POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Modern Fully Homomorphic Encryption (FHE)

Supervisor
prof. Antonio Lioy
ing. Daniele Canavese

Candidate
Fabio Trovero

April 2024

To my Family and Friends

Acknowledgements

I would like to express my deepest gratitude to Prof. Antonio Lioy and Ing. Daniele Canavese for their invaluable guidance, encouragement and unwavering support throughout my research journey. Their expertise, patience, and constructive feedback have been instrumental in shaping this thesis.

I extend my heartfelt appreciation to my family for their boundless love, motivation, and understanding. Their unwavering belief in me has been the cornerstone of my academic pursuits.

To my friends, I am grateful for your constant encouragement, understanding, and unwavering support. Your presence has brought joy and companionship to every step of this journey.

I would also like to thank all those who have supported me in various ways, whether through discussions, encouragement, or providing resources. Your contributions have been deeply appreciated and have made a significant difference in the completion of this thesis.

Thank you all for being an integral part of this work and for enriching my academic and personal growth in countless ways.

Contents

List of Figures 8

List of Tables 10

1 Introduction 11

2 Homomorphic Encryption 15

2.1 Definitions . . . 15
2.2 FHE Schemes . . . 17
2.2.1 Partially Homomorphic Encryption Schemes . . . 17
2.2.2 Somewhat Homomorphic Encryption Schemes . . . 18
2.2.3 Fully Homomorphic Encryption Schemes . . . 19
2.2.4 Ideal Lattice-based FHE Schemes . . . 19
2.2.5 FHE Schemes over Integers . . . 21
2.2.6 LWE-Based FHE Schemes . . . 21
2.2.7 NTRU-Like FHE Schemes . . . 22
2.2.8 Alternative taxonomy . . . 22
2.3 Security Properties . . . 23
2.4 Use cases . . . 25
2.4.1 FHE for machine learning . . . 25
2.4.2 HE in Fog Computing for IoT . . . 27
2.4.3 HE in cloud computing . . . 28

3 FHE Libraries and Compilers 33

3.1 FHE Libraries . . . 33
3.2 FHE Compilers and Accelerators . . . 35
3.3 Concrete Library by Zama . . . 37
3.3.1 Implementation . . . 37
3.3.2 Core API . . . 38
3.3.3 Crypto API . . . 39
3.3.4 Example of usage . . . 40
3.4 OpenFHE library . . . 40

3.4.1 Cryptographic capabilities . . . 41
3.4.2 Bootstapping, Noise Estimation and Scheme Switching . . . 41
3.4.3 Hardware Acceleration Support . . . 42
3.4.4 Usability Enhancements . . . 42
3.4.5 Example of usage . . . 43
3.5 Comparison between Concrete and OpenFHE . . . 44

4 Machine Learning 47
4.1 Type of Data and Machine Learning Techniques . . . 47
4.2 Machine Learning Algorithms . . . 48
4.2.1 Classification . . . 49
4.2.2 Regression . . . 51
4.3 Concrete-ml example of usage . . . 53

5 Methodology 55
5.1 Hardware architecture and software configuration . . . 55
5.2 Strategies . . . 56
5.2.1 Configuration Objects . . . 57
5.2.2 Dataset generation . . . 58
5.2.3 Univariate . . . 59
5.2.4 Results format . . . 59
5.3 Evaluation metrics for Machine Learning . . . 61
5.3.1 Classification evaluation metrics . . . 62
5.3.2 Regression evaluation metrics . . . 63

6 Experimental Results 65
6.1 Exhaustive test for input set . . . 65
6.2 Size and Time . . . 66
6.3 Basic operation . . . 67
6.3.1 Addition . . . 68
6.3.2 Subtraction . . . 70
6.3.3 Multiplication . . . 70
6.3.4 Division . . . 72
6.3.5 Power . . . 72
6.3.6 Square Root . . . 73
6.4 ML operation . . . 74
6.4.1 Logistic Regression . . . 74
6.4.2 Linear Support Vector Regression . . . 75
6.4.3 Linear SVM Classification . . . 76

7 Conclusion 81

Bibliography 83

A Distribution of data 87

A.1 data distribution addition . . . 87
A.2 data distribution subtraction . . . 87
A.3 data distribution multiplication . . . 88
A.4 data distribution division . . . 88
A.5 data distribution power . . . 88
A.6 data distribution square root . . . 88

B Mathematical Foundations 93

B.1 Number Theory . . . 93
B.2 Probability Theory . . . 94
B.3 Definitions . . . 94
B.3.1 Lattice . . . 94
B.4 Other Key Concepts . . . 95

C User Manual 97

C.1 Directory structures . . . 97
C.2 Instructions for installation . . . 98
C.3 How to use . . . 98
C.3.1 Mathematical Operations . . . 99
C.3.2 Machine Learning test . . . 99

D Developer Manual 101

D.1 Scripts Description . . . 101
D.2 How to write test with Concrete . . . 102
D.3 How to write test with Concrete-ml . . . 103

List of Figures

2.1 Client/Server HE scenario. 16
2.2 Bootstrapping technique. 21
2.3 Generation of FHE Schemes pros/cons. 22
2.4 IND-CPA security. 24
2.5 PPML model example. 26
2.6 Hybrid FHE applied to fog computing IoT. 28
2.7 HPRE for homomorphic evaluation of multiuser data. 29
2.8 FHAE by composition encrypt-then-sign. 30
2.9 MPC constructed with MKFHE. 31

3.1 Concrete deployment. 38
3.2 CONCRETE architecture. Dashed boxes refer to upcoming features. 39
3.3 Example of usage of the concrete library. 41
3.4 Layers in openFHE. 43
3.5 Boolean Fully Homomorphic examples with default bootstrapping. 45

4.1 Types of machine learning techniques 48
4.2 General structure of a predictive model. 48
4.3 Linear SVM optimize margin. 50
4.4 Example of a decision tree structure. 51
4.5 Example of a random forest structure. 52
4.6 Classification vs regression. 52
4.7 Concrete-ml example of usage. 53

5.1 FHE configuration object. 58
5.2 FHE configuration object with parallelism enabled. 58
5.3 Script for sum and sub dataset generation. 59
5.4 Example of univariate function. 60
5.5 Output data format. 60
5.6 Converter script to Excel sheets example. 61
5.7 Confusion matrix for classification. 62

6.1 Exhaustive test. 65
6.2 Add with a constant value. 66
6.3 Sum of the elements of the vector. 68
6.4 Steps involved in a FHE computation. 68
6.5 Comparison average time addition. 69
6.6 Performance trend for addition. 70
6.7 Comparison average time subtraction. 71
6.8 Comparison average time multiplication. 73
6.9 Comparison average time division. 74
6.10 Comparison average time power. 76
6.11 Comparison average time square root. 77
6.12 Comparison time logistic regression. 78
6.13 Dataset for logistic regression analysis. 79
6.14 Comparison time linear support vector regression. 79
6.15 Comparison time linear SVM classification. 80

A.1 Time distribution of the data with parallelization on for addition. 87
A.2 Time distribution of the data without FHE enable for addition. 87
A.3 Time distribution of the data with parallelization off for addition. 88
A.4 Time distribution of the data with parallelization off for substraction. 88
A.5 Time distribution of the data with parallelization on for substraction. 89
A.6 Time distribution of the data without FHE enable for substraction. 89
A.7 Time distribution of the data with parallelization off for multiplication. 89
A.8 Time distribution of the data with parallelization on for multiplication. 89
A.9 Time distribution of the data without FHE enable for multiplication. 90
A.10 Time distribution of the data with parallelization off for division. 90
A.11 Time distribution of the data with parallelization on for division. 90
A.12 Time distribution of the data without FHE enable for division. 90
A.13 Time distribution of the data with parallelization off for power. 91
A.14 Time distribution of the data with parallelization on for power. 91
A.15 Time distribution of the data without FHE enable for power. 91
A.16 Time distribution of the data with parallelization off for square root. 91
A.17 Time distribution of the data with parallelization on for square root. 92
A.18 Time distribution of the data without FHE enable for square root. 92

C.1 Project directory structure. 97

List of Tables

2.1 Timeline of the most important HE schemes and their application. 17

2.2 Homorphic Properties of most used PHE schemes. 18

2.3 Comparison of Well-Known Pre-Gentry Somewhat Fully Homomorphic Encryption (SWHE) Schemes. 19

3.1 Available Open-Source FHE Libraries. 33

3.2 Available FHE Compilers. 35

5.1 Specification of test platform. 56

5.2 Metrics for evaluation of classification. 63

6.1 Add function with data size of 28 bytes and function of 136 bytes. 66

6.2 Univariate function with data size of 176 bytes and function size of 136 bytes. . . . 67

6.3 Average computational time for addition. 69

6.4 Compilation and key generation time for addition. 70

6.5 Times trend for addition. 71

6.6 Average computational time for subtraction. 72

6.7 Compilation and key generation time for subtraction. 72

6.8 Average computational time for multiplication. 73

6.9 Compilation and key generation time for multiplication. 74

6.10 Average computational time for division. 75

6.11 Compilation and key generation time for division. 75

6.12 Average computational time for power. 76

6.13 Compilation and key generation time for power. 77

6.14 Average computational time for square root. 78

6.15 Compilation and key generation time for square root. 78

6.16 comparison of evaluation metrics logistic regression 78

6.17 comparison of evaluation metrics linear support vector regression. 79

6.18 comparison of evaluation metrics linear SVM classification. 80

7.1 Mathematical operation results. 81

7.2 Machine learning algorithms results. 81

Chapter 1

Introduction

In today's digital era, technology has become an integral part of our daily routine which includes communication, finance, healthcare, and commerce. In this scenario, cybersecurity and encryption are crucial to protect the privacy and security of our data and ensure its integrity. With the exponential increase in cyber threats nowadays, it is imperative to develop and implement strong cybersecurity strategies.

Cryptography, in particular, plays a critical role in safeguarding sensitive information and enabling secure communication on a large scale. It ensures the confidentiality, authenticity, and integrity of data exchanged between individuals and financial transactions conducted online. Cybersecurity and cryptography not only protect critical digital infrastructure but also the privacy and security of individual users, contributing significantly to building trust in our constantly evolving digital world.

Encryption typically serves as a crucial tool for protecting the confidentiality of sensitive data. However, traditional encryption methods and nearly all network security protocols are based on the exchange of keys between peers involved in the communication and require data decryption before any operations can be performed, compromising user privacy in scenarios like utilizing cloud services for file storage, sharing, and collaboration.

Additionally, concerns emerge regarding the retention of identifiable user elements by untrusted servers, providers, and major cloud operators even after users discontinue their service usage, posing significant privacy risks. Ideally, a scheme allowing operations on encrypted data without decryption would offer a perfect solution, preserving privacy while enabling desired computations [1]. This is where homomorphic encryption comes in, a special type of encryption that allows any third party to operate on the encrypted data without the need to decrypt it first and can help reach zero trust[1] security.

The concept of homomorphism in computer science and cryptography is derived from homomorphism in mathematics in which an application between two algebraic structures of the same type preserves the operations defined in them. An example concerning the addition is:

$$f(x_1 + x_2) = f(x_1) + f(x_2) \; : \; \forall x_1, x_2 \in V$$

where f is a generic function such that $f : V \to W$, V and W are two vectorial spaces and x_1 and x_2 are two random element of V.

There are encryption schemes known as homomorphic schemes that allow for one type of operation, or a limited number of operations. If a scheme allows only one type of operation, it is called a partially homomorphic encryption scheme. If it allows only a limited number of operations, it is known as a somewhat homomorphic encryption scheme. Such schemes have been

[1] https://www.ibm.com/topics/homomorphic-encryption

around for quite some time. For example, the Rivest-Shamir-Adelman (RSA) cryptosystem from 1978 is a partially homomorphic scheme over multiplication [2].

Fully Homomorphic Encryption (FHE) is a cryptographic technique that enables complex computation on encrypted data without the need for decryption an unlimited number of times. In 2009, Gentry presented the first FHE scheme in his work [3], which was considered the ultimate goal of cryptography at the time. An FHE scheme is defined by four operations: key generation, encryption, decryption and evaluation. The latter is the most important process since it is the one that permits the execution of the requested operations without the need for the private or public key of the client.

Gentry's early research is based on the mathematical notion of an ideal lattice, offering not just a mere scheme but a functional framework for realizing full homomorphism. Central to his work is the delineation of two essential processes crucial for achieving complete homomorphism: squashing[2] and bootstrapping[3] since one of the most crucial matters in the context of homomorphic encryption is that the format and size of the ciphertext after an evaluation must be preserved to sustained an unlimited number of operations.

FHE is crucial in addressing privacy concerns in various fields, including machine learning, cloud computing, and fog computing. However, to make this technique work, both addition and multiplication must be executed homomorphically because any Boolean (arithmetic) circuit, which is a set of connected gates where the evaluation is performed, is represented by XOR (addition) and AND (multiplication).

Most research efforts have concentrated on public key encryption schemes, while symmetric FHE schemes have received less attention within the scientific community, primarily due to their restricted applicability in the aforementioned domains. In addition, certain proposed symmetric key schemes continue to exhibit security vulnerabilities [4].

Regarding FHE schemes, two taxonomies are defined. The first divides the schemes according to the mathematical concepts that are applied and they are ideal lattice-based FHE, FHE schemes over integers, LWE-based FHE schemes and NTRU-Like FHE schemes. The other instead divides them into four generations each with its own pros and cons and most suitable applications area. In addition, it is very important to mention the security properties of fully homomorphic schemes, distinguishing four main properties which are: semantic security, function privacy, circuit privacy and quantum resistance.

Over the years, many libraries have been published that through APIs enable full homomorphism of which many are open source. In this work, the two most comprehensive, most user-friendly and most widely used which are concrete by Zama and openFHE are analyzed.

This work, comprehensively presents what Fully Homomorphic Encryption (FHE) is from the main definitions to an explanation of the various families and types and their applications and possible use cases.

After having defined the enhancements achievable through the implementation of full homomorphism, a query naturally emerges regarding the computational cost and performance of a homomorphic circuit across various scenarios. This thesis work aims to address this question first by analyzing the time and size of each step required to enable FHE and then the main mathematical operations that are used as the basis for any computation, which are: addition, subtraction, multiplication, division, power elevation and square root. Making the comparison in three different configurations: the one with FHE disabled, the one with FHE enabled, and the one with FHE enabled and parallelization enabled.

As mentioned before, since machine learning is one of the main areas in which FHE can be applied, as an additional test in this work the performance of some classification and regression algorithms was calculated. A comparison of performance was made both in terms of execution and

[2]tweaks to reduce the complexity of the algorithms

[3]tweks to reduce the error or noise in a ciphertext. It applies a re-encryption procedure to a given ciphertext c, resulting in a new and fresh version of it with a smaller error

inference times and by comparing the respective metrics with those computed using scikitlearn[4], a Python library that provides APIs for most machine-learning algorithms.

The purpose of this thesis is to evaluate the practicality of applying and enabling Fully Homomorphic Encryption. As mentioned in Chapter 2 and Chapter 3 at the beginning of the thesis, this cryptographic technique offers significant improvements in terms of data privacy and protection against cyber attacks. It is even considered a viable defence mechanism in the post-quantum era.

However, the results show that the technique is still in its infancy and requires excessive execution times, which results in significant overhead. As a consequence, it is currently challenging to identify areas in which FHE can be applied efficiently nowadays. Nevertheless, with further research and optimization, it is possible to apply this technique more efficiently in the future.

In particular, this thesis work consists of the following chapters. First, it is important to mention that in Appendix B are clarified some important mathematical concepts to understand what fully homomorphic encryption is based on. Chapter 2 defines some basic concepts behind homomorphism, then discusses the various types of FHE schemes their security properties and a detailed analysis of some possible use cases where this technology could be applied and where it would bring significant added value. Chapter 3 deals with a detailed investigation of today's available libraries and compilers denoting their characteristics. In addition, concrete and openFHE are analyzed in more detail and a comparison is made between the two, based on their features and what they offer, and how user-friendly they are from a developer's point of view. Chapter 4 addresses and introduces the basic concepts of machine learning and contains a description of some of the most widely used algorithms in classification and regression. In addition, an example of using the concrete-ml library is given to introduce its capabilities. Chapter 5 defines the context in which the tests were analyzed and conducted, defining the hardware structure and introducing what software was used. Furthermore, the strategies applied to conduct tests related to mathematical operations and evaluation metrics to compare machine learning algorithms were explained. Therefore, Chapter 6 exhibits the results and discusses them. Finally, Chapter 7 shows the conclusions that have been reached and suggests some future development.

[4]`https://scikit-learn.org/stable/`

Chapter 2

Homomorphic Encryption

The concept of Fully Homomorphic Encryption (FHE), originally referred to as privacy homomorphism, was first presented by Rivest et al. [2] in 1978. For over three decades, this concept was believed to be the ultimate goal of cryptography, until 2009, when Gentry revolutionized the field by presenting the first fully homomorphic encryption scheme in his PhD thesis [3]. Homomorphic encryption allows operations on plaintext without the need for decryption, hence it enables *arbitrary complex computation* on encrypted data. In other words, a series of operations can be carried out on ciphertexts, and these operations are reflected as additions and multiplications on the target plaintexts.

This has tremendous application potential since it allows privacy-preserving data processing, which can be adopted in fields such as machine learning, cloud computing, or in the different data processing layers of new generational networks [4].

The majority of research efforts for the FHE scheme focused on public key encryption schemes. Symmetric FHE schemes have gained less popularity among the scientific community, due to their limited applicability to cloud computing [4]. Furthermore, some proposed symmetric key schemes still suffer from security vulnerabilities [4].

To achieve the goal of FHE, the direct manipulation of the ciphertext, both addition and multiplication operations must be executed homomorphically since these two operations constitute a functionally complete set over finite rings. In particular, any Boolean(arithmetic) circuit can be represented using only XOR (Addition) and AND (multiplication) gate [4].

2.1 Definitions

Definition 1 *An encryption scheme is called homomorphic over an operation "$\star$" if it supports the following equation*

$$Enc(m_1) \star Enc(m_2) = Enc(m_1 \star m_2),\ \forall m_1, m_2 \in M\ [1]$$

where $Enc()$ is the encryption algorithm and M is the set of all possible messages.

Definition 2 *Given two ciphertexts $Enc(m_1)$ and $Enc(m_2)$, where m_1 and m_2 are the two plaintexts, it is possible to get:*

$$Dec(Enc(m_1) \star Enc(m_2)) = m_1 \star m_2$$

where "$\star$" is the sum or product operation, and $Dec()$ is the decryption algorithm.

A Homomorphic Encryption (HE) scheme is mainly characterized by four operations, *keyGen*, *Enc*, *Dec* and *Eval*, which are polynomial-time (PPT) algorithms. According to [4] they are defined as:

1. the public key-generation algorithm $KeyGen$ takes as input the security parameter λ and outputs the secret key sk, the public key pk, and the (public) evaluation key evk, which is needed to perform homomorphic operations over the ciphertext;

2. the public encryption algorithm Enc takes as input the public key pk and a message m from the message space. Subsequently, it outputs a ciphertext c;

3. the decryption algorithm Dec takes as input the secret key sk and a ciphertext c. Next, it outputs a plaintext m. The algorithm provides as output $\perp$ if the decryption algorithm cannot successfully recover the encrypted message;

4. the evaluation algorithm $Eval$ takes as input the evaluation key evk, a function f, and a t-uple of ciphertexts $(c_1, \dots, c_t)$. It outputs a ciphertext c_f such that it decrypts to the result of the evaluation of $(m_1, \dots, m_t)$ over f, i.e., $c_f = Eval_{evk}(f, (c_1, \dots, c_t))$ and $Dec_{sk}(cf) = f(m1, \dots, m_t)$. Note that the ciphertext c_f and $c = Enc_{pk}(f(m_1, \dots, m_t))$ are analogous in the sense that they decrypt to the same plaintext but different in their construction.

A simple cloud application scenario is illustrated in Figure 2.1, where the client (C) starts encrypting its private data and then sends it to the server (S). The client asks to compute a function $f()$ over its data, and it sends the query and the function to S. The latter computes the homomorphic operation over the encrypted data using the $Eval$ function and returns the encrypted result. Ultimately, the client can recover the data with its secret key and retrieve $f(m)$. It is significant to remark that for what concerns server-side operations, the server doesn't require the secret key of the client and is able to perform addition and multiplication over the encrypted data.

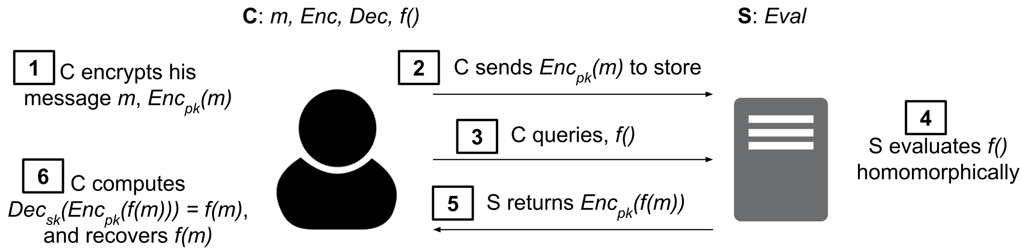

Figure 2.1. Client/Server HE scenario.

The most crucial matter in the context of homomorphic encryption is that the format and size of the ciphertext after an evaluation process must be preserved to sustain an unlimited number of operations. Commonly, any encryption procedure in the context of homomorphism adds a random element, which is called noise or error. This is mainly due to the cryptographic methods on which the homomorphic encryption is based. FHE has an intrinsic feature that can cause incorrect computations: every homomorphic operation increases the noise in the system. This noise can eventually lead to errors in ciphertext decryption after a certain number of additions or multiplications [4]. Behind these assumptions, it is possible to define the essential characteristics of a homomorphic scheme:

1. The maximum degree of a function that a scheme can evaluate correctly. More precisely, a scheme is considered $\mathcal{F}-$*homomorphic* if it can accurately evaluate any function f in $\mathcal{F}$ [4];

2. the length increase of the ciphertext after each homomorphic operation. Specifically, how much the ciphertext bit size rises after each evaluation.

Depending on the previous concepts, the following section explores the distinct definitions of homomorphic encryption schemes.

2.2 FHE Schemes

Until Gentry's turning point, all the attempts permitted just one type or, at most, a limited number of operations on the encrypted data. Indeed, it is possible to define three types of schemes with respect to the number of allowed operations [1]:

1. *Partially Homomorphic Encryption* (PHE), allows only one type of operation with an unlimited number of times;
2. *Somewhat Homomorphic Encryption* (SWHE) allows some types of operations a limited number of times;
3. *Fully Homomorphic Encryption* (FHE) allows an unlimited number of operations for an unlimited number of times.

Gentry's FHE scheme, based on the ideal lattice, is a power framework for achieving FHE, but it is not practically a realistic scheme due to its high level of computational complexity, as well as the size of its key and the amount of noise it generates. Consequently, a lot of follow-up progress and new schemes were presented in the following years.

According to T.V.T. Doan et al. [5], Table 2.1 illustrates the most widely used HE schemes with their timeline and application scenarios.

Scheme	***Year***	***Application***
RSA	1978	Banking and credit card transaction
ElGamal	1985	In Hybrid Systems
Paillier	1999	E-voting
BGN	2005	A Novel IoT Data Protection Based on BGN Cryptosystem
BGV	2011	For the security of integer polynomials
BFV	2012	A fast oblivious linear evaluation (OLE) protocol
FHEW	2014	Fast Fully Homomorphic Encryption over Torus
CKKS	2016	Homomorphic Machine Learning Big Data Pipeline for the Financial Services Sector
TFHE	2020	An Homomorphic LWE-based E-voting Scheme

Table 2.1. Timeline of the most important HE schemes and their application.

2.2.1 Partially Homomorphic Encryption Schemes

In the literature, there are many PHE examples, this section will list and, briefly, analyse the most significant:

1. *RSA* is introduced by Rivest et al. [6], and it is the first practicable achievement of the public key cryptosystem. The security of RSA is based on the hardness of the factoring problem of the product of two large prime numbers. RSA is only homomorphic over multiplication as explained below:

- *KeyGen(λ)*: two large prime numbers p and q are randomly chosen and a modulus $N = pq$ and $\Phi(N) = (p-1)(q-1)$ are computed. A secret large integer d is selected such that $gcxd(d, \Phi(N)) = 1$ and a public exponent e is computed by calculating the multiplicative inverse of d, $ed = 1 \ mod \Phi(N)$. Finally, $pk = (e, N)$ and $sk = (d, p, q)$;
- *Enc(pk,m $\in \mathbb{Z}_N$)*: a plaintext p is an integer between 0 and $N-1$. The corresponding ciphertext $c = Enc(p) = m^e(modN)$;
- *Dec(sk, c)*: a plaintext p can be recovered from a ciphertext c, $p = c^d(modN)$;
- *Homomorphic property over Mult(c_1, c_2)* [5]:

  $$c_1c_2 = Enc(p_1)Enc(p_2) = [m_1^e(modN)][m_2^e(modN)] = (m_1m_2)^e(modN) = Enc(m_1m_2);$$

2. *Goldwasser-Micali* is the first probabilistic public key encryption scheme, it is based on the *hardness of quadratic residuosity problem*. A quadric residue modulo n for a number a is when exists an integer x such that $x^2 \equiv a \ mod(n)$. The problem establishes whether a certain number q is quadratic modulo n or not. GM is homomorphic over addition;

3. *El-Gamal* improved the original Diffie-Hellman Key Exchange algorithm, it is based on a hybrid approach to encrypt the secret key of a symmetric encryption system. It doesn't support addition over ciphertext, hence is only homomorphic over multiplication;

4. *Benaloh* is an extension of GM cryptosystems, it permits encrypting the message as a block instead of as a stream of bits. The Benaloh algorithm is homomorphic over addition;

5. *Paillier* is a novel probabilistic encryption scheme based on the *composite residuosity problem*. It interrogates whether there exists an integer x such that $x^n \equiv a \ mod(n^2)$. It is homomorphic over addition.

To better analyse the PHE schemes and their properties concerning homomorphism, a tabular representation is used in Table 2.2. The table presents various PHE schemes and their specific characteristics in terms of homomorphism. This examination aims to clarify the distinct approaches employed in PHE schemes.

Scheme	*Year*	*Addition*	*Multiplication*
RSA	1978	-	✓
Goldwasser-Micali	1982	✓	-
ElGamal	1985	-	✓
Benaloh	1994	✓	-
Paillier	1999	✓	-

Table 2.2. Homorphic Properties of most used PHE schemes.

2.2.2 Somewhat Homomorphic Encryption Schemes

In this section, the focus is on the most significant Somewhat Homomorphic Encryption schemes, which were used as an intermediate stage to the first FHE scheme.

The literature contains numerous attempts to develop schemes for SWHE, and several efficient variants have been suggested over the years. An essential consideration is the assessment of these schemes in terms of potential vulnerabilities, including their security and feasibility. Table 2.3 presents the critical feature of the most renowned SWHE schemes before Gentry's work.

It is worth mentioning further *BGN* scheme which is, according to A. Acar et al. [1], one of the most significant steps toward an FHE. It was introduced by Boneh-Goh-Nissim (BGN) in 2005 in Boneh et al. [7]. It is based on the *Subgroup decision Problem*, which consists of determines whether an element is a member of a subgroup G_p of a group G of composite order $n = pq$, where p and q are distinct primes. It supports an arbitrary number of additions and one multiplication by keeping the ciphertext size constant.

Scheme	*Year*	*Evaluation Size*	*Ciphertext Size*
Yao	1982	arbitrary	grows at least linearly
SYY	1999	poly-many AND & one OR/NOT	grows exponentially
BGN	2005	unlimited add & 1 mult	constant
IP	2007	arbitrary	doesn't depend on the size of the function

Table 2.3. Comparison of Well-Known Pre-Gentry Somewhat Fully Homomorphic Encryption (SWHE) Schemes.

2.2.3 Fully Homomorphic Encryption Schemes

In the preceding chapter 2.1, the fundamental concepts necessary to delve into a detailed definition of Fully Homomorphic Encryption (FHE) schemes have been defined. According to A. Acar et al. [1], an encryption scheme is called an FHE scheme if it allows an unlimited number of evaluation operations on the encrypted data and the resulting output is within the ciphertext space. Gentry's proposal gives not only an FHE scheme but also a general framework to obtain an FHE scheme. Consequently, many researchers have tried to design a secure and practical FHE scheme after Gentry's outcome. Although his idea based on the ideal lattice showed great promise, it also faced certain challenges and limitations in terms of computational cost and applicability in real life, and its advanced mathematical notions make it complex and hard to implement. As a consequence, a multitude of new schemes and optimizations have emerged to address the aforementioned limitations. The security of these new approaches in achieving FHE schemes is predominantly rooted in the computational hardness of lattice-based problems. It is possible to categorize the FHE schemes under four main FHE families:

2.2.4 Ideal Lattice-based FHE Schemes

Gentry's initial work began with a Somewhat Homomorphic Encryption (SWHE) scheme based on ideal lattices. This scheme enabled the homomorphic evaluation of ciphertexts for a limited number of operations. Nevertheless, beyond a certain threshold, the decryption algorithm failed to accurately retrieve the original message from the ciphertext due to the accumulation of noise. To manage this limitation, Gentry developed innovative methods known as *squashing* and *bootstrapping*. By combing repeatedly these two techniques Gentry's approach enabled the execution of an arbitrary number of homomorphic operations while managing and reducing the noise.

To gain a better understanding of the concept of *noise*, according to A. Acar et al. [1], let's consider an encryption scheme over integers. In this scenario, the encryption of the bit b is the ciphertext $c = b + 2r + kp$, where the key p is an odd integer and r is a random number from the range $(-n/2, n/2)$ and k is an integer. The decryption works as follows: $b = (c \, mod \, p)$ and $(c \, mod \, p)$ is called *noise parameter*. If the noise parameter exceeds $|p/2|$, the decyption fails since $(c \, mod \, p)$ is not equal to $b + 2r$ anymore.

To achieve full homomorphism, Gentry presented the bootstrapping technique. However, the bootstrapping process can only be applied to bootstrappable ciphertexts, which are characterized by being noisy and having a small circuit depth. The circuit depth refers to the maximum number of operations that can be executed. To prepare the ciphertexts for bootstrapping, the circuit depth is initially reduced using the squashing technique.

- *Squashing*: As mentioned above, Gentry's bootstrapping technique is permitted solely for small depth decryption algorithms. Thus, he employed some tweaks to decrease the complexity of the algorithms. This method is called *squashing* and operates as follows according to A. Acar et al. [1]: First choose a set of vectors, whose sum equals the multiplicative inverse of the secret key $((B_J^{sk})^{-1})$. If the ciphertext is multiplied by the elements of this set,

the polynomial degree of the circuit is reduced to the level that the scheme can handle. The ciphertext is now *bootstrappable*. Anyhow, the hardness of the recovering of the secret key is now based on the assumption of the *Sparse Subset Sum Problem* (SSSP) [1], which adds another premise to the provable security of the scheme;

- *Bootstrapping*: As briefly explained earlier, it is a technique used to reduce the error or noise in a ciphertext. It involves a re-encryption procedure applied to a given ciphertext c, resulting in a new and fresh version of it, for example by encrypting it again under another key getting a new ciphertext, for the same initial plaintext, but with a smaller error. Marcolla et al. [4] propose that an explanation is achievable by initially considering a somewhat homomorphic encryption scheme, two pairs of keys (sk_1, pk_1) and (sk_2, pk_2), the encryption algorithm Enc, and the ciphertext $c = Enc_{pk_1}(m)$ that encrypt m under pk_1. The refreshing procedure for c works as follows:

  1. Encrypting the secret key sk_1 under pk_2: $Enc_{pk_2}(sk_1) = \overline{sk_1}$. The encryption of sk_1 may generate multiple ciphertexts [4];
  2. Encrypting the ciphertext c under $pk_2 : Enc_{pk_2}(c) = \overline{c}$.
  3. Decrypting homomorphically the new ciphertext employing the encrypted secret key: $Dec_{\overline{sk_1}}(\overline{c})$. In this way, an encryption of the same message under the second public key $Enc_{pk_2}(m)$ is obtained.

$$Eval_{evk}(Dec, \overline{c}, \overline{sk_1}) =$$
$$Eval_{evk}(Dec, Enc_{pk_2}(c), Enc_{pk_2}(sk_1)) =$$
$$\widehat{Enc}_{pk_2}(Dec_{sk_1}(c)) = \widehat{Enc}_{pk_2}(m)$$

  The value $\widehat{Enc}_{pk_2}(m)$ is equivalent to $Enc_{pk_2}(m)$ in a sense, it decrypts both m, i.e.,

$$Dec_{sk2}(\widehat{Enc}_{pk_2}(m)) = Dec_{sk2}(Enc_{pk_2}(m)) = m$$

As depicted in Figure 2.2, the bootstrapping technique consists of a re-encryption procedure where two different public and secret keys are generated and all the steps explained above are executed on the noisy ciphertext to get a fresh version, related to the identical plaintext. A scheme is named *bootstrappable* if it can evaluate its own decryption algorithm. Initially, the process begins with homomorphic decryption, which effectively eliminates the noise from the ciphertext. Nevertheless, as a result, a small amount of fresh noise is introduced to the ciphertext. Consequently, the ciphertext appears as if it has just been encrypted anew. This fresh ciphertext now permits additional homomorphic operations to be performed until the threshold point is once again reached. Gentry's bootstrapping approach is associated with a significant increase in computational cost, posing a considerable drawback to the practicality of Fully Homomorphic Encryption (FHE). In essence, the process of creating an FHE scheme involves producing a somewhat homomorphic encryption (SWHE) scheme, followed by the development of a squashing method, if necessary, to reduce the circuit depth of the decryption algorithm. Ultimately, the bootstrapping step is performed to obtain fresh ciphertext, finally achieving a fully homomorphic encryption scheme. By applying bootstrapping iteratively, it can compute an unlimited number of operations on the ciphertexts, thus successfully implementing the FHE scheme. However, it's crucial to be aware that each application of bootstrapping incurs a substantial computational overhead, which accumulates with the number of operations performed, making it a significant practical challenge in the adoption of FHE for real-world applications. Researchers continue to work on optimizing bootstrapping and investigating alternative techniques to mitigate this computational cost and improve the practicality of Fully Homomorphic Encryption.

[1] Given a set of integers $A = a_1, a_2, \ldots, a_n$, target integer t, and modulus M in $\mathbb{Z}$, an SSSP requires the finding of a sparse subset whose elements sum up to t modulo M. Similar to the subset sum problem, an SSSP is generally considered to be hard when n is large [8].

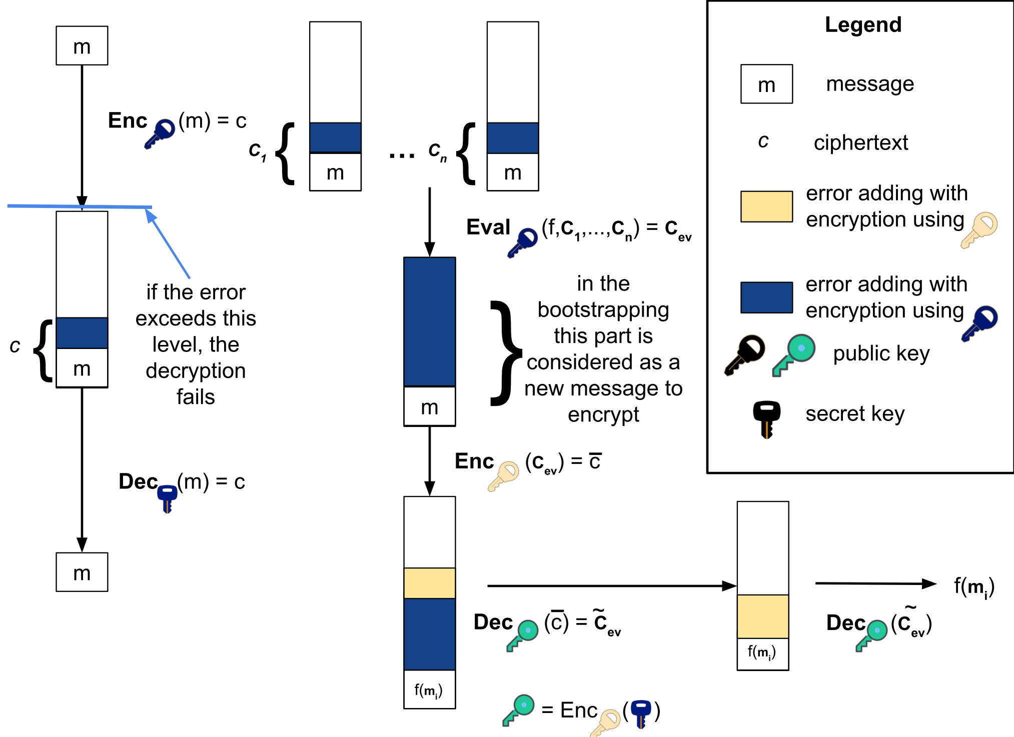

Figure 2.2. Bootstrapping technique.

2.2.5 FHE Schemes over Integers

A year later, in 2010, another SWHE scheme was presented by Van Dijk et al. [9]. This suggested scheme is over integers and its hardness is founded on the *Approximate – Greatest Common Divisor* (AGCD) problems [4], that attempt to recover p from a given set of $x_i = pq_i + r_i$, where p is the key. The scheme's conceptual simplicity, particularly in its symmetric version, is advantageous as it makes it easier to understand and implement. However, this simplicity comes at a cost in efficiency, therefore some attempts directly tried to enhance it. It is important to mention an noteworthy variant, which is batch FHE over integers, proposed by Cheon et al. [10], that has the ability to pack multiple ciphertexts into a single one;

2.2.6 LWE-Based FHE Schemes

Learning with Error problem (LWE), as explained in Appendix B, is considered one of the hardest problems to solve in a reasonable time for even post-quantum algorithms. In 2011, Brakerski and Vaikuntanathan employed the bootstrapping technique to suggest two FHE schemes based on LWE and RLWE problems and circular security assumptions. According to Marcolla et al. [4], the two researchers also introduce two methods called *relinearization* and *dimension – modulus reduction*. The first one is needed to reduce the multiplication ciphertext size from almost $n^2/2$ to the standard size $n + 1$. The dimension modulus reduction, instead, transforms a SWHE scheme into an FHE scheme converting a ciphertext c modulo q into another ciphertext c' modulo p, where p is sufficiently smaller than q. The effects produced by this operation are the decrease of the error when comparing the ciphertext before and after a modulus switching in addition to the modulus reduction, but the error level relative to its modulus is actually higher since this technique introduces some noise.

2.2.7 NTRU-Like FHE Schemes

NTRU is a lattice-based encryption scheme, it is significantly more efficient in both hardware and software performances compared to RSA and Goldreich, Gold-wasser, and Halevi (GGH)[2] cryptosystems. However, there were security concerns until 2011, when a study conducted by Stehlè and Steinfield [11] slightly adjusted the scheme to obtain a variant in which security is established on RLWE assumption. A new notion called *multikey* FHE supports computation on ciphertext encrypted under different keys [4]. The scheme's security relies on circular security, the RLWE problem, and the *decisional small polynomial ratio* (DSPR) problem. To achieve a secure NTRU-based scheme, it's necessary to substantially increase the parameters' sizes and it led to a significant decrease in efficiency compared to their earlier counterparts. As a result, NTRU-based schemes are no longer in use or supported by any library due to their lowered practicality.

	2nd Generation		3rd Generation	4th Generation
SCHEMES	BGV	B/FV	TFHE	CKKS
PROS / APPLICATIONS	Integer Arithmetic		Bitwise operations	Real Number Arithmetic
	efficient packing (SIMD)		efficient boolean circuits	fast polynomial approx.
	fast escalar multiplication		fast bootstrapping	fast multiplicative inverse
	fast linear functions		fast number comparison	efficient DFT
	efficient leveled design			efficient logistic regression
				efficient packing (SIMD)
				leveled design
CONS	slow bootstrapping		no support for batching	slow bootstrapping
	slow non-linear functions			slow non-linear functions

Figure 2.3. Generation of FHE Schemes pros/cons.

2.2.8 Alternative taxonomy

Marcolla et al. [4] suggest another division based on four generations:

1. *first Generation*: based on Ideal Lattices and on AGCD Problem;
2. *second Generation*: based on LWE and RLWE Problem and on NTRU. It includes:
   - *BGV* was invented by Brakerski, Gentry and Vaikutanathan in 2011. It is a leveled FHE which means that the parameter of the scheme depends polynomially on the maximum number of multiplications that can be executed [5];

[2]Goldreich, Gold-wasser, and Halevi (GGH) proposed an important type of PKE scheme, whose hardness is based on the *lattice reduction problems* [1]

- *B/FV* was invented by J. Fan and F.Vercauteren in 2012. They simplified the scheme and improved efficiency with a modulus switching trick, while also simplifying the bootstrapping analysis [5].

3. *third Generation*: based on LWE and RLWE Problem, with a different approach in doing homomorphic operations, presenting the *approximate eigenvector method*, that eliminates the need for key and modulus switching methods, reducing the error growth caused by homomorphic multiplications to a minimal polynomial factor.

   - *TFHE* was introduced in 2014 and provides two efficient bootstrapping methods called *AP* and *GINX* [5].

4. *fourth Generation*: based on LWE and RLWE. In 2017 was introduced a technique to construct a levelled homomorphic encryption scheme for approximate arithmetic numbers and include an open-source library(HEAAN) implementing the scheme(CKKS).

   - *CKKS* was invented in 2016 and allows approximate operations on ciphertext with vectors of real and complex values [5].

Based on the current state of knowledge, the most practical and widely adopted cryptographic schemes are BGV, B/FV, TFHE, and CKKS. The second-generation schemes, such as BGV and B/FV, are well-suited for working with finite fields in modular exact arithmetic. They maintain efficient packing techniques that enable SIMD (Single Instruction Multiple Data) instructions to perform computations over arrays of integers (batching). Therefore, these schemes excel when processing large arrays of numbers simultaneously. However, second-generation schemes may not be the best option for circuits that require bootstrapping (e.g., circuits with large multiplicative depth) or the implementation of nonlinear functions. In such cases, third-generation schemes like TFHE are more suitable, especially for bitwise operations represented as Boolean circuits. However, TFHE's limitation lies in the lack of support for CRT (Chinese Remainder Theorem) packing (batching), making it potentially outperformed by previous approaches when processing considerable amounts of data simultaneously. For arithmetic involving real numbers, the fourth-generation scheme, CKKS, is the optimal alternative due to its ability to handle real number arithmetic efficiently. According to Figure 2.3, each generation has its primary application. It's important to mention that while TFHE boasts the fastest bootstrapping procedure, the batching capability of both second and fourth-generation schemes enables parallel bootstrapping of multiple plaintexts. Specifically, CKKS exhibits a more efficient amortized bootstrapping than TFHE in particular cases. Nevertheless, this advantage does not extend to second-generation schemes due to their significantly lower number of available slots compared to CKKS. For instance, in BGV, the number of slots is only around 1000, whereas CKKS offers approximately 215 slots. This results in CKKS bootstrapping being more than one order of magnitude faster than BGV bootstrapping.

2.3 Security Properties

A homomorphic encryption scheme must have semantic security as an essential condition. Additionally, it can optionally provide function or circuit privacy [4]. Semantic security is formally defined by indistinguishability under chosen-plaintext attack (IND-CPA[3] security). In this scenario, an attacker can obtain encryptions of arbitrary plaintexts, but it cannot decrypt arbitrary ciphertexts. It's important to note that the encryption algorithm and the encryption key (pk) are publicly known, while the decryption key (sk) remains secret. The other essential security model to prove the security of an encryption algorithm is IND-CCA[4] standing for indistinguishability under chosen ciphertext attacks, in this context, two crucial concepts are considered: IND-CCA1[5]

[3] indistinguishability under chosen plaintext attack means that an adversary cannot distinguish between the encryption of two different plaintexts even when given the ciphertexts of those plaintexts. [12]

[4] indistinguishability under chosen ciphertext attack indicates that the adversary can actively interact with the oracles, but it is unable to distinguish the encryption of two different plaintexts. [13]

[5] indistinguishability under non-adaptive chosen ciphertext attacks the adversary is authorised to query the oracle only until it receives the challenge ciphertext. [13]

and IND-CCA2[6]. The former relates to indistinguishability under non-adaptive chosen ciphertext attacks, whereas the latter deals with the adaptive chosen ciphertext attacks.

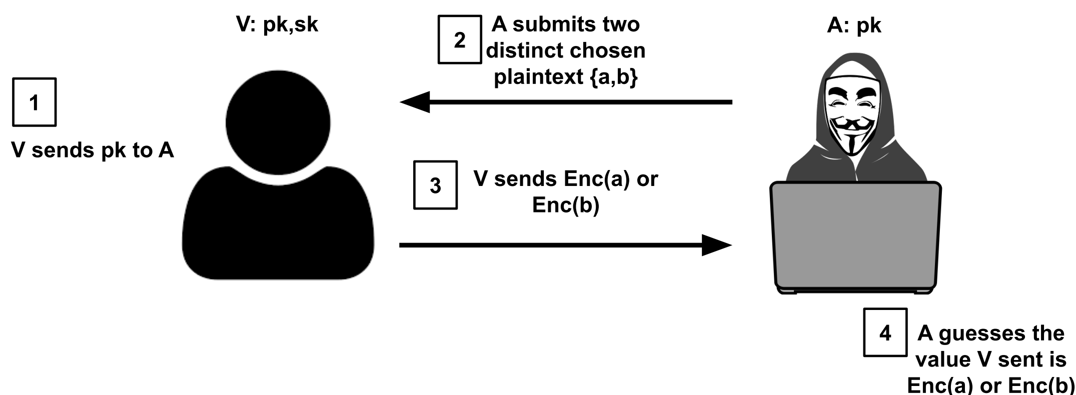

Figure 2.4. IND-CPA security.

As illustrated in Figure 2.4, IND-CPA is modelled as a scenario involving an adversary (A) and a verifier (V). In general, after generating the public key (*pk*), secret key (*sk*), and other security parameters for an encryption system, V sends the public key to A. From this point, A is free to perform any computations using *pk* and then selects two different plaintexts, a and b, and sends them to V. V randomly encrypts either a or b and sends the result back to A as a challenge. A's task is to determine whether the received value is the encryption of a or b. The security of the system is evaluated in terms of IND-CPA, consequently, it ensures that no adversary can output the correct value with a significantly better probability than 1/2.

The definition of IND-CCA is similar to IND-CPA, but in both IND-CCA1 and IND-CCA2, the attacker can request the decryption of any ciphertexts, except for the challenge sent by the verifier. In both cases, the attacker is allowed to make queries to the decryption oracle to decrypt arbitrary ciphertexts before step 3 in Figure 2.4 when the verifier sends the challenge to the adversary. However, after step 3, the adversary is not allowed to make further calls to the decryption device in IND-CCA1, while it is permitted in IND-CCA2. The security under IND-CCA2 implies the security under IND-CCA1, and the security under IND-CCA1 also implies the security under IND-CPA. In other words, an encryption scheme that is IND-CCA2 secure is both IND-CCA1 and IND-CPA secure [5]. It is worth noting that IND-CPA security is only achievable if the encryption scheme randomizes the ciphertext [4].

Finally, an encryption scheme that is secure against adversaries who observe encryption of the scheme's secret key under its public key is called *circular security* [4]. To achieve this, the current constructions of Fully Homomorphic Encryption (FHE) schemes needed the encryption of

[6]indistinguishability under adaptive chosen ciphertext attacks the adversary may continue to query the oracle even after it has obtained a challenge ciphertext. [13]

the secret key to be bootstrappable, resulting in the requirement for circular security in all known FHE constructions. Consequently, IND-CPA security must be ensured under circular security. While some FHE schemes lack proof of IND-CPA under circular security, it is often adopted as an additional assumption with the underlying security premises of the scheme. Optionally, the homomorphic encryption scheme can be *function private*, meaning that a ciphertext that has been homomorphically evaluated over a function f should not disclose any information about f, except for the outputs corresponding to the queried inputs [4]. Function privacy serves as a more relaxed version of the original *circuit privacy* concept, where the evaluated ciphertext should be statistically indistinguishable from a fresh ciphertext. It is important to note that for a scheme to be either circuit or function private, this property must hold even against an adversary possessing the secret key and capable of decrypting any ciphertext [4].

In addition, it is important to mention that the concept of FHE being quantum-resistant is widely acknowledged, but there are currently no definitive empirical studies that conclusively confirm its resilience against quantum attacks. The assertion that FHE falls within the realm of post-quantum encryption methods is based on the underlying principles of lattice cryptography and the assumption that quantum computers do not significantly simplify their decryption complexity. Nonetheless, ongoing research and analysis are necessary to validate these premises.

2.4 Use cases

In an increasingly interconnected and data-driven world, the security and privacy of sensitive information have become an important concern. Decrypting data prior to processing exposes it to potential security vulnerabilities. As an innovative solution to this challenge, Fully Homomorphic Encryption (FHE) has emerged as a new cryptographic technology that promises to revolutionize secure computing.

This chapter delves into practical use cases and potential applications of Fully Homomorphic Encryption schemes and cryptosystems, ranging from secure Cloud computing and data sharing to preserving privacy in artificial intelligence and machine learning. As researchers and industries continue to invest substantial efforts in FHE, it becomes increasingly crucial to understand the underlying mechanics and evaluate its feasibility and performance for specific use cases.

2.4.1 FHE for machine learning

This section offers an extensive overview of the combined subject concerning privacy-preserving techniques and machine learning (ML), an area that has collected considerable attention in recent research efforts. ML involves algorithms and computing systems designed to create models that learn and incorporate structural knowledge from input datasets. However, a notable limitation to its widespread adoption lies in the necessity for access to significant amounts of data to achieve high accuracy rates, and then raising concerns about data privacy and security. Fully Homomorphic Encryption (FHE) provides a solution by allowing arithmetic evaluations of encrypted real number data. This permits the development of $privacy-preserving\ ML$ (PPML) training algorithms, offering a potential way to address the aforementioned privacy and security issues. Moreover, FHE plays a crucial role in distributed ML, supporting confidential secure computing scenarios. In Figure 2.5 is illustrated as an example of a potential PPML model.

According to Marcolla et al. [4], there are different interesting ML fields to investigate and discuss their potential integration with FHE schemes.

- *Support Vector Machines (SVMs)* are learning models that analyze data for classification and regression analysis, and several privacy-preserving SVM computing techniques have been proposed in the literature. For instance, Laur et al. [14] introduced a privacy-preserving approach that utilized additively homomorphic encryption and secret sharing or secure multiparty computation (MPC) protocols for both SVM training and classification. Park et al. [15] devised an algorithm for the SVM training phase, leveraging homomorphic encryption to ensure efficient operations within an encrypted domain.

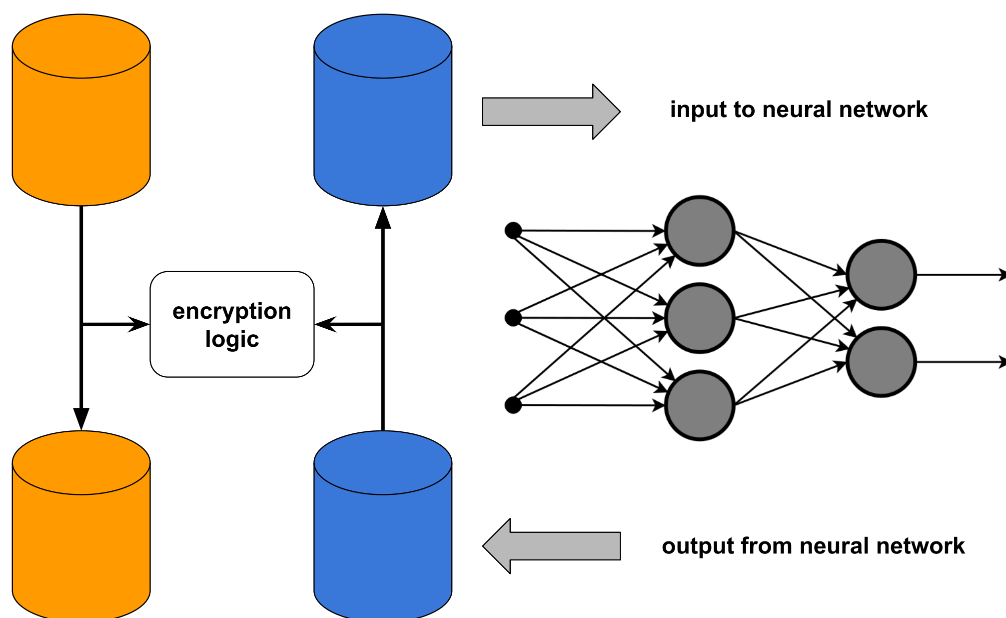

Figure 2.5. PPML model example.

Rahulamathavan et al. [16] presented a classification protocol for two-class and multiclass scenarios using SVMs, exploiting the power of Paillier's cryptosystem and secure two-party computation, where the client and server parties each held a share of the secret. In a more practical implementation, Makri et al. [17] introduced EPIC, an image classification system that employed an SVM computing scheme along with transfer learning-based feature extraction techniques [4];

- *Neural Networks and Other Machine Learning Models.* A neural network is an approach in artificial intelligence that teaches computers to process data. It is a type of machine learning process that uses interconnected nodes or neurons in a layered structure that elaborates relationships between high-dimensional input data. According to Marcolla et al. [4] different methods have been developed. For example, some employ training algorithms expressed as low-degree polynomials to train over encrypted data using FHE, but with limited accuracy. Others utilize high-performance ridge regression systems with homomorphic encoding (additively homomorphic encryption) and garbled circuits, evaluating them on extensive datasets. Mohassel and Zhang presented protocols for Privacy-Preserving Machine Learning (PPML). Aslett et al. demonstrated accurate training of ML models using homomorphic encryption [4]. Several studies explore the application of homomorphic encryption in various contexts, including hardware architecture, collaborative learning, CryptoNets, deep learning models, and statistical techniques toolboxes. They aim to enhance existing protocols by incorporating HE schemes, such as Pailler BGV or CKKS, for full homomorphism support.

In addition to research efforts, various commercial products are emerging to address real-world challenges across different industries. Zama's open-source technology proposes a solution that enables Machine Learning (ML) models, regardless of their underlying architecture or training method, to perform inference on encrypted user data using homomorphic encryption. The

potential applications of this technology extend to diverse fields such as medicine, image classification, autonomous environments, and data processing for smart cities. Intel and Ant Group have cooperated on an initiative to develop Privacy-Preserving Machine Learning using Intel's Software Guard Extensions (SGXs) and Ant Group's memory-safe and multiprocess library operating system for Intel SGX [4]. They leverage cryptographic techniques like homomorphic encryption and differential privacy in their solution. Duality Technologies is a company that offers privacy-preserving data collaboration platforms based on homomorphic encryption. Their scope is the advanced usage of FHE by developing a hardware accelerator for computations. These commercial initiatives demonstrate the growing interest and investment in privacy-preserving technologies, particularly in the context of secure data collaboration and confidential data processing in various domains.

Anyway, several issues remain, they continue to face challenges of extensive computational complexity, limited efficiencies, and inadequate applicability in real-world situations [4].

2.4.2 HE in Fog Computing for IoT

Fog computing is a decentralized computing architecture that extends cloud computing capabilities to the edge of the network, closer to where data is generated and consumed. In traditional cloud computing, data is sent to a centralized data centre, which processes and stores the information on servers located in remote nodes. While cloud computing offers immense scalability and processing power, it also introduces some limitations, especially in scenarios where low latency, real-time processing, and reduced data transmission costs are required. Fog computing, also known as edge computing, aims to overcome these issues by pushing computing resources, storage, and services closer to the edge of the network, where data are preprocessed. This proximity enables faster processing and real-time analysis, as data doesn't have to travel long distances to reach the cloud and back.

Key features include [4]:

- *low Latency*;
- *bandwidth conservation*, reducing the amount of data that needs to be transmitted to the cloud;
- *better scalability and resource utilization*;
- *security*, by processing sensitive data locally;
- *offline operation*, it can work even when the connection to the cloud is lost;
- *data processing* is event-driven (triggered by the device) and performed packet by packet.

Fog computing is well-suited for a wide range of applications, including smart cities, Internet of Things (IoT) devices, industrial IoT, healthcare monitoring, and real-time analytics.

Indeed, while Homomorphic Encryption (HE) holds significant potential for enhancing privacy in the IoT domain, it faces challenges in practical implementation. One major obstacle is its computational complexity, which can be resource-intensive and demand substantial processing power. IoT devices typically have limited hardware capabilities and computational constraints, making it difficult to perform the required encryption and decryption operations efficiently. Another challenge is ciphertext expansion, where the encrypted data becomes significantly larger than the corresponding plaintext. This expansion results in increased communication overhead, leading to higher data transmission costs and potentially straining the limited bandwidth available in current IoT communication standards. There are some limitations with solely using homomorphic encryption (HE) or symmetric key encryption (SKE), so researchers have been exploring hybrid protocols that combine both. This approach is known as hybrid homomorphic encryption (HHE), as shown in Figure 2.6. In HHE, an IoT device first encrypts data using an SKE scheme with a randomly generated key [4]. Then, the device encrypts this key with an HE scheme using the data collector's public key. SKE schemes are less complex and do not lead to ciphertext expansion.

Intermediate fog nodes can homomorphically evaluate the decryption circuit of the SKE scheme, which allows them to convert SK-encrypted data into HE-encrypted data. This process enables the data to be processed homomorphically [4].

One benefit is that the complexity and ciphertext expansion are shifted from IoT devices to fog nodes. The IoT device only needs to encrypt a short key homomorphically. Many studies have suggested AES as an SKE scheme because modern chips in IoT devices have hardware acceleration for AES [4]. However, the AES decryption circuit has a minimum multiplicative depth of 40, which raises the complexity of fog nodes.

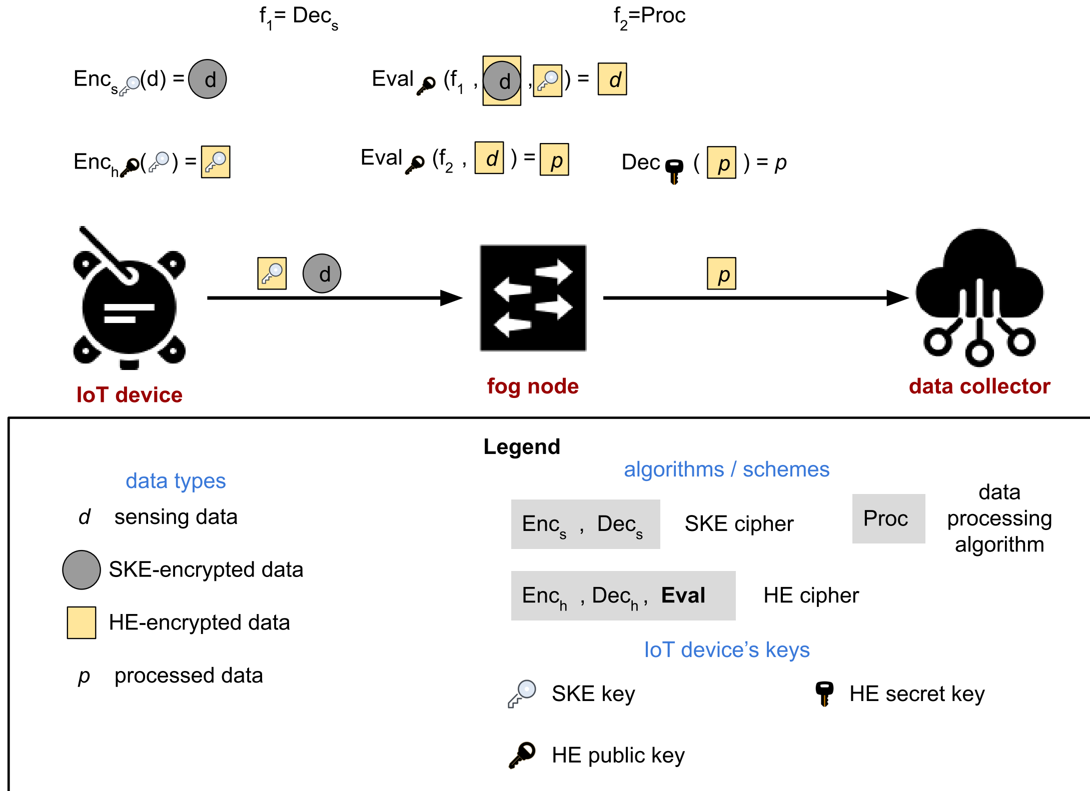

Figure 2.6. Hybrid FHE applied to fog computing IoT.

Some research suggests using public key encryption (PKE) instead of symmetric encryption, but it has a higher level of complexity and ciphertext expansion [4]. It is important to note that hybrid protocols can be effective when encrypting data with the receiver's public key (in the case of a data collector in a smart city scenario). However, if an IoT device encrypts data with its own public key and the communication is bidirectional (with the IoT device receiving encrypted processed data), the problem of ciphertext expansion cannot be avoided [4].

2.4.3 HE in cloud computing

Homomorphic encryption has the potential to be a crucial component for both fog computing and cloud computing. Fog computing is a decentralized infrastructure, while cloud computing is a centralized system that allows for query-based data processing across multiple application sessions with large datasets. However, some cloud service scenarios impose requirements that make plain HE schemes unsuitable [4]. This chapter explores the potential scenarios in detail, as suggested by Marcolla et al. [4]:

- *Homomorphic Proxy Reencryption.*

Most HE schemes only allow for homomorphic operations on ciphertexts that are encrypted with the same public key. Therefore, when dealing with ciphertexts from different users, they need to be transformed into ciphertexts encrypted with the same key. This process is referred to as *homomorphic proxy re-encryption* (HPRE). The PRE technique allows for the conversion of a ciphertext from one user (the delegator) to a ciphertext of another user (the delegatee) through a proxy. This enables the delegatee to decrypt the delegator's ciphertext without gaining access to the delegator's private key, while the proxy can convert the ciphertext without possessing knowledge of the plaintext or the user's keys. Gentry proposes a simple scheme to achieve HPRE [4], in Figure 2.7:

1. the secret key is encrypted homomorphically using the delegatee's public key;
2. encrypts the data with its own public key. Then, the proxy can evaluate the decryption circuit of the homomorphic scheme to re-encrypt the ciphertext with the delegatee's public key (this technique is equivalent to bootstrapping).

Although this method may not be resistant to weak collision attacks, recent studies utilizing key-switching techniques suggest HPRE schemes that avoid this problem. Despite recent advances, current methods are still vulnerable to strong collusion attacks and may reveal information about the delegator's secret key. Although HPRE is considered CPA secure, it may not provide sufficient protection in some cases;

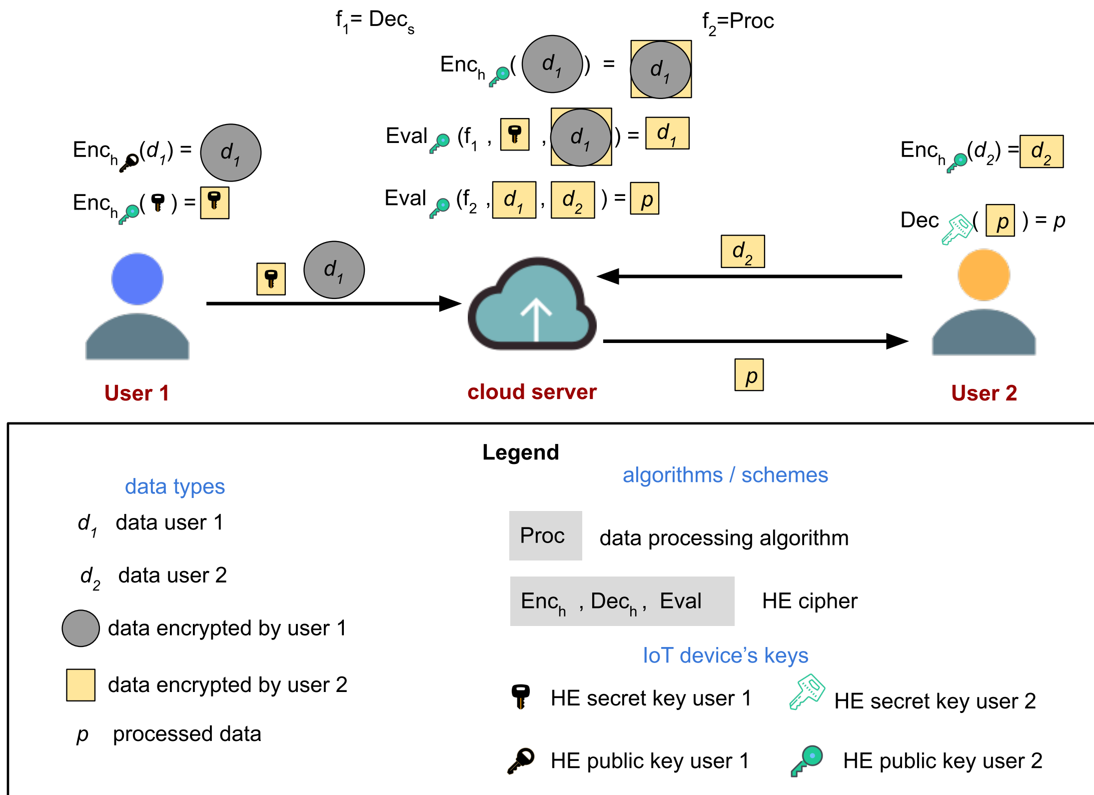

Figure 2.7. HPRE for homomorphic evaluation of multiuser data.

- *Homomorphic Authenticated Encryption*

  Sometimes, privacy isn't enough. In certain situations, it may be necessary to ensure that data has been processed accurately by the cloud service. To do this, the user should be able to confirm that the decrypted data is the outcome of a particular arithmetic circuit over the transmitted encrypted data. This can be achieved with *Homomorphic Authenticated Encryption* (HAE), which is created by combining *Homomorphic Encryption* (HE) and

Homomorphic Authentication (HA) [4], in Figure 2.8. Specifically, the user sends the ciphertexts and attaches homomorphic authenticators in the form of *Homorphic Signatures* (HSs). These signatures can be evaluated homomorphically, similar to ciphertexts, to produce a valid signature for the processed data. If both the HE and HA schemes are CPA secure, then the consequent HAE scheme is CCA1 secure [4]. Subsequent research provided constructions to achieve and enable fully HAE (FHAE). A potentially more efficient solution is the adoption of *Verifiable Computation* (VC) schemes that work over encrypted data. A VC scheme provides proof that each arithmetic gate of the arithmetic circuit has had its inputs processed. Furthermore, it is possible to provide such proof of computation over a partially private circuit (known only to the cloud service) as the cloud service can prove that part in zero knowledge [4];

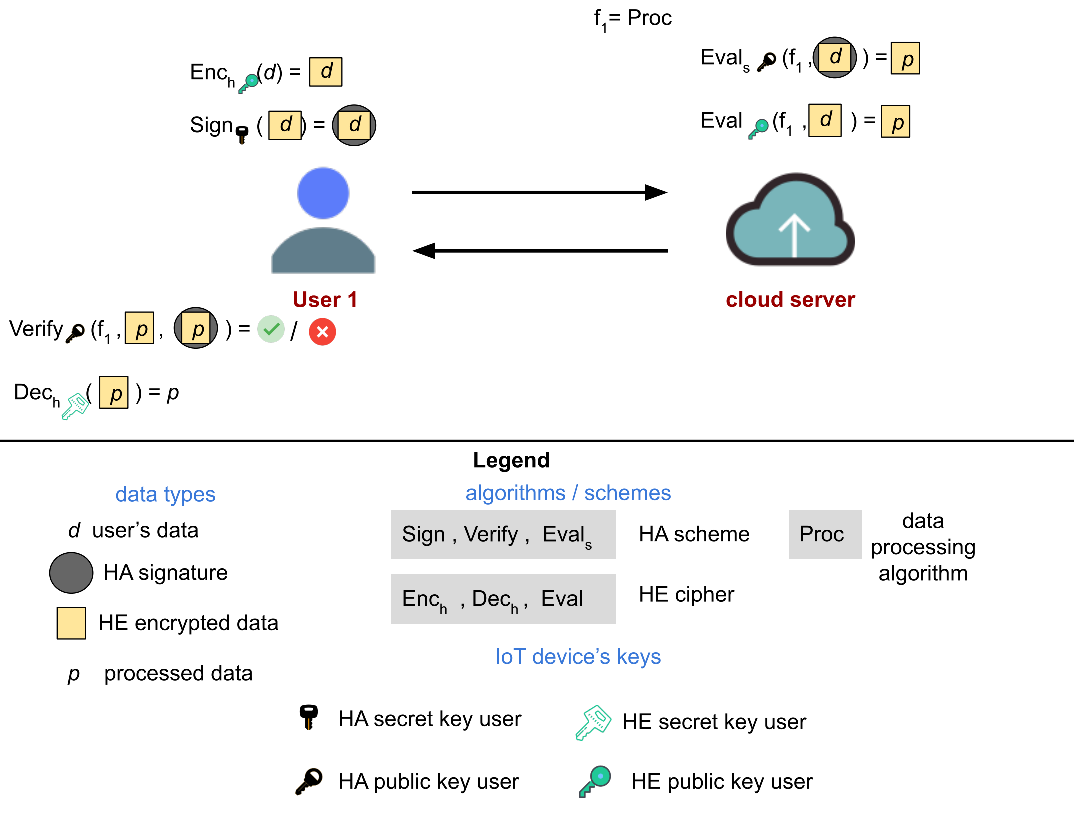

Figure 2.8. FHAE by composition encrypt-then-sign.

- *Homomorphic Encryption in Multiparty Computation*

  When multiple parties need to work together, using HE can be challenging. This is particularly true when various cloud services need to analyze a function that combines their private data without revealing any input information (except what can be inferred from the output). However, this issue can be solved by using *Secure Multiparty Computation* (MPC) [4]. MPC includes different types of protocol optimization for arithmetic and boolean circuits, which are based on secret-sharing techniques and garbled circuits, respectively. The computation is divided into two phases:

  1. the first phase occurs before the parties' inputs are determined. It involves creating cryptographic material, such as secret-shared elements or gabled circuits, which will be used to accelerate the second phase;
  2. during the second phase, the parties involved establish their respective inputs and privately evaluate the circuit. The consuming elements concept refers to the fact that the material used cannot be reused and must be generated anew for each execution.

HE plays a crucial role in MPC during the generation of preprocessing materials.

One way to create MPC is by using *Multikey Fully Homomorphic Encryption* (MKFHE) [4].

First, all parties encrypt their inputs using multiple keys and share the ciphertexts with everyone else, then each party performs the homomorphic evaluation of the circuit. In the second round, each of them partially decrypts the output and shares its results with others. These partial results can be combined locally by each party to obtain the final output.

With MKFHE, input privacy is ensured because a ciphertext cannot be decrypted without all partial decryptions from the secret key holders. However, if one party fails to deliver their share of the output, the entire MPC protocol will fail. Anyway, further research has solved the issue of resilience to failures, but it does require trust between the parties involved.

MKFHE is sufficient to provide MPC with passive security, however, in a setting where parties can deviate from the protocol specification (e.g., transmit wrong shares of the output), active security is needed and can be achieved by integrating zero-knowledge proofs [4].

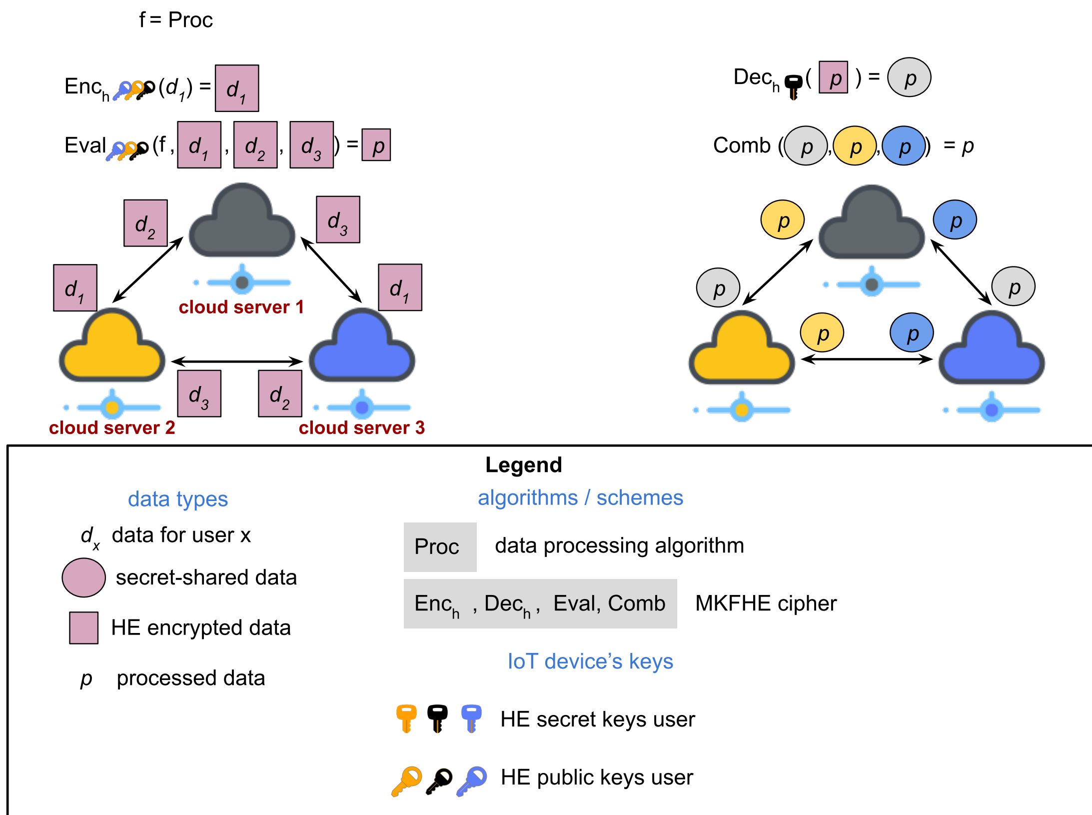

Figure 2.9. MPC constructed with MKFHE.

Marcolla et al. [4] provide an illustration of an MPC built with MKGHE in Figure 2.9. The figure only details the operations carried out by Cloud Server 1, but the other parties behave similarly. The algorithm *Enc* in the figure generates a ciphertext for multiple keys for simplicity, but in a real-world scenario, an expanded algorithm is used.

Chapter 3

FHE Libraries and Compilers

This chapter provides an overview of existing FHE libraries and compilers, their features, and the cryptographic schemes they support. The tools are studied to gain practical knowledge of implementing FHE in real-world settings.

3.1 FHE Libraries

The main objective of FHE libraries is to provide API access to FHE scheme operations. In addition to *KeyGen*, *Enc*, *Dec*, and *Eval* (Chapter 2.1), widely used libraries also offer features for ciphertext maintenance and manipulation, as well as homomorphic addition and multiplication methods. However, developers must have an in-depth knowledge of each API call to use them correctly in a privacy-preserving solution.

Library	Language	Scheme					Date of last commit
		BGV	B/FV	FHEW	TFHE	CKKS	
HElib	C++	✓	-	-	-	✓	1/10/2021
SEAL	C++/C#	✓	✓	-	-	✓	24/3/2022
PALISADE	C++	✓	✓	✓	✓	✓	30/4/2022
Lattigo	Go	-	✓	-	-	✓	13/6/2022
FHEW	C++	-	-	✓	-	-	30/5/2017
TFHE	C++/C	-	-	-	✓	-	16/9/2021
Concrete	Rust	-	-	-	✓	-	10/5/2022
HEAAN	C++	-	-	-	-	✓	27/1/2022
RNS-HEAAN	C++	-	-	-	-	✓	26/10/2018
FV-NFLib	C++	-	✓	-	-	-	26/7/2016
CuFHE	Cuda/C++	-	-	-	✓	-	9/2/2019
NuFHE	Python	-	-	-	✓	-	18/3/2020
OpenFHE	C++	✓	✓	✓	✓	✓	18/8/2022

Table 3.1. Available Open-Source FHE Libraries.

According to Marcolla et al. [4], Table 3.1 displays a list of well-known open-source FHE libraries, including the programming language used, supported FHE schemes, and the date of the last update release.

In the following list, a more exhaustive analysis will be conducted on the main libraries that have been used:

- *HElib* (Homorphic-Encryption Library) is the first open-source library implementing HE. Published in 2013, it focuses on using BGV and CKKS schemes, ciphertext packing, and optimizations. Till 2021 it is still under development, the authors implemented several algorithmic improvements, including faster homomorphic linear transformation, in order to make HELib 30/75 times faster than those previously built;

- *PALISADE* is an open-source C++ project that provides efficient implementations of lattice cryptographic building blocks. The library implements various HE schemes, such as BGV, BFV, CKKS, FHEW, and TFHE [5]. Furthermore, it supports multi-part extensions for digital signature schemes, proxy re-encryption, and program obfuscation. The PALISADE community has integrated the project into the next-gen OpenFHE (open-source FHE software library), which it will discuss later;

- *Lattigo* is a Go module that implements full-RNS ring learning with error-based homomorphic encryption primitives and multiparty homomorphic-based secure protocols. Its primary purpose is to support HE in distributed systems and microservices architectures, for which Go is a popular choice [18].

- *FHEW* is an open-source library provided by L. Ducas and D. Micciancio and makes use of the *Fastes Fourier Transform in the West* (FFTW). It supports the evaluation of arbitrary boolean circuits, but it has not been updated since 2017 [4].

- *TFHE* Fast Fully Homomorphic Encryption over Taurus, is an open-source library written in C/C++, and needs at least one fast Fourier transform (FFT) processor to run [4].

- *SEAL* (Simple Encrypted Arithmetic Library) is developed by the Cryptographic and Privacy Research Group at Microsoft. It was released in 2015 with the specific goal of providing a well-engineered and documented HE library, and it was conceived to be utilised both by experts and by non-experts with small or no cryptographic background [5]. In addition, there is a SEAL version written in Python, called SEAL-Python;

- *HEAAN* (Homorphic Encryption for Arithmetic of Approximate Numbers) is an open-source cross-platform software which implements CKKS with its complete properties [4];

- *RNS-HEAAN* is a variant of HEAAN written in C++, but it has not been updated since 2018 [4];

- *FV-NFLib* is designed for ideal lattice cryptography, written in C++ programming language. It is specifically tailored for polynomial rings quotient by a cyclotomic polynomial whose degree is a power of two. The library has algorithmic optimizations combined with programming optimization techniques for efficiency and better performance [4];

- *CuFHE* CUDA-accelerated Fully Homomorphic Encryption Library, is an open source library for FHE on CUDA-enabled GPUs. The library benefits from an improved CUDA implementation of the Number Theoretic Transform (NTT) [19];

- *NuFHE* implements the FHE algorithm from TFHE using CUDA and OpenCL in Python. It can use either FFT or purely integer NTT (DFT-like transform on a finite field) [20];

- *Concrete* is a Zama's variant of TFHE implemented in Rust, and it is a framework that enables developers to use homomorphic encryption in their applications without having to learn cryptography. All the complexity of FHE is hidden under the high-level APIs, while still being available in the low-level APIs. However, this library will be thoroughly examined in the subsequent section 3.3.

- *OpenFHE* is a new C++ library published in 2022 implemented in C++. Further details are in section 3.4.

According to T.V.T Doan et al. [5], in order to investigate the advantages and drawbacks of aforementioned libraries it is mandatory to define a set of criteria. Since all the previous libraries are open-source, which is fundamental for transparency reasons, the criteria are:

- *ease of use* implies that the library should be easy to integrate with existing systems and have clear documentation, examples, and a well-designed API for developers to use;
- *compatibility* describes the dependence of the library on a specific platform and/or hardware.
- *reliability* means that the library implementation is stable with few minimal bugs.

3.2 FHE Compilers and Accelerators

Fully homomorphic encryption compilers are high-level tools that abstract the technical APIs exposed by FHE libraries, hence a wider range of developers are capable of implementing privacy-preserving mechanisms securely. They address some of the most common engineering challenges that exist nowadays when designing FHE-based applications:

- *parameters' choice.* Defining secure and efficient parameter values for FHE schemes is challenging. Some FHE compilers offer automatic parameter generation based on predefined requirements [4];
- *plaintext encoding.* semantics of the plaintext message are strictly linked to the type of homomorphic computations that can be performed. Some context-specific FHE compilers can already be used to help in this particular item [4];
- *data-independent execution.* Since FHE operations are data-independent by nature, conducting data-dependent branching steps can break privacy properties. However, branching operations are possible by evaluating both branches and selecting the result at the end [4];
- *packing or batching.* FHE schemes letting for message packing or bathing into a single ciphertext can directly leverage SIMD instruction sets. Some FHE compilers already actively optimize for vectorized operations [4];
- *ciphertext Maintenance.* Managing noise growth during FHE operations is complex, so FHE compilers use advanced strategies to assist in this difficult operation [4].

The table 3.2 displays a list of FHE compilers, including the programming language they are written, the FHE libraries they use from the ones described in the previous Chapter 3.1, and their latest update date.

Compiler	Language	Library						Date of last commit
		HeLib	SEAL	PALISADE	FHEW	TFHE	HEAAN	
ALCHEMY	haskell	-	-	-	-	-	-	15/3/2020
Cingulata	C++	-	-	-	-	✓	-	7/12/2020
E3	C++	✓	✓	✓	✓	✓	-	31/5/2022
SHEEP	C++	✓	✓	✓	-	✓	-	7/4/2023
EVA	C++	-	✓	-	-	-	-	1/5/2021
Marble	C++	✓	✓	-	-	-	-	23/12/2020
RAMPARTS	Julia	-	-	✓	-	-	-	-
Transpiler	C++	-	-	✓	-	✓	-	15/9/2023
CHET	C++	-	✓	-	-	-	✓	-
nGraph-HE	C++	-	✓	-	-	-	-	3/1/2023
SEALion	C++	-	✓	-	-	-	-	-

Table 3.2. Available FHE Compilers.

The following is a list of compilers, accompanied by a brief explanation of each:

- *ALCHEMY* (https://github.com/cpeikert/ALCHEMY) is a modular system written by Crockett, Peikert, and Sharp [21] that simplifies and speeds up the use of FHE. Programmers can write computations on plaintexts in a domain-specific language (DSL), and the compiler will automatically generate the corresponding homomorphic computations on ciphertexts. The compiler can choose most of the parameters, generate keys and key-switching hints, schedule appropriate ciphertext maintenance operations, and more. Alchemy also allows the logging of empirical noise rates of ciphertexts throughout a computation, without requiring any changes to the original DSL code;

- *Cingulata* (https://github.com/CEA-LIST/Cingulata), is a compiler toolchain for executing C++ programs over encrypted data by means of fully homomorphic encryption techniques. Currently, it implements the TFHE Library;

- *Encrypt-Everything-Everywhere (E3)* (https://github.com/momalab/e3) is an easy-to-use open-source homomorphic encryption framework developed by the MoMA Lab at New York University Abu Dhabi. E3 allows non-crypto experts to add privacy to programs via FHE encryption and easy-to-use C++ operators;

- *SHEEP* (https://github.com/CEA-LIST/Cingulata) is a homomorphic encryption evaluation platform and aims to provide a platform for practitioners to evaluate the state-of-the-art fully homomorphic encryption technology in the context of their specific application. SHEEP makes it possible to conduct this evaluation across libraries implementing different HE schemes, which have varying levels of security;

- *Ecrypted Vector Arithmetic (EVA)* (https://github.com/microsoft/EVA) targets Microsoft SEAL, and currently supports the CKKS scheme for deep computation on encrypted approximate fixed-point arithmetic;

- *Marble* (https://github.com/MarbleHE/Marble) is a C++ middleware library that translates between user programs written close-to-plaintext-style and FHE computations based on FHE crypto libraries;

- *RAMPARTS* provides an environment for developing HE applications in Julia. It offers three main features. First, it uses symbolic execution to automatically build an optimized computation circuit. The compiler selects both the circuit size and multiplicative depth. Second, Ramparts automatically chooses the HE parameters for the generated circuit. Third, it automatically selects the plaintext encoding for input values and performs input and output data transformations [22];

- *Traspiler* (https://github.com/google/fully-homomorphic-encryption) is a versatile library that can convert C++ code into FHE-C++. Its modular architecture allows for customization of the FHE library, program description, and output language. This flexibility is intended to promote cooperation between researchers from diverse fields, with the shared goal of making FHE more efficient and widely applicable;

- *CHET* is an optimizing compiler that specializes in fully homomorphic encryption neural network inferencing. Its primary purpose is to make programming FHE applications easier. It is designed to support a domain-specific language for specifying tensor circuits. This is motivated by the need to perform neural network inference on encrypted medical and financial data [23];

- *nGraph-HE* (https://github.com/IntelAI/he-transformer) is an Intel graph compiler for artificial neural networks. Currently, it supports the CKKS scheme implemented by SEAL from Microsoft. The aim is to measure the performance of various HE schemes for deep learning;

- *SEALion* is an extensible framework for privacy-preserving machine learning, specifically in neural network inference. The framework has two layers: the first layer is built on TensorFlow and SEAL, and it exposes standard algebra and deep learning primitives. The second layer implements a Keras-like syntax for training and inference with neural networks [24].

While these tools have made it possible to significantly speed up Fully Homomorphic Encryption (FHE) schemes, the performance in comparison to computations on plaintext data is still not optimal. This has led to the development of *FHE hardware accelerators*, offering a possible option in place of highly optimized software implementations. As a result, a wider range of applications can now leverage FHE technology. Over the years, various optimized hardware solutions have been developed to accelerate both FHE and SWHE schemes, mainly focused on three implementation platforms: Graphic Processing Unit (GPU), Application Specific Integrated Circuit (ASIC), and Field-Programmable Gate Array (FPGA) [1].

3.3 Concrete Library by Zama

Concrete is an acronym for "*Concrete Operations oN Ciphertexts Rapidly by Extending TfhE*" [25], in fact as illustrated in Table 3.1 in the previous chapter 3.1, it implements and extends the TFHE scheme. This decision was made because, in the TFHE scheme, the bootstrapping procedure is relatively fast and could be "programmed" [25]. As mentioned above, homomorphic operations increase the noise in the ciphertext, and when the noise rises too much the ciphertext cannot be decrypted anymore. This is why it is very important to decrease the quantity of noise. Programmable bootstrapping (PBS) is a generalization of the bootstrapping technique allowing resetting the noise at a fixed level while at the same time homomorphically evaluating a function on the input ciphertext [25].

The library is written in Rust (`https://www.rust-lang.org/`), a new language known for its performance, security, memory safety, and efficient memory usage and access. TFHE is based on the learning with errors (LWE) problem and its variant Ring-LWE, hence it deals with two types of ciphertext. Its full potential is achieved by encrypting and bootstrapping any value, including real numbers, using encoding functions. In particular, it is not limited to boolean functions.

The main features of the library are [25]:

- ability to compile Python functions (that may include NumPy) to their FHE equivalents, to operate on encrypted data;
- support for large collection of operators;
- partial support for floating points;
- support for table lookups on integers;
- support for integration with Client / Server architectures.

In a usual deployment of the library, the client requests for specifications and requirements, based on which it generates the keys. After that, it sends the encrypted data and retrieves the result by decrypting it. The server responds with its own specifications and starts the evaluation process when it receives the encrypted data. Figure 3.1 provides a visual representation of this process.

In addition, Zama made available a dedicated machine-learning library called concrete-ml (`https://github.com/zama-ai/concrete-ml`).

3.3.1 Implementation

Concrete is built as a stack layer, as shown in Figure 3.2. The lower-level layer, called *Core* API, is accessible by FHE experts and aims to be efficient. The layer above, named *Crypto* API, is user-friendly and easy to use for any programmer, even with a limited understanding of cryptography. It offers additional capabilities, including automated metadata tracking of noise levels.

Going into further detail, Core API proposes a hardware generic API, and at the moment, it only provides access to the CPU-based code version, but a GPU version (following the same

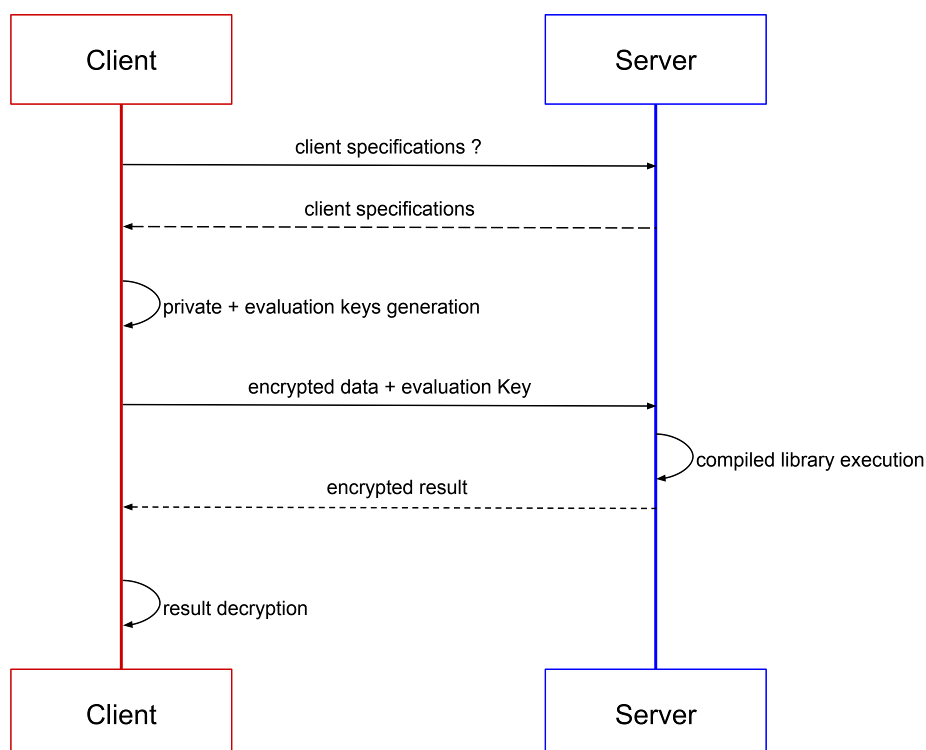

Figure 3.1. Concrete deployment.

API) is in preparation [25]. Crypto API, instead, is based on cryptographic objects defined as a structure. These structures also include metadata such as the noise distribution, the number of padding bits or the decoding parameters. They are mostly used to ensure the correctness of the computation.

The *Noise Propagation Estimator* (NPE) is a module devoted to monitoring the noise. In a nutshell, it contains the noise formula associated with each homomorphic operator. It uses metadata elements associated with the ciphertext as inputs, for example, the variances of the noise distribution. In the end, the noise variance of the output is updated [25].

As an additional feature, the *Fast Fourier Transform* (FFT) is used to speed up the polynomial products.

3.3.2 Core API

The core API is conceived as an abstraction of the hardware-dedicated code. The API provides multiple options that don't require any code modification, depending on the hardware available on the machine. For instance, it offers SIMD[1] optimizations that can be used if the CPU supports it, or even the option to run on a GPU if possible. The API strives to be as low-level as possible to achieve optimal performance.

The API is divided into two main modules:

[1]Single Instruction, Multiple Data refer to hardware elements that compute the same operation on multiple data operands concurrently

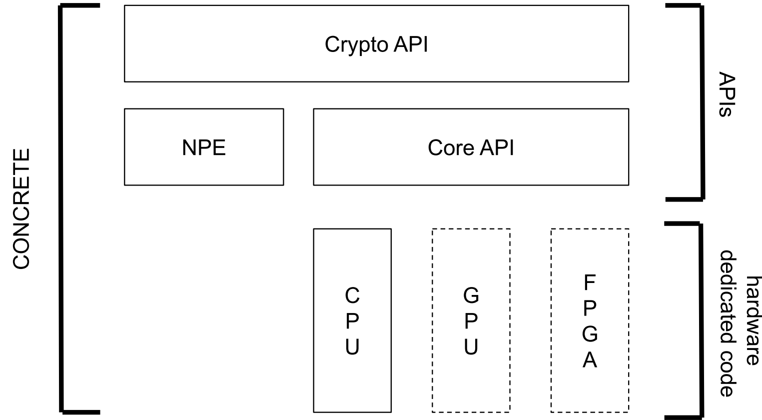

Figure 3.2. CONCRETE architecture. Dashed boxes refer to upcoming features.

- *math* including non-cryptographic operations such as adding, subtracting or multiplying (unsigned) integers or polynomials [25];
- *crypto* containing cryptographic-related operations such as encryption and decryption functions or bootstrapping and key switching functions [25].

In Rust, a *slice* is a sequence of elements that are contiguous. Unlike an array, its size is not known at compile time. Slices are commonly used to represent messages, plaintext, ciphertext, and keys. Messages are stored in slices of f64, while plaintext, ciphertext, and keys are stored in slices of unsigned integers u32 or u64.

- *ciphertexts* are stored in slices of unsigned 32/64-bit integers. In the core API, functions are able to operate on an array of ciphertexts, this is modelled as a single slice including several concatenated LWE ciphertexts [25];
- *secret Keys*, LWE or RLWE secret keys are sampled from an invariant binary distribution. Keys are kept in the same way as ciphertext [25];
- *random source.* Uniform random integers are generated with *OpenSSL* [25].

3.3.3 Crypto API

The *Crypto* API simplifies the use of the *Core* API. Typically, metadata are directly included and automatically updated as the Core API homomorphic computation progresses [25].

- *encoders and ciphertext.* A structure called an Encoder includes information about the real interval bounds, precision bits, padding bits, and encoding type. It also has metadata for

tracking noise, encoding, and the number of correct message bits during computations. The NPE modules are used to evaluate the noise as mentioned above [25];

- *padding bits.* Some homomorphic functions require padding bits to avoid losing significant bits of plaintext. Furthermore, padding bits act as a safety guard for computation correctness [25];
- *automatic generation of the look-up table.* The Crypto API can automatically create a lookup table for any function f that takes f64 inputs and outputs f64 values. This feature is mainly used in PBS. To generate the table, the API requires the encoding of both the inputs and outputs, as well as the function f itself [25].

To put it concisely, the Concrete library provides a convenient and effective way to perform homomorphic computations on encrypted data. It offers several encoding methods, including representing approximations of real numbers. With the PBS, it is possible to compute non-linear functions in addition to homomorphic addition and multiplications by scalar [25].

3.3.4 Example of usage

This section presents a step-by-step example of using the library provided in concrete documentation [26], as illustrated in Figure 3.3.

To begin with, we need to import the module `fhe` from the library. After that, we define the function, which in this example, lines 2 and 3, is just a simple addition. To compile the function, we have to define a `Compiler` by specifying the function we want to compile and the encrypted status of its inputs, line 5.

Next, we need to specify an input set, line 7, which represents the typical inputs of the function. This set is used to determine the bit widths and shapes of the variables within the function. The input set should be iterable and yield tuples of the same length as the number of arguments of the function being compiled. All inputs in the input set will be evaluated in the graph, which takes time.

After defining the input set, we can use the `compile` method of the `Compiler` class with the input set previously described to perform the compilation and get the resulting circuit back, line 8.

Finally, the most important operation is the `encrypt_run_decrypt` method of a *Circuit* class, which is written in line 14. This method is used to perform homomorphic evaluation.

3.4 OpenFHE library

As previously explained, there are several open-source libraries implementing FHE schemes. In this section, a detailed explanation of OpenFHE will be provided. OpenFHE was designed by some of the authors of PALISADE. HElib, HEAAN, and FHEW libraries and it supports all common FHE schemes [27].

Moreover, OpenFHE introduces several new features:

- it is expected that all FHE schemes that are supported will eventually incorporate bootstrapping and scheme switching functionalities [27];
- the library has the capability to accommodate multiple hardware acceleration back-ends through a standardized Hardware Abstraction Layer (HAL) [27];
- openFHE offers both user-friendly modes, where maintenance operations like modulus switching, key switching, and bootstrapping are automatically handled by the library, and compiler-friendly modes where these decisions are made by an external compiler [27].

```
1
2   def add(x, y):
3       return x + y
4
5   compiler = fhe.Compiler(add, {"x": "encrypted", "y": "clear"})
6
7   inputset = [(2, 3), (0, 0), (1, 6), (7, 7), (7, 1)]
8   circuit = compiler.compile(inputset)
9
10  x = 4
11  y = 4
12
13  clear_evaluation = add(x, y)
14  homomorphic_evaluation = circuit.encrypt_run_decrypt(x, y)
```

Figure 3.3. Example of usage of the concrete library.

3.4.1 Cryptographic capabilities

OpenFHE is a library designed to implement a range of FHE schemes for integer, real-number, and Boolean arithmetic. These schemes are all based on the difficulty of solving the ring variant of the Learning With Errors (RLWE) problem. Moreover, the library encompasses multiparty extensions to facilitate scenarios involving multiple secret keys or secret shares.

Since the library is designed to work with all commonly used schemes, it supports all the different classes of homomorphism. The first class utilizes *modular arithmetic over finite fields* and includes BGV and BFV schemes. The second class uses *boolean circuit and decision diagrams* and includes DM and CGGI schemes. The third class works with *approximate computation over vectors of real and complex numbers* and is represented by the CKKS scheme.

In terms of implementation, the first and third classes have a shared design that was initially created for BGV. This design allows for a significant number of multiplications without bootstrapping by utilizing the technique of modulus switching and enables homomorphic operations on vectors of integer or real numbers in a Single Instruction, Multiple Data (SIMD) manner. Additionally, DM and CGGI also have a common design, which is explained in [27].

As mentioned above OpenFHE supports multiparty extensions for BGV, BFV, and CKKS schemes: threshold FHE and Proxy ReEncryption (PRE). The Threshold Fully Homomorphic Encryption extension operates by using additive secret sharing and the additive key homomorphism properties of FHE schemes. While the computation follows the same path as single-key FHE, key generation and decryption are replaced with their distributed version [27].

The Proxy Re-Encryption (PRE) extension makes it possible to delegate an existing ciphertext to another party that has a different secret key. This is done by using FHE key switching to perform proxy encryption of the ciphertext with the encryption key. This key represents the encryption of the old secret key using the public key for the new secret key [27].

3.4.2 Bootstapping, Noise Estimation and Scheme Switching

The current scheme-specific information regarding *bootstrapping* availability for the core FHE schemes in openFHE is as follows [27]:

- bootstrapping is currently implemented for DM and CGGI;
- approximate bootstrapping is implemented for CKKS;

- there is a prototype implementation of thin BGV bootstrapping

One drawback of using this library is that it does not currently have *noise estimation* included. The user must indicate the multiplicative depth (and for certain schemes, the maximum amount of additions/key switching operations). OpenFHE then chooses all necessary parameters, such as the number of bits required for each multiplicative level. Subsequently, the library carries out operations on ciphertexts, utilizing scaling-modulus-switching for certain schemes, without estimating the noise level in each ciphertext. It is up to the users to ensure that pre-specified bounds are respected, according to [27] in the future version, a noise estimation is going to be added.

Scheme switching is beneficial for various applications. For instance, CKKS is best for polynomial evaluation in many machine-learning applications, while DM/CGGI is ideal for comparison and other discontinuous functions. This process requires bridging between schemes, typically using bootstrapping-based procedures.

OpenFHE supports various scheme-switching operations that enable different encryption schemes to work together. Scheme switching involves multiple techniques and processes that go beyond just changing the encryption scheme used for message encryption. It is helpful to categorize scheme-switching operations based on various factors: message space, key space and key material.

3.4.3 Hardware Acceleration Support

In order to effectively utilize OpenFHE, hardware acceleration is crucial. This library is specifically crafted to accommodate various hardware acceleration technologies, including AVX, GPU, FPGA, and ASIC. As FHE's computational challenges are mainly related to polynomial arithmetic required for implementing FHE schemes, these technologies can help overcome these bottlenecks.

According to recent research on hardware acceleration, FHE computations that use bootstrapping are limited not only by computing power but also by memory. A study on FHE hardware acceleration found that moving data is the most significant bottleneck in homomorphic computations based on BGV and CKKS. The study proposes a design that decreases data movement, additionally, both studies suggest that key switching is a memory bandwidth challenge, as it involves large evaluation keys and requires a series of alternating NTT and RNS operations with different data access patterns [27]. Hence, OpenFHE is designed to minimize the amount of data movement necessary.

The design of the library is modular and consists of several layers, as seen in Figure 3.4. The math and lattice/polynomial layers contain all the algorithms for polynomial arithmetic. The Hardware Abstraction Layer (HAL) aims to enable multiple instantiations of bottleneck polynomial and RNS operations within these layers. To accomplish this, C++ abstract classes are defined for functionalities that may have backend-specific implementation [27].

In the primitive math layer, abstract classes have been defined for integers, modular vectors, and NTT transformations. The lattice-polynomial layer defines abstract classes for polynomials, with a focus on the double-CRT polynomial representation [27].

The design aims to implement integers, modular vectors, NTT transformations, and polynomials with a backend-aware approach. It also uses a uniform implementation of cryptographic capabilities for all backends [27].

The current implementation of OpenFHE supports only software PRNGs based on random seeds.

3.4.4 Usability Enhancements

The goal of OpenFHE is to enhance the usability of Fully Homomorphic Encryption (FHE) schemes for application developers. While some FHE schemes like BFV are already user-friendly, others such as BGV and CKKS are more complex. BFV does not require explicit modulus

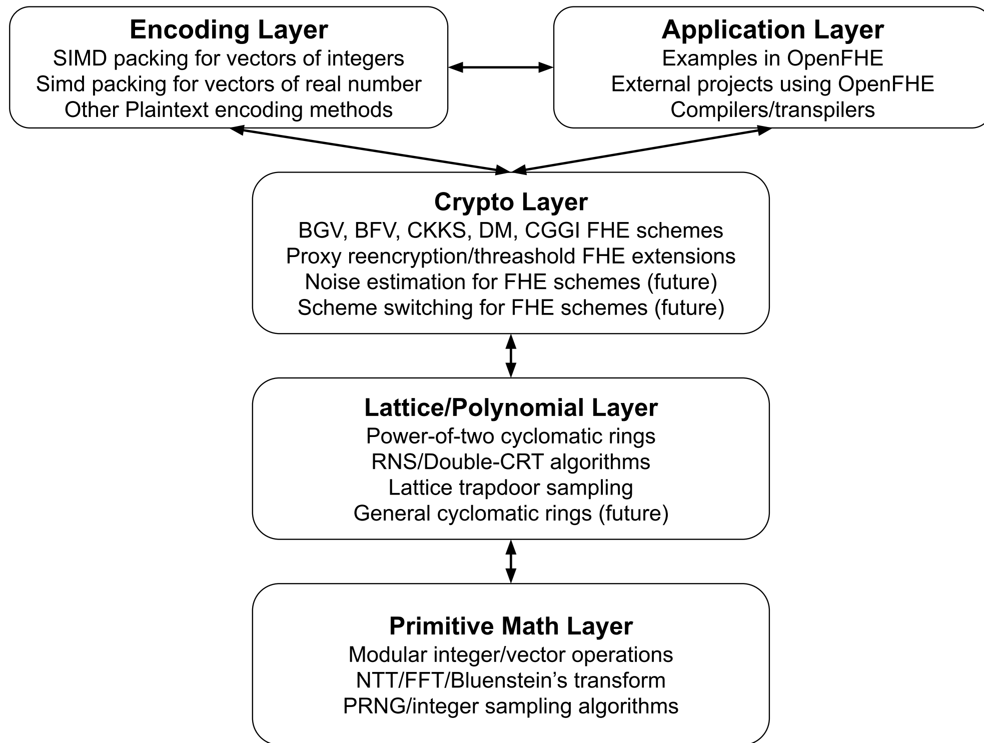

Figure 3.4. Layers in openFHE.

switching or key switching by the user, and BFV, DM, and CGGI schemes automate certain processes like bootstrapping and gate optimization. However, BGV necessitates precise noise estimation and strategic use of modulus switching, and CKKS demands a meticulous assessment of approximation errors and controlled truncation of the least significant bits through rescaling.

However, for more sophisticated use cases and improved performance, non-experts can utilize external FHE compilers. These compilers tackle complex tasks like selecting FHE parameters, determining optimal locations for modulus switching and key switching to enhance performance, and handling the intricate process of encoding input data into plaintexts compatible with FHE. One of the compilers that supports the OpenFHE library is *Google Transpiler*.

3.4.5 Example of usage

In this section, an example of using the OpenFHE library is presented, as extracted from the documentation associated with the work [27]. The illustration involves employing the *FHEW* scheme along with the default bootstrapping method, *GINX*, as depicted in Figure 3.5.

The first step is to set the *cryptoContext* with the method `BinFHEContext()` which returns an auto[2] object. Then call the function `GenerateBinFHEContext(STD128)` to actually set the parameters. *STD128* is the security level of 128 bits based on the LWE estimator and HE standard, there are other common options it is possible to choose which are *TOY*, *Medium*,*STD192* and *STD256*.

The second step is the *key generation* process. This procedure has two important phases. The first one involves the generation of a secret key, which can be done using the function `KeyGen()`.

[2]The auto keyword in C++ automatically assigns a data type based on the variable's initialization.

The second phase is to generate a key for bootstrapping, which requires the secret key and can be done with the function `BTKeyGen(sk)`.

The third step is the *encryption* procedure, in this particular example, the encryption process involves two ciphertexts representing a boolean true value (1). By default, freshly encrypted ciphertexts are bootstrapped by writing `Encrypt(sk,1)`. However, if it is wanted to get fresh encryption without bootstrapping it is required to include an additional parameter in this way: `Encrypt(sk,1,FRESH)`.

The fourth step is the *evaluation* which is the most important part of the FHE procedure. The following is a list describing two examples:

- `EvalBinGate(AND,ct1,ct2)` where `ct1` and `ct2` are the encrypted ciphertex. In the example in Figure 3.5 this function computes *1 AND 1*;

  Other arbitrary gate options are *OR*, *NAND*, and *NOR*;

- `EvalNOT(AND,ct1,ct2)`, where `ct2` is an encrypted ciphertext. This method performs *NOT 1*.

The final step is the one concerning how to perform the *decryption*, it is possible to call `Decrypt(sk,ctResult,&result)` method, where *sk* is the secret key, *ctResult* is the result of the evaluation, and *result* is an *LWEPlaintext* variable in which the result of the decryption will be stored.

3.5 Comparison between Concrete and OpenFHE

This section compares Concrete by Zama and OpenFHE, as explained in chapter 3.3 and chapter 3.4. The analysis justifies the selection of Concrete over OpenFHE as the preferred tool for evaluating FHE performance in the following chapters.

OpenFHE is a more complete library since it includes almost all the available FHE schemes and features, indeed it is modular and flexible and allows the user to achieve a high level of customisation. It provides a range of advanced tools for FHE, including a compiler, and a set of cryptographic primitives, but this is at the cost of being less user-friendly and more complex.

On the other hand, Concrete has the ability to work with a wide range of operators and partially supports floating points. Additionally, it comes equipped with a noise estimator and extensive metadata linked to the compilation circuit, which are essential for conducting a detailed analysis. Furthermore, it also allows for analysis over machine learning algorithms using *concrete-ml* as discussed in chapter 3.3.

Additionally, as it was explained in Chapters 3.3.4 and 3.4.5 with the examples, it can be noticed that even when it comes to coding, concrete is much more practical as it requires fewer steps and less configuration on the users's side.

In conclusion, concrete would be a better choice in terms of ease of use and features when conducting a comprehensive analysis of the FHE performance.

```
1     // Sample Program: Step 1: Set CryptoContext
2
3     auto cc = BinFHEContext();
4
5     cc.GenerateBinFHEContext(STD128);
6
7     // Sample Program: Step 2: Key Generation
8
9     auto sk = cc.KeyGen();
10    cc.BTKeyGen(sk);
11
12    // Sample Program: Step 3: Encryption
13
14    auto ct1 = cc.Encrypt(sk, 1);
15    auto ct2 = cc.Encrypt(sk, 1);
16
17    // Sample Program: Step 4: Evaluation
18
19    // Compute (1 AND 1) = 1
20    auto ctAND1 = cc.EvalBinGate(AND, ct1, ct2);
21    // Compute (NOT 1) = 0
22    auto ct2Not = cc.EvalNOT(ct2);
23    // Compute (1 AND (NOT 1)) = 0
24    auto ctAND2 = cc.EvalBinGate(AND, ct2Not, ct1);
25    // Computes OR of the results in ctAND1 and ctAND2 = 1
26    auto ctResult = cc.EvalBinGate(OR, ctAND1, ctAND2);
27
28    // Sample Program: Step 5: Decryption
29
30    LWEPlaintext result;
31    cc.Decrypt(sk, ctResult, &result);
```

Figure 3.5. Boolean Fully Homomorphic examples with default bootstrapping.

Chapter 4

Machine Learning

We live in the *Fourth Industrial Revolution* (Industry 4.0), an era of data, advanced analytics and data science, where everything around us is connected to a data source and everything in our lives is digitally recorded [28].

This chapter provides an overview of machine learning, including explanations of algorithms and comparison metrics, describing how it is possible to extract and manipulate data in various domains to build a data-driven automated and intelligent system. It is important to mention that the effectiveness and efficiency of machine learning (ML) solutions depend on the nature and characteristics of *data* and the performance of the *learning algorithms* [29].

4.1 Type of Data and Machine Learning Techniques

Learning is a process of *"using experience to gain expertise"* [30] and, as mentioned above the characteristics of data play a crucial role in the performance of machine learning algorithms, according to H. Sarker [29] data are divided into different types:

1. *structured* data: characterized by a well-defined structure following a standard hierarchy, hence they are highly organized and easily accessed. For instance a relational database;
2. *unstructured* data: characterized by a no pre-defined organization and structure, mostly representing text or multimedia material. For example PDF, audio or image files;
3. *semi-structured* data: usually stored in NoSQL databases and characterized by some organizational properties, hence are more manageable to access and analyse. For example, HTML, JSON or XML documents;
4. *metadata*: they are *"data about data"* [29]. They describe the relevant data details such as the file size, the date of generation and other properties.

In order to manipulate and analyse those types of data, distinct machine learning techniques are employed. According to H. Sarker [29], it is possible defined four categories:

1. *supervised* is characterized by employing labelled datasets for training algorithms to effectively classify data or accurately predict outcomes. The model adapts its weights during the input data feeding process. In particular, it involves mapping input to output based on input-output pairs;
2. *unsupervised* it is characterized by the analysis of unlabeled datasets. For example, it is used for identifying meaningful trends and structure [29];
3. *semi-supervised* is a hybrid variant of the previously mentioned types, it has the capability to operate with both labelled and unlabeled data. Its main purpose is to deliver better outcomes compared to those generated solely with labelled data;

4. *reinforcement* is concerning inferring the optimal behaviour in an environment through interaction and observations of the responses. This process is based on reward and penalty as feedback, and the goal is to obtain the maximum reward. It is widely applied in training Artificial Intelligence (AI) models and in robotics.

Figure 4.1 provides a summary of the various machine learning types discussed before. In the context of this work, it is primarily emphasized *supervised learning* models, as the current application of Fully Homomorphic Encryption requires labelled data.

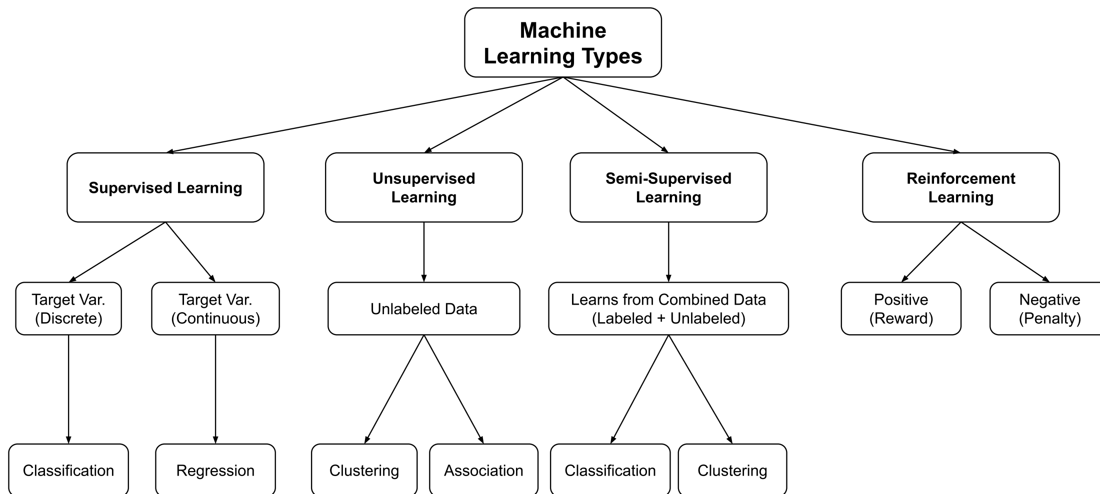

Figure 4.1. Types of machine learning techniques

4.2 Machine Learning Algorithms

The field of machine learning is vast and there are various algorithms that include *classification* and *regression* analysis, *data clustering*, *association rule learning* and *deep learning* methods [29].

In this section, *classification* and *regression* will be explored to briefly describe their most important characteristics and features.

A general structure of an ML predictive model trained from previously recorded data is illustrated in Figure 4.2. The model generates a prediction at the end of phase 2.

Figure 4.2. General structure of a predictive model.

4.2.1 Classification

Classification is the procedure of finding a model that describes and distinguishes classes or concepts. The model is based on the analysis of a collection of training data and it is employed to predict the class labels of objects for which the class labels are known [31].

The classification model can use both *structured* or *unstructured* data. In the literature there are many classification problems, this is a list of the most commonly employed [29]:

- *binary classification*: is a task that categorises new observations into one of two classes, *true or false* or *yes or not* [29]. Potential application fields are medical diagnosis, email analysis, financial data analysis, and image classification;
- *multiclass classification*: is a task with more than two class labels [31], but each sample can only be classified into a single class. Hence, each training point is included in one of N distinct classes and the purpose is to construct a function which, given a new data point, will accurately predict the class to which the new point belongs. For instance, it could be used to classify a network attack in a dataset, where attack categories are divided into different labels such as denial of service, user-to-root attack and root-to-local attack;
- *multi-label classification*: is a generalization of the *multiclass classification*, where a sample is assigned with multiple classes or labels [29]. Each data instance may simultaneously be owned by more than one class, hence while conventional classification deals with single-label problems, *multi-label classification* needs the support of predicting various mutually *non-exclusive* labels. For example, an image that includes multiple objects can be labelled with all the objects within it.

Over the years, many classification algorithms have been presented. These are the most popular and used [29]:

- *Naive Bayes (NB)*: is a supervised machine learning algorithm, based on the *Bayes'* theorem[1]. It is founded on the assumption that the predictive attributes are conditionally independent given the class, and it asserts that no hidden or latent attributes affect the prediction process [32];

  The algorithm is suitable for both binary and multi-class classification categories in different real-world circumstances, and it needs a small amount of training data to perform the task in a fast way [29]. Nevertheless, its performance may be affected by the assumption of attribute independence mentioned above [29];
- *Linear Discriminant Analysis (LDA)* or *Normal Discriminant Analysis (NDA)* follows a framework where the algorithm models the data distribution for each class and employs *Bayes'* theorem to classify new data points making predictions to estimate the probability of whether an input data set will belong to a particular output [33];
- *Logistic Regression (LR)* or *logit model* is a model which makes an estimation of the probability of an event occurring, based on a pre-defined dataset of independent input [33]. It typically uses a logistic function to compute the probabilities, which is referred to as the sigmoid function defined in this way:

  $$Logit(\pi) = 1/(1 + exp(-\pi)) \; [33]$$
  $$ln(\pi/(1-\pi)) = \beta_0 + \beta_1 * X_1 + \cdots + \beta_k * X_k \; [33]$$

  where $logit(\pi)$ is the dependent or response variable and X is the independent variable. β coefficient is commonly estimated via maximum likelihood estimation (MLE)[2].

[1] in probability and statistic, it describes the probability of an event based on previously related input occurrence. Hence, it is a formula employed for computing conditional probabilities [32].

[2] method that tests different values in order to optimize for the most acceptable fit of log odds.

The main problem is the constraint of the linearity between dependent and independent variables [29];

- *K-nearest neighbours (KNN)* can be employed in both classification and regression problems, but it is mostly used as a classification algorithm. It belongs to the category of *lazy classifiers* which means that this technique does not build the model from the training set but postpones it until it is used to classify the observations from the test set.

  KNN works on the assumption that similar points can be located near one another [33]. K is a hyperparameter of the model that indicates the number of observations in the *neighbouring* training set that the model must consider to perform a classification. The latter is computed from a simple majority vote of the k nearest neighbours of each point and its major issue is to choose the optimal number k to consider [29];

- *Support vector machine (SVM)* can be used for both classification and regression problems. SVM attempts to classify the observations in the data set by finding a *hyperplane* which succeeds in dividing the data into two distinct regions. Since there are an infinite number of hyperplanes, SVM looks for the *Maximum Marginal Hyperplane* (MMH), which is the hyperplane that can maximize the margin width, defined as the minimum distance between observations belonging to two different classes. The observations that define the margin are called *support vectors* as it is shown in Figure 4.3.

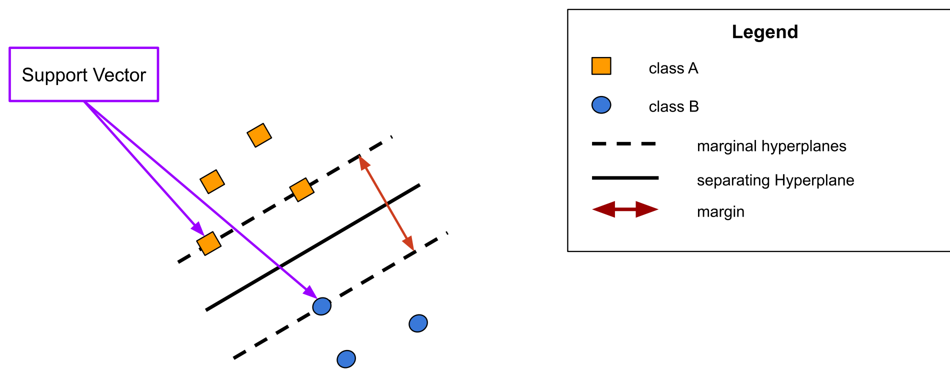

Figure 4.3. Linear SVM optimize margin.

  There are two types of SVM [29]:

  1. *simple or linear SVM* used to classify linear distinct and separable data, in a 2D space;
  2. *kernel or non-linear SVM* to perform classification of non-linear data which have features into higher or infinite dimensions, non just 2D space. It works in order to manage problems with an input set of multiple variables.

  Nevertheless, it doesn't achieve great accuracy when the input data includes noise [29];

- *Decision tree (DT)* is a non-parametric supervised machine learning algorithm, that can be employed in both classification and regression tasks.

  Figure 4.4 shows a usual tree structure, it is a hierarchical layout which consists of a *root* node, *internal* or *decisional* nodes and *leaf* or *terminal* nodes. There are various types of tree structures in literature, but they typically work in the following way: root and internal nodes are responsible for evaluating the available features, while the leaf nodes represent all the possible outcomes within the dataset [33].

  Decision tree learning is a strategy that uses a *divide-and-conquer* [3] technique. It searches

[3] it is a paradigm that divides problems into sub-problems that are similar to the original one, then solves the sub-problems and combines the solutions

for the best points to split the data within a tree. This procedure of splitting is then repeated recursively from the top down until all or most records have been classified under specific class labels. Whether or not all data points are classified into homogeneous sets depends largely on the complexity of the decision tree [29];

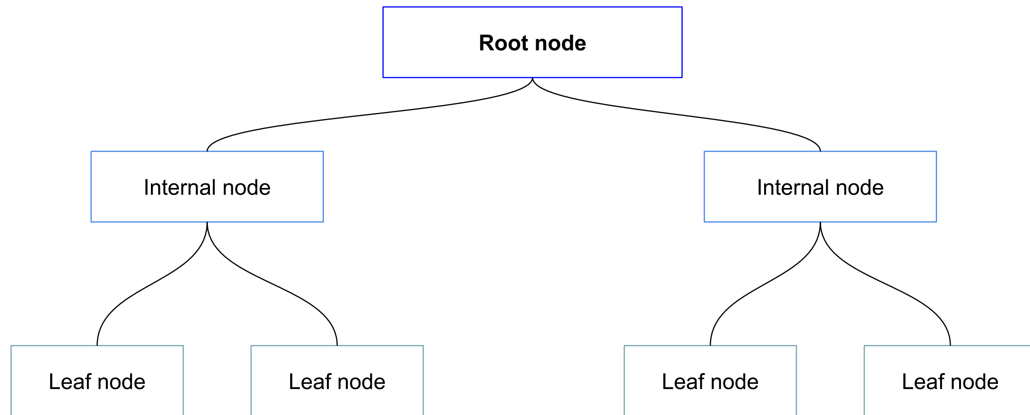

Figure 4.4. Example of a decision tree structure.

- *Random forest (RF).* Combination techniques enhance classification accuracy by combining multiple classifiers with low correlation, reducing variance and dependence on specific training sets. Achieving independence among models from the same training set is challenging, leading to the development of various techniques, such as *random forests*.

  As depicted in Figure 4.5, the random forest consists of a collection of decision trees made up of data from the dataset, then combines it to produce the final result. There exist certain methodologies to construct decision trees with controlled variation, which ultimately lead to augmented accuracy and performance. By implementing these techniques, one can ensure that the decision tree provides reliable and precise output, which is essential [29].

4.2.2 Regression

Regression is a statistical technique that is frequently employed to predict numerical outcomes. It also allows identifying *trends* over the available data. As shown in 4.6, the main difference between classification and regression is that the former predicts distinct class labels, whereas the latter enables the prediction of a continuous quantity. In classification the dotted line is a boundary that divides the two classes, instead in regression the dotted line models the linear relationship between variables [29].

Regression is commonly used in different fields such as financial, business and marketing. Furthermore, there are distinct types of regression algorithms, the most used are briefly described in the subsequent list:

- *simple and multiple linear regression.* Linear regression analysis is a statistical method that helps in predicting the value of one variable based on the value of another variable. The variable that we want to predict is referred to as the *dependent variable*, while the variable we use to predict the other variable's value is called the *independent variable* [33].

  Simple linear regression analyzes the relationship between two variables, the dependent and the independent, to explain how changes in one variable impact the other. It could referred to as:

$$y = a + bx + e$$

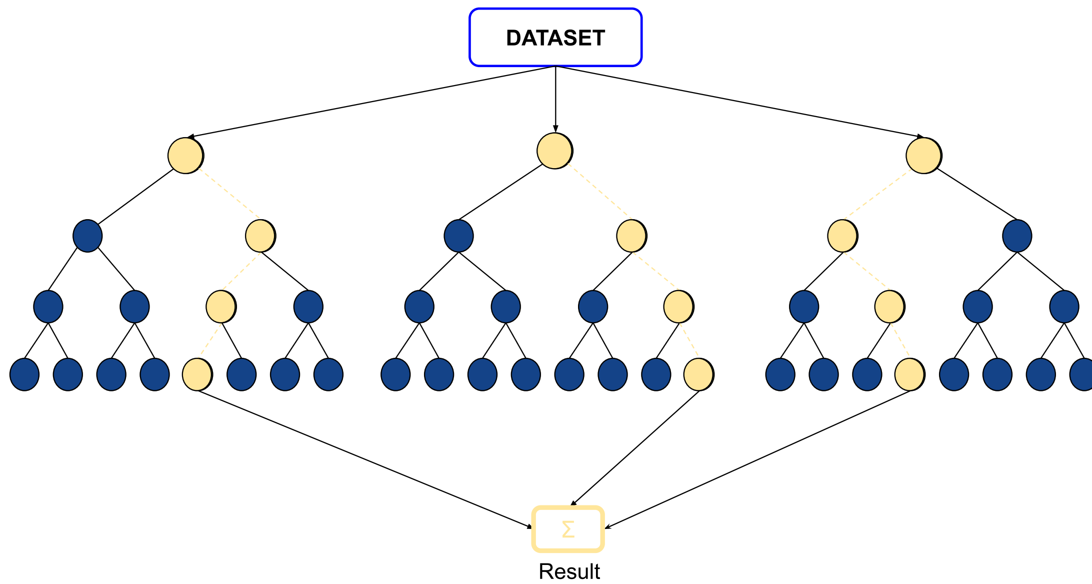

Figure 4.5. Example of a random forest structure.

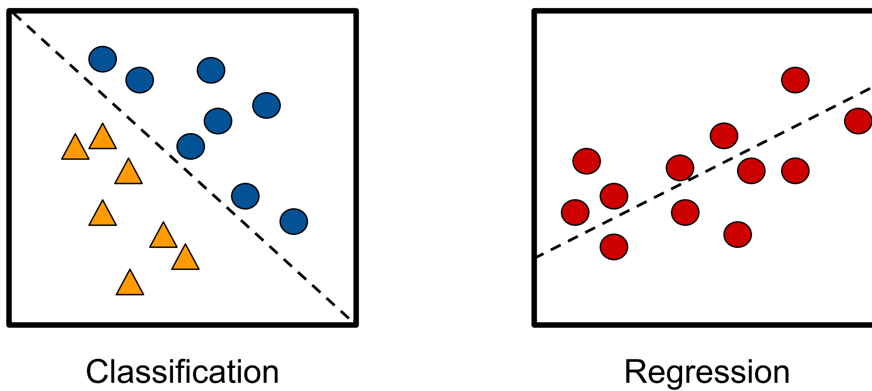

Figure 4.6. Classification vs regression.

where a is the interceptor, b is the slope of the line, e is the error term, y is the dependent variable and x is the independent variable [29].

Multiple linear regression, instead is an extension of simple linear regression that uses several dependent variables (x_n) to predict the outcome of a response or independent variable (y). It could be represented through the following equation:

$$y = a + b_1 x_1 + \cdots + b_n x_n + e$$

where a is the interceptor, b_n is the slope and e is the error term [29];

```
1   # Lets create a synthetic data-set
2   x, y = make_classification(n_samples=1000, class_sep=2, n_features=30,
        random_state=42)
3
4   # Split the data-set into a train and test set
5   X_train, X_test, y_train, y_test = train_test_split(
6       x, y, test_size=0.2, random_state=42
7   )
8
9   # Now we train in the clear and quantize the weights
10  model = LogisticRegression(n_bits=8)
11  model.fit(X_train, y_train)
12
13  # We can simulate the predictions in the clear
14  y_pred_clear = model.predict(X_test)
15
16  # We then compile on a representative set
17  model.compile(X_train)
18
19  # Finally we run the inference on encrypted inputs
20  y_pred_fhe = model.predict(X_test, fhe="execute")
```

Figure 4.7. Concrete-ml example of usage.

- *polynomial regression* is regression analysis in which the association between the independent variable x and the dependent variable y is not linear, but it is modelled as an $n^{(th)}$ degree polynomial in x [29];
- *LASSO and ridge regression* are two of the most important techniques widely used nowadays for building learning models; *Lasso (least absolute shrinkage and selection operator)* aims to enhance the prediction accuracy and interpretability of the resulting statistical model, by deleting the most important features; *Ridge* is employed when a data set has multicollinearity[4] [29].

4.3 Concrete-ml example of usage

In this section a brief explanation of how Concrete-ml by Zama works and its characteristics are presented.

Developers can now easily turn machine-learning models into their fully homomorphic equivalents without any preliminary knowledge of cryptography. This can be done by employing familiar APIs from scikit-learn and PyTorch [34].

In Figure 4.7, it is possible to notice that the first step is to define the dataset. In this case, it is created a dataset from scratch with 1000 entries for testing purposes. After that, the dataset is split into a training set and a test set. At this point, any machine-learning algorithm could be used to train the model, but in this instance, the code utilized a logistic regression model, which is a classification problem as discussed in Chapter 4.2.1. Finally, it is possible to simulate the prediction with clear inputs or run the inference on encrypted inputs.

The following is a summary of the process used to perform inference on encrypted data as defined in the documentation [34]:

[4]high correlation between two or more independent variables

1. The model is trained on clear data using scikit-learn. However, since Concrete only works with integers, as explained in Chapter 3.3, the model needs to be quantized before performing inference;
2. The quantized model is then converted to a concrete Python program under the hood, and subsequently compiled;
3. Once the model is compiled, inference can be performed on encrypted data.

Typically, the data are encrypted by the client, securely transmitted to the server, where they are processed, and then the result is decrypted by the client.

Chapter 5

Methodology

This chapter examines the methodology used for the subsequent investigations presented in the following paragraphs. As discussed in chapter 3.5, Concrete by Zama was chosen to conduct the analyses because it has the capability to work with large structures and is supported by metadata that allows the behaviour of specific functions to be estimated and analyzed.

5.1 Hardware architecture and software configuration

In the context of this work, a virtual machine running on Politecnico di Torino's server is employed to execute tasks. The technical specifications are listed in Table 5.1.

It is important to point out that the performance of the CPU is crucial for the execution of concrete library processes. In addition, we are using Python 3.9.10, as the library is compatible with version 3.8 and above. The installation and functioning of concrete rely on two packages, namely *wheel* [1] and *setuptools* [2].

Concrete-python version used is 2.4.0 which natively supports [26] only addition, subtraction and multiplication between two encrypted values. Furthermore, it has other limitations which are:

- *control flow constraints*: some control flow statements in Python are not supported, such as *if* or *while* statements that rely on encrypted values [26];
- *type costraint*: it is not possible to have floating point inputs or outputs. However, it is possible to have floating point intermediate values as long as they can be converted to an integer Table Lookup. For example, `(60 * np.sin(x)).astype(np.int64))` [26];
- *bit width constraints*: there is a limit to the bit width of encrypted values, and exceeding this limit will result in an error [26].

The installed version of *Concrete-ml* is 1.4, but it has certain limitations. For instance, it can only operate within 16-bit integers. As a result, machine learning models need to be quantized, which may lead to a loss of accuracy compared to the original model that operates on plaintext. Furthermore, *Concrete-ml* only supports FHE inference. Therefore, training has to be performed on plain data, which produces a model that is then converted to an FHE equivalent. Finally, there is currently no support available for pre-processing model inputs and post-processing model outputs. These processing stages may involve text-to-numerical feature transformation,

[1] Wheels are a crucial part of the Python ecosystem. They enable faster and more stable package installations.

[2] Setuptools is a stable and feature-rich library that is actively maintained. Its primary purpose is to simplify the process of packaging Python projects.

dimensionality reduction, kNN [3] or clustering, featurization, normalization, and the mixing of results of ensemble models [34].

Scikit-learn version 1.1.3 is a simple library that works with Python, it is employed for predicting data analysis and machine learning, but it does not support HE. In the context of this work, it is used to compare performance with respect to computational time cost and accuracy between machine learning models since *concrete-ml* library is written to be as similar as possible to *scikit-learn*.

For the purpose of data analysis and manipulation, the software version of *Pandas* being employed is 2.1.1. It is widely used and is a popular tool for data analysis because of its ease of use and powerful features.

The Python module named *time* version 3.12.1 is used to collect data related to the computational time of an operation. The function `perf_counter_ns()` returns the integer value in nanoseconds of the *performance counter* which is essentially a clock with the highest available resolution to measure short durations. It is important to note that the reference point of the returned value depends on the implementation of Python and the architecture of the CPU, so only the difference between the results of two calls is a valid measure.

Operating system	GNU/Linux Ubuntu 20.04.4 LTS
Kernel release	5.4.0-122-generic
Python	3.9.10
pipenv	23.3.2
wheel	0.41.2
setuptools	65.6.3
concrete-python	2.4.0
scikit-learn	1.1.3
pandas	2.1.1
concrete-ml	1.1.0
time	3.12.1
CPU	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
RAM	16 GB

Table 5.1. Specification of test platform.

After collecting the data, all analyses and results were processed employing Microsoft *Excel* from the Microsoft 365 suite.Chapter 5.2.4 covers the various stages of data preparation for analysis.

The following chapters will present comprehensive results categorized by the analyzed operation types.

5.2 Strategies

In the field of computer science, an exhaustive understanding of the fundamental operations that underlie any computation and algorithm is crucial. For this reason, analyses were conducted on these functions, starting with the simplest ones which are addition, subtraction, multiplication, and division.

[3]The term KNN refers to "K-Nearest Neighbour". It's a supervised machine-learning algorithm that can solve classification and regression problems. The number of nearest neighbours that need to be predicted or classified for a new unknown variable is expressed by the symbol 'K'.

The aim of these analyses was to study how applying FHE to these types of processes can modify their behaviour with respect to computational cost and accuracy.

In addition to these basic operations, further analyses were conducted on power and square root which are important in many areas of computer science, such as cryptography, data compression, and image processing.

Overall, through these analyses, the implications of applying FHE to a system are presented. This investigation delves into how FHE can impact the behaviour of the system in question, in order to provide valuable insights into practical applications.

5.2.1 Configuration Objects

This section will delve into the specifics of the configuration that has been selected and provide a complete explanation of its features and functionalities in the context of this work.

Each test was conducted aiming to compare performance by evaluating the execution time of the operations in three ways:

- *regular operations* using the base function implemented by default on Python. For instance, the addition is performed in this way: `sum = a+b` where a and b are two integers;
- *FHE operation* which are the equivalent operation in FHE with default configuration. Figure 5.1 shows the configuration object used in this part of the analysis.

  It is possible to notice that the only modifications made to the default configuration are enabling unsafe features to allow the key of the circuit to be saved for analysis of its elements and size. This was done by setting `enable_unsafe_features = True`, `use_insecure_key_cache = True`, and `insecure_key_cache_location = ".keys"`. It is important to mention the other parameters:

  - `show_statistic = True` print the circuit statistics during compilation;
  - `show_graph = True` print computational graph during computation;
  - `show_mlir = True` print MLIR [4] during compilation;
  - `show_optimizer = True` print optimizer output during compilation.

  Concrete Optimizer is a Rust library that finds the best fit cryptographic parameters for a given TFHE circuit. The objective is to reduce the computation time while ensuring security and error limitations are met;
- *FHE operation with parallelism enabled*, by setting properly the configuration object it is possible to enable parallelism during compilation and computation. As is shown in Figure 5.2 is it possible to notice that by modifying the default value of the following parameters it is possible to increase performance in specific cases.

  - `loop_parallelize = True` enable loop parallelization in the compiler;
  - `auto_parallelize = True` enable auto parallelization in the compiler;
  - `dataflow_parallelize = True` enable dataflow parallelization in the compiler;
  - `parameter_selection_strategy = fhe.ParameterSelectionStrategy.MONO` sets how cryptographic parameters are selected. When selection strategy *MONO* is specified, only a single set of parameters would be selected for all encrypted values. This is not ideal because it means some operations will be more costly. For example, in a scenario where 2-bit and 8-bit table lookups are used in a circuit, if *MONO* is used all table lookups would be as costly as 8-bit table lookups.

[4]MLIR (Multi-Level IR) is an intermediate representation of the code that simplifies the design and implementation of code generators, translators and optimizers at different levels of abstraction and also across application domains, hardware targets and execution environments [35].

```
1     fhe.Configuration(
2             enable_unsafe_features=True,
3             use_insecure_key_cache=True,
4             insecure_key_cache_location=".keys",
5             # show_statistics=True,
6             # show_graph=True,
7             # show_mlir=True,
8             # show_optimizer=True,
9             )
```

Figure 5.1. FHE configuration object.

```
1     fhe.Configuration(
2             enable_unsafe_features=True,
3             use_insecure_key_cache=True,
4             insecure_key_cache_location=".keys",
5             # show_statistics=True,
6             # show_graph=True,
7             # show_mlir=True,
8             # show_optimizer=True,
9
10            # to enable parallelization
11            loop_parallelize=True,
12            auto_parallelize=True,
13            dataflow_parallelize=True,
14            parameter_selection_strategy= "mono"
15     )
```

Figure 5.2. FHE configuration object with parallelism enabled.

The default value for the selection strategy is `fhe.ParameterSelectionStrategy.MULTI`, which means that multiple parameters would be selected, to be as optimal as possible in terms of execution time.
According to the Concrete documentation [26], the only possible way to enable loop and data parallelization is to force the value of the selection strategy to *MONO*. In the following chapters, we will analyze the results to determine which is the best configuration to use for the specific mathematical function.

By carefully examining these metrics, we can gain valuable insights into the performance efficiency and accuracy of the operations.

It is important to specify these features because clients cannot choose arbitrary cryptographic parameters in Concrete. Instead, all parameters are set by the optimizer during the circuit's compilation. It is significant to note that key generation depends on circuits, which have different optimal parameters.

5.2.2 Dataset generation

This section describes the selection process for datasets and the reasons behind the choices made.

A randomly generated dataset of elements was used for each operation that was analyzed. The script used to generate the addition and subtraction dataset is shown in Figure 5.3. The range was

```
1
2   vet = [(random.randint(0, 200), random.randint(0, 200)) for _ in
       range(10000)]
3
4   results = {
5       "vector": vet
6   }
7
8   with open('Data_10000.json', 'w') as file:
9       json.dump(results, file)
```

Figure 5.3. Script for sum and sub dataset generation.

set from 0 to 200 because the library allows bitwise operations only up to 16-bit. Otherwise, an overflow error occurred. More stringent ranges have been adopted for mathematical operations, such as power, which result in greatly increased values.

5.2.3 Univariate

As mentioned in Chapter 5.1 concrete support natively only addition, subtraction and multiplication. In order to overcome these constraints it was decided to use an *univariate*[5] function. Concrete provides some extensions, one of these is called `fhe.univariate(function)` which allows wrapping any univariate function to a single table lookup. The code depicted in Figure 5.4 showcases the application of univariate functions. It is worth noting that the function to be executed is enclosed within the univariate function. In this instance, a for loop is written to traverse the vector's components, nevertheless, any function type can be nested inside. Univariate functions work identically to standard functions and require an input set of elements for compilation. Line 25 serves to validate that the two functions behave comparably and yield identical outcomes.

The wrapped function must follow certain rules, which are:

- shouldn't have any side effects. For example, no modification of the global state;
- should be deterministic. For example, no random numbers;
- should have the same output shapes as its input;
- each output element should correspond to a single input element. For example, `output[0]` should only depend on `input[0]`.

If any of the constraints mentioned above are violated, the function's outcome is *undefined*.

The technique permits us to solve the limitations of loops and conditional statements. It does this by encapsulating and blackboxing the original function and evaluating it according to a given input range (for example, `f(0), f(1), f(2), ... , f(2**p - 1)`). Then creates a lookup table, but it comes at a cost. The technique is more complex and takes longer to compute.

5.2.4 Results format

In this section, it will be explained how data were collected and manipulated in order to be suitable for the analysis.

[5] In mathematics, a univariate object involves the use of only one variable.

```
1
2
3   def complex_univariate_function(x):
4
5       def per_element(element):
6           result = 0
7           for i in range(element):
8               result += i
9           return result
10
11      return np.vectorize(per_element)(x)
12
13  @fhe.compiler({"x": "encrypted"})
14  def f(x):
15      return fhe.univariate(complex_univariate_function)(x)
16
17  inputset = [np.random.randint(0, 5, size=(3, 2)) for _ in range(10)]
18  circuit = f.compile(inputset)
19
20  sample = np.array([
21      [0, 4],
22      [2, 1],
23      [3, 0],
24  ])
25  assert np.array_equal(circuit.encrypt_run_decrypt(sample),
        complex_univariate_function(sample))
```

Figure 5.4. Example of univariate function.

```
1
2
3   results = {
4   "results": "name",
5   "times fhe": vet_time_fhe,
6   "times clear": vet_time_clear,
7   "compilation time": comp_time,
8   "key generation time": time_keygen
9   }
```

Figure 5.5. Output data format.

During the process of conducting the analysis, it was necessary to obtain and record detailed data on several key aspects. Specifically, we collected information on the time required for the *computation*, *compilation*, and *key generation* phases. This data was carefully extrapolated and saved for further analysis and review. Time information was collected via the *time* library support offered by Python, and then saved within a JSON file.

In Figure 5.5 it is possible to notice the format of the data after the computation. They are saved in a JSON format with different fields:

- *results* is the arbitrary name of the computation performed;
- *times fhe* is a vector in which are contained the time for each single calculation. Its size

```
1
2
3     with open('results.json') as f:
4         data1 = json.load(f)
5
6     with open('results_p.json') as f:
7         data2 = json.load(f)
8
9     # Generate data frame with Pandas
10    df1 = pd.DataFrame(data1)
11    df2 = pd.DataFrame(data2)
12
13    # Write data frames into two distinct Excel sheets
14    with pd.ExcelWriter('output_p_div.xlsx') as writer:
15        df1.to_excel(writer, sheet_name='Sheet1', index=False)
16        df2.to_excel(writer, sheet_name='Sheet2', index=False)
```

Figure 5.6. Converter script to Excel sheets example.

depends on the dataset it is used;

- *time clear* is a vector in which data on the compilation times of the operation without FHE are saved. Its size is the same as the *times fhe* vector;
- *compilation time* is the time taken to compile the circuit;
- *key generation time* is the time taken to generate the keys for the specific circuit.

In Chapter 5.1, It is mentioned that Microsoft Excel is used to analyze data, generate graphs, and perform calculations on the results. To convert the JSON output file created in Figure 5.5 into two Excel sheets, one with parallelism disabled and one with parallelism enabled, it is used the script in Figure 5.6. The code from lines 2 to 7 is responsible for loading two JSON files that contain critical information regarding computations. Once the data is loaded, the Python library *pandas* is employed, which provides native support and comprehensive integration with Excel. The code then generates an *.xlsx* file, which is an Excel spreadsheet, that is populated with the data from the previously mentioned JSON files. This approach allows for efficient data conversion from the JSON files to the spreadsheet, ensuring that all the necessary information is accurately presented in an easily readable format.

All the results will be set out in the next chapter with all the considerations due to better understand their meanings.

5.3 Evaluation metrics for Machine Learning

This section aims to discuss the evaluation of machine learning algorithms to enable a comparison of performance between those with and without homomorphism enabled. The integration of homomorphism in machine learning algorithms enables secure data processing without the need for decryption, thus ensuring data privacy and confidentiality.

This analysis will enable us to gain insight into how applying FHE in the machine learning process could affect the performance and the results. In the context of this work only supervised learning types, more specifically classification and regression will be considered as mentioned in Chapter 4.1.

These two techniques have different evaluation metrics that will be discussed in detail in Chapter 5.3.1 and 5.3.2.

All the following metrics are available in *scikit-learn* library.

5.3.1 Classification evaluation metrics

There are no strict guidelines on how to evaluate a machine learning algorithm, but certain metrics can be used to assess its performance and features. According to Hossin M. et al. [36], some metrics can be defined to carry out a comparative evaluation.

The most important metric is the *confusion matrix*. As shown in Figure 5.7, *TP* and *TN* are the number of correctly classified positive and negative instances, while *FP* and *FN* are the number of misclassified negative and positive instances [36].

		ACTUAL VALUES	
		Positive	Negative
PREDICTED VALUES	Positive	True Positive (**TP**)	False Negative (**FN**)
	Negative	False Positive (**FP**)	True Negative (**TN**)

Figure 5.7. Confusion matrix for classification.

Based on the concept of *confusion matrix* several further metrics could be generated, Table 5.2 reported a list of the most used metrics for classification evaluation:

- *accuracy* is a metric to measure the ratio of correct predictions over the total number of samples evaluated;

- *error rate* is used to measure the ratio of incorrect prediction over the total amount of instances;
- *sensitivity* is employed to measure the fraction of positive patterns that are correctly classified;
- *specifity* is used to measure the fraction of negative patterns that are correctly classified;
- *precision* measure the positive patterns that are correctly predicted from the total predictive patterns in a positive class;
- *recall* is the fraction of positive patterns that are correctly classified;
- *F-measure* is the harmonic mean between recall and precision.

Metrics	***Formula***
Accuracy (acc)	$\frac{tp+tn}{tp+fp+tn+fn}$
Error Rate (err)	$\frac{fp+fn}{tp+fp+tn+fn}$
Sensitivity (sn)	$\frac{tp}{tp+fn}$
Specificity (sp)	$\frac{tp}{tn+fp}$
Precision (p)	$\frac{tp}{tp+fp}$
Recall (r)	$\frac{tp}{tp+tn}$
F-Measure (FM)	$\frac{2*p*r}{p+r}$

Table 5.2. Metrics for evaluation of classification.

5.3.2 Regression evaluation metrics

In order to evaluate regression algorithms there are several ways, but no strict rules. According to Tatachar V. [37] the best way to compare regression is by following the subsequent metrics:

- *mean squared error (MSE)* or *mean squared deviation* is the squared difference between the actual and predicted values. It measures how the best-fit line is close to the point. The formula is the following:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

  where n is the number of predictions, Y_i are the observed values and $\hat{Y}_i$ are the predicted values;

- *root mean squared error (RMSE)* or *root mean square deviation* is the the root of the *mean squared error (MSE)*. It measures the standard deviation of the errors. The formula is:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}$$

  where n is the number of predictions, Y_i are the observed values and $\hat{Y}_i$ are the predicted values;

- *mean absolute error (MEA)* or *mean absolute deviation* measures the average of the absolute difference between observed and predicted values. The formula is:

$$MEA = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \hat{Y}_i\right|$$

  where n is the number of predictions, Y_i are the observed values and $\hat{Y}_i$ are the predicted values;

- *R-squared (R^2)* or *coefficient of determination* measures the portion of the variance in the dependent variable that can be described by the independent variables. It tells us how good is the fitted line for the model. The formula is:

$$R^2 = 1 - \frac{SSR}{TSS}$$

  where SSR is the *sum of square residuals*[6] and TSS is the *total sum of squares*[7];

- *Adjusted* R^2 as the R^2 provides information on how good is the fit, but it is adapted for the number of predictors for the model. The formula is:

$$R^2_{adj} = 1 - \frac{(1 - R^2)(n - 1)}{n - p - 1}$$

  where R^2 is the *R-squared*, p is the number of predictors and n is the total number of samples.

[6] is the sum of the differences between the dependent value and the mean of the dependent variable [37].

[7] is the sum of squared differences between the observed dependent variables and the overall mean [37].

Chapter 6

Experimental Results

A brief explanation of the hardware and software parts used in the context of this work was explained in Chapter 5.1.

In this chapter, a detailed description of the tests and related analyses is provided. Firstly, general tests are performed and analyzed. These tests are exhaustive in nature and assess how the input set is related to the circuit compilation, as explained in Chapter 6.1. Additionally, time and size in each step of the FHE flow are examined in Chapter 6.2.

Secondly, basic operations such as addition, subtraction, multiplication, division, power, and square root are tested and analyzed in Chapter 6.3.

Finally, some tests on machine learning algorithms will be analyzed in Chapter 6.4.

6.1 Exhaustive test for input set

In this section, it will be analyzed how dependent the circuit compilation is on the input set which is a crucial step in the concrete library as discussed in Chapter 3.3.4. To ensure accuracy in any situation, exhaustive tests will be conducted with the aim of minimizing the amount of data that needs to be entered.

At first, a dataset is generated containing all possible combinations of couples of integers within the range of 0 to 255. This dataset is created to evaluate the accuracy of the values obtained using a simple function that adds two values, both with and without FHE enabled.

The code in Figure 6.1 is responsible for testing and verifying the accuracy of the values using both the normal function without FHE and the one with FHE enabled. The two functions have to produce the same result.

To gain a better understanding of how things behave in varying situations, a series of tests were performed with different configurations of the input set:

1. The input set is a vector of three pairs: [0, 0]; [100, 100]; [255, 255]. It contains the smallest value, the largest value, and an intermediate value. During the test, a *100% accuracy is achieved*;

```
1     for x in vet:
2      clear_evaluation = fun(x[0], x[1])
3      fhe_evaluation = circuit.encrypt_run_decrypt(x[0], x[1])
4      assert (clear_evaluation == fhe_evaluation)
```

Figure 6.1. Exhaustive test.

```
1   def function(x):
2    return x + 42
```

Figure 6.2. Add with a constant value.

2. The input set is a vector consisting of two pairs of values: [0,0] and [100,100]. The values in this set range from the smallest to an intermediate value. However, it is important to note that during testing, a complete accuracy of *100% cannot be achieved*, only 95% is reached. When the values exceed 128, some errors may occur in the precision of the calculation;
3. The input set is a vector consisting of two pairs of values: [100,100] and [255,255]. The values in this set range from an intermediate value to the greatest. It is important to mention that during testing, a complete accuracy of *100% is achieved*;
4. The input set is a vector consisting of two pairs of values: [0,0] and [255,255]. The values in this set range from the smallest to the greatest value. However, it is important to note that during testing, an accuracy of *100% is achieved*;
5. The input set is a vector consisting of one pair of values: [0,0], the smallest value. However, it is important to note that during testing, a compilation error appears, when values are greater than 2, revealing that the circuit expects integers on 1 bit as input but instead inputs of values greater than 2 are greater;
6. The input set is a vector consisting of one pair of values: [100,100], an intermediate value. However, it is important to note that during testing, a complete accuracy of *100% cannot be achieved*. When the values exceed 128, some errors may occur in the precision of the calculation;
7. The input set is a vector consisting of one pair of values: [255,255], the greatest value. However, it is important to note that during testing, an accuracy of *100% is achieved*.

Based on the test results, it was found that the input set can be created using only the largest value that will be entered. This can be done without the need for intermediate or initial values to achieve 100% accuracy in the calculation. A similar test was also conducted for univariate functions that implement a division between two numbers, and the same result was obtained.

6.2 Size and Time

Phase	*Type*	*Time [s]*	*Size [bytes]*
encrypted data	bytes	0.000475	4923
encrypted circuit	Circuit	0.0502	48
encrypted result	bytes	0.000119	4923
result	integer	0.0000417	28

Table 6.1. Add function with data size of 28 bytes and function of 136 bytes.

In this section, the time and size of each step of the FHE flow will be discussed. Figure 6.4 illustrates all the steps related to an FHE flow as described in Chapter 2.1.

The process of FHE computation involves four steps. Firstly, the plain data is encrypted with a public key and transformed into encrypted data to be transmitted for the evaluation process.

Phase	*Type*	*Time [s]*	*Size [bytes]*
encrypted data	bytes	2.93	73851
encrypted circuit	Circuit	0.130	48
encrypted result	bytes	0.352	73851
result	integer	0.000221	128

Table 6.2. Univariate function with data size of 176 bytes and function size of 136 bytes.

Secondly, the circuit compilation begins with the plain function and an example of possible input. The circuit is then compiled, producing an encrypted circuit that is ready to be transmitted alongside the encrypted data for evaluation.

Thirdly, the evaluation process takes place, with the encrypted data and the encrypted circuit serving as inputs. The output is an encrypted result.

Lastly, the encrypted result is decrypted to retrieve the plain result.

This particular test is designed to assess the size and time taken for each step of computation in two distinct scenarios: one with a straightforward addition of a constant, and the other with a univariate function. The latter is especially significant since it has the ability to wrap any function, even those that are not supported by the library, as was discussed in Chapter 5.2.3.

Table 6.1 shows the result related to the function reported in Figure 6.2 and the initial *data* is an integer of 28 bytes and the *function* is 136 bytes. It is possible to notice that the *encrypted data* size and *encrypted result* size are the same because they are the encryption of two integers of the same dimensions and the size of the generated keys is 4864 bytes.

When the function is compiled in a circuit it is transformed in type class *concrete FHE compilation* `circuit.Circuit`, which is an object defined in the library, and its size is 48 bytes. Moreover, the process that requires the most computational time in this case is the circuit compilation.

Table 6.2 illustrates the result related to the function reported in Figure 6.3. The initial *data* is a vector of class *numpy.ndarray* of size 176 bytes and the function has a dimension of 136 bytes.

In this scenario, it can be seen that the size of the result is different because as explained earlier in Chapter 5.1, concrete has limitations on the type of output and input which can only be an integer. Moreover, it can be noticed that the operation that takes more time is the encryption of the vector with the public key.

The size of the keys is 18216 bytes. Additionally, the bootstrapping key is defined for the more complex function with a size of 145584640 bytes.

In conclusion, since the second scenario in which a univariate function was used is generally more complex and works on larger input sizes, the times are all increased as one might have expected. Furthermore, if operations implemented in the library are used times are smaller, instead if univariate functions are used times and the size of the keys generated increases.

6.3 Basic operation

This section aims to analyze and compare the performance of the fundamental operations of addition in Chapter 6.3.1, subtraction in Chapter 6.3.2, multiplication in Chapter 6.3.3, division in Chapter 6.3.4, power in Chapter 6.3.5 and square root in Chapter 6.3.6 using the Concrete library. This investigation proves to be of primary importance since these operations form the core of any computational calculation. The choice to focus on these operations derives from their essential contribution to most computational tasks, both in software development contexts and in data analysis and computational science in general.

Each test was conducted to assess Fully Homomorphic Encryption across three configurations, aiming to compare performance by evaluating the execution time of *regular operations*, *FHE operations*, and those performed with *parallelism enabled*.

```
1     def complex_univariate_function(x):
2      def per_element(element):
3          result = 0
4          for i in range(element):
5              result += i
6          return result
7
8      return np.vectorize(per_element)(x)
```

Figure 6.3. Sum of the elements of the vector.

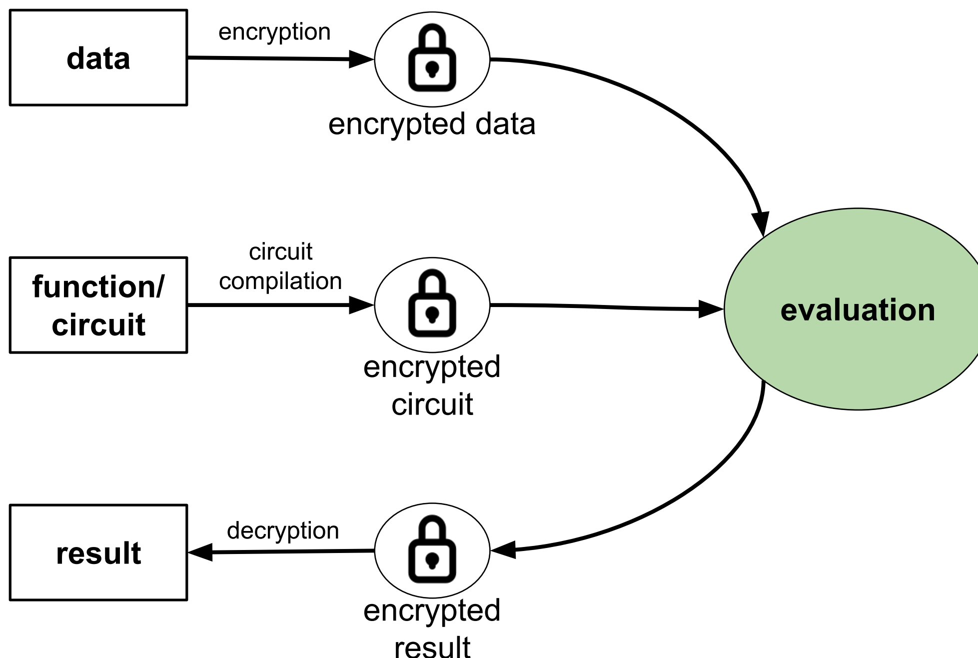

Figure 6.4. Steps involved in a FHE computation.

6.3.1 Addition

In this section, we are going to present the results related to the tests done to analyze in detail the performance of the sum with FHE-enabled using Concrete as mentioned in Chapter 5.1.

The sum holds great significance in the field of cryptography for various reasons. Firstly, it is a fundamental mathematical operation that serves as the basis for several cryptographic algorithms. For instance, the Advanced Encryption Standard (AES) algorithm, which is widely used for symmetric encryption, uses modular addition in combination with other operations like substitution and permutation to ensure the security of data. Secondly, addition is the foundation for more complicated cryptographic concepts such as digital signature and authentication algorithms that require the use of mathematical operations to ensure the authenticity and integrity of messages. Finally, the sum is also used for the generation and verification of one-time passwords (OTP) that provide an additional layer of security in online transactions and access to sensitive systems. Thus, the sum plays a crucial role in ensuring the confidentiality, integrity, and authenticity of

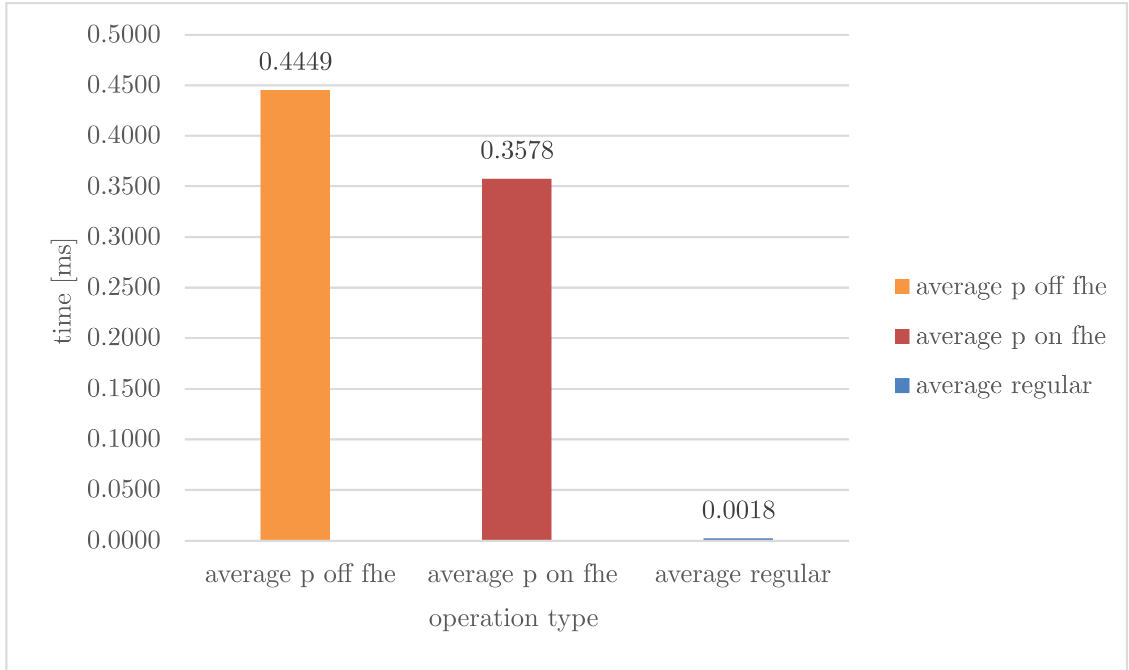

Figure 6.5. Comparison average time addition.

avg parall.off [ms]	*avg parall.on [ms]*	*avg regular [ms]*
0.449	0.358	0.0018

Table 6.3. Average computational time for addition.

cryptographic information.

To effectively test performance, a dataset containing 10000 pairs of values ranging from 0 to 200 was randomly generated using the method described in Chapter 5.2.2. The results obtained from this analysis are shown in Figure 6.5, Table 6.3 and Table 6.4. It can be observed that with active parallelization, the performance of individual sum operations improves by approximately 19.6%. Moreover, the key generation time also sees an improvement, although it remains more or less the same, while the circuit compilation time increases. The significant aspect is the substantial difference in speed between the sum with FHE enabled and the normal sum, with the former being almost three orders of magnitude slower than the latter.

In addition, the distribution of data for each analysis is shown in Figure A.1, A.3, and A.2. The standard deviation for normal FHE is 0.00123, while with parallelization enabled, it is 0.000883. The standard deviation of the data without using FHE is $6.54375E-06$. The standard deviation regarding the results with FHE enabled is very low, this indicates the data have less variability and greater consistency

In conclusion, a possible field that would be interesting to analyze concerns how the compilation and key generation times change and how the circuit is generated for the different operations presented in the chapter on mathematical operations. Table 6.5 and Figure 6.6 show the time trends for the addition.

Especially from Figure 6.6, it can be seen that what is most affected by changing the range of the input set is the key generation time, which increases exponentially. In contrast, all other times stay comparable.

CompTime p off [s]	*CompTime p on [s]*	*KeyGen p off [s]*	*KeyGen p on [s]*
0.114	0.279	0.000513	0.000325

Table 6.4. Compilation and key generation time for addition.

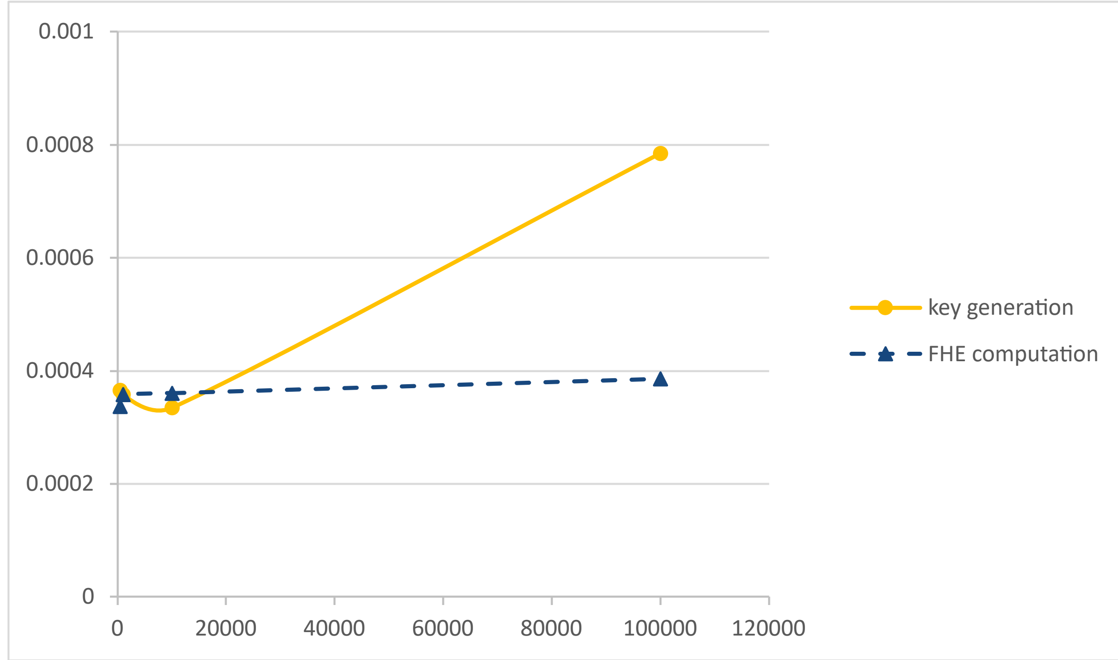

Figure 6.6. Performance trend for addition.

6.3.2 Subtraction

In this chapter, it will examine the performance of the subtraction operation. Subtraction is a crucial operation in computer science, as it plays a key role in various contexts, from binary arithmetic to cryptographic algorithms. Therefore, it contributes to the efficiency and security of computer systems. This is exactly why it is interesting to investigate and analyze the results.

The dataset used in Chapter 6.3.1 is the same one used here, which includes 10000 pairs of integers ranging from 0 to 200, collecting as much data as possible to conduct the analysis. The results of the analysis are presented in Figure 6.7, Table 6.6 and Table 6.7. It is evident from the results that enabling parallelization leads to an average increase in performance of individual sub by approximately 9.9%. This, in turn, results in a decrease in both compile time and key generation time. Therefore, it can be concluded that with active parallelization, all the most significant operations were faster on average. As mentioned in Chapter 6.3.1, the performance of FHE is significantly slower compared to the operation carried out without FHE by about 3 orders of magnitude.

The data distribution is illustrated in Figure A.4, A.5 and A.6. The standard deviation for normal FHE is $0.002719867s$, while the one with parallelization enabled is $0.000583899s$ and the one with FHE disabled is $4.83335E-06s$. It is evident that the standard deviation in the case of FHE operation is very low, indicating great consistency in the data.

6.3.3 Multiplication

This section focuses on the performance of the multiplication operation with FHE enabled. Multiplication is a fundamental operation used in various algorithms and data structures. It is used extensively in many computational operations, such as power elevation, polynomial equations'

range input set	*Compilation time [s]*	*KeyGen time [s]*	*FHE computation time [s]*
[0,500]	0.0795	0.000365	0.000337
[0,1000]	0.165	0.000358	0.000358
[0,10000]	0.076	0.000334	0.000361
[0,100000]	0.172	0.000785	0.000386

Table 6.5. Times trend for addition.

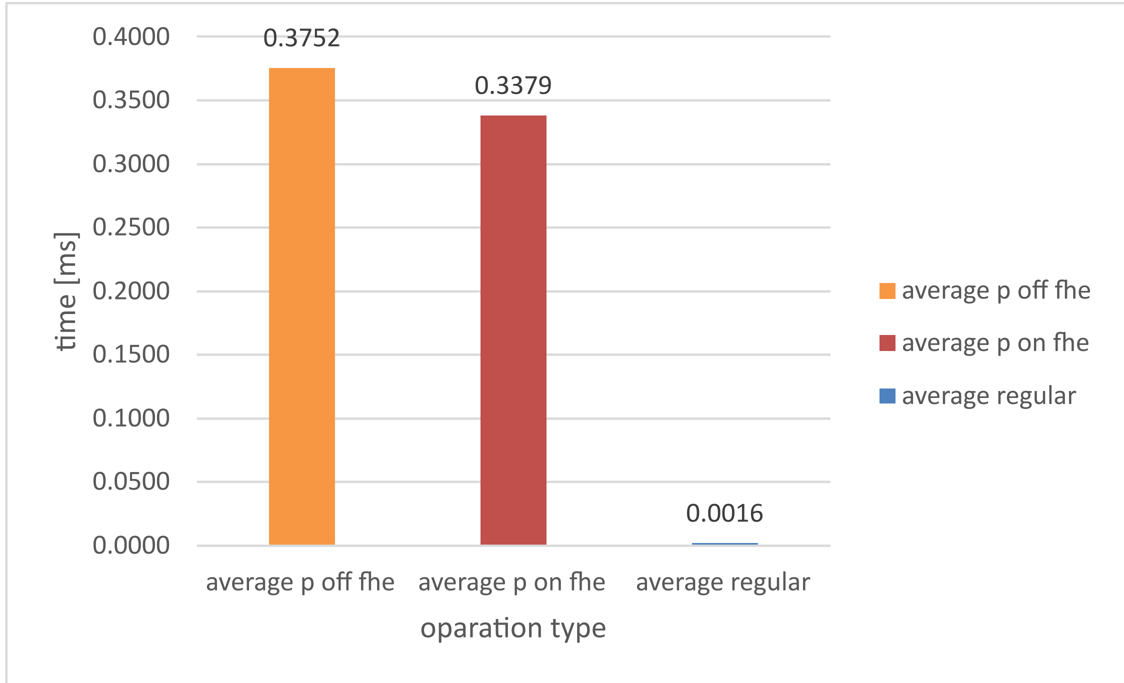

Figure 6.7. Comparison average time subtraction.

coefficient calculation, and the Fourier transform. The computational complexity of algorithms is often represented in terms of multiplication operations, which is crucial in algorithm design, optimization, and performance analysis. Moreover, multiplication is essential in computer security, as several cryptographic algorithms use it to perform mathematical computations.

The dataset contains 1000 pairs of integers between 0 and 100 to avoid exceeding result dimensions, as explained in Chapter 5.1.

The analysis results are presented in Figure 6.8, Table 6.8, and Table 6.9. Multiplication is a more complex operation than addition and subtraction, which were explained in Chapters 6.3.1 and 6.3.2, respectively. As a consequence, the computation time increases significantly, and there is a significant difference between the operations without fully homomorphic encryption (FHE) enabled and those with FHE enabled.

It is observable that the compilation and key generation times are comparable, but the execution times are higher with parallelization enabled than with parallelization disabled. This can be explained by the fact that enabling parallelization demands selecting the parameter selection strategy to multi as explained in Chapter 5.2.1, which causes the circuit to be represented in a different way. This leads to decreased performance, specifically in the case of multiplication.

Therefore, it could be possible to conclude that parallelization with concrete does not always lead to increased performance. It depends on the type of operation and input provided to the circuit. In this case, the performance decreases by about 26.9%.

Data distribution is illustrated in Figure A.8, A.7 and A.9. The standard deviation is $0.195609051s$

avg parall.off [ms]	*avg parall.on [ms]*	*avg regular [ms]*
0.3752	0.3379	0.0016

Table 6.6. Average computational time for subtraction.

CompTime p off [s]	*CompTime p on [s]*	*KeyGen p off [s]*	*KeyGen p on [s]*
0.877	0.207	0.000366	0.000687

Table 6.7. Compilation and key generation time for subtraction.

for normal FHE, $2.81083E-05s$ for parallelization enabled and $0.162656984s$ for the one with FHE disabled. Compared to addition and subtraction there is a greater variability of the data.

6.3.4 Division

This section focuses on examining the performance of the division operation using FHE. Division is a crucial operation in computer science as it plays a fundamental role in data manipulation, algorithm design, data structures, and numerical analysis. However, division is significantly more complex than addition, subtraction, and multiplication, making it interesting to investigate and analyze its performance in the context of this work.

As mentioned in Chapter 5.2.3, the concrete library does not directly implement division. Instead, it is necessary to use the univariate function to wrap its functionality. Unfortunately, this results in a loss of performance, and the times analyzed in the rest of the chapter are considerably higher compared to the regular operation in Python.

The analysis utilizes a dataset of 1000 pairs of integers ranging from 1 to 100. This choice is due to the limitations of the library, as mentioned in Chapter 6.3.3.

The results are presented in Figure 6.9, Table 6.10, and Table 6.11. It is apparent that a single division taking an average of 35 seconds is quite high. However, parallelization can increase the speed by up to 31%. Nevertheless, the average time remains relatively high.

Compared to the previous operation analyzed, compilation times are higher due to the increased complexity of the univariate functions. However, parallelization can help in speeding up the process. Key generation time is relatively the same for both parallelization-enabled and disabled modes.

Data distribution is presented in Figure A.10, A.11, A.12. The standard deviation for normal FHE is 1.056162337 seconds, while the one with parallelization enabled is 0.683154054 seconds. The standard deviation with FHE disabled is $5.49634E-05$ seconds.

6.3.5 Power

In this section, we focused on analyzing the performance of power in the context of full homomorphism. Power is a crucial operation in various fields such as mathematical expressions, computational efficiency, signal processing, encrypting, algorithm design, and computer graphics. It plays a significant role in solving complex problems.

The dataset used in this analysis consists of 1000 pairs of integers ranging from 1 to 15. The results are presented in Figure 6.10, Table 6.12, and Table 6.13. The results obtained are comparable with those achieved for division, as discussed in Chapter 6.3.4. However, due to the univariate function, computation times are quite high. Nevertheless, parallelization increases the performance by 30.4% for the average computation of the operation and compilation time of the circuit. Key generation time remains more or less constant.

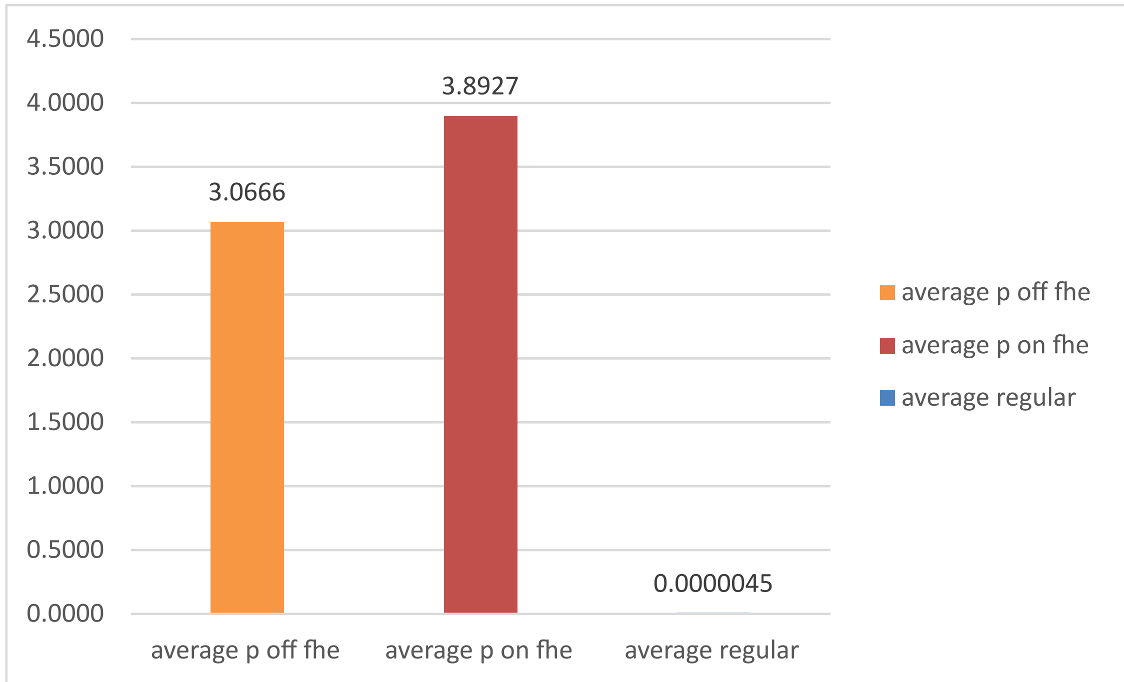

Figure 6.8. Comparison average time multiplication.

avg parall.off [s]	*avg parall.on [s]*	*avg regular [s]*
3.066	3.893	0.0000045

Table 6.8. Average computational time for multiplication.

Data distribution is presented in Figure A.13, A.14, A.15. The standard deviation for normal FHE is 1.388180959 seconds, while the one with parallelization enabled is 0.523836067 seconds. The standard deviation with FHE disabled is $3.44292E-05$ seconds.

6.3.6 Square Root

This section explores the analysis of square root operations in the context of full homomorphism. Square roots are fundamental in computer science as they are used in various computations such as mathematical calculations, numerical methods, signal processing, error analysis, geometric computation, and machine learning.

To collect as much data as possible to infer the average computation time, we used a dataset of 1000 random integers ranging from 0 to 255.

The results, which are presented in Figure 6.11, Table 6.14, and Table 6.15, are comparable to those achieved for multiplication, as discussed in Chapter 6.3.3. The time with parallelization enabled is higher by 15.5%, whereas the time for the compilation of the circuit and the key generation are comparable.

Data distribution is presented in Figure A.16, A.17, A.18. The standard deviation for normal FHE is 0.078124487 seconds, while the one with parallelization enabled is 0.105040925 seconds. The standard deviation with FHE disabled is $7.92562E-05$ seconds.

CompTime p off [s]	*CompTime p on [s]*	*KeyGen p off [s]*	*KeyGen p on [s]*
0.153	0.260	309.6	316.2

Table 6.9. Compilation and key generation time for multiplication.

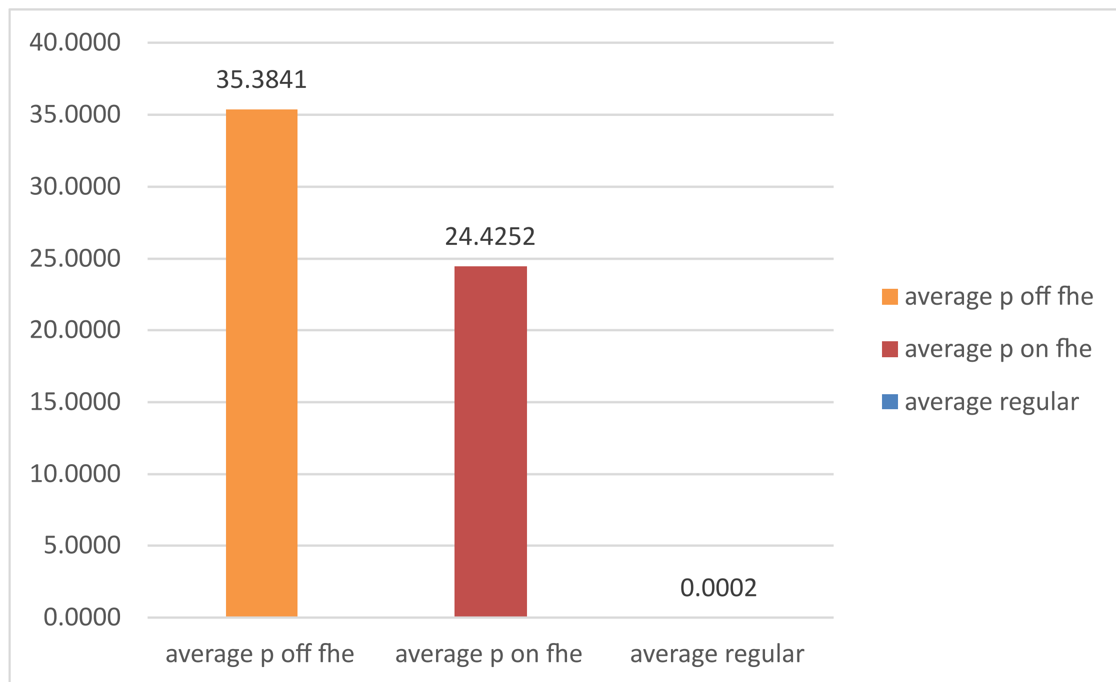

Figure 6.9. Comparison average time division.

6.4 ML operation

This section aims to analyze and compare the performance of some machine learning algorithms such as *logistic regression* in Chapter 6.4.1, *linear support vector regression* in Chapter 6.4.2 and *linear SVM classification* in Chapter 6.4.3.

This investigation not only compares algorithm execution times but also analyzes algorithm evaluation metrics as explained in Chapter 5.3.1 and 5.3.2, between those using *scikit-learn* and *concrete-ml*.

6.4.1 Logistic Regression

This section delves into the analysis of *logistic regression* which is a machine learning classification algorithm as explained in Chapter 4.2.1.

The dataset used for the investigation is built using `make_classification`[1] as shown in Figure 6.13 with 1000 samples and 2 different classes. This method is written in the scikitlearn library and allows the generation of a random n-class classification problem.

The figure in 6.12 displays the time it takes to make predictions using scikit-learn and concrete-ml libraries. However, it is important to note that concrete-ml requires circuit compilation which takes a significant amount of time. Therefore, to determine the overall time it takes to use the

[1] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

avg parall.off [s]	*avg parall.on [s]*	*avg regular [s]*
35.38	24.42	0.0002

Table 6.10. Average computational time for division.

CompTime p off [s]	*CompTime p on [s]*	*KeyGen p off [s]*	*KeyGen p on [s]*
9.44	1.15	25.14	26.15

Table 6.11. Compilation and key generation time for division.

FHE library, the time for circuit compilation and prediction using concrete-ml must be added. These values are significantly higher than those obtained using the scikit-learn library.

The confusion matrix generated using scikit-learn is as follows:

	Predicted Value	
Actual Value	$TP = 181$	$FN = 13$
	$FP = 1$	$TN = 205$

On the other hand, the confusion matrix created using concrete-ml is:

	Predicted Value	
Actual Value	$TP = 182$	$FN = 12$
	$FP = 1$	$TN = 205$

Although there is a slight difference between false positives and true negatives, the two matrices are very similar.

In addition in Table 6.16 are reported the calculations of the other comparison metrics presented in Chapter 5.3.1. It can be noticed that there is a little difference in the two calculations related to metrics.

6.4.2 Linear Support Vector Regression

This section explores the analysis of *linear support vector regression* which is a machine learning regression algorithm as mentioned in Chapter 4.2.2.

It is used `load_diabets`[2] dataset from scikit-learn library which is a dataset for regression and includes 442 samples and has a dimensionality of 10.

Figure 6.14 reported the time to make predictions using the two libraries. As mentioned in Chapter 6.4.1 to calculate the total time for the computation with concrete, the time value and the compilation time with full homomorphism needs to be added.

Furthermore, in Table 6.17 are presented the results regarding the evaluation metrics for regression based on the one written in Chapter 5.3.2. Although the values are not exactly the same, they are still very close and similar.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_diabetes.html

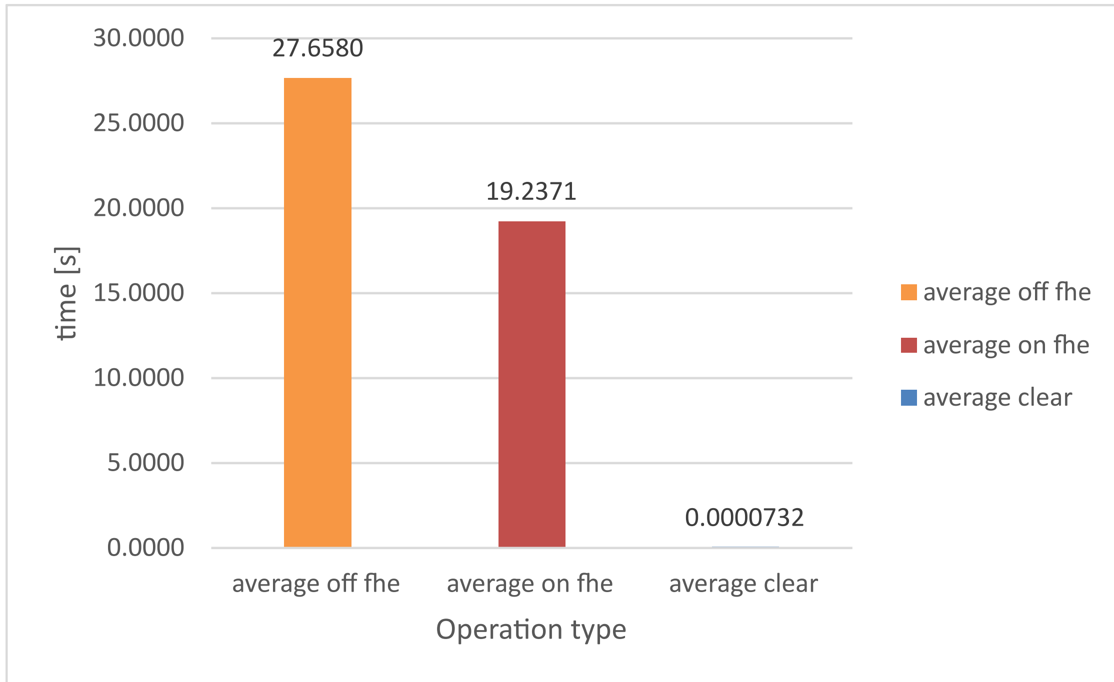

Figure 6.10. Comparison average time power.

avg parall.off [s]	*avg parall.on [s]*	*avg regular [s]*
27.66	19.23	0.000073

Table 6.12. Average computational time for power.

6.4.3 Linear SVM Classification

This section delves into the analysis of *linear SVM classification* which is a machine learning classification algorithm as explained in Chapter 4.2.1.

The dataset used for the investigation is the same used in Chapter 6.4.1 and shown in Figure 6.13 with 1000 samples and 2 different classes.

The figure in 6.15 displays the time it takes to make predictions using scikit-learn and concrete-ml libraries. The same observations written in previous Chapters 6.4.1 and 6.4.2 apply here as well.

The confusion matrix generated using scikit-learn is as follows:

	Predicted Value	
Actual Value	$TP = 87$	$FN = 2$
	$FP = 2$	$TN = 109$

On the other hand, the confusion matrix created using concrete-ml is:

CompTime p off [s]	*CompTime p on [s]*	*KeyGen p off [s]*	*KeyGen p on [s]*
13.9	0.549	17.53	18.4

Table 6.13. Compilation and key generation time for power.

Figure 6.11. Comparison average time square root.

	Predicted Value	
Actual Value	$TP = 86$	$FN = 1$
	$FP = 3$	$TN = 110$

Although there is a slight difference between false positives and true negatives, false negatives and true positives, the two matrices are mostly equivalent.

Furthermore, in Table 6.18 are reported the calculations of the other comparison metrics presented in Chapter 5.3.1. It can be noticed that there is a little difference in the calculations.

avg parall.off [s]	*avg parall.on [s]*	*avg regular [s]*
1.167	1.348	0.0000331

Table 6.14. Average computational time for square root.

CompTime p off [s]	*CompTime p on [s]*	*KeyGen p off [s]*	*KeyGen p on [s]*
0.1273	0.2312	144.1	149.8

Table 6.15. Compilation and key generation time for square root.

Figure 6.12. Comparison time logistic regression.

metrics	*scikit-learn*	*concrete-ml*
accuracy	0.965	0.967
error rate	0.350	0.0325
recall	0.932	0.938
specificity	0.995	0.995
sensitivity	0.933	0.938
precision	0.994	0.994
F-score	0.963	0.965

Table 6.16. comparison of evaluation metrics logistic regression .

```
1     X, y = make_classification(
2      n_features=30,
3      n_redundant=0,
4      n_informative=2,
5      random_state=42,
6      n_clusters_per_class=1,
7      class_sep=2,
8      n_samples=1000,
9  )
```

Figure 6.13. Dataset for logistic regression analysis.

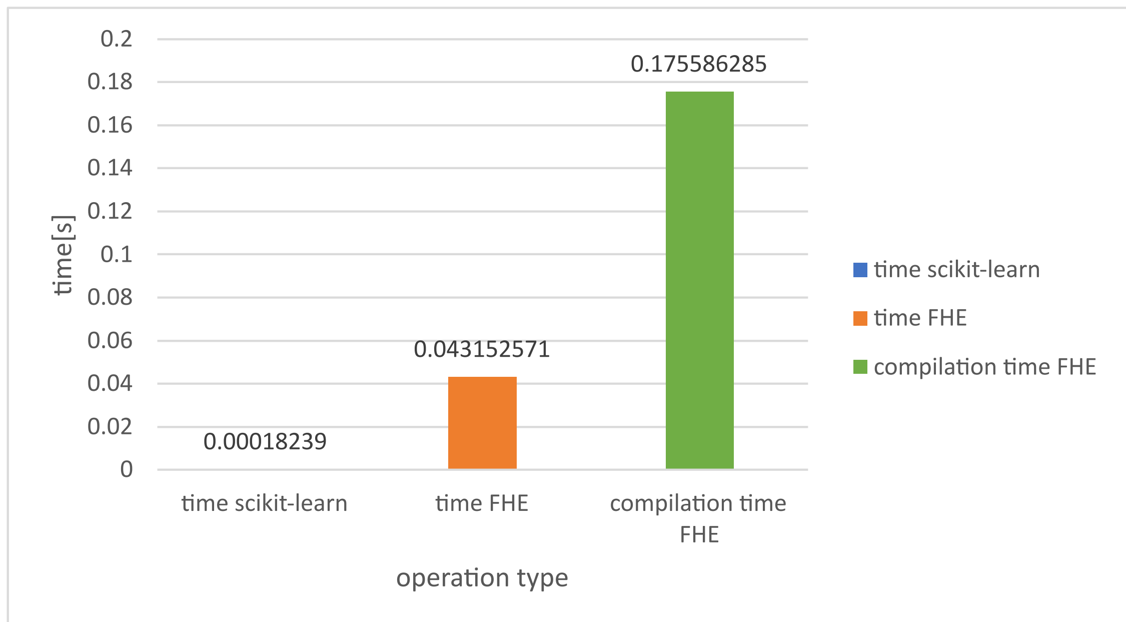

Figure 6.14. Comparison time linear support vector regression.

metrics	*scikit-learn*	*concrete-ml*
MSE	3802.299	3787.242
RMSE	61.663	61.541
MEA	49.305	49.369
R^2	0.3243	0.3270
$adj - R^2$	0.321	0.321

Table 6.17. comparison of evaluation metrics linear support vector regression.

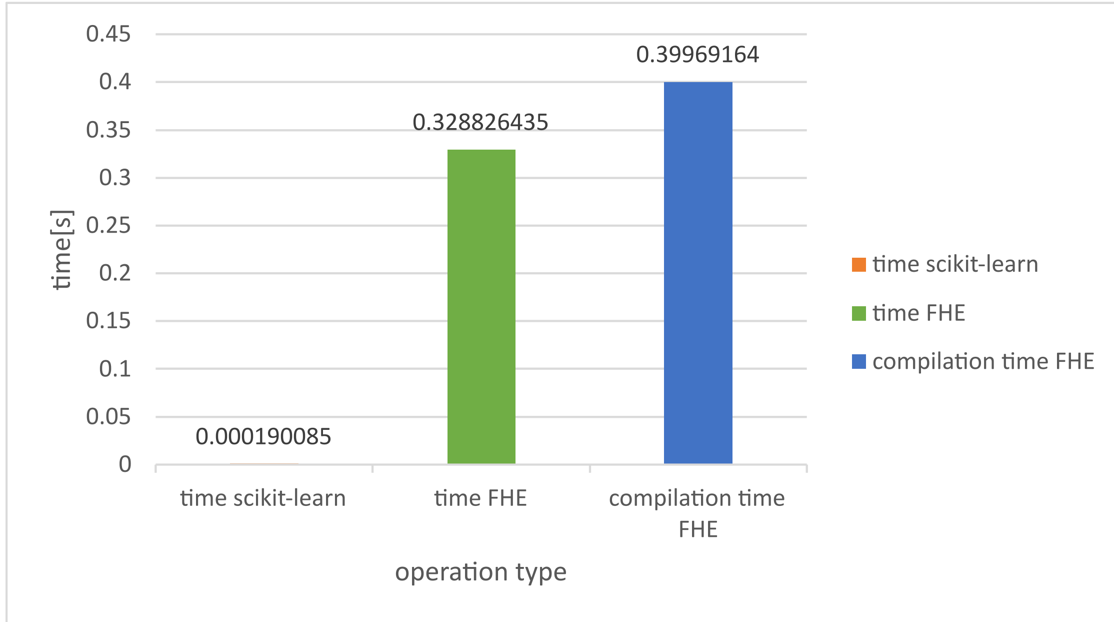

Figure 6.15. Comparison time linear SVM classification.

metrics	*scikit-learn*	*concrete-ml*
accuracy	0.970	0.980
error rate	0.0200	0.0200
recall	0.943	0.966
specificity	0.991	0.991
sensitivity	0.966	0.943
precision	0.988	0.989
F-score	0.966	0.977

Table 6.18. comparison of evaluation metrics linear SVM classification.

Chapter 7

Conclusion

This thesis work highlights how homomorphism applied to cryptography and cybersecurity, in general, is a noteworthy topic that can bring significant improvements, especially in the field of data privacy. In this scenario, it is important to focus on the concept of fully homomorphic encryption that allows an unlimited number of homomorphic operations an unlimited number of times.

The present work consists of an analysis of the performance resulting from the application of FHE on some basic operations regarding mathematics and some machine learning algorithms.

A brief representation of the analysis results is presented in Table 7.1 and Table 7.2. It is evident that the application of FHE to these operations causes a significant increase in computation time for the nowadays available libraries. However, when implementing additional security mechanisms, a tradeoff must always be evaluated between security and performance.

Operation type	*Average p off [s]*	*Average p on [s]*	*Average regular[s]*
addition	0.000449	0.0003578	0.0000018
subtraction	0.0003752	0.0003379	0.0000016
multiplication	3.066	3.893	0.0000045
division	35.38	24.42	0.0002
power	27.66	19.23	0.000073
square root	1.167	1.348	0.0000331

Table 7.1. Mathematical operation results.

ML algorithm	*concrete-ml [s]*	*compilation time FHE [s]*	*scikit-learn [s]*
Logistic Regression	0.6801	0.6467	0.002579
Linear SVR	0.04315	0.1755	0.0001823
linear SVM classification	0.3288	0.3996	0.0001901

Table 7.2. Machine learning algorithms results.

Considering the analysis and results outlined in this study, it is clear that there are numerous opportunities for further investigation and development in the field of fully homomorphic cryptography and application development using the Zama Concrete library. So it would certainly be interesting to go and analyze the performance of other mathematical operations from the approach that was used in this work. Furthermore, it would be interesting to analyze the same mathematical operations implemented in the other libraries that have been presented in the previous chapters.

When it comes to homomorphic libraries that implement machine learning algorithms, it would be beneficial to run a comprehensive test on all libraries that are well-developed. This would enable us to assess the performance of each algorithm and determine which ones are most suitable for the specific needs. Therefore, it is important to ensure that the library it was chosen is efficient, accurate, and secure.

Consequently, it is possible to analyze all the algorithms implemented in the machine learning library and explained in Chapter 4.2.1 and 4.2.2 which are:

- *linear models*[1]: poison regression, TweedieRegressor, GammaRegressor, Lasso, Ridge;
- *tree-based models*[2]: decision tree classifier and regressor, random forest classifier and regressor;
- *neural network*[3] for classification and regression;
- *nearest neighbors*[4].

[1] https://docs.zama.ai/concrete-ml/built-in-models/linear

[2] https://docs.zama.ai/concrete-ml/built-in-models/tree

[3] https://docs.zama.ai/concrete-ml/built-in-models/neural-networks

[4] https://docs.zama.ai/concrete-ml/built-in-models/nearest-neighbors

Bibliography

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation", ACM Computing Surveys (Csur), vol. 51, July 2018, pp. 1–35, DOI 10.1145/3214303

[2] R. L. Rivest, L. Adleman, M. L. Dertouzos, *et al.*, "On data banks and privacy homomorphisms", Foundations of secure computation, vol. 4, October 1978, pp. 169–180

[3] C. Gentry, "A fully homomorphic encryption scheme", Stanford university, 2009

[4] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on Fully Homomorphic Encryption, Theory, and Applications", Proceedings of the IEEE, vol. 110, October 2022, pp. 1572–1609, DOI 10.1109/JPROC.2022.3205665

[5] T. V. T. Doan, M.-L. Messai, G. Gavin, and J. Darmont, "A survey on implementations of homomorphic encryption schemes", The Journal of Supercomputing, vol. 79, September 2023, pp. 15098–15139, DOI 10.1007/s11227-023-05233-z

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, vol. 21, February 1978, pp. 120–126, DOI 10.1145/359340.359342

[7] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF Formulas on Ciphertexts", Theory of Cryptography, Cambridge (MA, USA), February 10-12, 2005, pp. 325–341, DOI 10.1007/978-3-540-30576-7_18

[8] M. S. Lee, "Sparse subset sum problem from Gentry-Halevi's fully homomorphic encryption", IET Information Security, vol. 11, January 2017, pp. 34–37, DOI 10.1049/iet-ifs.2015.0263

[9] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully Homomorphic Encryption over the Integers", Advances in Cryptology – EUROCRYPT 2010, Monaco and Nice, (France), May 30 - June 3, 2010, pp. 24–43, DOI 10.1007/978-3-642-13190-5_2

[10] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun, "Batch Fully Homomorphic Encryption over the Integers", Advances in Cryptology – EUROCRYPT 2013, Athens (Greece), May 26-30, 2013, pp. 315–335, DOI 10.1007/978-3-642-38348-9_20

[11] D. Stehlé and R. Steinfeld, "Making NTRU as Secure as Worst-Case Problems over Ideal Lattices", Advances in Cryptology – EUROCRYPT 2011, Tallinn (Estonia), May 15-19, 2011, pp. 27–47. `https://link.springer.com/chapter/10.1007/978-3-642-20465-4_4`

[12] T. V. Carstens, E. Ebrahimi, G. Tabia, and D. Unruh, "Relationships between quantum IND-CPA notions", `https://eprint.iacr.org/2020/596`

[13] R. Canetti, S. Raghuraman, S. Richelson, and V. Vaikuntanathan, "Chosen-Ciphertext Secure Fully Homomorphic Encryption", Public-Key Cryptography – PKC 2017, Amsterdam (The Netherlands), March 28-31, 2017, pp. 213–240, DOI 10.1007/978-3-662-54388-7_8

[14] S. Laur, H. Lipmaa, and T. Mielikäinen, "Cryptographically private support vector machines", 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia (PA, USA), August 20-23, 2006, pp. 618–624, DOI 10.1145/1150402.1150477

[15] S. Park, J. Byun, J. Lee, J. H. Cheon, and J. Lee, "HE-Friendly Algorithm for Privacy-Preserving SVM Training", IEEE Access, vol. 8, March 2020, pp. 57414–57425, DOI 10.1109/ACCESS.2020.2981818

[16] 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", December 2014, DOI 10.1109/TDSC.2013.51

[17] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: Efficient Private Image Classification (or: Learning from the Masters)", Topics in Cryptology – CT-RSA 2019, San Francisco (CA, USA), March 4-8, 2019, pp. 473–492, DOI 10.1007/978-3-030-12612-4_24

[18] tuneinsight, "Lattigo v5", https://github.com/tuneinsight/lattigo

[19] Center for Hardware Intelligence, Privacy & Security (CHIPS), "cuFHE", https://github.com/vernamlab/cuFHE

[20] NuCypher, "NuFHE", https://github.com/nucypher/nufhe

[21] E. Crockett, C. Peikert, and C. Sharp, "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY", 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto (Canada), October 15- 19, 2018, pp. 1020–1037, DOI 10.1145/3243734.3243828

[22] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications", 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, London (United Kingdom), November 11, 2019, pp. 57–68, DOI 10.1145/3338469.3358945

[23] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing", 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Phoenix (AZ, USA), June 22-26, 2019, pp. 142–156, DOI 10.1145/3314221.3314628

[24] T. v. Elsloo, G. Patrini, and H. Ivey-Law, "SEALion: a Framework for Neural Network Inference on Encrypted Data", https://arxiv.org/abs/1904.12840

[25] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE", WAHC 2020 - 8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, [Virtual], France, December 15, 2020

[26] Zama, "Concrete: TFHE Compiler that converts python programs into FHE equivalent", https://github.com/zama-ai/concrete

[27] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "OpenFHE: Open-Source Fully Homomorphic Encryption Library", 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles (CA, USA), November 7, 2022, pp. 53–63, DOI 10.1145/3560827.3563379

[28] L. Cao, "Data Science: A Comprehensive Overview", ACM Computing Surveys, vol. 50, June 2017, pp. 1–42, DOI 10.1145/3076253

[29] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions", SN computer science, vol. 2, January-March 2021, pp. 160–181, DOI 10.1007/s42979-021-00592-x

[30] S. Shalev-Shwartz and S. Ben-David, "Understanding machine learning: From theory to algorithms", Cambridge university press, 2014

[31] J. Han, J. Pei, and H. Tong, "Data mining: concepts and techniques", Morgan kaufmann, 2022

[32] G. H. John and P. Langley, "Estimating Continuous Distributions in Bayesian Classifiers", 2013, https://arxiv.org/abs/1302.4964

[33] IBM, https://www.ibm.com/

[34] Zama, "Concrete ML: a Privacy-Preserving Machine Learning Library using Fully Homomorphic Encryption for Data Scientists", https://github.com/zama-ai/concrete-ml

[35] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law", https://arxiv.org/abs/2002.11054

[36] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations", International journal of data mining & knowledge management process, vol. 5, March 2015, pp. 1–11, DOI 10.5121/ijdkp.2015.5201

[37] A. V. Tatachar, "Comparative Assessment of Regression Models Based On Model Evaluation Metrics", International Journal of Innovative Technology and Exploring Engineering, vol. 8,

September 2021, pp. 853–860. https://www.irjet.net/archives/V8/i9/IRJET-V8I9127.pdf

[38] A. Hülsing, T. Lange, and K. Smeets, “Rounded Gaussians: fast and secure constant-time sampling for lattice-based crypto”, IACR International Workshop on Public Key Cryptography, Rio de Janeiro (Brazil), March 25-29, 2018, pp. 728–757, DOI 10.1007/978-3-319-76581-5_25

[39] D. Micciancio and O. Regev, “Worst-case to average-case reductions based on Gaussian measures”, SIAM Journal on Computing, vol. 37, no. 1, 2007, pp. 267–302, DOI 10.1137/S0097539705447360

[40] D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa, “Efficient public key encryption based on ideal lattices”, International Conference on the Theory and Application of Cryptology and Information Security, Tokyo (Japan), December 6-10, 2009, pp. 617–635, DOI 10.1007/978-3-642-10366-7_36

Appendix A

Distribution of data

A.1 data distribution addition

This section lists all the graphs related to the distribution of data regarding the sum tests.

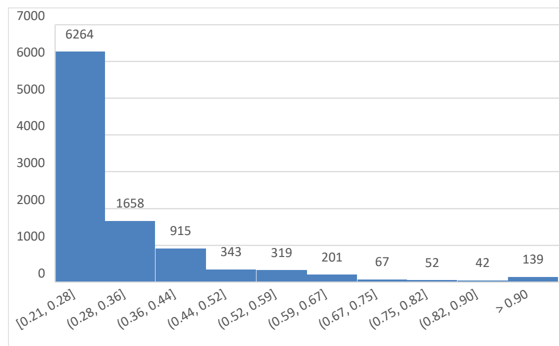

Figure A.1. Time distribution of the data with parallelization on for addition.

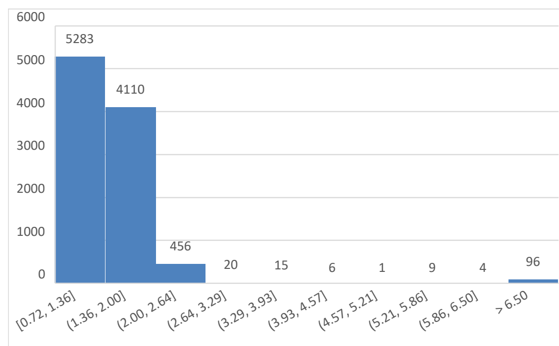

Figure A.2. Time distribution of the data without FHE enable for addition.

A.2 data distribution subtraction

This section lists all the graphs related to the distribution of data regarding the subtraction tests.

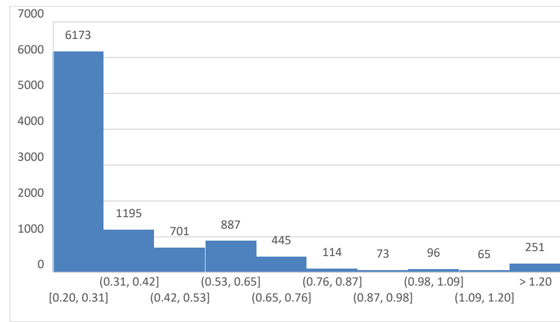

Figure A.3. Time distribution of the data with parallelization off for addition.

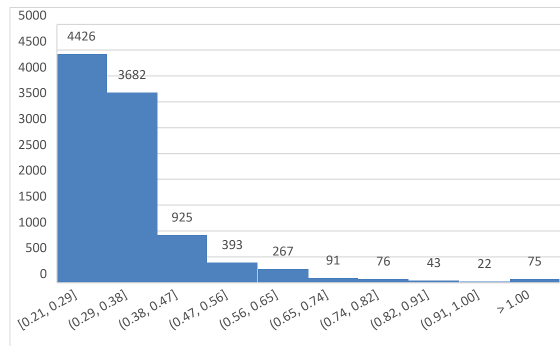

Figure A.4. Time distribution of the data with parallelization off for substraction.

A.3 data distribution multiplication

This section lists all the graphs related to the distribution of data regarding the multiplication tests.

A.4 data distribution division

This section lists all the graphs related to the distribution of data regarding the division tests.

A.5 data distribution power

This section lists all the graphs related to the distribution of data regarding the power tests.

A.6 data distribution square root

This section lists all the graphs related to the distribution of data regarding the square root tests.

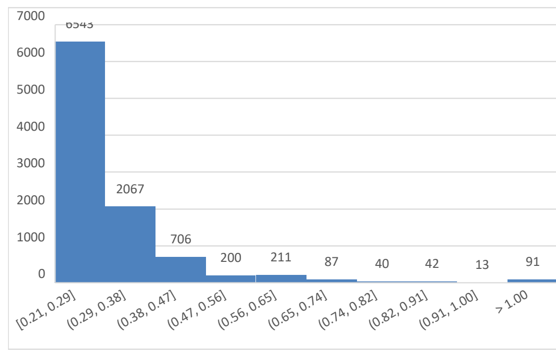

Figure A.5. Time distribution of the data with parallelization on for substraction.

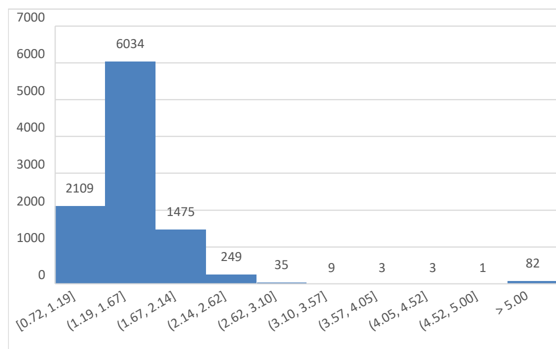

Figure A.6. Time distribution of the data without FHE enable for substraction.

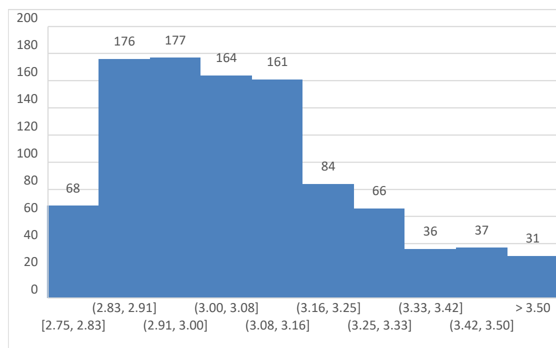

Figure A.7. Time distribution of the data with parallelization off for multiplication.

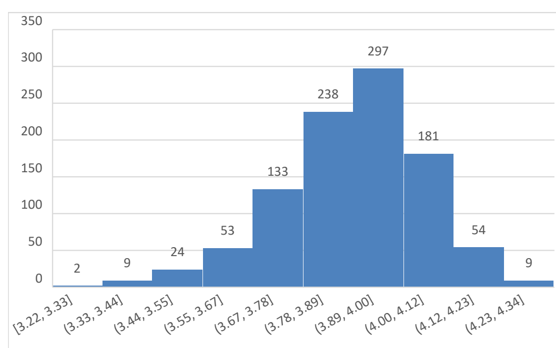

Figure A.8. Time distribution of the data with parallelization on for multiplication.

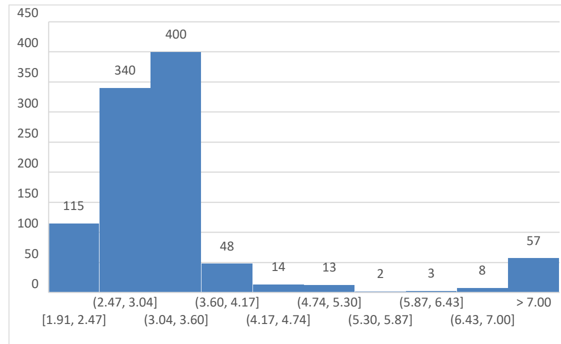

Figure A.9. Time distribution of the data without FHE enable for multiplication.

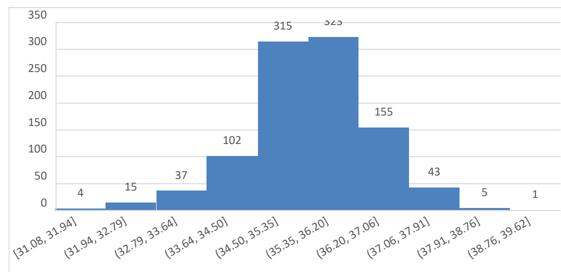

Figure A.10. Time distribution of the data with parallelization off for division.

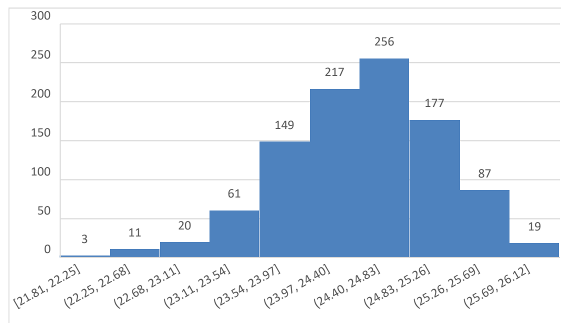

Figure A.11. Time distribution of the data with parallelization on for division.

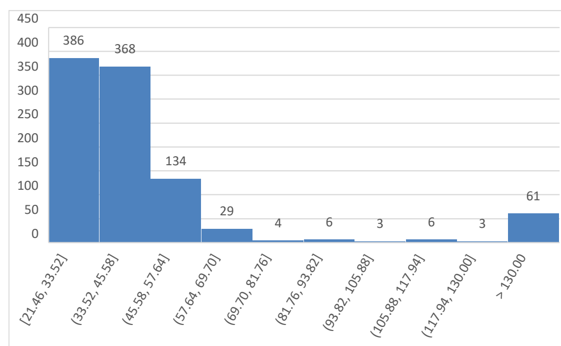

Figure A.12. Time distribution of the data without FHE enable for division.

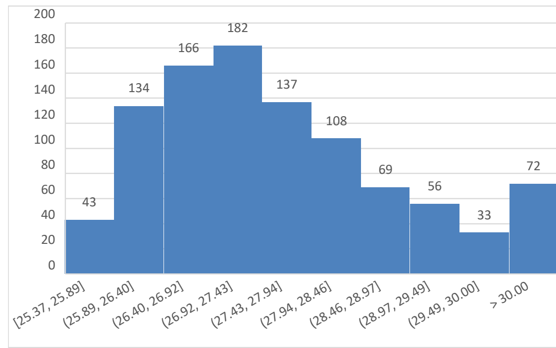

Figure A.13. Time distribution of the data with parallelization off for power.

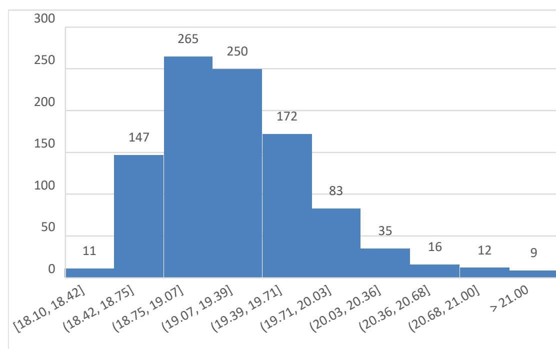

Figure A.14. Time distribution of the data with parallelization on for power.

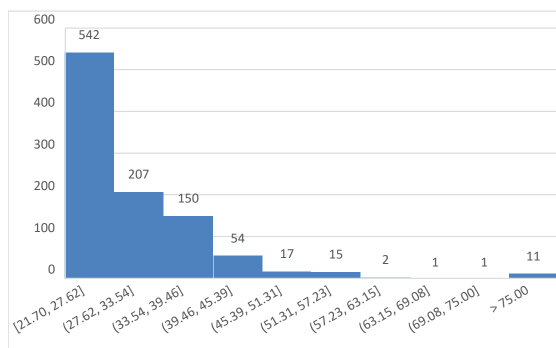

Figure A.15. Time distribution of the data without FHE enable for power.

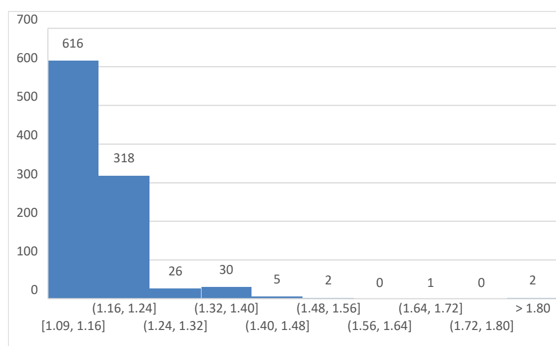

Figure A.16. Time distribution of the data with parallelization off for square root.

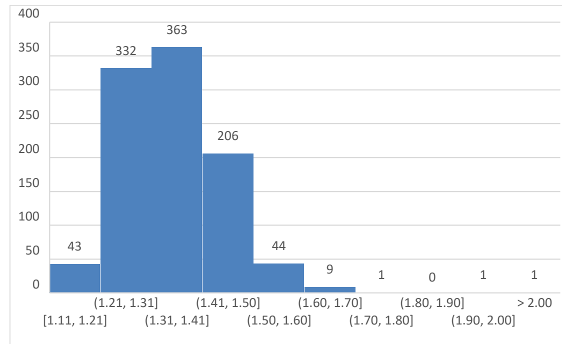

Figure A.17. Time distribution of the data with parallelization on for square root.

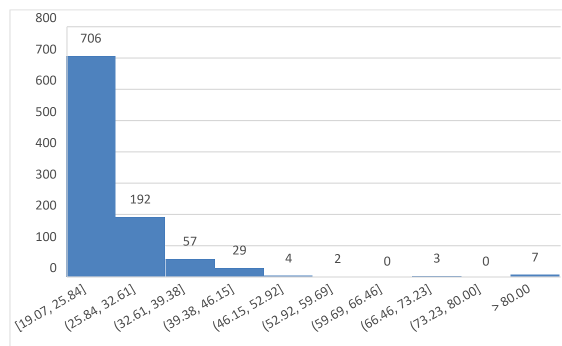

Figure A.18. Time distribution of the data without FHE enable for square root.

Appendix B

Mathematical Foundations

In the pursuit of knowledge and understanding, it is essential to establish a common language that permits clear and concise communication, for this reason, it is necessary to establish a strong foundation by identifying fundamental mathematical terms and concepts.

This chapter serves to define the essential mathematical foundations, such as number and probability theory as well as other related concepts, necessary for a thorough understanding of the discussions that follow.

B.1 Number Theory

According to the paper titled Marcolla et al. [4], the key definitions that hold significant value in the field of Number Theory are:

- A group G is a set that has an associative operation. This means that the order of operation does not matter. In addition, there exists an identity element of G and every element of G has an inverse. An identity element is an element that, when combined with another element of the group, returns that same element. An inverse element is an element that, when combined with another element of the group, returns the identity element. If the group operations are commutative, meaning that the order of the elements does not matter, the group is called an *abelian* group.

- *Fields*: A field is a set with two operations: addition and multiplication. A set is considered an abelian group under addition, where 0 is the identity element, and its nonzero elements are considered an abelian group under multiplication, where 1 is the identity element. The multiplication operation is distributive over addition. A "finite field" denoted as $\mathbb{F}_q$, is a field that contains q elements, and it exists only if q is a prime or prime power. A *number field* is a vector space that has a finite dimension over the rational number $\mathbb{Q}$.

- *Rings*: A ring is a generalization of a field because multiplication does not need to be commutative, and some elements may not have a multiplicative inverse.

- *Sets*: Let E be a set. It is possible to define E^n as the set of n-tuples $(e_1, \ldots, e_n)$ where each $e_i \in E$.;

- *Norms*: Let x be a vector in the set E. The ℓ-norm of x is defined as $||\boldsymbol{x}||_\ell := (\sum_i |x_i|^\ell)^{1/\ell}$, where x_i represents the elements in x. The *infinity*-norm of x is defined as $||\boldsymbol{x}||_\infty := max_i|x_i|$. The *Euclidean* norm of x is denoted by $||\boldsymbol{x}||$ and is equivalent to the two-norm. It can also be referred to as the length of a vector.

B.2 Probability Theory

According to the aforementioned paper [4], the essential definitions regarding Probability Theory are:

- *Negligible probability*, in the field of cryptography, the key to achieving provably secure schemes is identifying an attack with a negligible probability of success relative to the security parameter. In this regard, a function denoted by $f(k) : \mathbb{N} \to \mathbb{R}$, where k represents the length of the secret values, is considered negligible if, for any possible integer c, there exists a value N such that, for all $k > N$, it is guaranteed that $|f(k)| < (1/k^c)$. It is noteworthy that if a negligible function produces a probability as its output, that probability is also negligible.;

- *Gaussian Distribution*: the Gaussian distribution $\mathcal{X}$ is a common probability distribution for a random variable $x \in \mathbb{R}$. The distribution is defined by the following function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \text{ [4]}$$

  Here, μ is the mean or center of the distribution and σ is its standard deviation. In the field of Fully Homomorphic Encryption (FHE), the discrete Gaussian distribution $\mathbb{D}_{\mathbb{Z},\alpha q}$ is usually defined as a Gaussian distribution on the integers with center zero and width parameter αq. The distribution assigns a weight proportional to $exp(-\pi x^2/(\alpha q)^2)$ to all $x \in \mathbb{Z}$.. In other words, for any integer x, its weight is determined by the formula $exp(-\pi x^2/(\alpha q)^2)$ [38]

$$\mathbb{D}_{\mathbb{Z},\alpha q} = \frac{f(x)}{f(\mathbb{Z})} where f(\mathbb{Z}) = \sum_{x \in \mathbb{Z}} \frac{1}{\alpha q} e^{-\pi(\frac{z}{\alpha q})^2} \text{ [39].}$$

B.3 Definitions

Within the discourse on mathematical definition, to ensure a comprehensive understanding, it is necessary to include additional fundamental concepts:

B.3.1 Lattice

- $k-$*dimensional lattice*: refers to a discrete additive set of points generated by taking integer linear combinations of a basis set of k linearly independent vectors. Let $B = (b_1, \ldots, b_k)$ be linearly independent vectors in $\mathbb{R}^n$:

$$\mathcal{L} = \mathcal{L}(B) = \left\{ \sum_{i=1}^{k} \gamma_i b_i : \gamma_i \in \mathbb{Z}, b_i \in B \right\} \text{ [4];}$$

- The *paralelepiped* associated with the basis B is defined as:

$$\mathcal{P}(B) = \left\{ \sum_{i=1}^{k} x_i b_i : x_i \in [-1/2, 1/2] \right\} \text{ [4];}$$

- The rank k of the Lattice is defined as $dim(span(\mathcal{L})$;

- The *volume* of the Lattice is defined as: $Vol(\mathcal{L}) = (det(B^t B))^1/2$ [4];

- The *ideal lattice* $\mathcal{L}(\mathcal{I})$ is defined as an integer lattice $\mathcal{L}(B) \in \mathbb{Z}^n$ where $B = g \bmod f : g \in I, I \subseteq \mathbb{Z}[x]/\langle f \rangle$ and f is a monic polynomial of degree n;

- *Lattice Distance* for any vector t in $\mathbb{R}^n$ and any element v of the Lattice: $dist(t, \mathcal{L}) = min\|t - v\| : v \in \mathcal{L}$. In the previous definition, t is defined as the minimum distance between t and any element in the Lattice. This is an essential concept since most common attacks against FHE schemes are based on the notion of *distance*;

- *minimum distance* of Lattice that is the shortest nonzero vector in $\mathcal{L}$: $\lambda_1(\mathcal{L}) = min\{\|v\| : v \in \mathcal{L}, v \neq 0\}$.

B.4 Other Key Concepts

- *Learning With Errors Problem (LWE)* is a fundamental problem in lattice-based cryptography, especially its *decisional* version since it is widely used in cryptography to create secure encryption algorithms. Their definitions are given in the following way, as explained in [4]:

  1. considering $b \in \mathbb{Z}_q^n$ and a matrix $A\ (\mathbb{Z}_q)^{m \times n}$, the LWE problem consist in searching un unknown vector $s \in \mathbb{Z}_q^n$ such that:

     $$As + e = b \; mod \; p;$$

     where e is a sampled coordinate-wise from an error distribution $\mathcal{X}$. The target is to discover a vector $s \in \mathcal{Z}_q^n$ from a set of $m = n + 1$ *noisy* equations from:

     $$A_{s,\mathcal{X}} = \{(a_i, b_i = \langle a_i, s\rangle + e_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q : \; a_i \xleftarrow{\$} \mathbb{Z}_q^n, e_i \xleftarrow{\$} \mathcal{X}\};$$

  2. the *decision learning with errors* (DLWE) problem consists of discerning (with non-negligible advantage) m samples selected according to $A_{s,\mathcal{X}}$ (for uniformly random $s \in \mathbb{Z}_q^n$) from m samples determined according to the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

- The *Ring Learning with Errors Problem (RLWE)* is a cryptographic problem that can be seen as a *ring version* of LWE with some differences, in particular, RLWE is in $(R_q)^2$, where $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$ and $f(x) \in \mathbb{Z}[x]$ is a monic, irreducible polynomial of degree d and q is a prime [4]:

  1. The RLWE problem is to find $s \in R_q$ given access to arbitrarly nomerous independent samples $(a, b = s \cdot a + e) \in R_q \times R_q$;
  2. The *decision ring learning with errors* (DRLWE) problem is to determine with non-negligible advantage between independent and uniformly random samples in $R_q \times R_q$ and the same number of separate RLWE instances [4].

From a cryptographic point of view, a scheme based on the RLWE problem is more computationally efficient and more compact (in terms of ciphertext size) than those based on LWE [40].

In conclusion, establishing a common language of clear and concise communication is crucial to examining the intricacies of Fully Homomorphic Encryption, it is imperative to lay a strong foundation by identifying key mathematical terms and concepts.

Appendix C

User Manual

C.1 Directory structures

The project is divided into two main directories, one contains all the code that concerns testing for mathematical operations and all the analysis related to the Zama Concrete library. The second contains all the code related to Concrete-ml from Zama regarding machine learning algorithms. Figure C.1 shows a graphic representation of the structure.

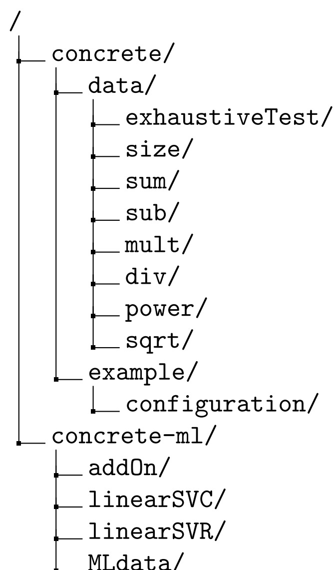

Figure C.1. Project directory structure.

- `data/` contains all the code related to the final tests, both code and results;
- `example/` contains some example tests and an example of configuration object as described in Chapter 5.2.1;
- `addOn/` contains some examples and test benches script;
- `MLdata/` contains script and data related to the tests conducted on the investigated machine learning algorithms;
- `linearSVC/` contains an example and test benches script for linear SVC algorithm;
- `linearSVR/` contains an example and test benches script for linear SVR algorithm.

C.2 Instructions for installation

This section presents a comprehensive guide on how to install the software that are essential to execute the scripts and tests developed in the implementation part of this thesis. All the necessary software and configurations required for testing and code execution can be found in Chapter 5.1. It is noteworthy that the instructions provided below are specifically intended for a *Linux/Debian* environment, as the library currently is not yet compatible with a Windows environment.

- firstly, it is possible to install Python 3.9 in Linux using the following command:

```
$ sudo apt-get install python3.9
```

- it is possible to install scikit-learn in Linux using the following command:

```
$ pip install -U scikit-learn
```

- it is possible to install pandas in Linux using the following command:

```
$ pip install pandas
```

- it is possible to install time in Linux using the following command:

```
$ pip install time
```

- to install concrete, follow the instructions given on the library Github[1] and run the following commands:

```
$ pip install -U pip wheel setuptools
$ pip install concrete-python
```

- to install concrete-ml, follow the instructions given on the library Github[2] and run the following commands

```
$ pip install -U pip wheel setuptools
$ pip install concrete-ml
```

  Concrete-ml is based on concrete so it is essential to install the latter first.

C.3 How to use

This section will describe how to launch the tests and use the code.

For the analysis described in Chapter 6.1 and Chapter 6.2, navigate to the `cocrete/data/size` directory and run the scripts `sumSize.py` and `uniSize.py`. The latter will display all the information about the types and sizes in each step, along with the execution time.

On the other hand, for the analysis in Chapter 6.1, navigate to the `concrete/data/exhaustiveTest` directory. Within this directory, two scripts are available for execution `sumExhaustive.py` and `uniExhaustive.py`, allowing observation of the data collection process for analysis.

[1] https://github.com/zama-ai/concrete

[2] https://github.com/zama-ai/concrete-ml

C.3.1 Mathematical Operations

In this section, there is a step-by-step guide outlining the initiation of tests that encompass various mathematical operations such as addition, subtraction, multiplication, division, power, and square root. All necessary scripts for conducting these tests, gathering data, and converting it into an Excel spreadsheet for further analysis are available in their corresponding subfolders, for example, `/concrete/data/div` for division. The latter includes:

- `divCollection.py` is the script for running and collecting data with parallelization not enabled;
- `divCollection_p.py` is the script for running and collecting data with parallelization enabled;
- `results.json` is the JSON file where the data related to the execution of `divCollection.py` are saved;
- `results_p.json` is the JSON file where the data related to the execution of `divCollection_p.py` are saved;
- `converter.py` is the script that converts data from the above-mentioned JSON file to an Excel spreadsheet;
- `output_p_div.xlsx` is the Excel file with all the data are stored;
- `runTest_div.py` is the script that permits the execution, collection and convert all the data with a single script. it executes in this order: `divCollection.py`, `divCollection_p.py`, `converter.py`.

Two methods are provided for running the tests. The first method involves executing individual scripts for each operation. To proceed, navigate to the folder of the operation it is desired to test and execute the following command, for instance, for the division operation:

```
$ python3 runTest_div.py
```

Executing this command will gather all essential data for that particular operation.

The second method enables the execution of all tests on all operations sequentially, aggregating the data in a single script. Simply navigate to the `concrete/data` folder and execute the following command:

```
$ python3 runTest.py
```

Please note that this option collects data from all tests. However, it's important to acknowledge that this process may require a significant amount of time to complete, especially as some tests are highly time-consuming.

After launching the tests and upon completion, within each folder, you will find an Excel file containing its sheets. One sheet will contain data with parallelization enabled, while the other will depict data with parallelization not enabled.

C.3.2 Machine Learning test

This section outlines the steps required to launch a test for machine learning algorithms. The first step is to navigate to the directory `concrete-ml/MLdata`, as mentioned in Chapter C.1. Once there, three Python scripts `linearSVR.py`, `logisticRegression.py` and `svmClass.py` will be available for use.

There are two methods to conduct the test. The first involves running each of the three tests individually, while the second and more efficient method involves running the following command:

```
$ python3 runTestML.py
```

This command will run each test sequentially and collect the data. The data collected will be stored in specific JSON files: `results_LR_FHE.json` and `results_LR_scikit.json` for logistic regression, `results_SVMC_FHE.json` and `results_SVMC_scikit.json` for linear SVM classification, and `results_lSVR_FHE.json` and `results_lSVR_scikit.json` for linear SVR algorithm.

The format of the file depends on the type of the algorithm used. If it is a classification algorithm, the format is as follows:

```
1     results_fhe = {
2         "results": "logReg FHE",
3         "compilation time": compilation_time_FHE,
4         "computation time": evaluate_time_FHE,
5         "error rate": error_rate_fhe,
6         "accuracy": acc_score_fhe,
7         "recall": recall_fhe,
8         "specificity": specificity_fhe,
9         "sensitivity": sens_fhe,
10        "precision": prec_fhe,
11        "F-score": F_fhe
12    }
```

On the other hand, if it is a regression algorithm, the format is as follows:

```
1     results_fhe = {
2         "results": "linearSVR FHE",
3         "compilation time": compilation_time,
4         "key generation time": key_generation_time,
5         "computation time": execution_time_fhe,
6         "MSE": mse_fhe,
7         "MEA": mea_fhe,
8         "R2": r2_fhe,
9         "adj-R2": adjusted_r2_fhe
10    }
```

Appendix D

Developer Manual

This section will provide a detailed description of the code that was used in the implementation of the thesis. It will also include instructions on how to make modifications to the code as needed.

Specifically, it is divided into three sections: the first section describes the scripts used and provides guidance on how to edit them appropriately in Chapter D.1. The second section in Chapter D.2 explains how to describe tests using Concrete and how to modify them as needed. The third in the Chapter D.3 section addresses the description of tests implemented using concrete-ml.

D.1 Scripts Description

Starting from the description that was provided in Chapter C.1, it will now analyze the various scripts in detail to explain how they work. The directory `concrete/data/` contains all the code related to Concrete library investigation:

- `runTest.py` is the script that allows to run all the script together:

```
1      current_directory = os.getcwd()
2
3      script_files = [['runTest_sum.py', '/sum'],
           ['runTest_sub.py','/sub'], ['runTest_mult.py','/mult'],
           ['runTest_sqrt.py','/sqrt'], ['runTest_div.py','/div'],
           ['runTest_power.py','/power']]
4
5      for script in script_files:
6          new_directory = current_directory + script[1]
7          print(new_directory)
8          os.chdir(new_directory)
9          print('run script', script[0])
10         try:
11             # Call the python command with the name of the current script
12             subprocess.run(["/usr/bin/python3.9", script[0]], check=True)
13         except subprocess.CalledProcessError as e:
14             print(f"script execution error {script}: {e}")
15
16         os.chdir(current_directory)
17
18     print("test execution completed for the mathematical operations")
```

This script accesses all the subdirectories that we include in `script_files` object and launches the respective tests individually. Hence, if it is wanted to add an additional new

written test in this script just add an entry in this object in this format: [name_of_the_script, subdirectory_name].

- runTest_operationName: inside each subdirectory, there is another script that allows running each python script in the correct order to collect and convert the data. The following is an example from the division:

```
1    import subprocess
2
3    script_files = ['divCollection.py', 'divCollection_p.py',
         'converter.py']
4
5    for script in script_files:
6        print('run script', script)
7        try:
8            # Call the python command with the name of the current script
9            subprocess.run(["/usr/bin/python3.9", script], check=True)
10       except subprocess.CalledProcessError as e:
11           print(f"script execution error {script}: {e}")
12
13
14   print("test execution completed for division")
```

  To write a new operation and create the corresponding script, ensure that within the script_files object, you maintain the order of execution as follows, or at least invoke the conversion script last.

Chapter 5 has already covered the explanation of how the other scripts operate with their respective configurations. Additionally, it delves into the distinction between functions directly implemented in the library and the univariate one, encapsulating functionalities that are not directly supported and how the dataset is generated using the corresponding script DataGeneration.py and how the range should be set.

D.2 How to write test with Concrete

In this section, the process of adding and writing new tests for the *Concrete* library will be outlined.

1. move to concrete/data and create a new subdirectory;
2. add or define a dataset;
3. this step involves referring to the library's documentation, specifically the Compatibility[1] section, to verify whether the operation it is intended to analyze is supported or not.

   If the operation is supported you can refer to the sum by modifying only the part that pertains to the function shown below:

```
1        def fun(x, y):
2        return (x + y)
3
4
5    compiler = fhe.Compiler(fun, {"x": "encrypted", "y": "encrypted"})
6
7    input_set = [(random.randint(1, 100), random.randint(1, 100)) for _
         in range(1000)]
```

[1] https://docs.zama.ai/concrete/getting-started/compatibility

In this scenario, it should proceed by modifying the operation returned by the function and potentially adjusting the number of parameters in both the function definition and the `fhe.compiler` definition. Subsequently, it's necessary to update the input set by incorporating parameters consistent with the newly defined function.

If the operation is not supported, it is necessary to use the univariate function as explained in Chapter 5.2.3. The following is an example taken from the division operation:

```
1      def divide(comb):
2          dividend = comb >> 8
3          divisor = comb & 0xFF
4          result = dividend//divisor
5          return result
6
7      def fun(x, y):
8          comb = (x << 8) | y
9          return fhe.univariate(divide)(comb)
10
11
12     compiler = fhe.Compiler(fun, {"x": "encrypted", "y": "encrypted"})
13
14     input_set = [(random.randint(1, 100), random.randint(1, 100)) for _
           in range(100)]
```

In this context, the function `fun` serves as a wrapper for the operation, consolidating the two input arguments into a single variable. This adaptation is necessary because the univariate function can only accept a single argument. If a different number of arguments is desired, the `comb` variable must be adjusted accordingly. Rather than employing the divide function, any alternative function meeting the specified bounds and constraints mentioned in Chapter 5.1 can be defined and enveloped. Finally, updates to both the `fhe.compile` function and the input set are necessary to align with the requirements of the new function;

4. write an equivalent script but with active parallelization as described in Chapter 5.2.1 by enabling in the configuration object:

```
1       loop_parallelize=True,
2       auto_parallelize=True,
3       dataflow_parallelize=True,
4       parameter_selection_strategy= "mono"
```

5. write the script to convert the data by adjusting the code reported in Chapter 5.2.4;

6. run the test in order as explained in Appendix C;

7. write a script on the base of the one presented in Chapter D.1 in order to run the test within a single command.

D.3 How to write test with Concrete-ml

In this section, the process of adding and writing new tests for the *Concrete-ml* library will be outlined.

1. move to `concrete-ml/MLdata`;

2. create a new Python script;

3. this step involves referring to the library's documentation. Specifically the linear model[2] section, tree-based model[3] section, the neural network[4] section and the nearest neighbour[5] section, to verify the supported machine learning algorithms.

4. import both concrete-ml and scikit-learn chosen algorithm;

5. regarding the datasets that can be used, there are two possible choices. The first is to use already generated datasets and import them at the beginning of the code. The second is to create the datasets ad-hoc with the functions made available by scikit-learn, `make_regression`[6] and `make_calssification`[7].

   The following is an example of using `make_classification`:

```
1       X, y = make_classification(
2               n_features=30,
3               n_redundant=0,
4               n_informative=2,
5               random_state=42,
6               n_clusters_per_class=1,
7               class_sep=2,
8               n_samples=1000,
9           )
```

6. in Chapter 5.3.2 and 5.3.1, they discussed the evaluation of performance metrics for regression or classification algorithms. However, if the algorithms being analyzed are not regression or classification algorithms, then it is necessary to determine the most suitable metrics for evaluation.

   If the algorithms are classification algorithms, then we can use the metrics defined below, starting with functions already implemented in the scikit-learn library:

```
1       conf_matrix_fhe = confusion_matrix(y_test,y_pred_fhe)
2       acc_score_fhe = accuracy_score(y_test,y_pred_fhe)
3       error_rate_fhe = 1 - acc_score_fhe
4       recall_fhe = recall_score(y_test,y_pred_fhe)
5       tn, fp, fn, tp = conf_matrix_fhe.ravel()
6       specificity_fhe = tn / (tn + fp)
7       sens_fhe = tp / (tp + fn)
8       prec_fhe = precision_score(y_test,y_pred_fhe)
9       F_fhe = f1_score(y_test,y_pred_fhe)
```

   On the other hand, if the algorithm is regression algorithm the metrics can be computed as described in the lines of code that follow:

```
1        mse_fhe = mean_squared_error(y_test,y_pred_fhe)
2        mea_fhe = mean_absolute_error(y_test,y_pred_fhe)
3        r2_fhe = r2_score(y_test, y_pred_fhe)
4        adjusted_r2_fhe = 1 - ((1 - r2_sklearn) * (n - 1) / (n - k - 1))
```

7. data collected will be stored in specific JSON files as mentioned in Chapter C.3.2:

   - `results_LR_FHE.json` and `results_LR_scikit.json` for logistic regression;

[2] https://docs.zama.ai/concrete-ml/built-in-models/linear

[3] https://docs.zama.ai/concrete-ml/built-in-models/tree

[4] https://docs.zama.ai/concrete-ml/built-in-models/neural-networks

[5] https://docs.zama.ai/concrete-ml/built-in-models/nearest-neighbors

[6] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html

[7] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

- `results_SVMC_scikit.json` for linear SVM classification;
- `results_lSVR_FHE.json` and `results_lSVR_scikit.json` for linear SVR algorithm.

The way in which data concerning times and metrics is stored should align with the formatting used by the algorithms implemented with scikit-learn. This alignment is crucial as it ensures that the behaviours of different configurations can be accurately analysed and compared.