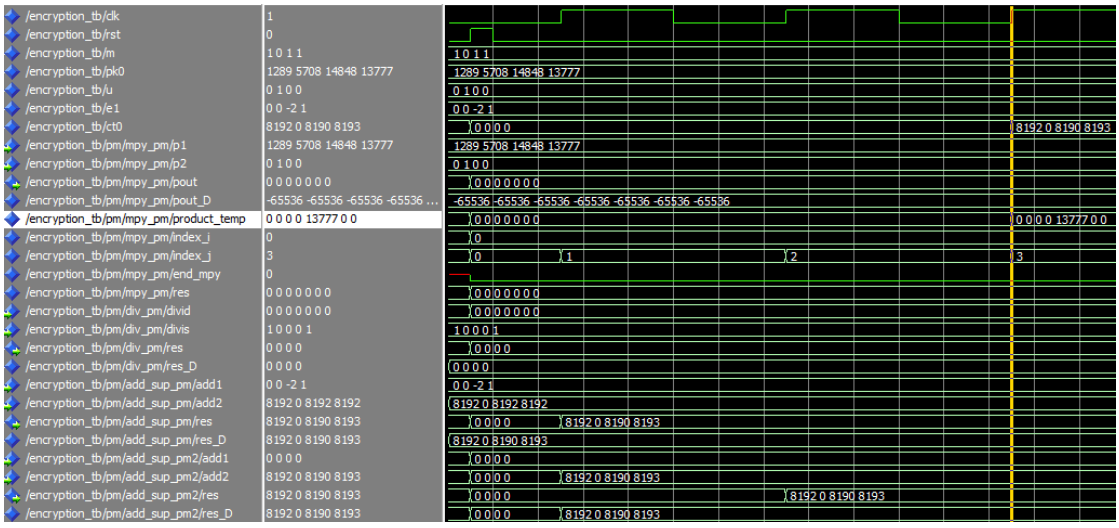


Figure 7.1: Start simulation of the encryption process

The last block to check is the addition between ciphertexts. The plaintexts are $m = 1011$ and $m' = 1101$. The encryption process provides:

$$ct_1 = [(10799, 1289, 13898, 6657), (4158, 6765, 8053, 13761)]$$

$$ct_2 = [(12335, 12087, 6998, 12364), (6780, 10923, 14819, 5432)]$$

The addition requires 3 clock cycles since the operations performed are:

1. addition

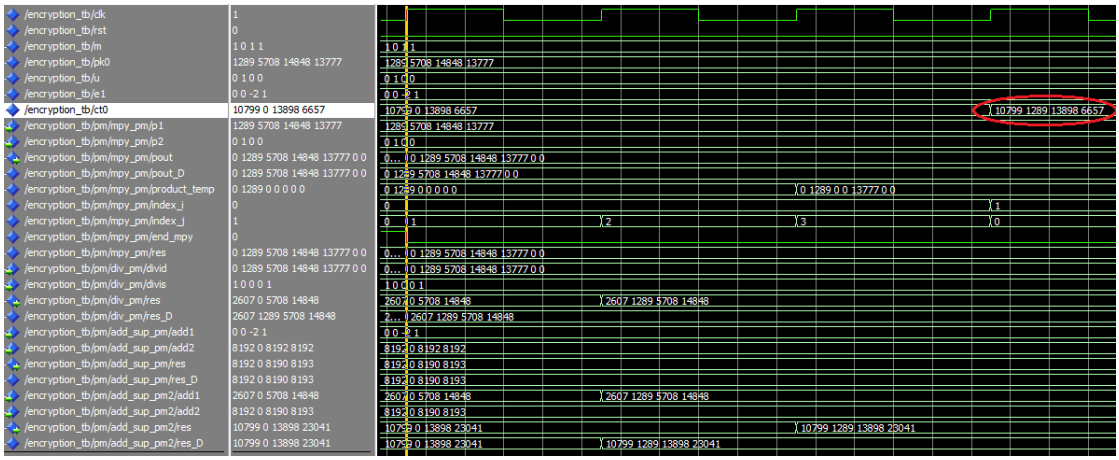


Figure 7.2: End simulation of the encryption process. The red circled number highlights the final result ct[0]

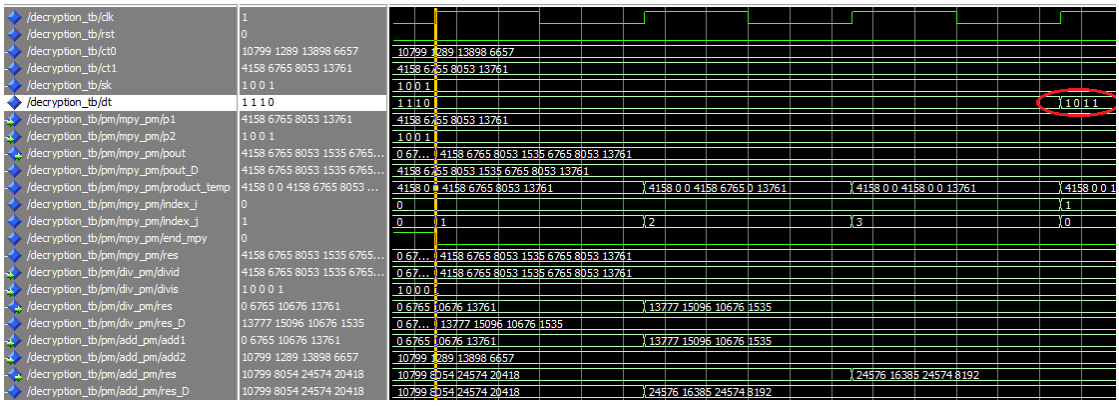


Figure 7.3: End simulation of the decryption process. The red circle highlights the final result dt which is equal to the original plaintext $m = 1011$

2. division

3. storing

The Figure 7.4 shows the simulation and the final results.

7.2 Logic synthesis

After verifying the correct behaviour of the circuits, the synthesis has been performed with *Synopsys Design Compiler* using a $65nm$ technology node. The logical

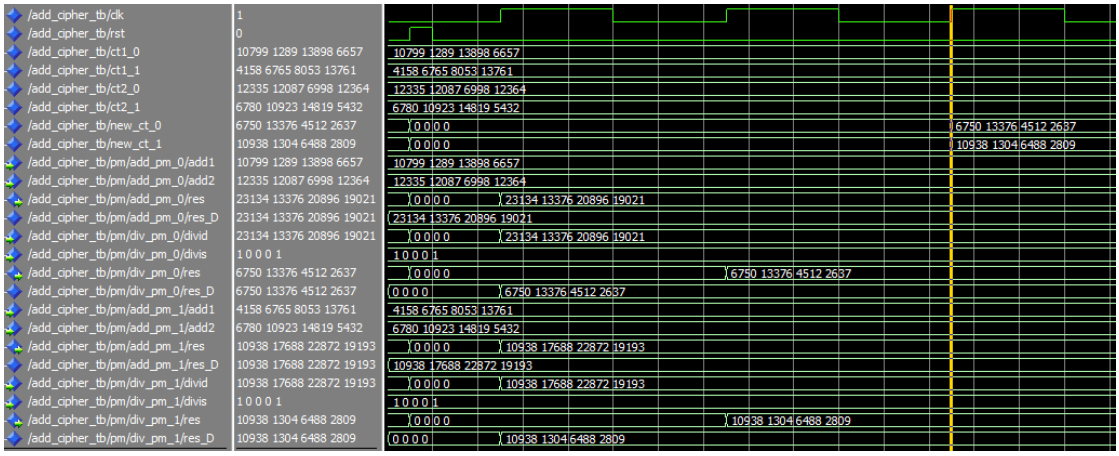


Figure 7.4: Simulation of the addition between ciphertexts. The red circle highlights the final result *new_ct*

synthesis process can be divided into the following steps:

- reading VHDL source files;
- applying constraints;
- start the synthesis;
- save the results.

During the first step, the source files are only read; the unit and all its components are analyzed. Going on, some constraints are applied. At first, a clock signal is created and then, to be closer to reality, also the *uncertainty* of the clock is considered since it can be affected by jitter. Moreover, each signal, at the input or at the output, could arrive with a certain delay with respect to the clock. So a maximum delay is set at both ports.
To verify the timing of the design the command *report_timing* is launched. This command shows the longest path in the design and if the timing requirement imposed with the create clock statement is met or not. When the report states "Met", it does not rule out the possibility to go faster. Hence, to find the maximum clock frequency of the design, the period has been forced to 0 when applying the constraints. Then a new synthesis is run and the timing result is used for the next synthesis. This process is repeated until a *slack* value equal to 0 is achieved and

the report states "Met". The last period set in the clock statement represents the minimum period T_{min}.
At this point, keeping the same clock constraints, the area and power consumption are observed with *the report_area* and *report_power* command respectively. Table 7.1 contains all the above information for the three implemented units.

Unit	f_{max}**[MHz]**	**Area** $[\mu m^2]$	**Power [mW]**
encryption	505	24815.88	9.15
decryption	505	11477.16	4.07
addition between ciphertexts	1052.6	5062.68	5.15

Table 7.1: Report data after the synthesis of the encryption, decryption and addition between ciphertexts unit

As expected, the max frequency of the encryption and decryption unit is the same since they take the same number of clock cycles to complete the execution. Also, the area and the power show a foreseeable trend: in fact, the encryption unit is composed of two equal instances, one to compute ct[0] and one to compute ct[1]. Both of them perform the well-known sequence of operations (multiplication, division and addition). Whereas the decryption is made up of only one instance, with only the difference of the rounding block.
The adder unit is extremely faster, as expected, due to the reduced number of operations to perform. The area is approximately half of the decryption block power: this gap, considering the two internal structures, is due to the overhead of the multiplier in the decryption unit, since the rounding block has no significant effect as demonstrated before. Finally, analyzing the power, in the adder unit, the increase has been caused by the internal power of the registers which are more than in the decryption block, since the operations are computed simultaneously on two inputs, ct_1 and ct_2.

7.2.1 Optimization

Now, the analysis is repeated on an optimized design. Synopsys offers the possibility to deal with the position of the registers in order to optimize the overall structure, issuing the *compile_ultra* command instead that the simple *compile*. The benefits relate not only to timing but also to area and power.

The following histograms compare each unit in the two different compile modes. Thanks to the high effort compile on the design, the minimum period has decreased by 12% for the encryption and decryption block and by around 5% for the adder. The area has reduced of 17%, 19% and 16% respectively. Whereas the power has decreased by 13%, 20% and 18% respectively.

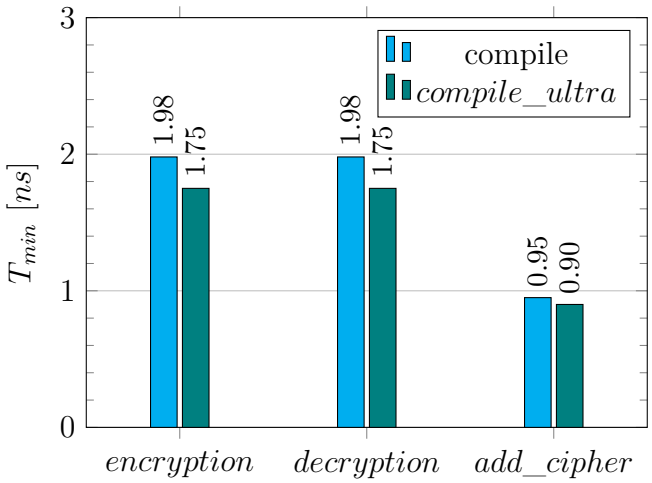


Figure 7.5: Comparison of T_{min} in the two compile modes

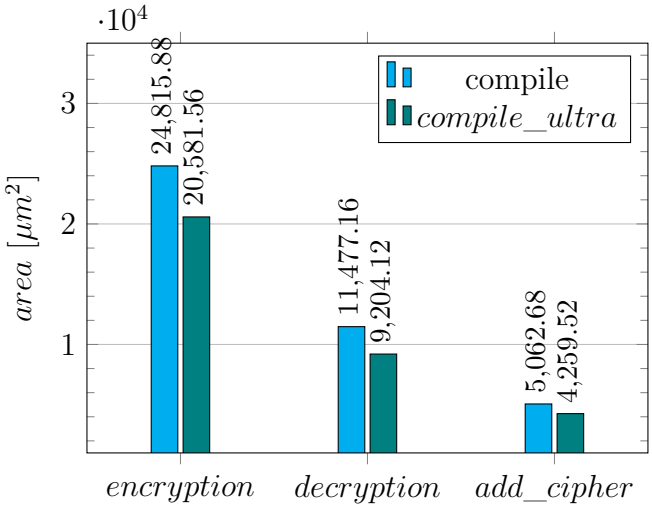


Figure 7.6: Comparison of occupation area in the two compile modes

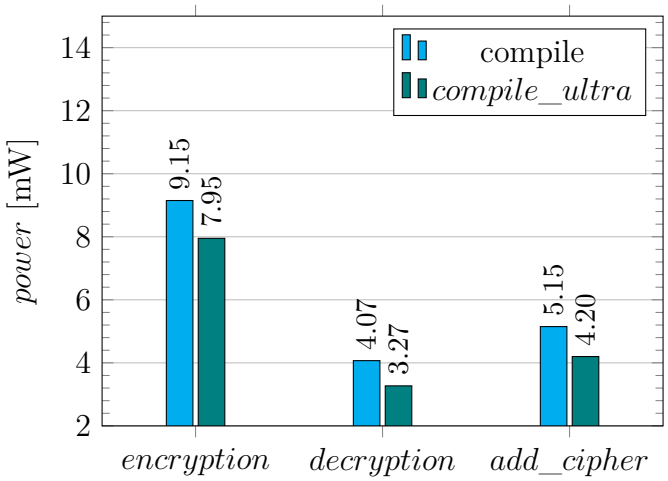


Figure 7.7: Comparison of power in the two compile modes

7.3 Future works

Starting from this design, there are several possible future works. At first, the main task is to complete the SHE scheme, implementing the remaining blocks to execute all operations in ciphertext space, such as the multiplication between ciphertexts and the following relinearization to restore the size. In particular, a widely known approach is the *Number theoretic transform* (**NTT**) based polynomial multiplication. It allows to reduce from the quadratic complexity of a standard polynomial multiplication to a quasi-linear complexity. Moreover, this implementation could optimize this design, taking the place of the polynomial multiplication in plaintext space.

The relinearization operation requires an *evaluation key*. It can be obtained by implementing the *EvalKeyGeneration* function in Python. Then, the relinearization can be performed with two main techniques [55]. Each of them has specific pros and cons, however, the second one called *modulus switching* results simpler to design.

Finally, as soon as the design is complete, the next step is to integrate it in neuromorphic hardware to execute SNN in an encrypted mode. The data owner encrypted the data and sends them to a third party, in this case to an SNN to obtain an encrypted prediction. The application of HE to neural networks implies also rethinking the whole training algorithm. The effect of this process leads to

ensure privacy both on the data and on the prediction since only the data owner can access their actual value.

Chapter 8

Conclusions

Recent developments of ANNs have led to the application of Machine Learning to different fields by trying to emulate the human brain in solving tasks through experience. This widespread has arisen two outcomes. The first was to reduce the power effort and it has enabled a breakthrough in AI: Spiking Neural Networks. Due to their biologically inspired behaviour, the communication between neurons is based on spikes, ensuring a low computational effort. The second challenge is related to the great availability of data, which causes privacy issues since the dataset could contain private and sensitive information.

A Privacy-Preserving Mechanism is Homomorphic Encryption which allows computation over encrypted data. A client encrypts and sends the data to a server that performs the computation without decrypting; finally, the data is sent to the client again who can decrypt it, since he only has the secure key. The cryptographic method includes multiple algorithms which have an expensive computational cost.

This thesis has provided a design of a Somewhat Homomorphic Encryption scheme. The implementation is split into two domains, software and hardware. The Key and other parameters generation functions are carried out in Python. They produce a set of parameters, such as public and private keys, which are drawn over particular distributions. These parameters are nothing else that input data to the encryption and decryption unit designed in VHDL. Each block involves a sequence of operations to encrypt the plaintext, to perform addition between ciphertexts and finally to decrypt the message. The correct behaviour of the whole design is verified with

Modelsim. Then, the performance is evaluated and the results demonstrate the highest complexity of the polynomial multiplier. Finally, the structure is optimized by modifying the position of the registers, achieving considerable improvements in occupation area and dissipated power.

Appendix A

Python functions

This Appendix reports the Python code of the functions used in the *Key Generation*(see Section 6.2).

```
import numpy as np
def gen_binary_poly(size):
    """Generates a polynomial with coeffecients in [0, 1]
    Args:
        size: number of coeffcients, size-1 being the degree of the
            polynomial.
    Returns:
        array of coefficients with the coeff[i] being
        the coeff of x ^ i.
    """
    return np.random.randint(0, 2, size, dtype=np.int64)


def gen_uniform_poly(size, modulus):
    """Generates a polynomial with coeffecients being integers in
    Z_modulus
    Args:
        size: number of coeffcients, size-1 being the degree of the
            polynomial.
    Returns:
        array of coefficients with the coeff[i] being
```

```
21          the coeff of x ^ i.
22      """
23      return np.random.randint(0, modulus, size, dtype=np.int64)
24
25
26  def gen_normal_poly(size, mean, std):
27      """Generates a polynomial with coeffecients in a normal
        distribution
28      of mean mean and a standard deviation std, then discretize it.
29      Args:
30          size: number of coeffcients, size-1 being the degree of the
31              polynomial.
32      Returns:
33          array of coefficients with the coeff[i] being
34          the coeff of x ^ i.
35      """
36      return np.int64(np.random.normal(mean, std, size=size))
```

To conclude, it is worth to describe the *random.randint* and *random.normal* functions from numpy package[65]:

- *random.randint(low, high=None, size=None, dtype=int)*:
  return random integers from low (inclusive) to high (exclusive) sampling from a "discrete uniform" distribution.

- *random.normal(loc=0.0, scale=1.0, size=None)*:
  draw random samples from a Gaussian distribution.

Appendix B

VHDL code

This Appendix reports the VHDL code to implement the functions and their operations as discussed in Section 6.2. Each of these operations is designed as a *component* in VHDL, and the whole encryption unit or decryption one connects them properly to get the final results. For the sake of order and simplicity, only the *architecture* part of the code is inserted.
At first, the following **package** is used:

```
1  package my_package is
2      constant parallelism     : integer range 1 to 64 := 4;
3      constant ciphertext_mod  : integer               := 2 ** 14;
4      constant plaintext_mod   : integer               := 2;
5      constant delta           : integer               := ciphertext_mod
       / plaintext_mod;
6      type vector is array (parallelism - 1 downto 0) of integer range
      -2 * ciphertext_mod to 2 * ciphertext_mod;
7      type vector_out is array (parallelism - 1 downto 0) of integer
      range -4*ciphertext_mod to 4*ciphertext_mod;
8      type vector_out_2 is array (parallelism - 1 downto 0) of integer
      range -6*ciphertext_mod to 6*ciphertext_mod;
9      type divisor is array (parallelism downto 0) of integer range 0
      to 1;
10     constant poly_mod        : divisor               := (0 => 1,
      parallelism => 1, others => 0);
```

```
11      type max_vector is array (2 * parallelism - 2 downto 0) of
        integer range -parallelism * ciphertext_mod to parallelism *
        ciphertext_mod;
12  end my_package;
```

The reason for different *types* of vector definitions is to avoid the *out-of-range* error at the beginning of the simulation. For each operation, the result data type is chosen considering the maximum possible output.

B.1 Polynomial multiplier

```
1   architecture mpy_arch of mpy is
2       signal pout_D       : max_vector;
3       signal product_temp : max_vector;
4       signal index_i      : integer;
5       signal index_j      : integer;
6       signal end_mpy      : std_logic;        --it is useful to flag
        the end of multiplication and so to enable the storage in pout_D
7       signal res          : max_vector;
8   begin
9       product:process(clk, rst)
10          variable product : max_vector;
11      begin
12          if (rst = '1') then
13              product := (others => 0);
14          elsif (clk'event and clk = '1') then
15              product(index_i + index_j) := p1(index_i) * p2(index_j);
16          end if;
17          product_temp <= product;
18      end process;
19
20      reg_update:process(clk, rst, end_mpy)
21          variable reg : max_vector;
22      begin
23          if (rst = '1') then
24              reg := (others => 0);
25          elsif (clk'event and clk = '0') then
```

```
26              reg(index_i + index_j) := reg(index_i + index_j) +
    product_temp(index_i + index_j);
27              if (end_mpy = '1') then
28                  mod_loop : for i in 0 to 2 * parallelism - 2 loop
29                      reg(i) := reg(i) mod ciphertext_mod;
30                  end loop mod_loop;
31                  pout_D <= reg;
32                  reg    := (others => 0);
33              end if;                          --
34          end if;
35      end process;
36
37      index_update:process(clk, rst)
38          variable i : integer;
39          variable j : integer;
40      begin
41          if (rst = '1') then
42              i := 0;
43              j := 0;
44          else
45              if (clk'event and clk = '1') then
46                  if (j < parallelism - 1) then
47                      j       := j + 1;
48                      end_mpy <= '0';
49                  elsif (i < parallelism - 1) then
50                      i       := index_i + 1;
51                      j       := 0;
52                      end_mpy <= '0';
53                  else
54                      i       := 0;
55                      j       := 0;
56                      end_mpy <= '1';
57                  end if;
58              end if;
59          end if;
60          index_i <= i;
61          index_j <= j;
62      end process;
63      reg_out_mpy:process(clk, rst)
64      begin
```

```
65          if (rst = '1') then
66              res <= (others => 0);
67          elsif (clk'event and clk = '1') then
68              if (end_mpy = '1') then
69                  res <= pout_D;
70              end if;
71          end if;
72      end process;
73      pout <= res;
74  end architecture mpy_arch;
```

B.2 Division

```
1  architecture division_arch of division is
2      signal res_D : vector;
3  begin
4      process(divid)
5          variable reminder    : max_vector;
6          variable rem_output : vector;
7          variable j           : integer range 0 to 2 * parallelism - 2;
8          variable i           : integer range 0 to 2 * parallelism;
9
10     begin
11         reminder := divid;
12         i        := (2 * parallelism - 2 - (divid'length - divis'
       length));
13         j        := 1;
14
15         i_loop : while (i <= 2 * parallelism - 2) loop
16             reminder(i) := reminder(i) - reminder(j - 1);
17             i           := i + 1;
18             j           := j + 1;
19         end loop i_loop;
20         rem_loop_3 : for z in (2 * parallelism - 2) downto 3 loop
21             rem_output(z - 3) := reminder(z) mod ciphertext_mod;
22         end loop rem_loop_3;
```

```
23          res_D <= rem_output;
24      end process;
25      reg_out_div:process(clk, rst)
26      begin
27          if (rst = '1') then
28              res <= (others => 0);
29          else
30              if (clk'event and clk = '1') then
31                  res <= res_D;
32              end if;
33          end if;
34      end process;
35  end architecture division_arch;
```

B.3 Addition

```
1   architecture add_arch of add is
2
3       signal res_D : vector_out;
4
5   begin
6       process(add1, add2)
7           variable sum : vector_out;
8       begin
9           sum_loop : for i in 0 to parallelism - 1 loop
10              sum(i) := add1(i) + add2(i);
11          end loop sum_loop;
12          res_D <= sum;
13      end process;
14
15      process(clk, rst)
16      begin
17          if (rst = '1') then
18              res <= (others => 0);
19          else
20              if (clk'event and clk = '1') then
```

```
21                  res <= res_D;
22              end if;
23          end if;
24      end process;
25  end architecture add_arch;
```

B.4 Encryption

```
1  architecture encryption_block_arch of encryption_block is
2
3      signal pk0_u_temp      : max_vector; --vector after the
       multiplication pk0*u
4      signal pk0_u           : vector;     --vector after the division
5      signal e1_delta        : vector_out;     --vector after the
       addition delta+e1
6      signal pk0_u_e1_delta : vector_out_2; --vector after the addition
        pk0_u+e1_delta
7
8      signal pk1_u_temp : max_vector;      --vector after the
       multiplication pk1*u
9      signal pk1_u       : vector;         --vector after the division
10     signal pk1_u_e2    : vector_out;     --vector after the addition
       pk1_u+e2
11
12     signal delta_m : vector;
13     signal ct0_temp:vector_out_2;
14     signal ct1_temp: vector_out;
15
16     component mpy is
17         port(
18             clk   : in  std_logic;
19             rst   : in  std_logic;
20             p1    : in  vector;
21             p2    : in  vector;
22             pout  : out max_vector
23         );
```

```
24      end component mpy;
25
26      component division is
27          port(
28              clk    : in  std_logic;
29              rst    : in  std_logic;
30              divid  : in  max_vector;
31              divis  : in  divisor;
32              res    : out vector
33          );
34      end component division;
35
36      component add is
37          port(
38              clk  : in  std_logic;
39              rst  : in  std_logic;
40              add1 : in  vector;
41              add2 : in  vector;
42              res  : out vector_out
43          );
44      end component add;
45
46      component add_sup is
47          port(
48              clk  : in  std_logic;
49              rst  : in  std_logic;
50              add1 : in  vector;
51              add2 : in  vector_out;
52              res  : out vector_out_2
53          );
54      end component add_sup;
55
56  begin
57
58      mpy_pm : mpy
59          port map(
60              clk  => clk,
61              rst  => rst,
62              p1   => pk0,
63              p2   => u,
```

```
64              pout => pk0_u_temp
65          );
66
67      div_pm : division
68          port map(
69              clk   => clk,
70              rst   => rst,
71              divid => pk0_u_temp,
72              divis => poly_mod,
73              res   => pk0_u
74          );
75
76      add_sup_pm : add
77          port map(
78              clk  => clk,
79              rst  => rst,
80              add1 => e1,
81              add2 => delta_m,
82              res  => e1_delta
83          );
84
85      add_sup_pm2 : add_sup
86          port map(
87              clk  => clk,
88              rst  => rst,
89              add1 => pk0_u,
90              add2 => e1_delta,
91              res  => pk0_u_e1_delta
92          );
93
94      mpy_pm2 : mpy
95          port map(
96              clk  => clk,
97              rst  => rst,
98              p1   => pk1,
99              p2   => u,
100             pout => pk1_u_temp
101         );
102
103     div_pm2 : division
```

```
104         port map(
105             clk    => clk,
106             rst    => rst,
107             divid  => pk1_u_temp,
108             divis  => poly_mod,
109             res    => pk1_u
110         );
111
112     add_sup_pm3 : add
113         port map(
114             clk  => clk,
115             rst  => rst,
116             add1 => pk1_u,
117             add2 => e2,
118             res  => pk1_u_e2
119         );
120
121     delta_m_computation:process(m)
122     begin
123         delta_loop : for index in 0 to parallelism - 1 loop
124             delta_m(index) <= delta * m(index);
125         end loop delta_loop;
126     end process;
127
128     ct0_end:process(pk0_u_e1_delta)
129     begin
130         ct0_loop : for index in 0 to parallelism - 1 loop
131             ct0_temp(index) <= pk0_u_e1_delta(index) mod
    ciphertext_mod;
132         end loop ct0_loop;
133     end process;
134
135     ct1_end:process(pk1_u_e2)
136     begin
137         ct1_loop : for index in 0 to parallelism - 1 loop
138             ct1_temp(index) <= pk1_u_e2(index) mod ciphertext_mod;
139         end loop ct1_loop;
140     end process;
141
142     reg_out_enc:process(clk, rst)
```

```
143     begin
144         if (rst = '1') then
145             ct0 <= (others => 0);
146             ct1 <= (others => 0);
147         else
148             if (clk'event and clk = '1') then
149                 ct0 <= ct0_temp;
150                 ct1 <= ct1_temp;
151             end if;
152         end if;
153     end process;
154 end architecture encryption_block_arch;
```

B.5 Decryption

```
1  architecture decryption_block_arch of decryption_block is
2
3      signal ct1_s_temp : max_vector;      --ct1*s
4      signal ct1_s      : vector;          --ct1*s after the division
5      signal plus_ct0   : vector_out;      --ct0+ct1*s
6      signal dt_D:vector_out;
7
8      component mpy is
9          port(
10             clk  : in  std_logic;
11             rst  : in  std_logic;
12             p1   : in  vector;
13             p2   : in  vector;
14             pout : out max_vector
15         );
16     end component mpy;
17
18     component division is
19         port(
20             clk   : in  std_logic;
21             rst   : in  std_logic;
```

```
22              divid : in  max_vector;
23              divis : in  divisor;
24              res   : out vector
25          );
26      end component division;
27
28      component add is
29          port(
30              clk  : in  std_logic;
31              rst  : in  std_logic;
32              add1 : in  vector;
33              add2 : in  vector;
34              res  : out vector_out
35          );
36      end component add;
37
38  begin
39      mpy_pm : mpy
40          port map(
41              clk  => clk,
42              rst  => rst,
43              p1   => ct1,
44              p2   => sk,
45              pout => ct1_s_temp
46          );
47
48      div_pm : division
49          port map(
50              clk   => clk,
51              rst   => rst,
52              divid => ct1_s_temp,
53              divis => poly_mod,
54              res   => ct1_s
55          );
56
57      add_pm : add
58          port map(
59              clk  => clk,
60              rst  => rst,
61              add1 => ct1_s,
```

```
62              add2 => ct0,
63              res  => plus_ct0
64          );
65      dt_end:process(plus_ct0)
66          variable dt_temp     : vector_out;
67          variable dt_end      : vector_out;
68      begin
69          mod_loop : for index in 0 to parallelism - 1 loop
70              dt_temp(index) := ((plus_ct0(index) mod ciphertext_mod) *
     plaintext_mod * 2) / ciphertext_mod;
71              dt_temp(index):=(dt_temp(index)+1)/2;
72              dt_end(index):=dt_temp(index) mod plaintext_mod;
73          end loop mod_loop;
74          dt_D <= (dt_end);
75      end process;
76
77      reg_out_dec:process(clk, rst)
78      begin
79          if (rst = '1') then
80              dt <= (others => 0);
81          else
82              if (clk'event and clk = '1') then
83                  dt <= dt_D;
84              end if;
85          end if;
86      end process;
87  end architecture decryption_block_arch;
```

B.6 Addition between ciphertexts

```
1  architecture add_cipher_block_arch of add_cipher_block is
2      signal ct_0_temp     : vector_out;
3      signal ct_1_temp     : vector_out;
4      signal new_ct_0_temp : vector;
5      signal new_ct_1_temp : vector;
6
```

```
7     component add is
8         port(
9             clk  : in  std_logic;
10            rst  : in  std_logic;
11            add1 : in  vector;
12            add2 : in  vector;
13            res  : out vector_out
14        );
15    end component add;
16
17    component div_cipher is
18        port(
19            clk   : in  std_logic;
20            rst   : in  std_logic;
21            divid : in  vector_out;
22            divis : in  divisor;
23            res   : out vector
24        );
25    end component div_cipher;
26
27 begin
28    add_pm_0 : add
29        port map(
30            clk  => clk,
31            rst  => rst,
32            add1 => ct1_0,
33            add2 => ct2_0,
34            res  => ct_0_temp
35        );
36
37    div_pm_0 : div_cipher
38        port map(
39            clk   => clk,
40            rst   => rst,
41            divid => ct_0_temp,
42            divis => poly_mod,
43            res   => new_ct_0_temp
44        );
45
46    add_pm_1 : add
```

```
47          port map(
48              clk  => clk,
49              rst  => rst,
50              add1 => ct1_1,
51              add2 => ct2_1,
52              res  => ct_1_temp
53          );
54
55      div_pm_1 : div_cipher
56          port map(
57              clk   => clk,
58              rst   => rst,
59              divid => ct_1_temp,
60              divis => poly_mod,
61              res   => new_ct_1_temp
62          );
63
64      reg_out_add_chiper : process(clk, rst)
65      begin
66          if (rst = '1') then
67              new_ct_0 <= (others => 0);
68              new_ct_1 <= (others => 0);
69          else
70              if (clk'event and clk = '1') then
71                  new_ct_0 <= new_ct_0_temp;
72                  new_ct_1 <= new_ct_1_temp;
73              end if;
74          end if;
75      end process;
76  end architecture add_cipher_block_arch;
```

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 2).

[2] Michael Nielsen. *Using neural nets to recognize handwritten digits*. 2019. URL: http://neuralnetworksanddeeplearning.com/chap1.html (cit. on p. 2).

[3] IBM Cloud Education. *Convolutional Neural Networks*. 2020. URL: https://www.ibm.com/cloud/learn/convolutional-neural-networks (cit. on p. 5).

[4] *A Comprehensive Guide to Convolutional Neural Networks*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (cit. on pp. 7, 8).

[5] Sambit Mohapatra, Heinrich Gotzig, Senthil Yogamani, Stefan Milz, and Raoul Zollner. *Exploring Deep Spiking Neural Networks for Automated Driving Applications*. 2019. DOI: 10.48550/ARXIV.1903.02080. URL: https://arxiv.org/abs/1903.02080 (cit. on pp. 8, 9).

[6] A. L. Hodgkin and A. F. Huxley. «A quantitative description of membrane current and its application to conduction and excitation in nerve». In: *The Journal of physiology vol. 117,4 (1952): 500-44* (1952). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413 (cit. on pp. 10, 11).

[7] Masoud Amiri, Soheila Nazari, and Karim Faez. «Digital realization of the proposed linear model of the Hodgkin-Huxley neuron». In: *International Journal of Circuit Theory and Applications* 47 (Feb. 2019). DOI: 10.1002/cta.2596 (cit. on p. 11).

[8] Nicolas Brunel and Mark van Rossum. «Quantitative investigations of electrical nerve excitation treated as polarization: Louis Lapicque 1907». In: *Biological Cybernetics* 97 (Dec. 2007), pp. 341–349. DOI: 10.1007/s00422-007-0189-6 (cit. on p. 12).

[9] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. «Spiking Neural Networks Hardware Implementations and Challenges». In: *ACM Journal on Emerging Technologies in Computing Systems* 15.2 (June 2019), pp. 1–35. DOI: 10.1145/3304103. URL: https://doi.org/10.1145%2F3304103 (cit. on pp. 13–15).

[10] E.M. Izhikevich. «Simple model of spiking neurons». In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572. DOI: 10.1109/TNN.2003.820440 (cit. on p. 15).

[11] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. «Training Deep Spiking Neural Networks Using Backpropagation». In: *Frontiers in Neuroscience* 10 (2016). ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00508. URL: https://www.frontiersin.org/article/10.3389/fnins.2016.00508 (cit. on p. 18).

[12] Chankyu Lee, Syed Shakib Sarwar, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Kaushik Roy. «Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures». In: *Frontiers in Neuroscience* 14 (2020). ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00119. URL: https://www.frontiersin.org/article/10.3389/fnins.2020.00119 (cit. on p. 18).

[13] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. «Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification». In: *Frontiers in Neuroscience* 11 (2017). ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. URL: https://www.frontiersin.org/article/10.3389/fnins.2017.00682 (cit. on p. 18).

[14] Gopalakrishnan Srinivasan, Priyadarshini Panda, and Kaushik Roy. «STDP-Based Unsupervised Feature Learning Using Convolution-over-Time in Spiking Neural Networks for Energy-Efficient Neuromorphic Computing». In: *J. Emerg. Technol. Comput. Syst.* 14.4 (Nov. 2018). ISSN: 1550-4832. DOI: 10.1145/3266229. URL: https://doi.org/10.1145/3266229 (cit. on p. 19).

[15] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. *Privacy in Deep Learning: A Survey*. 2020. DOI: 10.48550/ARXIV.2004.12254. URL: https://arxiv.org/abs/2004.12254 (cit. on pp. 21, 22, 25, 27, 30, 31).

[16] Sara Marie Mc Carthy, Arunesh Sinha, Milind Tambe, and Pratyusa Manadhata. «Data Exfiltration Detection and Prevention: Virtually Distributed POMDPs for Practically Safer Networks». In: Nov. 2016. DOI: 10.1007/978-3-319-47413-7 (cit. on p. 22).

[17] Ximeng Liu, Lehui Xie, Yaopeng Wang, Jian Zou, Jinbo Xiong, Zuobin Ying, and Athanasios V. Vasilakos. «Privacy and Security Issues in Deep Learning: A Survey». In: *IEEE Access* 9 (2021), pp. 4566–4593. DOI: 10.1109/ACCESS.2020.3045078 (cit. on pp. 22, 26, 28, 31).

[18] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. *Membership Inference Attacks against Machine Learning Models*. 2016. DOI: 10.48550/ARXIV.1610.05820. URL: https://arxiv.org/abs/1610.05820 (cit. on p. 23).

[19] Ahmed Salem, Yang Zhang, Mathias Humbert, Mario Fritz, and Michael Backes. *ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models*. June 2018 (cit. on p. 23).

[20] Congzheng Song and Vitaly Shmatikov. *Auditing Data Provenance in Text-Generation Models*. 2018. DOI: 10.48550/ARXIV.1811.00513. URL: https://arxiv.org/abs/1811.00513 (cit. on p. 23).

[21] Shagufta Mehnaz, Ninghui Li, and Elisa Bertino. *Black-box Model Inversion Attribute Inference Attacks on Classification Models*. Dec. 2020 (cit. on p. 23).

[22] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. «Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing». In: *Proceedings of the ... USENIX Security Symposium. UNIX Security Symposium* 2014 (Aug. 2014), pp. 17–32 (cit. on p. 23).

[23] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. «Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures». In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1322–1333. ISBN: 9781450338325. DOI: 10.1145/2810103.2813677. URL: https://doi.org/10.1145/2810103.2813677 (cit. on p. 23).

[24] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. *Stealing Machine Learning Models via Prediction APIs*. 2016. DOI: 10.48550/ARXIV.1609.02943. URL: https://arxiv.org/abs/1609.02943 (cit. on p. 24).

[25] Mathias P. M. Parisot, Balazs Pejo, and Dayana Spagnuelo. *Property Inference Attacks on Convolutional Neural Networks: Influence and Implications of Target Model's Complexity*. 2021. DOI: 10.48550/ARXIV.2104.13061. URL: https://arxiv.org/abs/2104.13061 (cit. on p. 24).

[26] Latanya Sweeney. «k-Anonymity: A Model for Protecting Privacy». In: *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 10.5 (Oct. 2002), pp. 557–570. ISSN: 0218-4885. DOI: 10.1142/S0218488502001648. URL: https://doi.org/10.1142/S0218488502001648 (cit. on pp. 24, 25).

[27] Arvind Narayanan and Vitaly Shmatikov. «Robust De-anonymization of Large Sparse Datasets». In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 2008, pp. 111–125. DOI: 10.1109/SP.2008.33 (cit. on p. 25).

[28] Charu C. Aggarwal. «On k-Anonymity and the Curse of Dimensionality». In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 901–909. ISBN: 1595931546 (cit. on p. 25).

[29] Cynthia Dwork. «Differential Privacy: A Survey of Results». In: vol. 4978. Apr. 2008, pp. 1–19. ISBN: 978-3-540-79227-7. DOI: 10.1007/978-3-540-79228-4_1 (cit. on p. 26).

[30] Chris Peikert. «A Decade of Lattice Cryptography». In: *Found. Trends Theor. Comput. Sci.* 10.4 (Mar. 2016), pp. 283–424. ISSN: 1551-305X. DOI: 10.1561/0400000074. URL: https://doi.org/10.1561/0400000074 (cit. on pp. 26, 33).

[31] Peter Kairouz, Jiachun Liao, Chong Huang, and L. Sankar. «Censored and Fair Universal Representations using Generative Adversarial Models». In: *arXiv: Learning* (2019) (cit. on p. 27).

[32] Nhat Hai Phan, Yue Wang, Xintao Wu, and Dejing Dou. «Differential Privacy Preservation for Deep Auto-Encoders: an Application of Human Behavior Prediction (AAAI-16) [oral presentation]». In: Feb. 2016 (cit. on p. 28).

[33] Nicolas Papernot, Martín Abadi, Úlfar Erlingsson, Ian Goodfellow, and Kunal Talwar. *Semi-supervised Knowledge Transfer for Deep Learning from Private Training Data*. 2016. DOI: 10.48550/ARXIV.1610.05755. URL: https://arxiv.org/abs/1610.05755 (cit. on p. 28).

[34] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. «GAZELLE: A Low Latency Framework for Secure Neural Network Inference». In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1651–1669. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar (cit. on p. 29).

[35] Nisarg Raval, Ashwin Machanavajjhala, and Jerry Pan. «Olympus: Sensor Privacy through Utility Aware Obfuscation». In: *Proceedings on Privacy Enhancing Technologies* 2019.1 (2019), pp. 5–25. DOI: doi:10.2478/popets-2019-0002. URL: https://doi.org/10.2478/popets-2019-0002 (cit. on p. 29).

[36] Seyed Ali Osia, Ali Taheri, Ali Shahin Shamsabadi, Kleomenis Katevas, Hamed Haddadi, and Hamid R. Rabiee. *Deep Private-Feature Extraction*. 2018. DOI: 10.48550/ARXIV.1802.03151. URL: https://arxiv.org/abs/1802.03151 (cit. on p. 29).

[37] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. «Shredder: Learning Noise Distributions to Protect Inference Privacy». In: Mar. 2020, pp. 3–18. DOI: 10.1145/3373376.3378522 (cit. on p. 29).

[38] Peter Kairouz et al. *Advances and Open Problems in Federated Learning*. 2019. DOI: 10.48550/ARXIV.1912.04977. URL: https://arxiv.org/abs/1912.04977 (cit. on p. 30).

[39] Otkrist Gupta and Ramesh Raskar. *Distributed learning of deep neural network over multiple agents*. 2018. DOI: 10.48550/ARXIV.1810.06060. URL: https://arxiv.org/abs/1810.06060 (cit. on p. 31).

[40] VF Rocha, Julio López, and V Falcão Da Rocha. *An Overview on Homomorphic Encryption Algorithms*. 2019 (cit. on p. 32).

[41] Joël Cathebras. «Hardware acceleration for homomorphic encryption». PhD thesis. Université Paris-Saclay, 2018 (cit. on p. 33).

[42] Priasnyomo Santoso, Elkin Rilvani, Ahmad Trisnawan, Krisna Adiyarta, Darmawan Napitupulu, Tata Sutabri, and Robbi Rahim. «Systematic literature review: comparison study of symmetric key and asymmetric key algorithm». In: *IOP Conference Series: Materials Science and Engineering* 420 (Oct. 2018), p. 012111. DOI: 10.1088/1757-899X/420/1/012111 (cit. on p. 34).

[43] Craig Gentry. «Computing Arbitrary Functions of Encrypted Data». In: *Commun. ACM* 53 (Mar. 2010), pp. 97–105. DOI: 10.1145/1666420.1666444 (cit. on pp. 34, 40).

[44] Paulo Martins, Leonel Sousa, and Artur Mariano. «A survey on fully homomorphic encryption: An engineering perspective». In: *ACM Computing Surveys (CSUR)* 50.6 (2017), pp. 1–33 (cit. on p. 36).

[45] Ron Rothblum. «Homomorphic Encryption: From Private-Key to Public-Key». In: *Theory of Cryptography*. Ed. by Yuval Ishai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 219–234. ISBN: 978-3-642-19571-6 (cit. on p. 36).

[46] R. L. Rivest, A. Shamir, and L. Adleman. «A Method for Obtaining Digital Signatures and Public-Key Cryptosystems». In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (cit. on p. 37).

[47] Ronald L. Rivest and Michael L. Dertouzos. «ON DATA BANKS AND PRIVACY HOMOMORPHISMS». In: 1978 (cit. on p. 37).

[48] Aadesh Neupane. *A brief history on Homomorphic learning: A privacy-focused approach to machine learning*. 2020. DOI: 10.48550/ARXIV.2009.04587. URL: https://arxiv.org/abs/2009.04587 (cit. on p. 37).

[49] T. Sander, A. Young, and Moti Yung. «Non-interactive cryptocomputing for NC/sup 1/». In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 554–566. DOI: 10.1109/SFFCS.1999.814630 (cit. on p. 37).

[50] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. «Evaluating 2-DNF formulas on ciphertexts». In: *Theory of cryptography conference*. Springer. 2005, pp. 325–341 (cit. on p. 37).

[51] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009 (cit. on p. 37).

[52] Craig Gentry and Shai Halevi. «Implementing gentry's fully-homomorphic encryption scheme». In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2011, pp. 129–148 (cit. on pp. 38, 44).

[53] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. «(Leveled) Fully Homomorphic Encryption without Bootstrapping». In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325. ISBN: 9781450311151. DOI: 10.1145/2090236.2090262. URL: https://doi.org/10.1145/2090236.2090262 (cit. on pp. 39, 41).

[54] Zvika Brakerski. «Fully homomorphic encryption without modulus switching from classical GapSVP». In: *Annual Cryptology Conference*. Springer. 2012, pp. 868–886 (cit. on p. 39).

[55] Junfeng Fan and Frederik Vercauteren. «Somewhat practical fully homomorphic encryption». In: *Cryptology ePrint Archive* (2012) (cit. on pp. 39, 40, 45, 65).

[56] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. «Homomorphic encryption for arithmetic of approximate numbers». In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 409–437 (cit. on p. 39).

[57] Craig Gentry, Amit Sahai, and Brent Waters. «Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based». In: *Annual Cryptology Conference*. Springer. 2013, pp. 75–92 (cit. on p. 39).

[58] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. «SHIELD: scalable homomorphic implementation of encrypted data-classifiers». In: *IEEE Transactions on Computers* 65.9 (2015), pp. 2848–2858 (cit. on p. 39).

[59] Jacob Alperin-Sheriff and Chris Peikert. «Faster bootstrapping with polynomial error». In: *Annual Cryptology Conference*. Springer. 2014, pp. 297–314 (cit. on p. 40).

[60] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. «TFHE: fast fully homomorphic encryption over the torus». In: *Journal of Cryptology* 33.1 (2020), pp. 34–91 (cit. on p. 40).

[61] Oded Regev. «On lattices, learning with errors, random linear codes, and cryptography». In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40 (cit. on p. 41).

[62] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. «On Ideal Lattices and Learning with Errors over Rings». In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13190-5 (cit. on p. 41).

[63] Ravital Solomon. *An Intro to Fully Homomorphic Encryption for Engineers*. 2020. URL: `https://blog.nucypher.com/an-engineers-guide-to-fully-homomorphic-encryption/` (cit. on p. 42).

[64] Ayoub Benaissa. *Build an homomorphic encryption scheme from scratch with Python*. 2020. URL: https://blog.openmined.org/build-an-homomorphic-encryption-scheme-from-scratch-with-python/ (cit. on p. 50).

[65] *Random sampling (numpy.random)*. URL: https://numpy.org/doc/stable/reference/random/index.html (cit. on p. 70).