Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Ågot Marianne Stornes

Cryptographically Private Linear Regression

Master's thesis in Mathematical Sciences
Supervisor: Kristian Gjøsteen
December 2021

NTNU
Norwegian University of
Science and Technology

Ågot Marianne Stornes

Cryptographically Private Linear Regression

Master's thesis in Mathematical Sciences
Supervisor: Kristian Gjøsteen
December 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Norwegian University of
Science and Technology

Abstract

Machine learning and big data analysis are important topics in the digital world, but they come with privacy concerns. In this text, we will look at at how one can use homomorphic encryption to preserve the privacy of both the data and the algorithm used to analyze the data. We will look at at linear regression, and briefly at logistic regression, as these are powerful tools that lay the foundation for other algorithms. We will create an algorithm for how to do this, and then prove it secure. We will also look at a toy implementation of the system, to show proof of concept.

Sammendrag

Maskinlæring og stordatabehandling er viktige tema i den digitale verden, men her finnes det ufordringer med å holde dataene privat. I denne teksten skal vi se på hvordan en kan bruke homomorf kryptering for å holde både dataene og algoritmen brukt til å analysere privat. Vi skal se på lineær og logistisk regresjon, siden dette er kraftige verktøy som legger grunnlaget for mange andre algoritmer. Vi skal så lage en algoritme for å gjøre dette, og bevise at den er kryptografisk sikker. Til slutt skal vi lage en liten implementasjon av systemet mitt, for å vise at det er gjennomførbart i praksis.

Dedication

To Marianne, for believing, and to Laura, the light in my world

Contents

Chapter 1

Introduction

Machine learning and big data analysis has given us big gains in functionality and are very powerful tools. This does not come without challenges. Sometimes the data the scientists wishes to analyse is of a sensitive nature. It could be medical data, position data, other privacy related issues, or simply just trade secrets. Companies often do not have the know-how to analyse these data themselves, so they will buy services from an analyst firm. The analyst firm on their side might not have the computing power to run their algorithm, so they rent computing time from a cloud service provider. This leads to the data being spread out to several outside companies, and a lot of people could hypothetically gain access to the data. This could lead to privacy scandals, or trade espionage if the data is not protected.

Several solutions has been proposed to deal with this, from adding noise to the data, obscuring the data, to cryptographically protecting the data. All of these comes with their advantages and challenges. Ever since statistical methods for big data analysis started to gain traction in the early 2000's, there has been attempts to preserve the privacy of the data. For example Vaidya, Yu, and Jiang wrote an article on privacy-preserving support vector machines in 2006[1]. These methods are focused on how to anonymise the data before analysis, not on how to protect it with cryptography. Anonymising or fuzzing the data is an interesting problem, but it leans more towards statistics and statistical methods. In this thesis, we want to look more at cryptographic protocols to protect the data.

With the emergence of homomorphic encryption, it has been possible to do simple calculations on encrypted data. This opened a new world of options in protecting the data sets. Now, one could encrypt the data and send it off, instead of - or in addition to - concealing the sensitive elements of the data, using the

already established methods of privacy-preserving.

Several authors wrote about homomorphic encryption, but it was not until Gentry's text on Fully Homomorphic Encryption in 2009[2], it was feasible to do homomorphic encryption on arbitrary data. Later on, Brakerski, Gentry, and Vaikuntanathan[3] refined this, to a scheme without so-called bootstrapping, which improved the performance and strength of the cryptographic scheme. This allowed researchers to create cryptographic privacy-preserving schemes, such as the one designed by Rahulamathavan et.al, Privacy-Preserving Multi-Class Support Vector Machine for Outsourcing the Data Classification in Cloud [4] in 2014.

In this text, we will look at machine learning problems, and look for an algorithm or a collection of algorithms that could be made cryptographically private using the homomorphic encryption scheme by Brakerski et.al, BGV [3]. We also need to find something that has not already been done before, but since this field is fairly new, it was possible to find several algorithms that had not been studied by others.

We will then write an algorithm that does homomorphic encryption on linear regression, and we will also look briefly at logistic regression. This is because linear regression is a very powerful tool, that has many applications. In addition, linear regression is the basis for polynomial regression, so it could be possible to extend the work at a later point.

A sketch of the algorithm that will be developed in this text, can be seen in Figure 1.1. The algorithm owner will train the machine learning algorithm, using some training data, and then encrypt the classifier. The data owner will gather the data and encrypt this. They will then send the encrypted data and classifier into the circuit evaluation. This circuit step is what will be studied in this text.

Finally, to show proof of concept, we will make a toy implementation of the system, using the PALISADE library, and the code will be written in in C++.

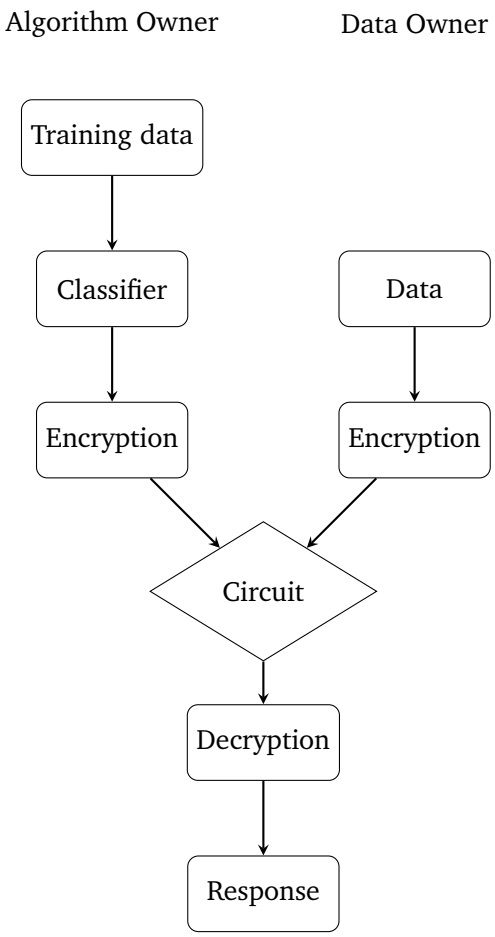


Figure 1.1: The general idea of the algorithm

Chapter 2

Algebraic background

In this chapter we will introduce some concepts that is needed to understand the rest of the text. However, we will assume these topics are known to the reader, and will not go into detail.

2.1 Rings

In this section, we will talk briefly about rings, and introduce some notation on rings used in this text. More on rings can be found in for example Bhattacharya et. al.'s book Basic Abstract Algebra [5].

2.1.1 Polynomial rings

Definition 1 (Polynomial ring)**.** A polynomial ring $R[X]$ is the set of elements, called polynomials, of the form

$$p = p_0 + p_1 X + p_2 X^2 + \cdots + p_m X^m \tag{2.1}$$

where the p_i belongs to a ring R, and X is a variable.

In this text, the ring R where the elements are taken from will usually be $\mathbb{Z}$. This means that the p_i's will be integers.

Definition 2 (Quotient polynomial ring)**.** a quotient polynomial ring is a quotient ring where the ideal divided out is defined by a polynomial in the polynomial ring R.

$$R = \mathbb{Z}[X]/f(X) \tag{2.2}$$

In this text, this $f(X)$ will be on the form $X^d + 1$, where d is a power of 2.

Abuse of notation

In this text, we will sometimes refer to R_q as the ring with coefficients less than q, where q is an integer. So if it is an ordinary integer ring, $R_2 = \{0, 1\}$. If it is a polynomial ring, we refer to the ring where the polynomials have coefficients in $\{0, 1\}$.

2.2 Discrete probability distributions

Since we are working over discrete spaces, we need to define discrete probability. A probability distribution χ that is continuous could be made discrete by the discretisation process. An example of this, is making a histogram out of continuous data.

Definition 3 (Discrete uniform distribution)**.** Let $S = \mathbb{Z}_n$. The probability of drawing any $m \in S$ is

$$P(m) = \frac{1}{n} \tag{2.3}$$

The discrete uniform distribution can be states as: A known, finite number of events is equally likely to happen.

Definition 4 (Discrete Gaussian distribution)**.** For any $s > 0$ define the Gaussian function on $\mathbb{R}^n$ centered at the point $\mathbf{c}$ with parameter s:

$$\forall \mathbf{x} \in \mathbb{R}^n, \rho_{s,\mathbf{c}}(\mathbf{x}) = e^{(-\pi||\mathbf{x}-\mathbf{c}||^2/s^2)} \tag{2.4}$$

If this was a one-dimensional Gaussian, $\mathbf{c}$ would be the expected value, or the centre of the distribution, and s would be the variance. When the subscripts of ρ, s and $\mathbf{c}$, are omitted, they are implicitly 1 and $\mathbf{0}$ respectively.

2.3 Lattices

In this section, we will briefly discuss lattices, as they are the foundation of the BGV scheme. More on lattices can be found in for example Lenstra's text Lattices [6].

Definition 5 (Basis)**.** Let $\mathbb{R}^n$ be the usual n-dimensional vector space over $\mathbb{R}$. A basis $\mathbf{B}$ is a set of l linearily independent vectors in $\mathbb{R}^n$, where $l \leq n$.

Definition 6 (Lattice)**.** Let $\mathbb{R}^n$ be the usual n-dimensional vector space over $\mathbb{R}$, let R be a ring, and let $\mathbf{B} = \{v_1, \ldots, v_n\}$ be a basis for $\mathbb{R}^n$. Then the integral lattice $\mathcal{L}$ in $\mathbb{R}^n$ generated by $\mathbf{B}$ is given by:

$$\mathcal{L} = \{\sum_{i=1}^{n} a_i v_i | a_i \in R, v_i \in \mathbb{Z}\}$$

The rank of a lattice is l, and the dimension is n. If the rank equals the dimension, the lattice is called a full rank lattice.

Definition 7 (Fundamental domain)**.** The fundamental domain $\mathcal{F}(\mathbf{B})$ of a lattice, is the area that is spanned by

$$\mathcal{F}(\mathbf{B}) = \{t_1 v_1, t_2 v_2, \ldots, t_n v_n : 0 \leq t_i < 1\}$$

The fundamental domain is an area spanned by the lattice points. Any point in $\mathbb{R}^n$ can be reached by first pointing to the fundamental domain, and then by some small vector pointing to a point inside the fundamental domain. If the basis vectors are "good" it is easy to find this combination of vectors, but if the basis vectors are "bad", the fundamental domain will have an odd shape, and the combination of vectors to point to a point will be hard to find. This is the basis of the closest vector problem, which is related to the Shortest Vector Problem, but this reduction is of scope for this text.

The length of vectors in a lattice is usually calculated using the ℓ^2 norm.

Definition 8 (ℓ^2 norm)**.** The ℓ^2 norm is calculated using the following formula:

$$||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \tag{2.5}$$

Where x_i is the i'th coordinate coefficient of the vector.

2.3.1 Shortest Vector Problem

The existence of vector length gives rise to the question; what is the shortest vector in the lattice? This is a NP-hard problem, that has seen a lot of research.

Definition 9 (Shortest Vector Problem (SVP))**.** Given a lattice $\mathcal{L}$, a basis $\mathbf{B}$, and a norm L, the Shortest Vector Problem is to find the shortest vector $\mathbf{v} \neq \mathbf{0}$ given $\mathbf{B}$ measured by L.

In general, finding *a* short vector is relatively easy. However, finding *the shortest vector* is very hard. How can one know that the short vector one has found is the shortest possible? There are algorithms that find short vectors, for example the LLL-algorithm by Lenstra, Lenstra, and Lovász [7].

The LLL-algorithm takes any basis for a lattice, and create an LLL-reduced basis. An LLL-reduced basis consists of short, nearly orthogonal vectors. The first vector in the LLL-reduced basis is not much larger than the shortest vector, and thus can the LLL-algorithm be used to find a candidate for the shortest vector[7].

2.4 Learing With Errors

Learning with errors is a computational problem, to distinguish between a random vector, and a inner product. It is considered hard to solve, and is used in many cryptographic protocols.

The LWE problem is defined as follows in Brakerski et. al [3]:

Definition 10 (Learning with errors problem (LWE))**.** Let n be an integer dimension, let $q \geq 2$ be an integer, and let χ be a distribution over $\mathbb{Z}$. The $\text{LWE}_{n,q,\chi}$ problem is to distinguish the following two distributions:

1. Sample $(\mathbf{a}_i, b_i)$ uniformly over $\mathbb{Z}_q^{n+1}$.
2. Draw $\mathbf{s} \leftarrow \mathbb{Z}_q^n$ uniformly, and sample $(\mathbf{a}_i, b_i) \in \mathbb{Z}_q^{n+1}$, by sampling $\mathbf{a}_i \leftarrow \mathbb{Z}_q^n$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = \langle \mathbf{a}, \mathbf{s} \rangle + e_i$,

The $\text{LWE}_{n,q,\chi}$ assumption is that this problem is infeasible.

It is important to note, that both the s and the e_i in LWE needs to be "short" to ensure that computation is feasible. However, the "shortness" of these two will impact how secure the system is, so an appropriate shortness needs to be chosen. If s and e_i are too short, it is possible to find them and distinguish these two distributions.

LWE is useful because the shortest vector problem over ideal lattices can be reduced to it [3]. To use LWE for a cryptosystem, one chooses the noise distribution from χ as a discrete Gaussian distribution.

Chapter 3

Machine Learning

In this chapter, we will briefly discuss types of machine learning, some of the problems that machine learning is used for, and some associated algorithms. This is because we later want to use fully homomorphic encryption to do machine learning. We will later choose one of these algorithms to take a closer look at how it can be used with homomorphic encryption, so in this chapter we will be on the lookout for machine learning algorithms with a simple response function, before making a choice on what algorithm to use in the remainder of this text. So we will briefly go trough a large amount of various algorithms, and judge who could be candidates for the algorithm we want to make in the end.

This chapter follows the texts by James et.al.[8], Goodfellow et.al. [9], Murphy [10], and Bishop [11], but any choices made and the selection of algorithms is done by the author.

3.1 Types of machine learning

3.1.1 What is machine learning?

The following definition of machine learning can be found in the Deep Learning book by Goodfellow. et. al: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [9].

The tasks T are the problems the machine learning algorithms are trying to solve. This is what we will focus on in the beginning of this text. The performance P is somewhat set to the side in this text, as the performance is measured in the number of errors the algorithm does. This is somewhat to the side of the purpose of

this text. The experience E is the type of experiences the algorithm is allowed. This is the input type and the amount of human interaction the algorithm is allowed.

Machine learning algorithms are in general divided into three main groups [10]. Supervised learning, unsupervised learning, and reinforcement learning. In this chapter we will take a closer look at supervised, and unsupervised learning.

Reinforcement learning is out of scope for this text, as it involves interaction with a dynamic learning environment. In this context, a learning algorithm acts in, and thus influences an environment, and at some point, after one or more actions, receives a reward signal from the environment to learn from, so the algorithm requires a reward signal to be sent from the environment when the algorithm is to make decisions, so continuous communication is required. [11]

3.1.2 Supervised learning

A machine learning task is considered supervised if it involves learning a mapping function based on examples consisting of input-output pairs of said function. It uses training data or examples to make a generalized model of the data set, and then it uses this generalized model to map input to output in the real world data.

Supervised learning could for example be classification, where the algorithm is trained using images of cats and dogs, with the associated correct label. When the algorithm has seen many pictures of cats and dogs with the label attached, it is able to distinguish cats from dogs in the input set, without human labels. Another example of supervised learning, is regression, where the input data points is fitted to a regression curve. This regression curve is then checked by a human in various ways to check if it actually represents the data. Checking the output data is not unique to regression, checking how the model fits the data is done both for regression and for classification. The reason this type of check is easy to do in a supervised setting, is that it is possible to set aside a part of the input data set, and use this as a so-called test set to check the fit of the data. There are many ways of dividing this test set from the training set, but this discussion is more a statistics problem, and a side-note in my text. More on classification and regression in the next Section.

It is important to think what kind of data sets we are going to look at, and how finely grained the data set should be. The training set should be adjusted so the algorithm sees data that it will encounter in the real world, as training a data set introduces bias.

3.1.3 Unsupervised learning

Unsupervised learning tries to find patterns in unlabelled data. It uses for example probability densities to find which data points that are "close", and assumes they are of the same type, or belong together somehow. Given real world data, the algorithm would look at what training data is around that point, and output the value that the surrounding points has.

As in the example with cats and dogs, the algorithm will just receive the images as input, and the algorithm has to determine the differences itself, with no human generated labels. There are many similarities between cats and dogs, for example they both have four legs and fur, but cats have clearly visible whiskers, whereas dogs usually do not. The algorithm could put all pictures with an animal with whiskers in one category and label it "cat", and then look for whiskers in any incoming data point pictures. The challenge with the unsupervised setting is that the dataset division a human might be interested in need not be the most salient division for a learning algorithm. Without labels, the algorithm might ignore the species divide and rather choose to separate cats and dogs with green eyes from those with brown. Or perhaps it divides the dataset into groups with an indoor or outdoor background. Both can be interesting separations, and provide insight into the dataset, but the possibility illustrates the inherent challenge associated with unsupervised learning.

3.1.4 Semi supervised learning

Semi supervised learning lies between supervised and unsupervised learning. Labeling an entire data set using humans could be very costly and/or time consuming, so semi supervised learning tries to improve unsupervised learning by labeling only part of the training data set, and using the partial labeling as a guide for the training.

Again in the example with cats and dogs, semi supervised learning will create clusters with "whiskers" and "not whiskers", and then look at the labels in the set and see how many cat-labels and dog-labels there is in each. The algorithm could then determine the "whiskers" set to be an animal type by majority vote of the labels in the cluster. The algorithm will assume that the unlabeled data is the same type as the labeled data, as they share similar traits.

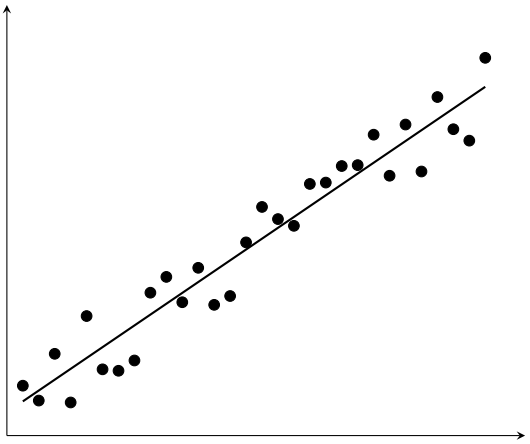


Figure 3.1: Standard linear regression

3.2 Machine learning algorithms

There are many different problems where machine learning can be used. In this Section, We will discuss two of them, with some associated algorithms.

3.2.1 Regression

Regression is a supervised learning method, where given training data, we want to fit a continuous function, to easily look up the input-output pairs of the sample data. Typically the algorithm is asked to use a function $f : \mathbb{R}^n \to \mathbb{R}$, and given an input vector $\mathbf{x}$, find $y = f(\mathbf{x})$. This function could take many forms, some of them which we will see in this section.

Linear regression analysis

Many statistical learning approaches can be seen as generalizations or extensions of linear regression, so linear regression is very important[8]. The input in linear regression problems is points in n-dimensional space, and the output is a function value number in 1 dimension. The problem is the find the function that fits a line to the points, so that the error is appropriately small.

$$Y \approx \beta_0 + \beta_1 x + \epsilon \tag{3.1}$$

Where Y is the output, β_0 is the intersection with the y-axis, β_1 is the estimated slope, and ϵ is the error term. β_0 and β_1 are known as the parameters of the equation. When we have fitted the model using the training data, we can use this

equation to do regression:

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 x \tag{3.2}$$

An illustration of linear regression can be found in Figure 3.1.

In more complex cases, the input $\mathbf{x}$ is a vector, and the slope coefficient $\boldsymbol{\beta}_1$ is also a vector, and these form an inner product. This is how we get the multi dimensional case $f : \mathbb{R}^n \to \mathbb{R}$, instead of a completely linear case where $f : \mathbb{R} \to \mathbb{R}$.

Least squares method - Linear regression

In practice, the parameters are unknown, so these needs to be fitted with training data. The most common way of doing this is by using least squares method.

We have the same $\hat{Y} = \hat{f}(X) + \epsilon$, but we measure take $\epsilon_i = y_i - \hat{y}_i$ for each point in the data set and the assumed f, and then we minimize this by finding a good S by this equation, where n is the number of observations in the training data.

$$S = \sum_{i=1}^{n} \epsilon_i^2 \tag{3.3}$$

Re-writing this S, gives the minimizers of $\hat{\beta}_0$ and $\hat{\beta}_1$ as follows:

$$\begin{aligned} \hat{\beta}_1 &= \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}, \\ \hat{\beta}_0 &= \bar{y} - \hat{\beta}_1 \bar{x} \end{aligned} \tag{3.4}$$

Where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. Using these minimizers, we are able to find $\hat{f}$.

After finding $\hat{f}$, it is used in exactly the same way as in regular linear regression. An illustration of this method can be seen in Figure 3.2.

Logistic regression - non linear regression

The main goal of logistic regression, is to model the probability of a discrete outcome, given an input vector. This is based on Bernoulli statistics, whereas linear regression is usually based on Gauss distribution. In binomial (also known as binary) logistic regression, the outcome is binary, for example true/false, or yes/no. However, the regression line is smooth. An illustration of this method can be seen in Figure 3.3. The regression line is based on the sigmoid function, and the re-

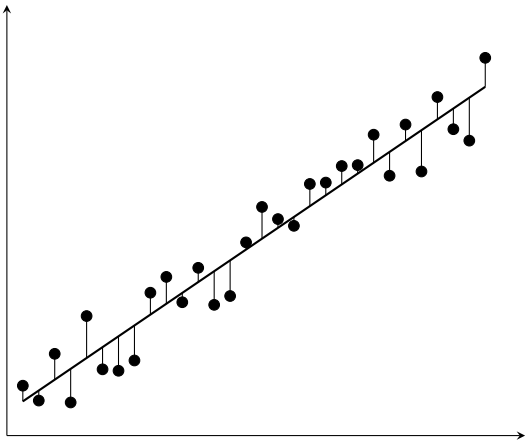


Figure 3.2: Least squares regression

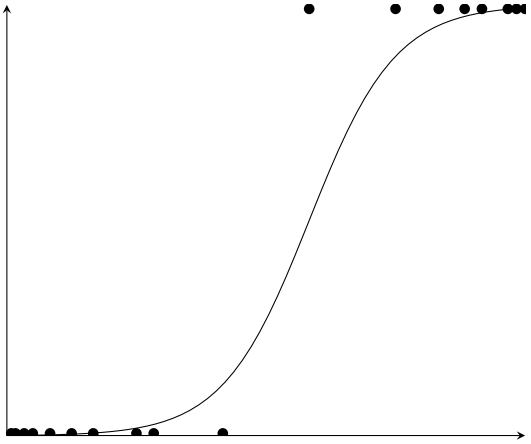


Figure 3.3: Logistic regression

sponse function for this regression is as follows:

$$\hat{Y} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}} \tag{3.5}$$

To find the β's, one could use for example the Maximum Likelihood method. Explaining maximum likelihood is out of scope for this text, but the method is well known, and could be found in any statistics textbook.

Support vector regression

Support vector machines are often seen in relation to classification, but is also used to do regression. An explanation on support vector machines can be found under the section on support vector classification.

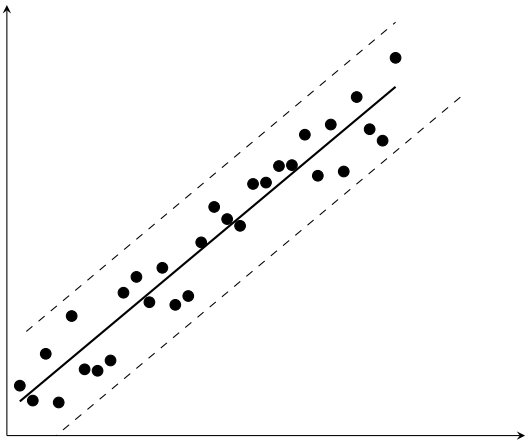


Figure 3.4: Support vector regression

In support vector classification, the goal is to maximise the distance between points, but support vector regression flips that around, and aims to gather as many points as possible inside the margins of the hyperplane. An illustration of this can be seen in Figure 3.4.

The response function ends up with the following form if the output is one dimensional, and it is clear that this is the same as for any linear regression.

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 x \tag{3.6}$$

K nearest neighbours - Regression

The k-nearest neighbours method for regression tires to learn from the values near the data point. The f is trained by looking for the k nearest neighbours, by making an average of the values of the k nearest neighbours. The size of k is usually chosen by cross validation, as it needs to be tweaked to avoid over fitting of data. Often this is used in combination with a penalty for the distance between the point and the neighbours, to minimize the impact of far away neighbours in a sparse neighbourhood. The process is repeated until a continuous line f has been made. Then the points of the training data is discarded, and f is used as a regression line. In the case of KNN, f is not necessarily describable with a smooth function, so making a generalized equation for the whole of f is not possible, but f could be described with a piecewise defined function. A sketch of KNN can be seen in the illustration in Figure 3.5.

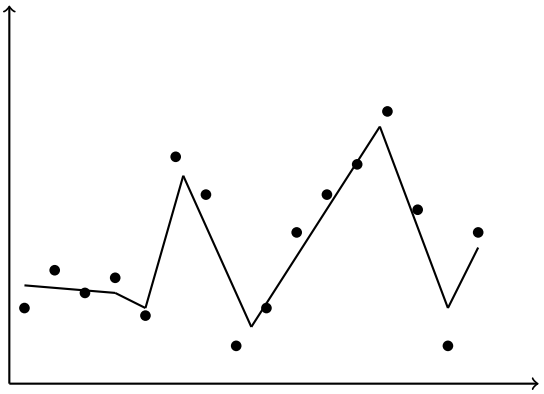


Figure 3.5: k nearest neighbours regression

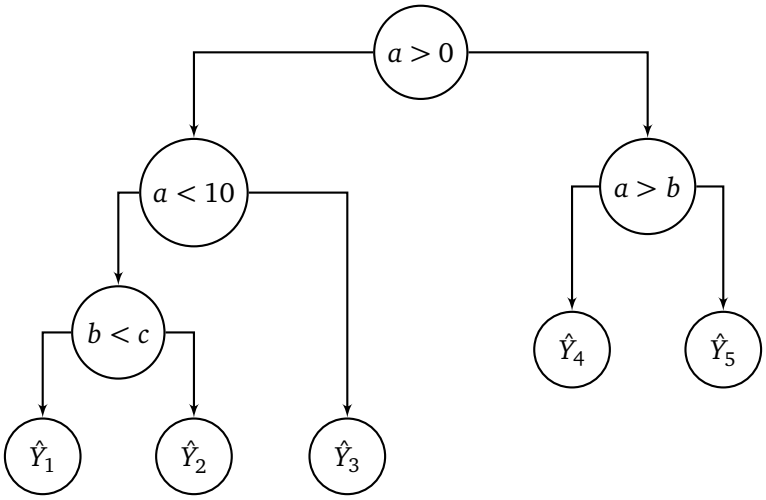


Figure 3.6: A decision tree

Decision trees - regression

Decision trees function like a flow chart, where each node has a true/false condition, and the two outputs leads either to a new true/false condition, or to a leaf node with the response value.

Usually, a tree is grown greedily top-down, so we start by making one split into two boxes. We just make the best possible split into two, given the training data. We then split each of these two boxes into two boxes, etc. Knowing when to stop growing the tree is important to avoid overfitting. For example a possible stop point could be when there are ten or fewer data points in each node.

In more detail, we aim to create boxes $R_1, R_2, \ldots, R_J$ in the data, where each

box minimize the residual sum of squares, via the equation for RSS:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2 \tag{3.7}$$

where $\hat{y}_{R_j}$ is the mean response for the training observations in the j'th box. Each box is a terminal node, or a leaf node, of the tree. An example of a tree can be found in Figure 3.6.

After the tree is grown, the response values of the leaf nodes is calculated into a mean, and this mean is the response value for the given leaf node. The data points themselves are discarded.

Regression is done by following the tree structure trough each true/false condition, and when arriving in a leaf node, outputting the calculated mean. This means that there is no clear function f in this case, rather a binary tree with true/false conditions in each internal node.

3.2.2 Classification

Classification is a supervised learning task. Here, the computer is asked to define which of n output categories the input data is. There is a function

$$f : \mathbb{R}^m \longrightarrow \{1, \ldots, n\} \tag{3.8}$$

that classifies data into n categories. Given an input vector $\mathbf{x}$, the algorithm tries to find $y = f(\mathbf{x})$, where y is a category of data. [9]

Support vector machines - classification

Support vector machines uses the training data to construct a n-dimensional hyperplane that separates the data points into categories, trying to maximize the distance from all points to the hyperplane. The vectors that define the hyperplane are called the support vectors. Misclassifications are penalized, so the algorithm will attempt to put as many points on the correct side of the hyperplane as possible. As can be seen in Figure 3.7, the goal is to get the data points outside of the support vector margins, as opposed to in the regression case, where the goal is to get the points inside the margins.

In the most basic example, where the data is two-dimensional, the hyperplane is a line. In the multi-dimensional case, the hyperplane is a true multi dimensional

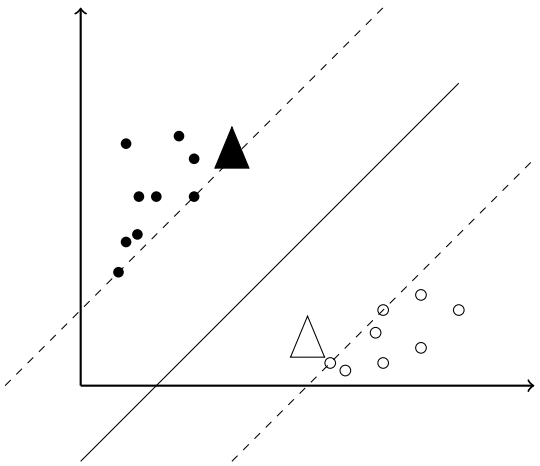


Figure 3.7: Support Vector Classification

hyperplane. One ends up with a function that is an inner product of two vectors plus a constant. The $\boldsymbol{\beta}_1$ and β_0 are the parameters.

$$f(\mathbf{x}) = \boldsymbol{\beta}_1 \cdot \mathbf{x} + \beta_0 \tag{3.9}$$

One then calculates on which side of this hyperplane the data point is. To classify the data into the correct category, one inserts the data $\mathbf{x}$ into f and look at the sign of the response.

Support vector machines makes the assumption that it is possible to find a linear separation between the points in some (possibly high) dimension n, but that is not necessarily the case. If the data can not be linearly separated, SVMs can not be used.

K Nearest Neighbours - classification

As in the regression case, the k nearest neighbour algorithm takes a point, and checks the type of the k nearest neighbours. It does not make a mean of the values, rather looks at the type of points. The majority vote of the k nearest neighbours determines the type of the point being examined. This is done for every point, until regions are mapped out, where each region is a class membership. The training points are then discarded, and the regions are left. There is no clear function f in this case, as there can be many borders. While they are continuous, they can be very complicated and not smooth at all. An example of a 4 nearest neighbour sample can be seen in Figure 3.8, where the triangles are the input data point,

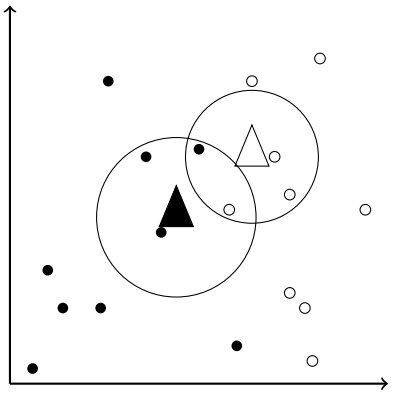


Figure 3.8: 4 nearest neighbour classification example

and the circles around the triangles circle the 4 nearest neighbours.

Decision trees - classification

Classification decision trees are trained in the same way as regression decision trees. However, instead of averaging out a numeric value in the leaf nodes, there is a majority vote in the leaf node members to decide a class membership. The response is a class membership.

Again, the tree is built using Residual Sum of Squares, and again it is important to choose a good number of data points in each leaf. Too small number, and the tree is overfitted, and too large number, there is a growing chance of misclassification. As with regression decision trees, there is no clear f, but rather a set of true/false flowchart instructions, to decide the class membership.

3.2.3 Other machine learning topics

There are many machine learning topics, and I will not closely discuss them all. They will be mentioned, as they were studied while working with this text, but decided to not write up extensively about them for various reasons, as will briefly be discussed in this section. In general, the f of these methods can be very complex, so they are not suitable for the needs of this paper.

Clustering

Clustering is typically an unsupervised learning method. Clustering is very similar to classification, but the aim is to find patterns in the data without any human input. Just like in classification, we have a function $f : \mathbb{R}^m \longrightarrow \{1, \ldots, n\}$. Given an

input vector $\mathbf{x}$, the algorithm tries to find $y = f(\mathbf{x})$, where y is a cluster type [9]. There are many clustering algorithms, and some of them are similar to classification algorithms, however they are more complex because they aim to have zero human input.

In many cases, clustering also is used to find the appropriate number of clusters. So again with the previous example with cats and dogs, a task is to not only find which is which, but also to find that there are two categories. If there is also pictures of dinosaurs in the data set, the algorithm is supposed to find that the appropriate number of clusters is 3 instead of 2, and then sort the pictures into the correct cluster.

Clustering often has complex response functions depending on the algorithm used, so it is not suitable for this text.

Model selection

When presented with training data, the data scientists needs to decide which algorithm to use. There are many options, and it can be difficult to know which model to use. Similar looking data sets can give different results, so it is important to not just choose "the standard" algorithm, but to look for the one that fits this particular data set that the scientists are working with.

In model selection, one tries to compare various models and see which one fits the data set, and validating the results. This is done before training the data with a specific model. Considering how this text is on how to secure the data after the model is chosen and trained, this does not fall in the scope of this text.

Dimensionality reduction and pre-processing

Dimensionality reduction is about transforming the data from high-dimensional to as low-dimensional as possible, while retaining the relevant information. Often having high dimensional data is a problem, as it could cause noise in the model, so it is desired to have low-dimensional data if possible.

Pre-processing is an umbrella term talking about many steps to prevent noise and errors in the data. It is important to prepossess the data, but not remove relevant data points. For example it could be about handling null data and standardise the data input, or transforming text input so it is possible to use it for calculations. For example changing "Male/female" variable to "1/0".

Both dimensionality reduction and pre-processing is done before the data is sent to the model, so it is not in scope of this text.

Anomaly detection

In anomaly detection, the algorithm looks for anomalies in the data. Anomalies could be a cluster from a different probability distribution, or it could be a single data point with very high distance from the rest. This is for example used in finance, to spot fraudulent transactions. Depending on what type of anomaly is to be detected, a different f is used.

Anomaly detection works on a large data set to spot the anomaly, whereas in this text, we want to focus on algorithms where the input is a single data point.

Neural networks

Neural networks is a large category of algorithms in machine learning, where there is no clear function, rather a complex set of functions. There is no simple function from input to output, but a relation of a more complex, non-linear kind. There are several layers of "nodes", and weighted paths between them. These paths simulate neurons in a brain. The nodes are typically not human readable, and the weights of the paths are dependent on the training data set.

The neural net could have hundreds or thousands of nodes in each layer, and tens of layers, and thus thousands upon thousands of paths trough the neural net. In Figure 3.9 there is an attempt to draw a simplified version of a neural net, to illustrate the layers and the nodes and the paths.

As neural nets has no clear response function, it is out of scope for my text.

3.3 What has been done before

There has been many advances in privacy preserving machine learning, but most methods found in the work with this thesis, has been statistics related. These are not quite what we have been looking for, so we will not write about them, although many of them were very interesting, they were out of scope for this text.

Some authors have written about cryptographically private machine learning. In this section we will discuss two papers about Support Vector Classification, as that is a very strong algorithm that has many use cases.

3.3.1 Support vector machines classification

Laur, Lipmaa, and Mielikäinen has written about cryptographically private support vector machines [12]. In their text, they create an algorithm that is cryptograph-

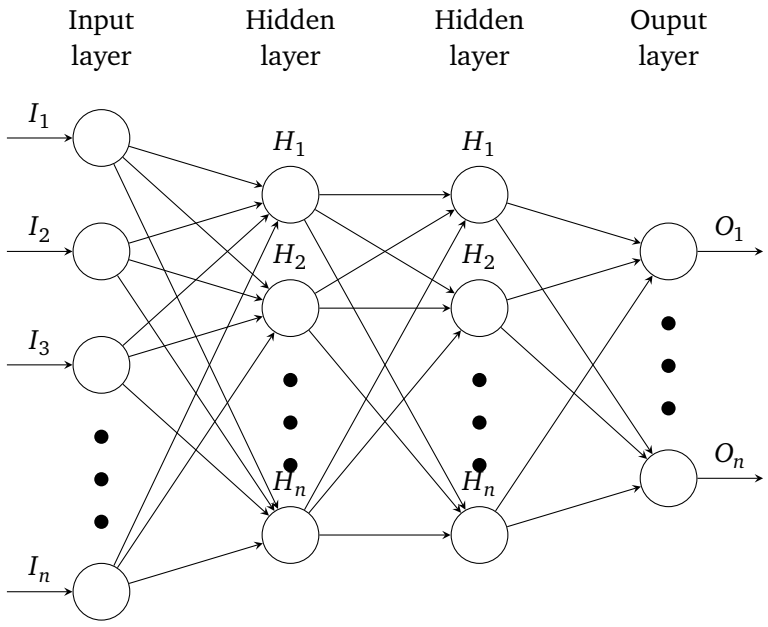


Figure 3.9: A simplified neural net

ically private for support vector machine classification.

In their text, they assume there are two parties that have each their own data set, and want to do analysis on the shared data set, without actually sharing data. They use oblivious transfer to do this. They claim that oblivious transfer is a cumbersome algorithm, and goes into detail with ways of speeding it up without compromising security, mainly by reducing the bit size of the input to the algorithm.

Later in the text, they go into a different scenario; private prediction. Private prediction assumes the model is already trained, and the client sends a vector with the pre trained weights to the server, and the server has the feature vectors. An alternate algorithm is that the client has some part of the data and the server has some other part of the data. Who has which part of the data does not really matter, as the algorithm is generalised in the text. This algorithm uses circuit evaluation to ensure privacy, but the paper does not go into detail on how the circuit evaluation is done, it just states in Theorem 3; "Assume that [the steps] are computed correctly and privately"[12].

The last part of the paper briefly discusses private training algorithms. This algorithm leaks the number of computed rounds, which says something about how many iterations of the algorithm has been done.

3.3.2 Multi class support vector machine classifications

Rahulamathavan, Phan, Veluru, Cumanan, and Rajarajan has written about cryptographically secure multi class support vector machines. [4] Support vector machines has traditionally been used to classify data into two groups, but there exists methods to divide the data into multiple classes. Generally, one decouples the multi class problem into many two class problems, and then combines these.

In their text, they assume that the client has some data they want to do computations on, and the server has the model. They base the security on the Paillier cryptosystem, but the authors claim that any homomorphic cryptosystem can be used.

The paper first goes trough the two-class problem, with a detailed break down of how the algorithm works, and a security proof. With this algorithm, it follows the definition of secure two-party computation:

Definition 11 (Secure two-party computation)**.** A secure two-party protocol should not reveal more than the information that can be induced by looking at that party's input and output. [13]

In section 4 of the paper, they go into the multi-class problem. They sketch two ways of doing this, one with a one-versus-all approach, where they for n subclasses train n subproblems with each their own support vector machine. They separate out one class, and group it against every other class together.

The other way of doing multi-class support vector machines, is a one-versus-one approach. For n classes, they train $\frac{n(n-1)}{2}$ support vector machines, each using data from only two classes. There are pros and cons to each of these methods, but this is beyond the scope of this paper.

Rahulamathavan et.al.'s paper first goes trough the 1VA problem, and then the 1V1 problem, and then compares them in a performance analysis. They also test their assumption with real-world data.

3.4 Choosing an algorithm

In this text we are assuming that the model has already been trained, and the algorithm only looks at the input feature vectors and returns a response. This is because training a model can be very complicated, and we want to find a model that works in a more general case. We find regression to be an interesting problem, and as will be explained later, many of the regression models have similarities

that can be generalised. In addition, classification problems has been done by others, as seen in Section 3.3. When it comes to unsupervised machine learning topics, when they are trained, they end up having a similar response function to the corresponding supervised algorithms, as can be seen in Goodfellow's Deep learning [9]. Because of this, we will focus on regression in this thesis.

As we can see from Section 3.2.1 on regression, most of the regression algorithms are variants of $\hat{Y} = f(\mathbf{x})$, where f is a smooth function $\mathbb{R}^n \to \mathbb{R}$. One important exception is decision trees. For this reason, we will not look at decision trees, as however interesting they are, we want something that can be generalised. We will also not consider support vector machines, as this has been done before by among others Laur et.al [12], as could be seen in Section 3.3.1.

Linear regression

The linear regression equation is very basic, and there are several methods how to find the two feature vector parameters $\hat{\beta}_i$, but since we in this text assume that the method is already fitted, we only need to look at this equation and how it is being used. I will assume we have a multi dimensional equation $f : \mathbb{R}^n \to \mathbb{R}$,

$$\hat{Y} = \hat{\beta}_0 + \hat{\boldsymbol{\beta}}_1 \mathbf{x} \tag{3.10}$$

This equation is described in detail in Section 3.2.1, but we will repeat it briefly here. There are two parameters, $\hat{\beta}_0$ and $\hat{\boldsymbol{\beta}}_1$, where $\hat{\beta}_0 \in \mathbb{R}$ is a number and $\hat{\boldsymbol{\beta}}_1 \in \mathbb{R}^n$ is a vector of numbers, and $\mathbf{x}$ is a vector of numbers. $\mathbf{x}$ comes from the data owner, and the $\hat{\beta}_i$'s belong to the algorithm owner.

While it is clear that this is a very basic equation, but there are still some challenges. We would have to write an algorithm for the inner product, and calculate the depth of the algebraic circuit.

Support Vector Regression

In Support Vector regression, f ends up having the same form as ordinary linear regression. The only difference between these two is the way the $\hat{\beta}_i$'s are found, but as previously mentioned, we assume that the model is already fitted. So in the end, this method has the same f as ordinary linear regression.

Logistic regression

As in the case of linear regression, we are assuming the equation is already trained, so the question on how the $\hat{\beta}_i$'s are found is not in the scope of this text. When the model is fitted, we end up with the following f in logistic regression:

$$\hat{Y} = \frac{1}{1 + e^{-(\hat{\beta}_0 + \hat{\boldsymbol{\beta}}_1 \mathbf{x})}} \tag{3.11}$$

This f is often called the sigmoid function, but what is interesting is the exponent in the denominator. That is on the form

$$-(\hat{\beta}_0 + \hat{\boldsymbol{\beta}}_1 \mathbf{x}) \tag{3.12}$$

This is very similar to the linear regression case, so looking at the linear regression case will also make progress on the logistic regression case. Looking at logistic regression has some additional challenges, namely that there is an exponent. As we will explain in Section 4.9.3, there is no method for exponentiation in the cryptosystem we are looking at, so some trick needs to be done to make it work. There is also the problem of dividing, and as we will explain in more detail in Section 4.9.3, division is a hard problem.

K nearest neighbours regression

In the K nearest neighbours regression algorithm, f is piece-wise defined. Since the input data is encrypted, there is no way to look up the correct part of f. When $\mathbf{x}$ is encrypted, it is not possible to know which are the correct values that corresponds to the appropriate part of f. If it was a way to know, the ciphertext would leak information about the plaintext. Since the algorithm owner does not tell the data owner what the divisions of f is, as this would leak information about the algorithm, there is no way for the data owner to encrypt the correct parts of $\mathbf{x}$ in an appropriate manner. Thus K nearest neighbours is unsuitable for this project.

3.4.1 Making a choice

Later in this text, we will see what capabilities the cryptographic protocols will have. Without spoiling too much, the protocol only supports addition and multiplication. This limits what we are able to do, and rules out most of the algorithms. For example there is no way to compare the size of two numbers, so making a decision tree where we have to compare which number is greatest, is not possible.

There are still some options left, and it is possible to find an algorithm to look at.

As we can see, linear regression, support vector regression, and logistic regression has some variation over $\hat{\beta}_0 + \hat{\boldsymbol{\beta}}_i \mathbf{x}$. This is interesting, as it shows that there are several algorithms that use a similar equation. We want to look at something that can be generalised, so this seems to be a good candidate. For linear regression and support vector regression, it is the exact same equation that is used, but for logistic regression there needs to be some adjustments. We will discuss these adjustments later, in Section 6.1.3.

In the remainder of the text, we will study this equation, and especially the linear combination $\hat{\boldsymbol{\beta}}_1 \mathbf{x}$.

Chapter 4

Homomorphic encryption

In this chapter we will give a brief summary of a cryptosystem, and some security terms, before defining homomorphic encryption. We will then talk about the BGV cryptosystem in more detail, look at an implementation of the system and what that means in practice.

4.1 General cryptography

4.1.1 Definition of cryptosystem

A public key cryptosystem is a triple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, where the key generating algorithm $\mathcal{K}$ returns the secret key sk, and the public key pk, $\mathcal{E}$ is the encryption algorithm, and $\mathcal{D}$ is the decryption algorithm. Let $\mathcal{P}$ be the plaintext space, and $\mathcal{C}$ be the ciphertext space. The encryption with pk takes $\mathcal{P} \longrightarrow \mathcal{C}$. We have $\forall m \in \mathcal{P}$, $\mathcal{E}_{\mathsf{pk}}(m \in \mathcal{P}) = c \in \mathcal{C}$ such that $\mathcal{D}_{\mathsf{sk}}(\mathcal{E}_{\mathsf{pk}}(m)) = m$.

4.1.2 Compact cipher text

We say that a homomorphic encryption scheme $\mathcal{E}$ is compact if there is a polynomial f such that for every value of the security parameter λ, $\mathcal{E}$'s decryption algorithm can be expressed as a circuit of size at most $f(\lambda)$. In other words, the length of the ciphertext is independent of the length of the plaintext.

4.1.3 Circuit correct

A circuit $\mathfrak{C}$ is a set of instructions in an algorithm. A logical circuit is a combination of logic gates such as AND, OR, XOR, etc, that form a set of logical instructions.

An algebraic circuit is a combination of $+$ and $\cdot$ to form an algebraic expression. $\mathcal{K}$ is the set of circuits, i.e. the set of all possible circuits in the algorithm.

- Π is correct if for any encrypted plaintext, we have the probability

$$P(\mathcal{D}(\mathsf{sk}, \mathcal{E}(\mathsf{pk}, m)) = m) = 1 \tag{4.1}$$

- Π is $\mathcal{K}$-correct if $\forall \mathfrak{C} \in \mathcal{K}$ the probability

$$P(\mathcal{D}(k, \mathfrak{C}(\mathcal{E}(k, m_i))) = \mathfrak{C}(m_i)) = 1 - \epsilon \tag{4.2}$$

- For all circuits $\mathfrak{C} \in \mathcal{K}$ and all ciphertexts $c \in \mathcal{C}$, we have

$$\mathcal{D}(k, \mathfrak{C}(k, c_i)) = \mathfrak{C}(\mathcal{D}(k, c)_1, \ldots, \mathcal{D}(k, c)_i) \tag{4.3}$$

This means that the order of the circuit $\mathfrak{C}$ and the decryption $\mathcal{D}$ does not matter. However, the ciphertexts in $\mathcal{C}$ are usually *fresh*. This means that they are straight from $\mathcal{E}$, or they have very little noise. What constitutes "very little" noise, depends on the security parameters and the depth of the circuits to be evaluated, so it varies on a case by case basis.

4.1.4 Circuit private

Circuit private means that it should be hard to see which circuit that has been used, also when the decryption is known. This means that the evaluation of the circuit can not be deterministic, as then the adversary could find the circuit by trial and error. All ciphertexts should have the same distribution, but none should be equal.

4.1.5 Circular secure

Encrypting the private key should not leak information about the key, or the data. Because of how bootstrapping works, if we do not assume circular secure, the best we can get is leveled FHE. Bootstrapping requires keys to partially decrypt, and to have full FHE, we need one single key to decrypt everything. If we can not have one single key, and this can not be decrypted without leaks, the best we have is a partial key to partially decrypt.

4.2 Types of security

There are many types of security. In this section, we will briefly look at a few of them.

4.2.1 Indistinguishability under chosen-plaintext attack (IND-CPA)

Under IND-CPA, an adversary is unable to distinguish two ciphertext from each other. The adversary has a sample of two plaintexts, and is presented with one of them, then it is impossible for the adversary to know which one they are presented with.

1. The challenger chal generates a key pair pk, sk, based on some security parameter, and publishes pk to the adversary adv. chal retains sk.
2. adv may perform a polynomially bounded number of encryptions.
3. adv submits two distinct chosen plaintexts m_0, m_1 to chal.
4. the challenger selects a bit $b \in \{0, 1\}$ uniformly at random, and sends the corresponding ciphertext $\mathcal{E}(\mathsf{pk}, c_b)$ back to the adversary.
5. adv is free to do as many computations or encryptions as they wish, before outputting a guess of the value of b.

A cryptosystem is IND-CPA if a probabilistic, polynomial time, attacker only has negligible advantage over a random guess. An attacker "wins" if they have more than $\frac{1}{2} + \epsilon(k)$ probability to guess correctly. Where $\epsilon(k)$ is a negligible function with security parameter k. That means that there is for any polynomial $p(x)$, we have

$$|\epsilon(k)| \leq \left| \frac{1}{p(x)} \right| \tag{4.4}$$

This means we can squeeze the $\epsilon(k)$ as small as we wish. [14]

4.2.2 Real-or-Random security

Real-or-Random security measures the indistinguishability between a real encrypted ciphertext, and a ciphertext made from a random input.

In Real-or-Random security, the chal computes the ciphertext of either a random string, or the ciphertext of a chosen plaintext, and the adv needs to distinguish between these two, and decide if it is a random string or a real plaintext that has been encrypted. adv is allowed to ask as many test queries as they want.

More about Real-or-Random security can be found for example in the text

by Abdalla et.al Password-Based Authenticated Key Exchange in the Three-Party Setting [15].

4.2.3 Semantically secure

Semantically secure is when only a negligible amount of information can be extracted about the plaintext. That means a probabilistic, polynomial time, algorithm given the ciphertext c of a message m and the length of the message $|m|$, can not obtain any information on the plaintext, than a probabilistic polynomial time algorithm that only has the access to the length of the plaintext, and no access to the ciphertext.

4.2.4 Types of adversaries

Honest adversary An honest attacker will follow the protocol as listed.

Semi-honest adversary A semi-honest attacker will follow the protocol, but will try to get more information out of the protocol than what is given freely.

Curious adversary A curious adversary will attempt to learn as much as possible from every bit of information given. A honest-but-curious adversary will follow the protocol, but learn as much as possible from the legitimate outputs they are given. A honest-but-curious adversary is often called a semi-honest adversary.

Malicious adversary A malicious adversary can deviate from the protocol, abort it, restart it, or in other ways attempt to break the protocol. A malicious adversary model is the most difficult model to protect oneself from.

4.3 Homomorphic encryption

4.3.1 Homomorphic cryptosystem

A cryptosystem is additive homomorphic if for each key pair $(\mathsf{sk}, \mathsf{pk})$, messages $m_1, m_2 \in \mathcal{P}$, we have

$$\mathcal{E}(m_1) + \mathcal{E}(m_2) = \mathcal{E}(m_1 + m_2) \tag{4.5}$$

where $+$ is the group operation in $\mathcal{P}$.

A cryptosystem is multiplicative homomorphic if for each key pair $(\mathsf{sk}, \mathsf{pk})$, messages $m_1, m_2 \in \mathcal{P}$, we have

$$\mathcal{E}(m_1) \cdot \mathcal{E}(m_2) = \mathcal{E}(m_1 \cdot m_2) \tag{4.6}$$

where $\cdot$ is the group operation in $\mathcal{P}$.

In these systems, we can compute on ciphertexts, using these identities:

$$\begin{aligned} \mathcal{E}(m_1) + \mathcal{E}(m_2) &= \mathcal{E}(m_1 + m_2 \mod N) \quad \forall m_1, m_2 \in \mathcal{P} \\ \mathcal{E}(m_1) \cdot \mathcal{E}(m_2) &= \mathcal{E}(m_1 \cdot m_2 \mod N) \quad m_1 \in \mathcal{P} \\ \mathcal{E}(y) \cdot \mathcal{E}(m) &= \mathcal{E}(y \cdot m \mod N) \quad m \in \mathcal{P}, y \in \mathbb{Z} \\ \mathcal{E}(m) \cdot \mathcal{E}(1) &= \mathcal{E}(m) \quad m \in \mathcal{P} \end{aligned} \tag{4.7}$$

4.3.2 History

Gentry [2] made the first Fully Homomorphic scheme in 2009. He made a construction of a somewhat homomorphic encryption scheme, based on lattices. Then he used something he called bootstrapping, to transform the scheme into a leveled homomorphic scheme. He then showed that a bootstrappable SHE scheme can be converted into a FHE scheme using recursive self-embedding. To combat the noise that appears when doing circuit calculations, bootstrapping refreshes the ciphertext, obtaining a new ciphertext that encrypts the same value as before, but with lower noise levels. Gentry refreshes the ciphertext periodically, or whenever the noise levels gets too high, and in this way it is possible to compute an arbitrary deep circuit without making the noise levels too high.

It was not until Brakerski, Gentry, and Vaikuntanathan in 2011 [3], we were able to do homomorphic encryption without bootstrapping. I will take a closer look at this cryptosystem in the following sections.

4.4 Definitions

4.4.1 Types of homomorphic encryption

A *fully homomorphic encryption scheme* can do as many circuits as we would like, and still be able to decrypt without errors.

A *leveled homomorphic encryption scheme*, also known as a *somewhat homomorphic encryption scheme*, can only compute a bounded amount of circuits without

encountering errors in the decryption process. In other words, the encryption (and thus, decryption) depends on an integer d, where d is the depth of the circuits.

4.4.2 Bootstrapping

When doing circuits on the ciphertext, FHE adds noise to the ciphertexts. If too much noise is added, it will no longer be possible to decrypt. To fix this, Gentry made somewhat homomorphic encryption, which redesigns the decryption algorithm into a circuit, and passes the ciphertext trough along with an encrypted key, to remove noise. The output of the partial decryption is a ciphertext. For each step in the larger circuit, do this partial decryption to remove noise. This partial decryption is called bootstrapping, and it adds significant computation time to the circuits.

4.5 A fully homomorphic system without bootstrapping - BGV

In this section we will discuss a fully homomorphic cryptosystem without bootstrapping, by Brakerski, Gentry, and Vaikuntanathan [3]. This chapter follows that text. Some of the proofs are very technical, and is therefore omitted. They can be found in the BGV text.

4.5.1 Summary of the system

BGV is introduced by first introducing a basic system with no homomorphic operations. Then two techniques to replace the bootstrapping in the original Gentry, namely key switching/dimension reduction, and modulus switching. These two techniques are used together to do homomorphic encryption.

We will only look at the case where learning with error is used, as ring learning with errors is more complex, and the implementation we looked at uses learning with errors. As in the chapter on basic encryption, we will not consider the impact of the security parameter, so we will disregard this.

4.5.2 Basic scheme

- `E.Setup`: The algorithm is initialised. The parameters of `params` is chosen.

1. We choose a μ-bit modulus q and choose the other parameters n, N, χ, appropriately so that the scheme is based on GLWE that achieves the appropriate security against known attacks.
2. Let $R = \mathbb{Z}$
3. Let $\mathsf{params} = (q, n, N, \chi)$.
   - q is the modulus of the group.
   - n is the dimension of the keyspace.
   - $N = \lceil (2n+1)\log q \rceil$.
   - χ is the probability density distribution to draw randomly from, aka the noise.

- $\mathsf{E.SecretKeyGen(params)}$: Takes the parameters as input, and outputs the secret key.
  1. Draw $k' \leftarrow \chi^n$.
  2. Set $\mathsf{sk} = k \leftarrow (1, s'[1], s'[2], \cdots, s'[n]) \in R_q^{n+1}$

- $\mathsf{E.PublicKeyGen(params, sk)}$: Generates the public key. It takes the parameters and the secret key as input, and outputs the public key
  1. Generate a matrix **A'**$\leftarrow R_q^{N+n}$ uniformly.
  2. Generate a vector $e \leftarrow \chi^N$.
  3. Set $\mathbf{b} \leftarrow \mathbf{A}\mathbf{s}' + 2\mathbf{e}$.
  4. Set $\mathbf{A}$ to be the $(n+1)$ column matrix consisting of b followed by the n columns of $-\mathbf{A}'$.
  ! (Observe: $\mathbf{A}\mathbf{s} = 2\mathbf{e}$).
  5. Set the public key to be $\mathsf{pk} = \mathbf{A}$.

- $\mathsf{E.Enc(params, pk, }m\mathsf{)}$: Takes the parameters, the public key, and a plaintext message $m \in R_2$ as input, and outputs a ciphertext $c \in \mathcal{C}$
  1. Set $m \leftarrow (m, 0, 0, \cdots, 0) \in R_q^{n+1}$.
  2. Sample $\mathbf{r} \leftarrow R_2^N$.
  3. Output the ciphertext $c \leftarrow m + \mathbf{A}^T\mathbf{r} \in R_q^{n+1}$

- $\mathsf{E.Dec(params, sk, }c\mathsf{)}$: Takes the parameters, the secret key, and a ciphertext $c \in \mathcal{C}$ as input, and outputs a plaintext $m \in R_2$
  1. Output $m \leftarrow [[\langle c, k\rangle]_q]_2$.
  2. This can be written as $m = [[L_c(k)]_q]_2$, where $L_c(\mathbf{x})$ is a ciphertext dependent linear equation over the coefficients of x such that $L_c(\mathbf{x}) = \langle c, \mathbf{x}\rangle$.

4.5.3 Key Switching

Suppose we have two ciphertexts, c_1 and c_2, encrypting m_1 and m_2 under the same key k. Assuming the noise is small enough when we do multiplication, we get $m_1 \cdot m_2 = [[L_{c_1}(\mathbf{x}) \cdot L_{c(\mathbf{x})}]_q]_2$. This can be viewed as a linear equation $L^{long}_{c_1,c}(x \otimes x)$, decryptable by the secret key $k \otimes k$. This key is long. Increasing dimension like this will be a problem, as both the ciphertext and the key will get very long, so efficiency will be a problem,. To solve this, we encrypt this with a new key k_2 to make a new ciphertext c_3.

The BGV key switching is even more general than this. It is used for dimensionality reduction after a multiplication. It is used to transform a ciphertext c_1 encrypted under the key k_1 to another ciphertext c_2 encrypted under a different key k_2. The dimension of c_2, k_2 could, but it is not necessarily of lower dimension than c_1, k_1.

Below is the details of the BGV key switching scheme.

They first have two subroutines to expand the vectors c, k into the higher dimensional vectors c', k', such that $\langle c', k' \rangle = \langle c, k \rangle \mod q$.

- BitDecomposition($\mathbf{x} \in R_q^n, q$) decomposes $\mathbf{x}$ into its bit representation.
  1. input $\mathbf{x}$
  2. calculate $\mathbf{x} = \sum_{j=0}^{\lfloor logq \rfloor} 2^j \cdot \mathbf{u}_j$ (The vectors $\mathbf{u}_j$ are in $\mathbb{R}_2^n$)
  3. output $(u_0, u_1, \ldots, u_{\lfloor logq \rfloor})$ the bit decomposition of x
- PowersOf2($\mathbf{x} \in R_q^n, q$) outputs a vector
  1. input $\mathbf{x}$
  2. calculate

```
for i in (0, floor(log q))
    x*2^i
```

  3. output the vector $(x, 2 \cdot x, 2^2 \cdot x, \ldots, 2^{\lfloor logq \rfloor} \cdot x)$

It turns out that we get an optimisation here

Theorem 1. *For vectors c, k of equal length, we have*

$$\langle \mathit{BitDecomposition}(c, q), \mathit{PowersOf2}(k, q) \rangle = \langle c, k \rangle \mod q$$

The proof of this is quite technical, and can be found in the BGV text [3].

We now look at the key switching itself. It has two algorithms that combine together to key switching SwitchKeyGen and SwitchKey.

- $\mathsf{SwitchKeyGen}(k_1, k_2, n_1, n_2, q)$. This takes as input the two keys k_1, k_2, the dimensions of these keys, n_1, n_2 and the modulus q. It outputs a vector composed by the new key plus the old key k_1 stretched to powers of 2
  1. input (k_1, k_2, n_1, n_2, q)
  2. calculate $N = n_1 \cdot \lceil log q \rceil$
  3. run $A \leftarrow \mathsf{E.PublicKeyGen}(k_2, N)$ to find a new key
  4. calculate $B \leftarrow A + \mathsf{PowersOf2}(k_1, q)$
  5. set $\tau_{k_1 \to k_2} = B$
  6. output $\tau_{k_1 \to k_2}$
- $\mathsf{SwitchKey}(\tau_{k_1 \to k_2}, c_1)$ takes as input the τ vector and the ciphertext c_1, and outputs a new ciphtertext c_2.
  1. input $(\tau_{k_1 \to k_2}, C_1)$
  2. calculate $c_2 = \mathsf{BitDecomposition}(c_1)^T \cdot \tau_{k_1 \to k_2}$
  3. output c_2

In SwitchKeyGen, the matrix A is encryptions of 0 under the key k_2. Then we add pieces of the key k_1. So in a way, the matrix B is encryptions of pieces of the key k_1 under the key k_2.

Theorem 2 (correctness)**.** *Let $k_1, k_2, q, n_1, n_2, A, B = \tau_{k_1 \to k_2}$ be as in SwitchKeyGen(k_1, k_2), and let $A \cdot k_2 = 2e_2 \in R_q^N$. Let $c_1 \in R_q^{n_1}$ and $c_2 \to$ SwitchKey$(\tau_{k_1 \to k_2}, c_1)$. Then*

$$\langle c_2, k_2 \rangle = 2\langle \mathit{BitDecompostion}(c_1), e_2 \rangle + \langle c_1, k_1 \rangle \mod q$$

Proof.

$$\begin{aligned}
\langle C_2, k_2 \rangle &= \mathsf{BitDecomposition}(c_1)^T \cdot B \cdot k_2 \\
&= \mathsf{BitDecomposition}(c_1)^T \cdot (2e_2 + \mathsf{PowersOf2}(k_1) \\
&= 2\langle \mathsf{BitDecomposition}(c_1), e_2 \rangle + \langle \mathsf{BitDecomposition}(c_1), \mathsf{PowersOf2}(k_1) \\
&= 2\langle \mathsf{BitDecomposition}(c_1), e_2 \rangle + \langle c_1, k_1 \rangle
\end{aligned}$$

■

We see that the dot product of $\langle \mathsf{BitDecomposition}(c_1), e_2 \rangle$ is small because $\mathsf{BitDecomposition}(c_1) \in R_2^N$. In general we have that c_2 is a valid encryption of m under the key k_2, with noise a little larger than under k_1 by an additive factor.

4.5.4 Modulus Switching

c is an encryption of m under modulo q, $m = [[\langle c,k\rangle]_q]_2$ and the key k is a short vector. Suppose that c' is a scaling of c, so c' is the vector closest to $\frac{p}{q}c$ such that $c' = c \mod 2$. Then c' is a valid encryption of m under k modulo p. So $m = [[\langle c,k\rangle]_p]_2$. This means that we can change the modulus to something smaller, and still achieve a valid encryption under the same secret key.

Definition 12 (Scaling)**.** For integer vector $\mathbf{x}$ and integers $q > p > m$, we define $\mathbf{x}' \leftarrow \mathsf{Scale}(\mathbf{x},q,p,r)$ to be the R-vector closest to $(p/q)\cdot\mathbf{x}$ that satisfies $\mathbf{x}' = \mathbf{x} \mod r$.

Definition 13 (ℓ_1 Norm for R)**.** The usual $\ell_1(\mathbf{x})$ norm for the reals is $\sum_i ||x[i]||$. We extend this to R. $\ell_1^{(R)}(\mathbf{x})$ for $\mathbf{x}\in R^n$ is $\sum_i ||x[i]||$

Theorem 3. *Let d be the degree of the ring. Let $q > p > r$ be positive integers satisfying $q = p = 1 \mod r$. Let $c \in R^n$ and $c' \leftarrow \mathsf{Scale}(c,q,p,r)$. Then for any $k \in R^n$ with $||[\langle c,k\rangle]_q|| < q/2 - (q/p)\cdot(r/2)\cdot\sqrt{d}\cdot\gamma(R)\cdot\ell_1^{(R)}(k)$, we have*

$$[\langle c',k\rangle]_p = [\langle c,k\rangle]_q \mod r$$

and

$$||[\langle c',k\rangle]_p|| < (p/q)\cdot||[\langle c,k\rangle]_q|| + (r/2)\cdot\sqrt{d}\cdot\gamma(R)\cdot\ell_1^{(R)}(k)$$

Proof. We have

$$[\langle c,k\rangle]_q = \langle c,k\rangle - jq$$

for some $j \in R$. For the same j, let

$$e_p = \langle c',k\rangle - jp \in R$$

Note that $e_p = [\langle c',k\rangle]_p \mod p$. We claim that $||e_p||$ is so so small that $e_p = [\langle c',k\rangle]_p$. We have:

$$\begin{aligned}
||e_p|| &= ||-jp + \langle (p/q)\cdot c,k\rangle + \langle c' - (p/q)\cdot c,k\rangle|| \\
&\le ||-jp + \langle (p/q)\cdot c,k\rangle|| + ||\langle c' - (p/q)\cdot c,k\rangle|| \\
&\le (p/q)\cdot||[\langle c,k\rangle]_q|| + \gamma(R)\cdot\sum_{i=1}^{n}||c'[i] - (p/q)\cdot c[i]||\cdot||s[i]|| \\
&\le (p/q)\cdot||[\langle c,k\rangle]_q|| + \gamma(R)\cdot(r/2)\cdot\sqrt{d}\cdot\ell_1^{(R)}(k) \\
&\le p/2
\end{aligned}$$

Then, modulo r, we have $[\langle c', k\rangle]_p = e_p = \langle c', k\rangle - jp = \langle c, k\rangle - jq = [\langle c, k\rangle]_q$. ■

This theorem means that an evaluator who doesn't know the secret key, but knows a bound on the length of the secret key, can transform a ciphertext c that encrypts m under a key k for a modulus q ($m = [[\langle c, k\rangle]_q]_r$) can transform it into a ciphertext c' that encrypts m under the same key k for modulo p. ($m = [[\langle c, k\rangle]_p]_r$)

Corollary 3.1. *Let p and q be two odd moduli. Suppose c is an encryption of bit m under key k for modulus q, $m = [[\langle c, k\rangle]_q]_r$. Moreover, suppose that k is a fairly short key, and the "noise" $eq \leftarrow [\langle c, k\rangle]_q$ has small magnitude. Assume that $||eq|| < q/2 - (q/p)\cdot(r/2)\cdot\sqrt{d}\cdot\gamma(R)\cdot\ell_1^{(R)}(k)$. Then $c' \leftarrow \mathsf{Scale}(c, q, p, r)$ is an encryption of of bit m under key k for modulus p. $m = [[\langle c, k\rangle]_p]_r$. The noise $e_p = [\langle c', k\rangle]_p$ of the new ciphertext has magnitude at most $(p/q)\cdot||[\langle c, k\rangle]_q|| + \gamma(R)\cdot(r/2)\cdot\sqrt{d}\cdot\ell_1^{(R)}(k)$.*

Assuming $p < q$ and k has coefficients that are small in relationship with q, this permits the evaluator to reduce the magnitude of the noise without knowing the secret key.

4.6 FHE based on LWE without bootstrapping

In this scheme, Brakerski et.al. use a parameter L indicating the levels of the arithmetic circuit they want the FHE scheme to be capable of evaluating.

In this Section we will only look at the LWE problem variety of the algorithm.

- $\mathsf{FHE.Setup}(1^\lambda, 1^L)$ initialises the algorithm.
    1. Input: Security parameter λ, a number of levels L.
    2. Calculate $\mu = \mu(\lambda, L) = \theta(\log\lambda + \log L)$.
    3. To obtain a ladder of decreasing moduli from $q_L((L+1)\cdot\mu$ bits down to q_0 (μ bits). Run the following code:
       from $j = L$ to $j = 0$,
       run $\mathsf{params}_j \leftarrow \mathsf{E.Setup}(1^\lambda, 1^{j+1}\cdot\mu)$
    4. For $j = L-1$ to 0, replace the value of d_j in params_j with $d = d_L$ and the distribution χ with χ_L. This means that the ring dimension and noise distribution does not depend on the circuit level, but the vector dimension j still might.
- $\mathsf{FHE.Keygen}(\mathsf{params}_j)$ generates the keys needed. Repeat for $j = L$ to $j = 0$
    1. run $k_j \leftarrow \mathsf{E.SecretKeyGen}(\mathsf{params}_j)$

2. run $A_j \leftarrow \mathsf{E.PublicKeyGen}(\mathsf{params}_j, \mathsf{sk}_j)$
3. set $k'_j \leftarrow k_j \otimes k_j$
4. set $k''_j \leftarrow \mathsf{BitDecomposition}(k'_j, q_j)$
5. Run $\tau_{k''_{j+1} \to k_j} \leftarrow \mathsf{SwitchKeyGen}(k''_j, k_{j-1})$ (omit this step when $j = L$)

The secret key is the vector $\mathsf{sk} = [k_j]$, and the public key is the pair of vectors $\mathsf{pk} = ([A_j], [\tau_{k''_{j+1} \to k_j}]$.

- $\mathsf{FHE.Encrypt}(\mathsf{params}, \mathsf{pk}, m)$ encrypts the message m.
  1. Run $\mathsf{E.Enc}$ on a message $m \in R_2$
- $\mathsf{FHE.Decrypt}(\mathsf{params}, \mathsf{sk}, c)$ decrypts the ciphertext c
  1. Run $\mathsf{E.Dec}(k_j, c)$ where k_j is the key that belongs to c
- $\mathsf{FHE.Refresh}(c, \tau_{k''_{j+1} \to k_j}, q_j, q_{j-1})$ Takes a ciphertext c encrypted under k'_j, the information to facilitate key switching in $\tau_{k''_{j+1} \to k_j}$, the current modulus, q_j, the next modulus, q_{j-i}
  1. Expand: Set $c_1 \leftarrow \mathsf{PowersOf2}(c, q_j)$ (Observe: $\langle c_1, k''_j \rangle = \langle c, k'_j \rangle \mod q_j$ by lemma 2)
  2. Switch modulus: Set $c_2 \leftarrow \mathsf{Scale}(c_1, q_j, q_{j-1}, 2)$, a ciphertext under the key k''_j for the modulus q_{j-1}
  3. Switch keys: Output $c_3 \leftarrow \mathsf{SwitchKey}(\tau_{k''_{j+1} \to k_j}, c_2, q_{j-1})$, a ciphertext c_3 under the key k_{j-1} for modulus q_{j-1}.
- $\mathsf{FHE.Add}(\mathsf{pk}, c_1, c_2)$ Takes two ciphertexts and adds them together.
  1. If the ciphertexts are not encrypted under the same k_j, use $\mathsf{FHE.Refresh}$ to make it so.
  2. Set $c_3 \leftarrow c_1 + c_2 \mod q_j$. This is now a ciphertext under k'_j because we do $k'_j = k_j \otimes k_j$.
  3. output $c_4 \leftarrow \mathsf{FHE.Refresh}(c_3, \tau_{k''_{j+1} \to k_j}, q_j, q_{j-1})$

  Since addition does not introduce that much noise as multiplication, it might not be necessary to do the refresh step every time.
- $\mathsf{FHE.Multiply}(\mathsf{pk}, c_1, c_2)$ Takes two ciphertexts and multiplies them together.
  1. If the ciphertexts are not encrypted under the same k_j use $\mathsf{FHE.Refresh}$ to make them.
  2. Multiply: c_3 is the coefficients of the equation $L^{long}_{c_1,c_2}(x \otimes x)$
  3. Find the new key: $k'_j = k_j \otimes k_j$
  4. output $c_4 \leftarrow \mathsf{FHE.Refresh}(c_3, \tau_{k''_{j+1} \to k_j}, q_j, q_{j-1})$

We will discuss the equation $L^{long}_{c_1,c_2}(x \otimes x)$. We have two ciphertexts $c_0 = (c_0, c_1), c_1 = (c_0', c_1')$. The decryption of these are $\mathcal{D}(c_0) = c_0 + c_1 \cdot s$, $\mathcal{D}(c_1) = c_0' + c_1' \cdot s$ We now multiply these two together, to obtain the following:

$$\mathcal{D}(c_0 \cdot c_1) = c_0 c_0' + (c_1 c_0' + c_0 c_1') \cdot s + c_1 c_1' \cdot s^2 \tag{4.8}$$

Now we think about what we need to find that decrypts to what we want. We create a ciphertext that decrypts with the key $k_{new} = (1, s, s^2)$

Equation (4.8) is a linear equation on the form $a + bX + cX^2$, also known as $L^{long}_{c_1,c_2}(X \otimes X)$. So it is clear to see that this scheme grows both in the direction of noise, but also in the length of the number of components in the ciphertext.

4.7 Correctness and performance

It is clear that there is some noise added in each step, but the question remains how much noise is actually added. In this section we will give a brief summary of the discussion given in the BGV paper [3].

4.7.1 Corectness of the various parts of the algorithm

Correctness of E.Enc

The parameters params in E.Enc depend on the security parameter λ and the depth of the circuits, L. The noise in E.Enc comes from the distribution χ. Recall that we have decided to disregard λ, and this makes no difference in the noise, as it only depends on L.

Theorem 4. *Let n_L and q_L be the parameters associated with FHE.Enc. Let B_χ be a bound such that R-elements sampled from the noise distribution χ have length at most B_χ with overwhelming probability. The length of the noise in ciphertexts output by FHE.Enc is at most $1 + 2 \cdot ((2n_L + 1) \log q_L) \cdot B_\chi$*

Proof. Recall that $k \leftarrow$ E.SecretKeyGen and $\mathbf{A} \leftarrow$E.PublicKeyGen(k, N) for $N = (2n_L + 1) \log q_L$ where $\mathbf{A} \cdot k = 2\mathbf{e}$ for $\mathbf{e} \leftarrow \chi$. Recall that encryption works as follows: $\mathbf{c} \leftarrow \mathbf{m} + \mathbf{a}^T \mathbf{r} \mod q$ where $\mathbf{r} \in R_2^N$. We have that the noise of this ciphertext is $[\langle \mathbf{c}, k \rangle]_q = [m + 2\langle \mathbf{r}, \mathbf{e} \rangle]_q$, whose magnitude is at most $1 + 2 \cdot \sum_{j=1}^{N} ||\mathbf{r}[j]|| \cdot ||\mathbf{e}[j]|| \leq 1 + 2 \cdot N \cdot B_\chi$ ∎

The correctness follows if the noise levels are less than $q/2$.

Correctness of **FHE.Add** and **FHE.Multiply**

FHE.Multiply is the most interesting one of these two, as multiplication is the most complex algorithm. $\mathsf{FHE.Multiply}(\mathsf{pk}, c_1, c_2)$ uses the key k_j and modulus q_j. It has noise $e_i = [L_{c_i}(k_j)]_{q_j}$, where $L_{c_i}(k_j)$ is the dot product $\langle c_i, k_j \rangle$. As discussed earlier in the text, to multiply together two ciphertexts, we multiply two linear equations to get a quadratic equation $L^{long}_{c_1,c_2}(\mathbf{x} \otimes \mathbf{x}) = Q_{c_1,c_2}(\mathbf{x}) \leftarrow L_{c_1}(\mathbf{x}) \cdot L_{c_2}(\mathbf{x})$. The coefficients of this long quadratic equation is the new ciphertext c_3. So if c_1 and c_2 has noise at most B, then c_3 has noise at most B^2. To be able to decrypt, I need that $B^2 < q_j/2$.

FHE.Add follows a similar argument, but here, we add together the noise of two ciphertext with the same noise levels. So if c_1 and c_2 has noise less than B, then c_3 has noise less than $2B$. Again, we need to have noise levels $2B < q_j/2$.

Correctness of **FHE.Refresh**

FHE.Refresh is the most complex of these. It has three steps; Expand, Switch Moduli, and Switch Keys.

Expand takes as input a long ciphertext c encrypted under a long key $k'_j = k_j \otimes k_j$ for the modulus q_j. It applies the PowersOf2 algorithm, which expands c encrypted under k'_j to obtain c_1 encrypted under k''_j. No noise is added in this step.

Switch Moduli tales as input a ciphertext c_1 under the key k''_j for the modulus q_j. It outputs $c_2 \leftarrow \mathsf{Scale}(c_1, q_j, q_{j-1}, 2)$, which is a ciphertext under key k''_j and modulus q_{j-1}. If the noise of c_1 is $B < q_j/2 - (q_j/q_{j-1}) \cdot t_j$, the noise of c_2 is bounded by $(q_j/q_{j-1}) \cdot B + t_j$. t_j is defined by the key k''_j, as it is a vector in $R_2^{t_j}$, and $t_j \leq \binom{n_j+1}{2} \cdot \lceil \log q_j \rceil$. The noise of c_2 needs to be smaller than $q_{j-1}/2$.

Switch Key takes as input a ciphtertext c_2 under key k''_j for modulus q_{j-1}, and sets $c_3 \leftarrow \mathsf{SwitchKey}(\tau_{k''_j \to k_{j-1}}, c_2, q_{j-1})$, a ciphertext under key k_{j-1} and modulus q_{j-1}. Let c_2 be a ciphertext under the key k''_j for modulus q_{j-1}, such that the noise has length less than B. Let c_3 as described. Then the noise of c_3 will be at most $B + 2 \cdot \binom{n_j+1}{2} \cdot \lceil \log q_j \rceil^2 \cdot B_\chi$. As long as the new noise is less than $q_{j-1}/2$, there is correctness.

4.7.2 Putting these together

It has been shown that as long as the noise is small enough, correctness is ensured, but the parameters needs to be set so that this actually will happen. It is proven

in the BGV text [3] that this holds. The proof uses induction on the length of the largest modulus, and it is shown that the modulus needs to be approximately $(j+1)\cdot\theta(\log\lambda+\log L)$ bits, where λ is the security parameter and L is the depth of the circuit. So completely disregarding the security parameter is not wise, but it needs to be appropriately large, and judging that is out of scope for this text, as it varies by use case and security requirements.

4.7.3 A word on performance

The following theorem on performance is proved in the BGV paper [3]:

Theorem 5. *BGV is a correct L-leveled FHE scheme with the ability to evaluate circuits of depth L, with Add and Multiply gates. The per-gate computation time is* $\mathcal{O}(\lambda^3\cdot L^5)$.

It is clear that the depth of the circuit makes the computation time grow very fast, so there are practical limitations to how deep the circuits could be. The text goes on to describe some optimisations.

Batching attempts to parallelise some of the function by using the Chinese remainder theorem, and prime ideals of the plaintext space. Basically switching out the plaintext space R_2 with another plaintext space R_p, and then using the fact that $R_p \cong R_{p_1}\times\cdots\times R_{p_n}$.

4.8 PALISADE

PALISADE is a open source lattice-based homomorphic cryptography library written i C++ [16]. PALISADE implements, among other schemes, the BGV scheme. PALISADE also have a python port, however this is somewhat slower than the C++ version. PALISADE also offers other post-quantum cryptographic protocols, and multi-party computation. We have only used it to look at implementations of the BGV scheme.

There are other protocols that support homomorphic encryption. HElib[17] is one of them, however it is less user friendly, and gives less options in choice of protocols. There are currently no other implementations of the BGV scheme than these two, but other libraries support other algorithms.

4.9 Ring structure/field/transformation

Depending on the `MATHBACKEND` type chosen in PALISADE [18], the plaintext modulus chosen, and the memory constraints in the host system, there is a limit on how big integers can be. Also since the encryption takes place in a finite group $\mathbb{Z}$, but the real numbers is infinite, we need a way to limit the size of the numbers used. For integers, a size constraint needs to be in place. Decimal numbers needs to be converted to integers, and the precision needs to be bounded. In this section, we will look at several steps needed to create this transformation.

4.9.1 Narrowing down the number space

The first thing to do, is to look at what values will actually appear in the algorithm. $\mathbb{R}$ is very big, and $\mathbb{R}^n$ is even bigger. A good way to do this, is to look at the input data of the training and see what range the data is within, and look at the output data of the training set, and see what output values are realistic. Of course there could be outliers here, some extreme value in the actual data set that could not be predicted, but here we have to look at the probabilities of what could happen vs the cost of the added space. It will also speed up things if we center the calculations around the center of the data, whatever the center might be, instead of just default choosing 0 as the center. A transformation or linear shift of the data could be useful.

Dealing with negative numbers

One way of dealing with negative numbers, is to predict that they will show up, and then shift the entire data set by some number. That may or may not be possible in reality.

Another way of dealing with negative numbers, is to add a sign, and thus have a signed value. The problem with this is that in encryption there could be issues with this sign, and the cryptosystem might not be able to handle it. Luckily, PALISADES handles negative numbers, as long as the signed numbers are not too big.

A third way of doing it, is to say that $-1 = n-1$, because $n-1 \equiv -1 \mod n$. Doing this will not be an issue in my case, as the group size can be chosen to be much larger than the number space needed for the data input/output.

4.9.2 Binary numbers

An example of a way to pack $\mathbb{R}$ into $\mathbb{Z}$, is the binary numbers with a cutoff on size, such as int_64 in C++. Due to the natural size restrictions in memory, binary numbers already has a size cutoff. It is important that we measure the required accuracy of the numbers vs the size restrictions inherent to the system used.

4.9.3 Fixed point arithmetic

One way computers has been doing calculations for decades, is fixed point arithmetic. Fixed point arithmetic works by in advance deciding how many digits precision is needed, and multiply the number by this amount to get a integer instead of a floating point number. Any remaining digits will need to be rounded off. In the following example, I have used base 10 expansion, and multiplied each number with 100, and then rounded off any remainder.

Floating point	Fixed point
0.23	23
345	34500
0.01	1
53.203	5320

Table 4.1: Examples of floating point and the corresponding fixed point

There is no reason to not use a different base expansion, base 2 is a popular one due to binary numbers. The main problem is that depending on the base and precision, a large p is needed to fit the size of the numbers. A common example is 64 bits, but then the rest of the parameters will be very large. Again, it is useful to evaluate how much precision is needed and what size the numbers will be in the end.

Addition and subtraction in fixed point arithmetic

Addition and subtraction in fixed point arithmetic works as expected, the same way as adding or subtracting the underlying integers, as long as there is no overflow. Here is an example in base 10. It works the same way in other bases, there is no practical difference between the various bases in how the algorithm takes place, as will be seen later in this Section.

In base 2, over $\mathbb{F}_2$, addition is equivalent to xor, and the carry operation is equivalent to and. Here is an example of the naive way of doing addition in $\mathbb{F}_2$:

^{1}	
257	257
+133	-133
390	124

Table 4.2: Addition base 10

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1} \\ 1011 \\ +\ 111 \\ \hline 10010 \end{array}$$

Table 4.3: Addition in $\mathbb{F}_2$

When calculating the depth of this circuit, it is clear that the depth grows very quickly. The first operation done, is the rightmost addition, $1 \oplus 1 = 0$, carry 1. This carry can be written as $(1 \oplus 1)\wedge$, so the next operation is the second column from the right, $(1 \oplus 1) \wedge (1 \oplus 1) = 1$, with 1 carry. This second carry can be written as $((1 \oplus 1) \wedge (1 \oplus 1))\wedge$. The third operation is the third row, which can be written as $((1\oplus1)\wedge(1\oplus1))\wedge(0\oplus1) = 0$, carry 1. This carry equals $(((1\oplus1)\wedge(1\oplus1))\wedge(0\oplus1))\wedge$. The fourth operation is $(((1 \oplus 1) \wedge (1 \oplus 1)) \wedge (0 \oplus 1)) \wedge (1 \oplus 0) = 0$, carry 1, and this final operation is thus $((((1 \oplus 1) \wedge (1 \oplus 1)) \wedge (0 \oplus 1)) \wedge (1 \oplus 0)) \wedge (0 \oplus 0) = 1$. This gives a circuit depth of 9.

As you see this grows very quickly. 1011 and 111 are very small numbers, and there is no reason to believe the numbers will be this small. In addition, in many of the machine learning algorithms, there are linear combinations and large sums with many summands. The circuit gets very deep very quickly.

This can be generalised in the following way: Assume we are adding two numbers, a and b, so that $a + b = r$. We do binary expansion of the numbers so $a = a_n, \ldots, a_2, a_1$, vice versa for b. I use c_i for carry.

$$\begin{array}{r} c_{n+1}c_n \ldots c_2c_1 \\ a_n \ldots a_3a_2a_1 \\ +b_n \ldots b_3b_2b_1 \\ \hline r_{n+1}r_n \ldots r_3r_2r_1 \end{array}$$

Table 4.4: Addition generalised

In the same way as above, $r_1 = a_1 \oplus b_1$, and $c_1 = (a_1 \oplus b_1)$. We then have

1011	summand
+ 111	summand
1100	temporary sum
+0110	carry
10010	final sum

Table 4.5: The same example as before, but writing the carry differently

$$\begin{array}{r} a_n \dots a_2 a_1 \\ b_n \dots b_2 b_1 \\ \hline t_n \dots t_2 t_1 \\ c_{n+1} c_n \dots c_2 c_1 \\ \hline r_{n+1} r_n \dots r_2 r_1 \end{array}$$

Table 4.6: The addition carry trick generalized

$r_2 = c_1 \oplus (a_2 \oplus b_2)$. In general we have

$$\begin{aligned} r_k &= c_{k-1} \oplus a_k \oplus b_k \\ c_k &= c_{k-1} \wedge (a_{k-1} \oplus b_{k-1}) \end{aligned} \tag{4.9}$$

In general, if we do it this way, we can expect a circuit depth of $\mathcal{O}(\lg n)$, where n is the length of the largest number. This is straightforward to develop, but the results can be found in Discrete Mathematics and Its Applications [19]. It is clear that the main problem is calculating c_k, but it is also clear that c_k actually is only really dependent on a_{k-1} and b_{k-1}, as the rest is just dragged along. Having unlimited fan-in gates, it is possible to parallellise this, using the carry-look-ahead method, as described in a lecture by Barrington and Maciel[20]. In Table 4.5 is the same example again, doing a trick with where we write the carry.

Generalising this as we did previously with $r = a + b$, we get the following set up, as can be seen in Table 4.6.

Here we end up with a circuit of constant depth 3, with $\mathcal{O}(1)$, so given unlimited fan-in gates, it is in the complexity class AC^0 [20]. A very similar argument could be done for subtraction, and is thus left out.

Multiplication in fixed point arithmetic

Multiplication in fixed point arithmetic is harder. The result will have a scaling factor which is the original scaling factor squared. So here, rounding needs to happen, to avoid the numbers growing exponentially with many multiplications.

1	0	1	1	·	1	1	1
1	1	1	1	1			
			1	0	1	1	
		1	0	1	1		
	1	0	1	1			
1	0	0	1	1	0	1	

Table 4.7: Mulitiplication in $\mathbb{F}_2$

As an example, multiplying $2.57 \cdot 1.33$ will be $257 \cdot 133 = 34181$ and we clearly see that $2.57 \cdot 1.33 \neq 341.81$, so something happens here with the scaling factor that we need to take into account. $2.57 \cdot 1.33 = 3.4181$, but we only care about precision to two decimals, so $2.57 \cdot 1.33 \approx 3.42$. This means that whenever we multiply, we need to round off, or at least remember somehow that the scaling factor is increased.

Keeping this in mind, we will now look at $\mathbb{F}_2$, and calculate the circuit complexity. In the same way as for addition, this is straightforward to develop, but the results can be found in Discrete Mathematics and Its Applications [19]. Multiplication is logical or, $\vee$, and as before, addition is xor, $\oplus$, and carry is and, $\wedge$.

In Table 4.7 is an example, using the elementary school method of multiplying numbers together to multiply numbers in $\mathbb{F}_2$.

It is clear that the multiplication problem largely reduces down to adding together at most $n+1$ numbers, were n is the size of the largest of the factors a and b. In general, one can say that naive schoolbook multiplication uses n additions of n bits, so the complexity will be $\mathcal{O}(n^2)$, as can be seen in the text by Rosen [19].

The question that rises, is how fast can we add n numbers of n size together. There is a trick called the 3-2 trick, where we find numbers a, b, c, d, e such that $a+b+c = d+e$. Add, in parallel, every integer in a, b, c such that $d_i = a_i \oplus b_i \oplus c_i$, and $e_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$. Now the three numbers has been condensed into only two, in NC^0. Repeating this process for all n numbers until there are only two numbers left, and then using the trick from addition to add them together.

However, since we are not actually doing integer multiplication, but fixed point arithmetic, there is also a round off step, or a memory allocation of the increased precision.

There are also other ways to speed up multiplications, and a complexity of $\mathcal{O}(n^{1.585})$ is achieved by using a divide-and-conquer process. Explaining this method is out of scope for this text, but the algorithm can be found in Discrete Mathematics and Its Applications by Rosen [19].

Exponentiation

Exponentiation in fixed point arithmetic is sometimes done using Taylor power series, and the identity

$$a^x = e^{x \ln a} \tag{4.10}$$

This means that exponentiation is a complex issue. Looking at how to do this efficiently is a master thesis on its own, and is thus out of scope for this quick introduction.

Another way of doing exponentiation, when the exponent x is an integer, is to use lookup tables and binary expansion of x. This is a more efficient way of doing things, but can only be done for integer values of x.

Division in fixed point arithmetic

Regular division is fairly easy in base 2, where there are only two options on the quotient, either 0 or 1. However, in other bases, there are several options. In base 10, there are 10 possibilities for each digit in the quotient, and there is quite some trial and error involved. In general it is possible to do schoolbook division in $\mathcal{O}(n^2)$ [19]. We are not doing regular division, we are doing fixed point arithmetic, and this further complicates things.

To do fixed point division, we first take the integer quotient of the numbers, with rounding off, and then assume the scaling factor is the quotient of the scaling factors of the respective numbers. This causes rounding off twice, so precision suffers.

Here is an example: $2.570 \div 1.33$ turns into $2570 \div 133 = 19$ (rounded) and the scaling factor is $1000 \div 100 = 10$, so with the scaling factor added, we get 1.9, so $2.57 \div 1.33 = 1.9$. Comparing this to the actual result, which is closer to $2.570 \div 1.33 \approx 1.932331$. This lack of accuracy gets worse the more different the scaling factors are, and the first round off step with the integer division is also not great for accuracy. Changing the scaling factors before dividing can reduce the round off error, but then memory needs to be allocated to remember the changed scaling factors. It is not an easy question of rounding off vs memory issues, as when the numbers increase, the efficiency suffers.

Division by a constant

One way of speeding up division, is to consider the divisor d as a constant, and then multiply with d^{-1}. In the S-box of the AES algorithm, they exponent with

264 instead of dividing [21]. This is because of the cyclic nature of the underlying group. $d^{254} = d^{-1}$ when the order of the group is 256. This is in general seen as a less computationally heavy thing to do, and this sketches the outline of how hard division actually is.

4.9.4 Choosing an appropriate base

Usually, we either store numbers in base 10 or in base 2, but it could be computationally more efficient to use a different base. Every base will give rise to a new set of cryptographic parameters, and they will each have their pros and cons.

4.9.5 Chinese remainder theorem trick

Choosing several different bases that all has shallow circuits, and then using the Chinese remainder theorem to put them back together, could be a way to avoid too deep circuits.

If the size of the group is $n = p_1 \cdot p_2 \cdot \ldots \cdot p_m$, where p_i is a prime factor of n, we have a group isomorphy of the composed subgroups of size p_i. As an example, $\mathbb{Z}/n \cong \mathbb{Z}/p_1^{a_1} \times \mathbb{Z}/p_2^{a_2} \times \ldots \times \mathbb{Z}/p_n^{a_n}$.

It is possible to use CRT to create many separate smaller circuits with small p_i, and then combine them to the full group modulo. This can speed up calculation times quite a bit, as calculating in smaller groups is more efficient.

4.9.6 What does this mean?

It is clear from this Section that the adjustments needed to do efficient computations, are very intricate. For example does the fixed point division and the exponentiation problem show that there is good reason to not choose algorithms with division and exponentiation in the encrypted circuit. Some of these points are mostly to point out that doing these circuits in practice is not trivial, but luckily, PALISADE does most of the hard work for us. If we had chosen a more complex algorithm, we would have to pay closer attention to the points in this Section. It is still a good practice to look trough the input numbers to ensure they are not too large. This is simply because too large numbers will give errors in the calculation, if the plaintext modulus is not also chosen to be large. Choosing a large plaintext modulus slows down the calculations. One way of speeding up calculations even with a complex circuit and a large plaintext modulus, is to speed up the logic itself. Then this chapter is very useful, but doing this is a computer science project,

not a mathematics project, and is thus out of scope for this text.

Chapter 5

Secure Classification and architecture

Machine learning algorithm owners has spent a lot of time and money on developing, training, and fine tuning their machine learning algorithm, and companies might not want to simply give this away. In addition, an algorithm can require updates and patches, and sending off the algorithm could mean that the algorithm never gets these updates. However, the data owner wants analysis done on their data, but might not be able to send their data to the analysis company for privacy reasons. Both parties has valid reasons for wanting to keep their part secret. This is where homomorphic encryption comes into play.

After choosing an algorithm in Section 3.4, we need to look at how exactly the circuits are going to appear. In this section we will discuss the general idea, and in the next chapter, we will see an implementation of the specific algorithm that was chosen.

5.1 Architecture

5.1.1 General idea

The initial idea of the algorithm can be seen in Figure 1.1. The machine learning algorithm is trained using some training data, and then the classifier is encrypted on the client side by the owner of the machine learning algorithm. The data point to be analyzed is encrypted on the client side by the owner of the data. These two encrypted pieces of information is then sent to the diamond shaped box in the drawing, where the circuit is evaluated. What kind of algorithm is used is

Algorithm owner	Trusted third party	Data owner
Train Algorithm		
Encrypt algorithm		Encrypt Data
$\mathcal{E}(\mathsf{pk}, A) = a \rightarrow$		$\leftarrow \mathcal{E}(\mathsf{pk}, D) = b$
Evaluate Circuit		Evaluate Circuit
$\mathfrak{C}(a, b) = c \rightarrow$		$\leftarrow \mathfrak{C}(a, b) = c'$
	Decrypt c, c'	
	$\mathcal{D}(\mathsf{sk}, c) = d$	
	$\mathcal{D}(\mathsf{sk}, c') = d'$	
	Verify $d = d'$	
	$d \rightarrow$	
		Read the output d

Table 5.1: The generalised algorithm

Algorithm owner	Trusted third party	Data owner
Train Algorithm		
Encrypt algorithm		Encrypt Data
$\mathcal{E}(\mathsf{pk}, A) = a \rightarrow$		$\leftarrow \mathcal{E}(\mathsf{pk}, D) = b$
Evaluate Circuit		Evaluate Circuit
$\mathfrak{C}(a, b) = c \rightarrow$		$\leftarrow \mathfrak{C}(a, b) = c'$
	Verify $c = c'$	
	Decrypt c	
	$\mathcal{D}(\mathsf{sk}, c) = d$	
	$d \rightarrow$	
		Read the output d

Table 5.2: Algorithm in case of deterministic circuit evaluation

known to all parties, so it is just the parameters that needs to be kept secret. After evaluation, the result is decrypted and the output can be read. In this work, it is assumed that we introduce a trusted third party that verifies encryption and does decryption.

In Table 5.1, it is shown more specifically what happens in the algorithm, and the details of who does what.

If the circuit evaluation $\mathfrak{C}$ is deterministic, one does not have to decrypt before comparing, c and c' will be equal because there is no randomness introduced in the calculation. This can be seen in Table 5.2. A pitfall with deterministic evaluation, is that it could be possible to find the exact circuit parameters trough trial and error.

5.1.2 Why evaluate the circuit twice?

Everything that needs to be protected, is encrypted, and the algorithm used is known, so theoretically anyone with the computing power available could compute the circuit. However, there are some pitfalls.

If the algorithm owner evaluates the circuit, there is a chance the algorithm owner will change the data. After all, there is nothing stopping the algorithm owner from making their own $\mathcal{E}(\mathsf{pk}, D') = b'$. The algorithm owner also possess a copy of the key needed, and could simply discard the b from the data owner, and replace it with any desired b'.

If the data owner evaluates the circuit, there is a chance the data owner will change the parameters of the algorithm. In the same argument as for the algorithm owner, the data owner could make their own $\mathcal{E}(\mathsf{pk}, A') = a'$ to get a desired output result.

This leaves two options. Either a trusted third party does the circuit evaluation, or both parties does the circuit evaluation individually, and a trusted third party verifies the result. Both of these are viable options, and has their own pros and cons.

If the trusted third party calculates the circuit, the majority of the computation is done by the third party. This could be costly, as renting computation time at a data center can be expensive. In addition, since the trusted third party has the decryption key, theoretically the trusted third party could be malicious and decrypt the inputs, and thus learn everything about both the data and the algorithm. Protecting the data and the key is important, and in addition to the principle that the trusted third party should not do more work than possible, it leaves this as a less than ideal solution.

If both parties does the computation, and have the trusted third party verify, the computation work is done twice, but in separate instances. This mitigates the possibility of errors, as any errors will be detected when the decryption verification fails. In addition, the brunt of the computation work is done by the parties who want to do the computation. Any possibility of cheating by injecting a false a' or b' on one end would fail, as the circuit verification step would fail.

There is still the option that the parties send a false a' or b'. If any of the parties sends a false input, the output will be false, skewed, or straight up garbage. For this reason we will in this text assume all parties are at least semi-honest participants.

A note on the semi-honest model

There are many cases of algorithms that becomes trivial if we assume the semi-honest model compared to the malicious model. For example, in a coin toss situation, if all parties are semi-honest, the coin will always be fair, and no issues will arise. If one of the parties is malicious, the coin can be weighted, or there could be no coin toss at all. One can say that the semi-honest model is a weaker assumption than the malicious model [22]. Since we assume all parties follow the protocol, or else the resulting output is garbage, it is sufficient to say that we have a semi-honest adversary, even though this is a weaker assumption.

5.2 What do we need?

The parties want their part of the equation to be kept secret. What is formally needed to do this?

5.2.1 Secrecy

The first thing needed, is secrecy. There are several things that should be kept secret in this algorithm, and some of it will be harder to achieve than others.

Protecting the input data

The input data should be protected. Only the data owner should know what this is. The same goes for the input from the algorithm owner, only the algorithm owner should know what this is. The main advantage to both the algorithm owner and the data owner doing the circuit calculation, is to protect the input data from being decrypted by the trusted third party. Since neither the algorithm owner nor the data owner posses the decryption key, neither of them will be able to decrypt any part of the circuit, and thus will not be able to learn anything about the other party's input.

The output data should not leak more than neseccary

The output data will leak a little bit of information on the algorithm, as each iteration will give the answer to $f(\mathbf{x}) = \mathbf{y}$. With a linear equation in two dimensions, it is enough to have three output data points to accurately determine the slope of the line. This is unavoidable to some extent, due to the nature of the structure of the equations in question. Of course, with a more complex equation, more data

points is needed to see the contours of the underlying equation f, but given potentially thousands of runs of the algorithm, f will eventually leak. It is therefore important that the output does not leak more than necessary.

5.2.2 Integrity

Integrity is the full confidence that the data has not been manipulated or tampered with. It is already briefly discussed in the above Section, but we will discuss this more in depth here.

With any communication and sending of information, there is the risk that a man in the middle (MITM) will tamper with the information. The advantage when both parties does the circuit computation, is that the MITM can not tamper with the encrypted data a and b, but would have to interfer with c and c', or wait until the circuit is decrypted and interfer with d. The reasoning behind this, is that if a MITM interfers with say a, and replaces this with a', the circuit verification step will fail. A possible solution to this, is to use the usual strategy of signing the packets before sending them, so every party will know, and can verify, who sent what.

However, MITM is not the only option of destroying message integrity. One of the parties could be malicious, and send false packets, but as discussed earlier, if one party is malicious, the output will be skewed, or even straight up garbage. This is why we assume all parties are at least semi-honest and tries to follow the protocol as given.

In this text, we will assume that the channels are secure, and that there is no packet loss in transfer of the data. Thus, signing the packets will not be necessary in this simplified model.

Chapter 6

Security analysis and instantiation

In this chapter we will describe the model in detail, and then prove it to be secure. There is also an implementation that we will take a closer look at. As a reminder, we have chosen to look at algorithms where the response function f is of the type

$$\hat{Y} = \hat{\beta}_0 + \hat{\boldsymbol{\beta}}_1 \cdot \mathbf{x} \tag{6.1}$$

6.1 Description of my model

6.1.1 Encryption

First we will first describe how encryption is done in detail. All encryption is done using the public key pk.

Encrypting the classifier

The classifier is on this form: $\hat{Y} = \hat{\beta}_0 + \hat{\boldsymbol{\beta}}_1 \cdot \mathbf{x}$

The two $\hat{\beta}_i$'s are different. The first one, $\hat{\beta}_0$, is a number, and $\hat{\boldsymbol{\beta}}_1$ is a vector of numbers. Encrypting the $\hat{\beta}_i$'s together will create a long ciphertext, and due to the way the implementation works, it can not be used as it is. This is discussed in Section 6.3. Thus they clearly needs to be encrypted individually. Now it should be pointed out that the $\hat{\boldsymbol{\beta}}_1 \cdot \mathbf{x}$ is an inner product of two vectors, so each entry of $\hat{\boldsymbol{\beta}}_1$ needs to be encrypted individually for the same reasons. Encrypting the classifier turns into encrypting $m+1$ numbers individually, where m is the number

of coefficients of $\hat{\boldsymbol{\beta}}_1$. In the multi dimensional case with two β's, where $\hat{\boldsymbol{\beta}}_1$ is a vector, I get these two encryption equations:

$$\begin{aligned} &\mathcal{E}(\mathsf{pk}, \hat{\beta}_0) \\ &\mathcal{E}(\mathsf{pk}, \hat{\boldsymbol{\beta}}_1) = \mathcal{E}(\mathsf{pk}, \hat{\boldsymbol{\beta}}_{11}), \mathcal{E}(\mathsf{pk}, \hat{\boldsymbol{\beta}}_{12}), \ldots, \mathcal{E}(\mathsf{pk}, \hat{\boldsymbol{\beta}}_{1m}) \end{aligned} \tag{6.2}$$

Encrypting the input data

The data $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$ is a vector of numbers. The vector has length m, the same as the length of $\hat{\boldsymbol{\beta}}_1$. Using the same argument as for the classifier case, each entry of $\mathbf{x}$ needs to be encrypted individually, to be able to compute the inner product.

$$\mathcal{E}(\mathsf{pk}, \mathbf{x}) = \mathcal{E}(\mathsf{pk}, \mathbf{x}_1), \mathcal{E}(\mathsf{pk}, \mathbf{x}_2), \ldots, \mathcal{E}(\mathsf{pk}, \mathbf{x}_m) \tag{6.3}$$

As can be seen in the equations, both the classifier and the input data vector is encrypted under the same key.

6.1.2 Evaluating the circuit

We wish to evaluate a circuit using FHE.Add and FHE.Multiply from the BGV scheme.

Real number support vs fixed point arithmetic

In BGV, all plaintexts needs to be in $R_2 = R/2R$. All the coefficients are in $\{0, 1\}$, so at some point, the plaintext needs to be converted to binary. We have previously discussed ways of converting real numbers into fixed point values and binary numbers, and since BGV only supports binary numbers, it is clear that those techniques are needed here, however only when the message is still in plaintext form. There is no way of knowing how to round off or do precision tuning in the ciphertext, so we need to remember the changes done to the ciphertext when decrypting. In the implementation with PALISADE, they solve this by adjusting the parameters in MATHBACKEND.

Defining the inner product

When we defined the FHE scheme, we did not define any inner product, FHE.InnerProduct, and there is also no inner product algorithm in the PALISADE implementation of

BGV. This means that we need to define the inner product in a meaningful manner. The inner product of the Euclidean space is generally given as:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i \tag{6.4}$$

It is clear that this inner product uses only addition and multiplication, and thus can be used with FHE.Add and FHE.Multiply. However, we need to be careful because it is not straightforward how to do it.

First we need to do FHE.Multiply on each $x_i y_i = z_i$, and store the z_i's. This process could be parallelised if the processor power is available, as each $x_i y_i$ is independent of the others. We now have the expression

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} \mathbf{z}_i \tag{6.5}$$

This sum is n terms long, but each term is independent of each other. Again we have the question of how fast we can add n numbers together, but this time using FHE.Add as the addition operator. The BGV addition operator does not posses the ability to do the 3-2 addition trick, so another method should be used. One suggestion is to divide the sum up in pairs, and parallelising the process again. We get the following equation

$$\langle \mathbf{x}, \mathbf{y} \rangle = (\mathbf{z}_1 + \mathbf{z}_2) + (\mathbf{z}_3 + \mathbf{z}_4) + \ldots + (\mathbf{z}_{n-1} + \mathbf{z}_n) \tag{6.6}$$

In case there is an odd number of terms, just leave the last one for now. Each of the $(\mathbf{z}_{i-1} + \mathbf{z}_i)$ could be added together in parallel, and then we get a shorter sum of terms. There is n numbers, and we are adding them pairwise. We then add the result of those pairwise. This keeps going until there is just two terms left, and adding these together for the complete sum. This sum forms a structure that is similar to a binary tree, where each leaf node is product $\mathbf{z}_i$, and each node is the result of a sum of two previous nodes.

This method does not help with the number of summations in total, but it does even out the noise. If we think back to Section 4.9.3 on fixed point multiplication, we remember how the depth of the circuit grew very fast if we did schoolbook addition with the carries that followed all the way trough. If we do the calculation and just start from the left and add $((\mathbf{z}_1 + \mathbf{z}_2) + \mathbf{z}_3) + \ldots$, we get noise that carries all the way from the first addition, and the levels will quickly

be very uneven between the leftmost number and the next number to be added. Parallelising this process and adding in a tree-like structure, helps evening out the noise distribution, ensuring that the noise levels does not grow too much.

This process could have a very large depth, depending on how many terms there are in the sum. In general, it is clear that there are at most $n-1$ additions needed for the entire inner product, where n is the length of $\mathbf{x}$ and $\hat{\boldsymbol{\beta}}_1$. This is because this problem reduces down to the known problem of how many internal nodes there are in a binary tree with n leaves.

More about binary trees and this equation can be found in Introduction to Algorithms by Cormen et.al. [23].

Putting everything together

After running the inner product, we need to add $\hat{\beta}_0$ to the inner product. This generates one more addition, so we have at most $2n$ runs of FHE.Add and exactly n runs of FHE.Multiply in this circuit. The depth of the inner product circuit needs to be recorded, or else the keys will not match, and it will be a problem in the decryption step. It is known that the depth of any binary tree with m nodes is $\lfloor \log m \rfloor$ [23], so the total high level depth L of the inner product circuit is bounded above by

$$L \leq \lfloor \log n \rfloor + 1 \tag{6.7}$$

It is not an equality, because there might be an odd number of terms in the sum, and there might be some zeros in the calculations that collapse the tree structure of the calculation. There is not possible to save much in the depth, as even though there are some zeros, the tree will probably still be close to full.

If there are zeros in $\hat{\boldsymbol{\beta}}_1$, it means that the corresponding data in $\mathbf{x}$ is disregarded, and this will probably not be the case. If there are many zeros in $\mathbf{x}$, it is either missing data, or badly formatted data. It is in the best interest of the data owner to have well defined and easy to compute data, so it is not very probable that they will send bad data into the very costly calculation.

One problem could arise with the fact that the inner product circuit is deep, and has had a lot of work done to it, whereas the encryption of $\hat{\beta}_0$ is fresh. This means that we will have to use FHE.Refresh at least once, to refresh the inner product ciphertext to match $\hat{\beta}_0$. To ensure the correct key s_j is used, one needs to keep track of the depth of the inner product ciphertext, but this can easily be calculated in addition to keeping track of it with a simple counter.

6.1.3 Adjusting for logistic regression

The circuit only calculates $f(x) = \hat{Y} = \hat{\beta}_0 + \hat{\boldsymbol{\beta}}_1 \cdot \mathbf{x}$, but logistic regression uses the sigmoid curve. Calculating $e^{f(x)}$ when there is no exponent operation, is a very complex issue. One suggestion is to use the Taylor expansion of e^x, but even that gets very complicated fast. In addition, we have the sigmoid function itself, and thus we would have to combine several Taylor series to get the correct one. The circuits would also get very deep, and there would be the issue of the error in the tail of the expansion.

To avoid these issues, it is suggested to just calculate $f(x)$, and then do the sigmoid fitting after the fact. So the algorithm runs as planned, but after the data owner has received the response variable d, they themselves calculate the sigmoid function, using this equation:

$$Y = \frac{1}{1 + e^{-d}} \tag{6.8}$$

Considering how we can find d from Y, it is clear that it is sufficient to use the calculation power on just doing d.

$$d = -\log(\frac{1}{Y} - 1) \tag{6.9}$$

6.1.4 More about the key

The keen reader might have noticed that the inner key s_j gets used only once for each level of the circuit, and then discarded. This means that in key generation, we need to know an upper bound of the depth of the circuit, to ensure the key is not too short. We have already discussed the bound for the depth of the circuit in the previous section, so we have

$$j = \lfloor \log m \rfloor + 1 \tag{6.10}$$

The equality in this equation is because it is possible that the upper bound of the depth is fully used, and the key length needs to be at least the level L of the circuit. The key does not need to be longer than L, as the keys that have $j > L$ are never used.

The FHE.KeyGen takes as input this depth, so it is necessary to calculate it in advance, but in this case the calculation is easy, and there is as discussed not much variation in the possible depth of the circuit.

6.1.5 Decryption

The decryption is done by a trusted third party using FHE.Decrypt and the key sk. Both the data owner and the algorithm owner needs to send their circuit calculation result to the trusted third party for decryption. The parties also needs to submit the index j of the key to the trusted third party, as the decryption needs to use the decryption key corresponding to the key used in the last circuit evaluation. Usually this $j = L$, but there could be cases where the depth of the circuit is not full, so the index j needs to be attached.

6.2 Security proof

In this section we will prove the security of the model, using the principles of real-or-random indistinguishability.

6.2.1 Reduction to FHE

If we manage to reduce the security of the system to the security in FHE, we have guaranteed secrecy and integrity, as FHE guarantees this. This can be seen in the paper for the BGV scheme [3].

In the system, we have only used circuit operations that exists in BGV, namely FHE.Add and FHE.Multiply. We have also found a bound on the size of the circuit. This means that the system can use the BGV scheme.

One pitfall could be the fact that we are sending vectors of encrypted entries, a and b, so the length of these vectors are seen by an adversary. The algorithm used is known, but the length of the vectors is not defined in the algorithm, so this is an information leak. However, there is no link between the dimension of the input vectors a and b, and the value of the output number d. As we have assumed the algorithm is already trained, this vector length is fixed. The algorithm owner needs to tell the data owner the expected input length, or the data owner needs to tell the algorithm owner the size of the data vectors, so it can be assumed that this is known, and not a true leak of information.

6.2.2 Statement of the security theorem

Theorem 6. *Given an adversary that can distinguish between the system given in Table 5.1, and a simulator given in Table 6.1, it is possible to create an adversary that will break the BGV cryptosystem.*

Algorithm owner	Trusted third party	Data owner
Draw random data		Draw random data
$r_1 \xleftarrow{r} \chi$		$r_2 \xleftarrow{r} \chi$
Encrypt random data		Encrypt random data
$\mathcal{E}(\mathsf{pk}, r_1) = a \rightarrow$		$\leftarrow \mathcal{E}(\mathsf{pk}, r_2) = b$
Evaluate c		Evaluate c'
$\mathfrak{C}(a, b) = c \rightarrow$		$\leftarrow \mathfrak{C}(a, b) = c'$
	Discard c, c'	
	Calculate $d = F(A, D)$	
	$d \rightarrow$	Read the output d

Table 6.1: Simulator of the random system

The goal is to do a simulation based reduction proof, and to show that the system is real-or-random indistinguishable under a chosen plain text attack.

6.2.3 Simulator

First we build a simulator that simulates the cryptosystem, but uses random data. This simulator takes A, D, the input of the real cryptosystem, as input to the trusted third party. The trusted third party calculates $F(A, D) = d$. In addition, pk is given as input to the algorithm owner and the data owner, and r_i is a vector of random values drawn from a distribution χ, with a similar distribution as the actual input data. This can be seen in Table 6.1.

Since the circuit is known by any adversaries adv, and the adversary is able to compute the circuit themselves, it is important that c is actually computed from a and b. If c is found in some other way, it will be trivial for the adversary to see if it is a real or a random cryptosystem, as they could easily calculate c_{adv} and see that it does not match c or c'.

If it is not possible to see the difference between this simulator and the actual system, then the system is real-or-random indistinguishable secure under a chosen plain text attack.

6.2.4 Setting up the game

We want to set up a game system of Game A and Game B where Game A acts as the adversary in Game B, such that if the adversary of Game A, adv_A, manages to win Game A, Game B is won as well. We make the game such that my system is in Game A, and BGV is in Game B. See Figure 6.1 for an illustration of this.

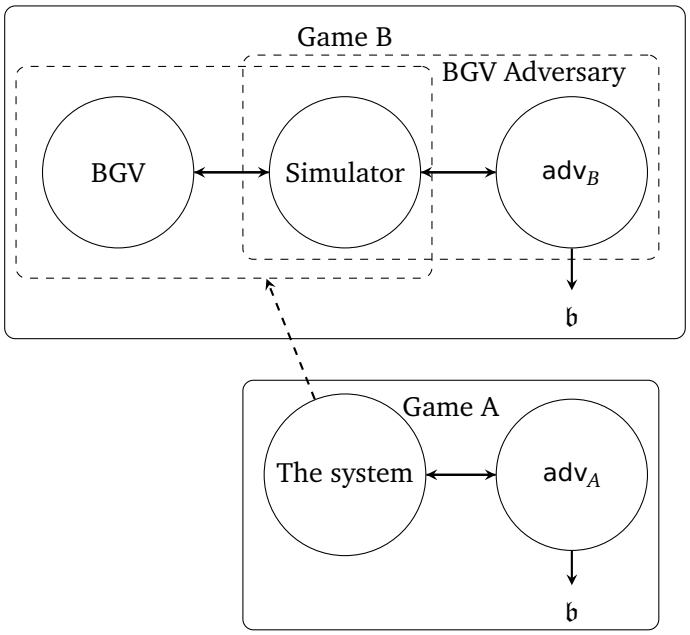


Figure 6.1: How the game system is set up

Game A

The adversary adv_A will follow the following protocol to the algorithm oracle:

1. adv_A chooses A_A, B_A and sends it to the system
2. The system chooses a bit $\mathfrak{b}_A \in \{0,1\}$ at random from χ_2
   - If $\mathfrak{b}_A = 0$, run the algorithm as given in Table 5.1
   - If $\mathfrak{b}_A = 1$, run the simulator as given in Table 6.1
3. adv makes a guess on $\mathfrak{b}_A$

To win, adv needs to be able to distinguish between the two types with probability more than

$$P(win) \geq \frac{1}{2} + \epsilon \tag{6.11}$$

where ϵ is some negligible probability as outlined in Equation (4.4).

Game B

The adversary adv_B will follow the following protocol to the BGV oracle:

1. adv_B asks for as many plaintext encryptions as they wish
2. BGV chooses a bit $\mathfrak{b}_B \in \{0,1\}$ at random from χ_2

3. BGV encrypts $c = \mathcal{E}(k, \mathfrak{b}_B)$
4. BGV sends c to adv_B
5. adv_B makes a guess on $\mathfrak{b}_B$

adv_B wins if they guess correctly with a non-negligible probability, as in Game A and Equation (6.11).

Putting Game A and Game B together - Game C

The goal is to write an algorithm so that Game A acts as the adversary in Game B, and if the adversary adv_C wins Game A, then that gives a win for Game B.

A_C and D_C is chosen by the adversary adv_C in advance. These could be any values, chosen in any way by adv_C.

adv_C	The algorithm	BGV
Choose $(A_C, D_C) \rightarrow$		
	Request encryption of $(\mathsf{pk}, A_C) \rightarrow$ $(\mathsf{pk}, D_C) \rightarrow$	
		Draw $\mathfrak{b}_0 \xleftarrow{r} \chi$ •If $\mathfrak{b}_0 = 0$: $a = \mathsf{FHE.Enc}(\mathsf{pk}, A_C)$ $b = \mathsf{FHE.Enc}(\mathsf{pk}, D_C)$ •If $\mathfrak{b}_0 = 1$ Draw $r_1, r_2 \xleftarrow{r} \chi$ $a = \mathsf{FHE.Enc}(\mathsf{pk}, r_1)$ $b = \mathsf{FHE.Enc}(\mathsf{pk}, r_2)$ Return $\leftarrow (a, b)$
Calculate $\mathfrak{C}(a, b) = c$ Send $c \rightarrow$		
	TTP discards c TTP calulates d $d = F(A_C, D_C)$ Return $\leftarrow (a, b, c, d)$	
Guess $\mathfrak{b}_0$		

Table 6.2: Game A and Game B put together

The adversary adv_C does not have access to encryption. If they had, it would be trivial for the adversary to calculate $\mathcal{E}(\mathsf{pk}, d)$ and check if this is equal to c. Another option for adv_C is to encrypt $\mathcal{E}(\mathsf{pk}, A_C)$ and check if this is equal to a.

The goal of the adversary adv_C is to guess if A_C, D_C was encrypted or if a random string was encrypted, so adv_C wants to guess the value of $\mathfrak{b}_0$. The adversary adv_C does not have any of the keys, so they are neither able to encrypt nor decrypt. This means that there is no way for adv_C to verify the encryption.

The encryption $\mathcal{E}(\mathsf{pk}, A_{\mathfrak{b}_0}) = a$ and $\mathcal{E}(\mathsf{pk}, D_{\mathfrak{b}_0}) = b$ is returned to the adversary, who calculates $c = \mathfrak{C}(a, b)$. This is done because we want the simulator to behave as closely to the algorithm as possible. Since the adversary does not have a copy of pk, there is no way for the adversary to verify if they are doing computations on A_A, D_A or on random data.

It is not strictly necessary to return d, as d is already known to adv_C, as they chose (A_C, D_C). As a reminder, d is calculated by the plaintext inputs, $d = \mathfrak{C}(A_C, D_C)$. It is necessary to send data to the TTP, or else adv_C will easily see the difference between the simulation and the actual algorithm, as no communication happens. adv_C will learn nothing about a, b, c from d.

6.2.5 Discussion and proof

The goal in this section, is to show that Game A and Game B is the same. The content of the dashed box to the left in Game B in Figure 6.1 can be shown to be the same as the circle with The System in Game A the same figure. If this is the case it is possible to create Game C. We then show that the probabilities of breaking Game A and Game B is the same, and thus the system is as secure as BGV.

The security of Game B is the security of BGV. This reduces down to the hardness of LWE. The reason we are not discussing Ring Learning With Errors, is that the scheme shown in this text is based on LWE. There is an alternative scheme using RLWE, but that has not been discussed in this text. Showing the hardness of LWE is out of scope for this text, but the proof can be found in Regev's text on Learing With Errors [24]. The idea of the proof is that the amount of pairs possible to choose at random is very high, and then given two pairs, it would be impossible to guess which is a random pair and which is an inner product. So encryptions of 0 and 1 are statistically indistinguishable.

In Game A, the adversary adv_A needs to distinguish between a real encryption and a random encryption made by a simulator. In Game B, the adversary adv_B needs to distinguish between an encryption of a bit $\mathfrak{b}_B \in \{0, 1\}$. It is possible to set up Game A such that the plaintexts chosen by adv_A is bits, such that $A_A, D_A = \mathfrak{b}_{A_A}, \mathfrak{b}_{D_A} \in \{0, 1\}$. This way, Game A is changed to distinguishing chosen bits from

randomness. Since the randomness is chosen from a similar range of values, it is fair to say that the randomness in Game A is also a bit. Hence, Game A and Game B are both to distinguish encryptions of bits.

The main difference between Game A and Game B is who chooses the encryption. In Game A, adv_A chooses the plaintext, and the simulator chooses if it is real or random data. In Game B, adv_B chooses nothing before receiving the plaintext they should assess. adv_B is allowed to make encryptions, before guessing the bit $\mathfrak{b}_B$. It is therefore no problem that the adversary in Game A adv_A does several encryptions before guessing. In practice, it does not matter what input is sent from adv_A, as they only receive one encryption back, and they need to guess $\mathfrak{b}_A$ from that, and that is the same guess as adv_B needs to make.

It is clear that Game A and Game B are the same, and thus we are able to make the combined Game C.

6.3 Implementation

We made a toy implementation of the system, using PALISADE. In this Section, we will go trough and discuss the code implementation example that can be found in Appendix A.

6.3.1 Fetching the values of ciphertext[i]

Ideally, the system should take a vector as input, and just encrypt that vector as it is. The problem with this approach, is that due to how PALISADE is set up, there is no way of pointing to an entry in a ciphertext vector. Encrypting one single value, leads to a ciphertext vector with six entries, five called EVAL 1 trough 5 and one to keep track of the modulus. So the usual strategy of pointing to vector[i] does not work in this case. This means that encrypting a whole vector in one chunk will not work, as the algorithm is based on that it is possible to fetch single values from the ciphertext. This means that the vectors will need to be cut into pieces and each entry needs to be encrypted by itself. The way the code is written, this means that the code is somewhat chunky, but this could be improved in the future.

6.3.2 Code

As can be seen in Appendix A, the code hard codes the values of $\beta_0, \boldsymbol{\beta}_1, \mathbf{x}$, but this part of the code could be re-written to take some vectors and numbers as input

from the user instead. No attempt as been made to decentralise the process, so that several users can give input independently, as that would require a server and communicating securely over a network. Also there has been no attempt to decentralised the decryption to a trusted third party, for the same reasons.

It is important to note that the plaintext modulus chosen sets a heavy restriction on the numbers used. Firstly, the input numbers can not be larger than the plaintext modulus. Secondly, trough experiments, we found that numbers that push the limit, are prone to giving errors in the calculations. An example of this can be seen in Listing A.4 in Appendix A.

Another thing to note, is that due to the way the BGV implementation is done, there is no decimal value support. This means that we will have to do fixed point adjustments and/or round off the values, as discussed in Section 4.9.3.

6.3.3 Time and correctness analysis

The code has only been run locally on a computer room computer. The CPU specifications of this computer can be seen in Listing A.2 in Appendix A.

Dimension equals 2

First we implemented the code with the dimension of $\boldsymbol{\beta}_1$ and $\mathbf{x}$ equal to 2. That gives a very simple code, and it ran in only a matter of milliseconds. A variety of values for $\beta_0, \boldsymbol{\beta}_1, \mathbf{x}$ was tested, and there was no significant change in the run time of the code, even when numbers were as large as possible within the plaintext modulus chosen. A correct code execution example output when the dimension equals 2, can be seen in Listing A.3 in Appendix A.

The code handles negative numbers, and outputs the correct value when some or all inputs are negative, with no significant changes in the run time of the algorithm.

We also found that if we set all values of $\{\beta_0, \boldsymbol{\beta}_1, \mathbf{x}\} = 65536$, which is 1 less than the plaintext modulus, there is an error in the calculation. It is also interesting to see that the time used increases significantly, total CPU time usage goes up from 0.325s, to 2.189s, without any change in dimension, as can be seen in Listing A.4 in Appendix A. This suggests that the numbers can not be too large compared to the plaintext modulus q. Since the values of $\boldsymbol{\beta}_1$ and $\mathbf{x}$ are multiplied together, they all need to be less than the square root of the plaintext modulus, to avoid overflow. In practice, they should be even smaller, since that will also give some

space for the addition. No attempt has been made to find these maximal values, since it depends on the amount of additions, and thus the length of the vectors.

Higher dimensions

After doing this, we tried to implement higher dimensional vectors $\boldsymbol{\beta}_1, \mathbf{x}$, which is the code shown in Listing A.1. The code example shows how it is set up for dimension 8.

For dimension 4, we got a slight increase in the time used, this can be seen in Listing A.5 in Appendix A. This increase is an approximate doubling of the time used from the correct runs of dimension 2.

We then tried doubling the length of the vectors, to dimension 8, and got approximately a doubling of the time used from dimension 4. The time used can be seen in Listing A.6 in Appendix A.

This is somewhat consistent with what I have written in Section 4.7.3, where the time used scales with $\sim L^5$, where L is the depth of the circuit. A doubling of the length of the vector gives one more level in the addition tree, and for small values of L, we get the following equation:

$$\frac{L_2^5}{L_1^5} = \frac{5^5}{4^5} \approx 3 \tag{6.12}$$

So in practice, I should expect to get even larger run time for dimension 8. However, the run time expected is given in $\mathcal{O}$, and for this small differences in run time, it could be that the initial start up cost of the algorithm is so large that the actual homomorphic steps gets drowned out. Another factor is that there are very few multiplications, and seeing a doubling in runtime each time a doubling in the number of additions is done, is as expected. This doubling can be seen in Figure 6.2, where the correct runs of the algorithm is plotted with the time used in each case. As one can clearly see, there is a near linear relationship on the time increased when the vectors are short. Interestingly enough, the slope of the line is approximately 0.3, which hints to the suggested rate of growth.

6.3.4 Discussion on the algorithm

In general, if the vectors get very large, it is important to note that the runtime will increase quickly. Both because of the homomorphic steps, but also just how the algorithm is written. The homomorphic steps will have a runtime increase

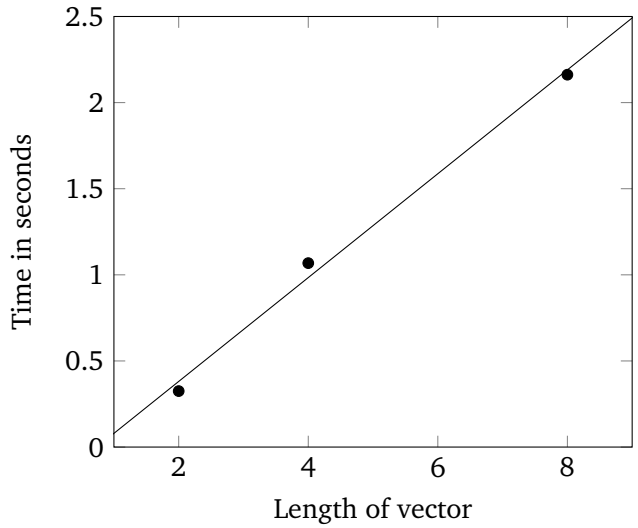


Figure 6.2: Runtime of the algorithm for vector length 2, 4 and 8

depending on the depth of the circuit. Mostly because of the binary tree structure of the inner product calculation, as that adds $\lfloor \log m \rfloor$ levels, as can be seen in Equation (6.7). In addition, the longer the vector, the more encryption steps there is, so more time will be spent on that, so the initialisation time will grow. This can be somewhat fixed if the code is cleaned up. In the end, the homomorphic steps will be the most significant part.

Since the BGV library in PALISADE does not support fetching a single value from an encrypted vector, the values needs to be encrypted separately. This leads to many variables declared. It is possible to put all of these in a vector, where each entry of the vector is an encrypted instance, with its own depth and all information attached. This was not implemented, as no long vectors were tried as input so it was possible to still track all values.

The code can be improved in several places. The first is the loading of the vector. The input vectors could be taken from a file or other user input, and then automatically divided into pieces and encrypted. This is to replace the hard coding of the vectors, as happens now. The second part is the additions, the binary tree structure could be implemented in a nicer and more automated way. Now, we need to know in advance how large the vector is, and then we need to hard code the addition steps, but this could be done in a recursive way. The last part is the output, it is possible to make the output nicer so it is more usable directly, depending on what the output is supposed to be used for. The final and most labor intensive part would be to set up the protocol as it is intended, with input from different

sources, and communication between the parties. One way to do this, would be to change the implementation in this text to taking ciphertexts as inputs, and just do the computation, and then write a different method to do the communications between the two parties and to the trusted third party.

Chapter 7

Conclusion

In this text, we have explored how homomorphic encryption can be used to create a privacy-preserving linear regression analysis. We have looked at how other researchers has done similar things, and found that linear regression is not done as a special case, so we made an algorithm for this, based on the BGV system. We also looked briefly at logistic regression, and made a sketch on how to do logistic regression with the system. In this implementation, only linear regression was done. It would be very easy to take the implementation and change it to fit the logistic regression algorithm sketched in this text.

The practical implications of this work are very straightforward. Someone who wants to do cryptographically private linear regression, could just take the algorithm and use it as it is. The proof of concept shows that the circuit calculation works as intended, but the implementation needs to be reworked if it is going to follow the algorithm completely as shown in this text.

In future works, it could be possible to look at a system that allows for malicious actors of this system. We did not go into this, as we assumed the parties would be semi-honest, but in principle there is no reason why not one or several of the parties could be malicious. It would also be possible to look more at other types of regression, maybe extend the algorithm to polynomial regression, or other types of non-linear regression.

Bibliography

[1] X. J. Jaideep Vaidya Hwanjo Yu, 'Privacy-preserving svm classification,' 2006. DOI: 10.1007/s10115-007-0073-7.

[2] C. Gentry, 'A fully homomorphic encryption scheme,' crypto.stanford.edu/craig, Ph.D. dissertation, Stanford University, 2009.

[3] Z. Brakerski, C. Gentry and V. Vaikuntanathan, *Fully homomorphic encryption without bootstrapping*, Cryptology ePrint Archive, Report 2011/277, https://eprint.iacr.org/2011/277, 2011.

[4] Y. Rahulamathavan, R. C.-W. Phan, S. Veluru, K. Cumanan and M. Rajarajan, 'Privacy-preserving multi-class support vector machine for outsourcing the data classification in cloud,' *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 2014. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6682897&tag=1.

[5] P. Bhattacharya, S. Jain and S. Nagpaul, *Basic Abstract Algebra*. Cambridge University Press, 1994, ISBN: 9780521466295.

[6] H. W. LENSTRA, *Lattices*. Algorithmic Number Theory, MSRI Publications, Volume 44, 2008.

[7] O. Regev, *Lll algorithm*, https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/lll.pdf, 2004.

[8] T. H. Gareth James Daniela Witten and R. Tibshirani, *An Introduction to Statistical Learning*. Springer, 2014.

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

[10] K. P. Murphy, *Probabilistic Machine Learning: An introduction*. MIT Press, 2021. [Online]. Available: probml.ai.

[11] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2007.

[12] S. Laur, H. Lipmaa and T. Mielikäinen, 'Cryptographically private support vector machines,' *IACR Cryptol. ePrint Arch.*, p. 198, 2006. [Online]. Available: http://eprint.iacr.org/2006/198.

[13] O. Goldreich, 'Secure multi-party computation,' *Manuscript. Preliminary Version*, Mar. 1999.

[14] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007, ISBN: 978-1-58488-551-1.

[15] M. Abdalla, P.-A. Fouque and D. Pointcheval, 'Password-based authenticated key exchange in the three-party setting,' in *Public Key Cryptography - PKC 2005*, S. Vaudenay, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 65–84, ISBN: 978-3-540-30580-4.

[16] *Palisade homomorphic encryption software library*, Accessed: 2021-06-10. [Online]. Available: https://palisade-crypto.org/.

[17] *Helib homomorphic encryption software library*, Accessed: 2021-06-06. [Online]. Available: https://github.com/homenc/HElib.

[18] *Palisade math documentation*, Accessed: 2021-10-18. [Online]. Available: https://gitlab.com/palisade/palisade-release/blob/master/src/core/include/math/README.md.

[19] K. H. Rosen, *Discrete Mathematics and Its Applications: Custom 7th edition,* McGraw-Hill Higher Education, 2015, ISBN: 9780077174521.

[20] D. M. Barrington and A. Maciel, *Lecture 2: The complexity of some problems*, https://lin-web.clarkson.edu/~alexis/PCMI/Notes/lectureB02.pdf, Jul. 2000.

[21] D. R. Stinson, *Cryptography: Theory and Practice*, 3rd. USA: CRC Press, Inc., 2006, ISBN: 1-58488-508-4.

[22] Y. Lindell, *How to simulate it – a tutorial on the simulation proof technique*, 2021. [Online]. Available: https://eprint.iacr.org/2016/046.pdf.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 3nd. The MIT Press, 2009, ISBN: 978-0-262-53305-8.

[24] O. Regev, 'The learning with errors problem (invited survey),' in *2010 IEEE 25th Annual Conference on Computational Complexity*, 2010, pp. 191–204. DOI: 10.1109/CCC.2010.26.

Appendix A

Code

A.1 Implementation

```
#include "palisade.h"

using namespace lbcrypto;

int main() {
  // Step 1 - Set CryptoContext

  // Setting the main parameters
  int plaintextModulus = 65537;
  double stdDeviation = 3.2;
  SecurityLevel securityLevelType = HEStd_128_classic;
  uint32_t depth = 5;

  // Instantiating the crypto context, where the last two parameters
     are key refresh type and modulus switching type.
  CryptoContext<DCRTPoly> cryptoContext =
      CryptoContextFactory<DCRTPoly>::genCryptoContextBGVrns(
          depth, plaintextModulus, securityLevelType, stdDeviation,
    depth, OPTIMIZED, BV);

  // Enabling features that I wish to use
  cryptoContext->Enable(ENCRYPTION);
  cryptoContext->Enable(SHE);
  cryptoContext->Enable(LEVELEDSHE);

  // Step 2 - Key Generation

  // Initializing Public Key Containers
```

```
27  LPKeyPair<DCRTPoly> keyPair;
28
29  // Generating a public/private key pair
30  keyPair = cryptoContext->KeyGen();
31
32  // Generating the relinearization key
33  cryptoContext->EvalMultKeyGen(keyPair.secretKey);
34
35  // Step 3 - Encryption
36
37  // beta0 is encoded
38  std::vector<int64_t> beta0 = {-3};
39  Plaintext plaintextbeta0 = cryptoContext->MakePackedPlaintext(
      beta0);
40  // beta1 vector is encoded
41  std::vector<int64_t> beta11 = {7};
42  Plaintext plaintextbeta11 = cryptoContext->MakePackedPlaintext(
      beta11);
43  std::vector<int64_t> beta12 = {6};
44  Plaintext plaintextbeta12 = cryptoContext->MakePackedPlaintext(
      beta12);
45  std::vector<int64_t> beta13 = {-7};
46  Plaintext plaintextbeta13 = cryptoContext->MakePackedPlaintext(
      beta13);
47  std::vector<int64_t> beta14 = {-6};
48  Plaintext plaintextbeta14 = cryptoContext->MakePackedPlaintext(
      beta14);
49  std::vector<int64_t> beta15 = {2};
50  Plaintext plaintextbeta15 = cryptoContext->MakePackedPlaintext(
      beta15);
51  std::vector<int64_t> beta16 = {5};
52  Plaintext plaintextbeta16 = cryptoContext->MakePackedPlaintext(
      beta16);
53  std::vector<int64_t> beta17 = {9};
54  Plaintext plaintextbeta17 = cryptoContext->MakePackedPlaintext(
      beta17);
55  std::vector<int64_t> beta18 = {1};
56  Plaintext plaintextbeta18 = cryptoContext->MakePackedPlaintext(
      beta18);
57  // xvector vector is encoded
58  std::vector<int64_t> xvector1 = {5};
59  Plaintext plaintextxvector1 = cryptoContext->MakePackedPlaintext(
      xvector1);
60  std::vector<int64_t> xvector2 = {10};
61  Plaintext plaintextxvector2 =cryptoContext->MakePackedPlaintext(
```

```
  xvector2);
std::vector<int64_t> xvector3 = {5};
Plaintext plaintextxvector3 = cryptoContext->MakePackedPlaintext(
  xvector3);
std::vector<int64_t> xvector4 = {-5};
Plaintext plaintextxvector4 = cryptoContext->MakePackedPlaintext(
  xvector4);
std::vector<int64_t> xvector5 = {-2};
Plaintext plaintextxvector5 = cryptoContext->MakePackedPlaintext(
  xvector5);
std::vector<int64_t> xvector6 = {9};
Plaintext plaintextxvector6 = cryptoContext->MakePackedPlaintext(
  xvector6);
std::vector<int64_t> xvector7 = {1};
Plaintext plaintextxvector7 = cryptoContext->MakePackedPlaintext(
  xvector7);
std::vector<int64_t> xvector8 = {3};
Plaintext plaintextxvector8 = cryptoContext->MakePackedPlaintext(
  xvector8);


// The encoded vectors are encrypted
auto ciphertextbeta0 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta0);
auto ciphertextbeta1_1 = cryptoContext->Encrypt(keyPair.publicKey,
   plaintextbeta11);
auto ciphertextbeta1_2 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta12);
auto ciphertextbeta1_3 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta13);
auto ciphertextbeta1_4 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta14);
auto ciphertextbeta1_5 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta15);
auto ciphertextbeta1_6 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta16);
auto ciphertextbeta1_7 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta17);
auto ciphertextbeta1_8 = cryptoContext->Encrypt(keyPair.publicKey,
  plaintextbeta18);
auto ciphertextxvector_1 = cryptoContext->Encrypt(keyPair.
  publicKey, plaintextxvector1);
auto ciphertextxvector_2 = cryptoContext->Encrypt(keyPair.
  publicKey,plaintextxvector2);
auto ciphertextxvector_3 = cryptoContext->Encrypt(keyPair.
```

```
    publicKey,plaintextxvector3);
89  auto ciphertextxvector_4 = cryptoContext->Encrypt(keyPair.
    publicKey,plaintextxvector4);
90  auto ciphertextxvector_5 = cryptoContext->Encrypt(keyPair.
    publicKey,plaintextxvector5);
91  auto ciphertextxvector_6 = cryptoContext->Encrypt(keyPair.
    publicKey,plaintextxvector6);
92  auto ciphertextxvector_7 = cryptoContext->Encrypt(keyPair.
    publicKey,plaintextxvector7);
93  auto ciphertextxvector_8 = cryptoContext->Encrypt(keyPair.
    publicKey,plaintextxvector8);
94
95  // Step 4 - Evaluation of the circuit
96
97  // Inner product of dimension 8
98  // multiply together pairwise, this adds 1 level evenly across
    everything
99  auto ciphertextPairwiseMult_1 = cryptoContext->EvalMult(
    ciphertextbeta1_1,ciphertextxvector_1);
100 auto ciphertextPairwiseMult_2 = cryptoContext->EvalMult(
    ciphertextbeta1_2,ciphertextxvector_2);
101 auto ciphertextPairwiseMult_3 = cryptoContext->EvalMult(
    ciphertextbeta1_3,ciphertextxvector_3);
102 auto ciphertextPairwiseMult_4 = cryptoContext->EvalMult(
    ciphertextbeta1_4,ciphertextxvector_4);
103 auto ciphertextPairwiseMult_5 = cryptoContext->EvalMult(
    ciphertextbeta1_5,ciphertextxvector_5);
104 auto ciphertextPairwiseMult_6 = cryptoContext->EvalMult(
    ciphertextbeta1_6,ciphertextxvector_6);
105 auto ciphertextPairwiseMult_7 = cryptoContext->EvalMult(
    ciphertextbeta1_7,ciphertextxvector_7);
106 auto ciphertextPairwiseMult_8 = cryptoContext->EvalMult(
    ciphertextbeta1_8,ciphertextxvector_8);
107
108 // Sum pairwise in a tree like structure, this adds floor of log n
     levels
109 auto ciphertextSummed1 = cryptoContext->EvalAdd(
    ciphertextPairwiseMult_1, ciphertextPairwiseMult_2);
110 auto ciphertextSummed2 = cryptoContext->EvalAdd(
    ciphertextPairwiseMult_3, ciphertextPairwiseMult_4);
111 auto ciphertextSummed3 = cryptoContext->EvalAdd(
    ciphertextPairwiseMult_5, ciphertextPairwiseMult_6);
112 auto ciphertextSummed4 = cryptoContext->EvalAdd(
    ciphertextPairwiseMult_7, ciphertextPairwiseMult_8);
113
```

```
114   auto ciphertextSummed1_1 = cryptoContext->EvalAdd(
        ciphertextSummed1, ciphertextSummed2);
115   auto ciphertextSummed1_2 = cryptoContext->EvalAdd(
        ciphertextSummed3, ciphertextSummed4);
116
117   auto ciphertextSummed = cryptoContext->EvalAdd(ciphertextSummed1_1
        , ciphertextSummed1_2);
118
119   // Add the constant term, this adds 1 level total
120   auto ciphertextAddedBeta0 = cryptoContext->EvalAdd(ciphertextbeta0
        , ciphertextSummed);
121
122   // Step 5 - Decryption
123
124   // Decrypt the result of the circuit Evaluation
125   Plaintext plaintextCircuit;
126   cryptoContext->Decrypt(keyPair.secretKey, ciphertextAddedBeta0,&
        plaintextCircuit);
127
128   // Output results
129   std::cout << "\nResult of encrypted circuit:" << std::endl;
130   std::cout << "#beta0 + beta1*x = " << plaintextCircuit << std::
        endl;
131
132
133   return 0;
134 }
```

Listing A.1: My algorithm using PALISADE

A.2 lspcu

```
1  lesesal5411:~$ lscpu
2  Architecture:        x86_64
3  CPU op-mode(s):      32-bit, 64-bit
4  Byte Order:          Little Endian
5  CPU(s):              8
6  On-line CPU(s) list: 0-7
7  Thread(s) per core:  2
8  Core(s) per socket:  4
9  Socket(s):           1
10 NUMA node(s):        1
11 Vendor ID:           GenuineIntel
12 CPU family:          6
13 Model:               60
```

```
Model name:          Intel(R) Core(TM) i7-4790S CPU @ 3.20GHz
Stepping:            3
CPU MHz:             3172.131
CPU max MHz:         4000.0000
CPU min MHz:         800.0000
BogoMIPS:            6385.15
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            8192K
NUMA node0 CPU(s):   0-7
[...]
```

Listing A.2: lscpu on the system used

A.3 Time used

```
$ time ./bin/examples/pke/pke

Results of homomorphic computations
#beta0 + beta1*x: ( 33 ... )

real  0m0.103s
user  0m0.325s
sys 0m0.010s
```

Listing A.3: Time used when the dimension equals 2

```
Results of homomorphic computations
#beta0 + beta1*x: ( 1 ... )

real  0m0.383s
user  0m2.189s
sys 0m0.012s
```

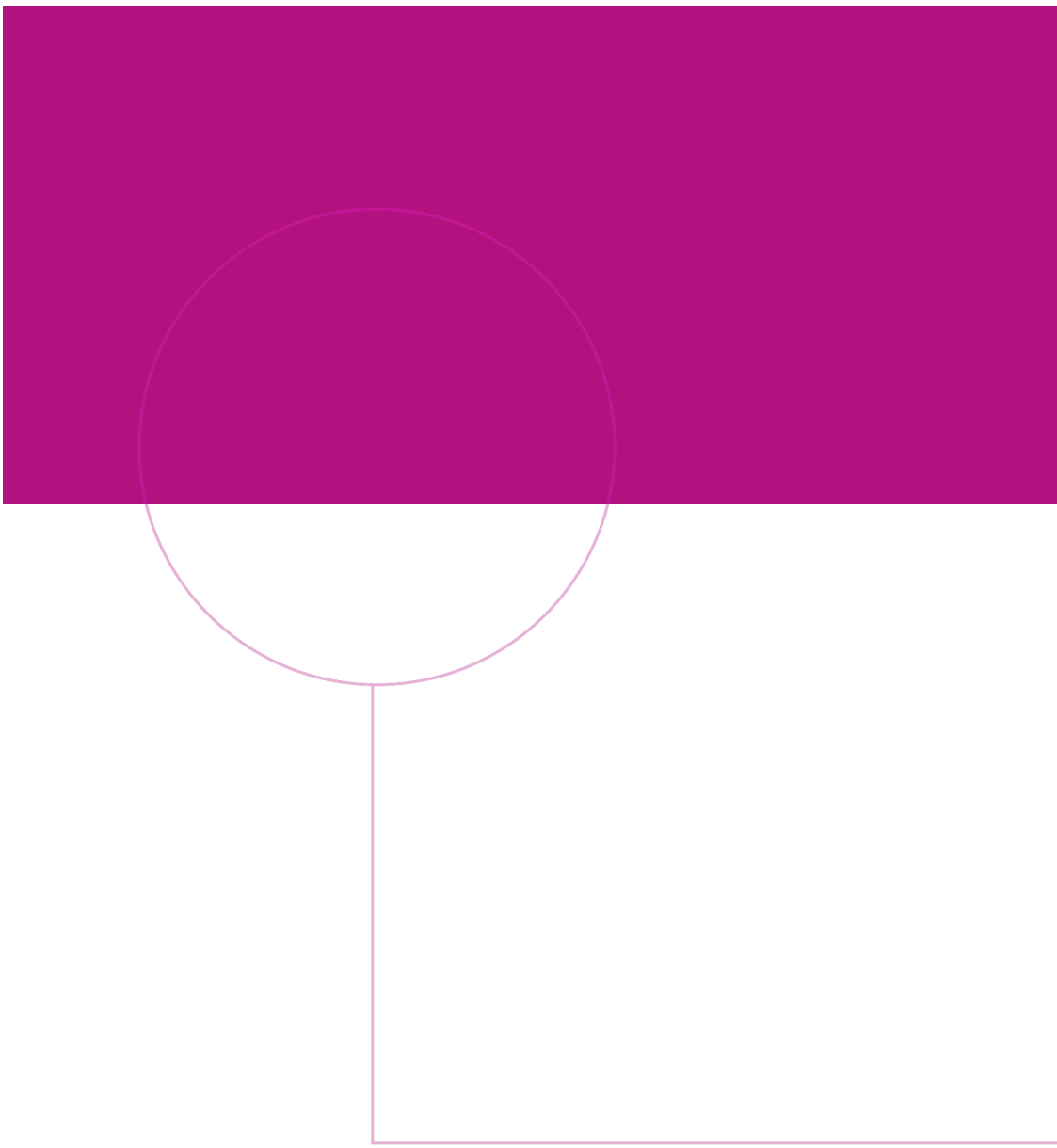
Listing A.4: Time used when every number equals 65536 and dimension is 2

```
Results of homomorphic computations
#beta0 + beta1*x: ( 162 ... )

real  0m0.317s
user  0m1.068s
sys 0m0.025s
```

Listing A.5: Time used when the dimension of the vectors equals 4

```
Results of homomorphic computations
#beta0 + beta1*x: ( 140 ... )

real  0m0.610s
user  0m2.162s
sys 0m0.041s
```

Listing A.6: Time used when the dimension of the vectors equals 8



NTNU

Norwegian University of
Science and Technology