Towards Improved Homomorphic Encryption for Privacy-Preserving Deep Learning

by

José Cabrero-Holgueras

A dissertation submitted by in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Computer Science and Technology

Universidad Carlos III de Madrid

Advisor:

Prof. Dr. Sergio Pastrana

March 2023

This PhD dissertation has been mostly developed at CERN, supported by the CERN Doctoral Student Programme (from November 1st, 2019 to October 31st, 2022).

A mis padres, quienes han sido los pilares de mi vida, siempre han creído en mí y me han apoyado en todas las decisiones que he tomado. Gracias por hacerme ser quien soy.

Acknowledgements

There are a lot of things that made this thesis a not-so-professionally-rewarding time. If there is a reason for this thesis to be written today, it is because of the people that were a part of this journey.

First, I would like to express my sincere gratitude to Sergio Pastrana, my director, and supervisor in this thesis. Despite being a remote Ph.D. and the fact that this topic was outside his area of expertise, he accepted the challenge and invested considerable time and effort in keeping up to date with my work and maximizing its success. Sergio honesty, integrity, and dedication are exceptional values that are rare to find in the academic world. I am deeply grateful for the support and encouragement that Sergio gave me, as it was fundamental for the success of this thesis and my development as a researcher.

I would also like to express my most profound appreciation to Marco Manca for his mentoring throughout the thesis. Our conversations were instrumental in gaining a deeper understanding of research, the value of my work, and how to properly motivate it.

I am deeply indebted to Alberto Di Meglio, my CERN supervisor, for giving me the opportunity to participate in such a unique and enriching experience.

I feel immensely grateful to have had the opportunity to work with such an extraordinary group of people at CERN. Every single one of my colleagues has left a lasting impression on me, creating unforgettable memories that I will treasure forever. I especially want to express my gratitude to Alexia Yiannouli, Anna Manou, Gabriele Morello, Ignacio Peluaga, Lars Soerlie, and Renato Cardoso, as they were the most fantastic group of friends, who I shared with unforgettable memories.

I cannot forget Team Kernel, a group of friends that, throughout these years, became my family in France. I refer to Guillermo Izquierdo, Irene García, Javier López and Saúl Alonso. They are a fantastic group of brilliant, humble, and generous people. I cannot stress enough the comfort and joy our gatherings brought me (especially during pandemic times). When we started working together back in 2018, I never imagined life would have these adventures to come. I must specially thank Javier López for his support, helpful suggestions, and the countless long dinners we enjoyed at his place. He has been an outstanding friend in every way, often putting others before himself, and I am forever grateful for his kindness.

I am incredibly thankful to my friends in Madrid, who were always keen to meet me

when I was around, and some of them I have known for more than 20 years. I refer to Santiago Díaz, Lorena Lopesino, Iker Higuera, and Luís Martínez, who always organized amazing plans for me when I arrived.

I also extend my gratitude to my university colleagues, Laura Martín and Alejandro Rey, for our engaging meetings during my visits to Spain.

I would like to take a moment to acknowledge my online friends and videogame partners, whose presence and constant companionship have added immeasurable joy and positivity to my life. I extend my profound gratitude to Nacho Martínez, Rodrigo Salgado, and Sergio López. Without their presence, my life would not have been as enjoyable, and I am forever grateful for their friendship.

I extend my thanks to the COSEC research group, particularly to Eduardo Blazquez and Antonio Nappa, for their exceptional assistance and warm hospitality.

I am incredibly grateful to my parents, Carmen, Jose, and sister, Elena. They truly know what this journey has meant for me and all the struggles I have been through. I know, in their way, they have also struggled for me and looked to help me in every possible way. They are the people I need the most and a fundamental part of my life. They have given me everything and raised me to be the person I am today. I am immensely indebted to them and will always treasure their support, which has been invaluable.

Above all, I want to thank Nerea Luna Picón, my lifetime partner. She is the best person I know; she is loyal, supportive, and kind. From the moment we met, Nerea has been a pillar of my life, helping me through decisions, even those that were tough for both of us. Throughout this journey, Nerea was a constant source of positivity and encouragement, especially during those lowest moments when quitting the Ph.D. seemed more than a feasible option. She has always been there to lend a helping hand or simply sit by my side and help me push through. I am confident that without her unwavering support, I would not be writing these words today. I am forever grateful to you, Nerea. This thesis is as much yours as it is mine.

Published and submitted content

The following publications from the author have been included in this thesis:

- José Cabrero-Holgueras and Sergio Pastrana. "SoK: Privacy-Preserving Computation Techniques for Deep Learning". In: *Proceedings on Privacy Enhancing Technologies* 2021.4 (2021), pp. 139–162
  - The contents are included in Chapters 2 and 3.
- José Cabrero-Holgueras and Sergio Pastrana. "Towards Realistic Privacy-Preserving Deep Learning over Encrypted Medical Data". In: *Frontiers in Cardiovascular Medicine* 10 (), p. 641
  - Preprint version available: José Cabrero-Holgueras and Sergio Pastrana. *Towards Realistic Privacy-Preserving Deep Learning Inference Over Encrypted Data*. http://dx.doi.org/10.2139/ssrn.4140183. 2022.
  - The contents are included in Chapter 4.
- José Cabrero-Holgueras and Sergio Pastrana. "Towards Automated Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming". In: *Expert Systems with Applications* (2023)
  - Preprint version available: José Cabrero-Holgueras and Sergio Pastrana. "Towards Automated Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming". In: *arXiv preprint arXiv:2302.08930* (2023).
  - The contents are included in Chapter 5.
- José Cabrero-Holgueras and Sergio Pastrana. "HEFactory: A Symbolic Execution Compiler for Privacy-Preserving Deep Learning with Homomorphic Encryption". In: *Software X* (2023)
  - The contents are included in Chapter 6.

The material of the sources above are included in this thesis and are not singled out with typographic means and references.

Other research merits

The following articles and contributions are other research merits of this thesis:

- José Cabrero-Holgueras and Sergio Pastrana. "A methodology for large-scale identification of related accounts in underground forums". In: *Computers & Security* 111 (2021), p. 102489
- José Cabrero-Holgueras. "Usable Homomorphic Encryption for Private Telemedicine on the Cloud". In: *Italian Telemedicine Society Conference (SIT)* (2021)
  - This extended abstract was awarded the best abstract price at the Italian Telemedicine Society Conference 2021.

Abstract

Deep Learning (DL) has supposed a remarkable transformation for many fields, heralded by some as a new technological revolution. The advent of large scale models has increased the demands for data and computing platforms, for which cloud computing has become the go-to solution. However, the permeability of DL and cloud computing are reduced in privacy-enforcing areas that deal with sensitive data. These areas imperatively call for privacy-enhancing technologies that enable responsible, ethical, and privacy-compliant use of data in potentially hostile environments.

To this end, the cryptography community has addressed these concerns with what is known as Privacy-Preserving Computation Techniques (PPCTs), a set of tools that enable privacy-enhancing protocols where cleartext access to information is no longer tenable. Of these techniques, Homomorphic Encryption (HE) stands out for its ability to perform operations over encrypted data without compromising data confidentiality or privacy. However, despite its promise, HE is still a relatively nascent solution with efficiency and usability limitations. Improving the efficiency of HE has been a longstanding challenge in the field of cryptography, and with improvements, the complexity of the techniques has increased, especially for non-experts.

In this thesis, we address the problem of the complexity of HE when applied to DL. We begin by systematizing existing knowledge in the field through an in-depth analysis of state-of-the-art for privacy-preserving deep learning, identifying key trends, research gaps, and issues associated with current approaches. One such identified gap lies in the necessity for using vectorized algorithms with Packed Homomorphic Encryption (PaHE), a state-of-the-art technique to reduce the overhead of HE in complex areas. This thesis comprehensively analyzes existing algorithms and proposes new ones for using DL with PaHE, presenting a formal analysis and usage guidelines for their implementation.

Parameter selection of HE schemes is another recurring challenge in the literature, given that it plays a critical role in determining not only the security of the instantiation but also the precision, performance, and degree of security of the scheme. To address this challenge, this thesis proposes a novel system combining fuzzy logic with linear programming tasks to produce secure parametrizations based on high-level user input arguments without requiring low-level knowledge of the underlying primitives.

Finally, this thesis describes *HEFactory*, a symbolic execution compiler designed to streamline the process of producing HE code and integrating it with Python. *HEFactory* implements the previous proposals presented in this thesis in an easy-to-use tool. It provides a unique architecture that layers the challenges associated with HE and produces simplified operations interpretable by low-level HE libraries. *HEFactory* significantly reduces the overall complexity to code DL applications using HE, resulting in an 80% length reduction from expert-written code while maintaining equivalent accuracy and efficiency.

Resumen

El aprendizaje profundo ha supuesto una notable transformación para muchos campos que algunos han calificado como una nueva revolución tecnológica. La aparición de modelos masivos ha aumentado la demanda de datos y plataformas informáticas, para lo cual, la computación en la nube se ha convertido en la solución a la que recurrir. Sin embargo, la permeabilidad del aprendizaje profundo y la computación en la nube se reduce en los ámbitos de la privacidad que manejan con datos sensibles. Estas áreas exigen imperativamente el uso de tecnologías de mejora de la privacidad que permitan un uso responsable, ético y respetuoso con la privacidad de los datos en entornos potencialmente hostiles.

Con este fin, la comunidad criptográfica ha abordado estas preocupaciones con las denominadas técnicas de la preservación de la privacidad en el cómputo, un conjunto de herramientas que permiten protocolos de mejora de la privacidad donde el acceso a la información en texto claro ya no es sostenible. Entre estas técnicas, el cifrado homomórfico destaca por su capacidad para realizar operaciones sobre datos cifrados sin comprometer la confidencialidad o privacidad de la información. Sin embargo, a pesar de lo prometedor de esta técnica, sigue siendo una solución relativamente incipiente con limitaciones de eficiencia y usabilidad. La mejora de la eficiencia del cifrado homomórfico en la criptografía ha sido todo un reto, y, con las mejoras, la complejidad de las técnicas ha aumentado, especialmente para los usuarios no expertos.

En esta tesis, abordamos el problema de la complejidad del cifrado homomórfico cuando se aplica al aprendizaje profundo. Comenzamos sistematizando el conocimiento existente en el campo a través de un análisis exhaustivo del estado del arte para el aprendizaje profundo que preserva la privacidad, identificando las tendencias clave, las lagunas de investigación y los problemas asociados con los enfoques actuales. Una de las lagunas identificadas radica en el uso de algoritmos vectorizados con cifrado homomórfico empaquetado, que es una técnica del estado del arte que reduce el coste del cifrado homomórfico en áreas complejas. Esta tesis analiza exhaustivamente los algoritmos existentes y propone nuevos algoritmos para el uso de aprendizaje profundo utilizando cifrado homomórfico empaquetado, presentando un análisis formal y unas pautas de uso para su implementación.

La selección de parámetros de los esquemas del cifrado homomórfico es otro reto recurrente en la literatura, dado que juega un papel crítico a la hora de determinar no sólo la seguridad de la instanciación, sino también la precisión, el rendimiento y el grado de se-

guridad del esquema. Para abordar este reto, esta tesis propone un sistema innovador que combina la lógica difusa con tareas de programación lineal para producir parametrizaciones seguras basadas en argumentos de entrada de alto nivel sin requerir conocimientos de bajo nivel de las primitivas subyacentes.

Por último, esta tesis propone *HEFactory*, un compilador de ejecución simbólica diseñado para agilizar el proceso de producción de código de cifrado homomórfico e integrarlo con Python. *HEFactory* es la culminación de las propuestas presentadas en esta tesis, proporcionando una arquitectura única que estratifica los retos asociados con el cifrado homomórfico, produciendo operaciones simplificadas que pueden ser interpretadas por bibliotecas de bajo nivel. Este enfoque permite a *HEFactory* reducir significativamente la longitud total del código, lo que supone una reducción del 80% en la complejidad de programación de aplicaciones de aprendizaje profundo que usan cifrado homomórfico en comparación con el código escrito por expertos, manteniendo una precisión equivalente.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interfaces. 32, 39, 46, 52, 53, 82, 107, 108, 110, 111, 125, 126

AST Abstract Syntax Tree. 113–117, 119

BNN Binary Neural Network. 38, 39

CNN Convolutional Neural Network. xxiii, 4, 7, 8, 10, 38, 39, 44, 45, 57, 60, 61, 63, 69, 79, 80, 112, 125, 133

CPU Central Processing Unit. 15

CRF Convolution Resulting Format. 131–133

DL Deep Learning. xi, xii, xxiii, 1–8, 10, 11, 13, 15–17, 25, 31–33, 35–44, 46, 48, 50–53, 55–57, 59–61, 64, 65, 67, 69, 72, 74, 80, 105–108, 111–113, 115, 124–128, 130, 141, 142

DNN Deep Neural Network. 38–40, 43

DO Data Owner. 108, 109, 111, 120, 121, 125

DP Data Processor. 108, 109, 111, 120

DP Differential Privacy. 14–16, 32, 42, 58

FHE Fully Homomorphic Encryption. 3, 17, 37, 38, 43, 69, 83

FIP Fuzzy Inference Process. 88–93

FL Federated Learning. 15, 16, 32

FL Fuzzy Logic. 81, 83, 84, 87–90, 92–94, 97, 102, 103

GC Garbled Circuits. 35, 40, 42, 44

GPU Graphics Processing Unit. 15

HBC Honest-but-Curious. 13, 27, 40, 42, 51, 58

HE Homomorphic Encryption. xi, xii, 2–8, 10, 11, 14–17, 19, 21, 25, 31, 33–35, 37, 38, 40, 41, 43–53, 55–61, 64, 66, 69–71, 79, 81–88, 90, 95, 97–103, 105–115, 117–122, 124, 126–128, 130, 136–142

IR Intermediate Representation Code. 108, 111

IR Initial Representation. 60, 61, 66, 67, 74, 76, 78, 108, 109, 111, 113, 114, 119, 121, 123

LHE Leveled Homomorphic Encryption. 3, 17, 24, 34, 36, 37, 41, 42, 70, 79, 85, 135

LP Linear Programming. 81, 83, 84, 87, 88, 91–98, 102, 103

LWE Learning with Errors. xxi, 3, 17–20, 56, 58, 87, 105, 136

ML Machine Learning. 11, 36

MLaaS Machine Learning as a Service. 32, 37, 47, 48, 108, 109

NN Neural Network. 7–11, 36, 38, 42, 74, 143

OT Oblivious Transfer. 25–27, 34, 36, 39, 42

PaHE Packed Homomorphic Encryption. xi, 4, 55, 56, 58, 60, 61, 79–81, 105, 113, 127

PHE Partially Homomorphic Encryption. 16, 36

PPCTs Privacy-Preserving Computation Techniques. xi, 2, 4, 5, 15, 31–33, 35, 36, 41, 43, 46, 50–53, 55, 127, 128

PPDL Privacy-Preserving Deep Learning. 7, 11–13, 15, 16, 32, 34–37, 41–43, 46, 47, 49–53, 106, 127, 128, 142

RC Row-Column. 56, 60, 64, 67, 74, 131–134

RLWE Ring Learning with Errors. 17–19, 58, 84

RT Result Transformation. 60, 61, 66, 67, 73, 76, 78, 79, 131–133

SCBF Streamlined Backward Convolution Format. 62–64

SCRF Stridden Convolution Resulting Format. 131–133

SHE Somewhat Homomorphic Encryption. 17, 34, 38

SIMD Single Instruction Multiple Data. 25, 34, 38, 39, 45, 46, 56, 57, 59, 60, 64, 65, 69, 71, 80, 99, 113, 131, 142

SISD Single Instruction Single Data. 34

SMPC Secure Multiparty Computation. 2, 7, 14–16, 25–27, 29, 31, 33–37, 39–44, 46–49

SS Secret Sharing. 27, 34

TEE Trusted Execution Environment. 15, 36, 44, 51

WAN Wide Area Networks. 49

Yao's GC Yao's Garbled Circuits. 39, 40

ZKP Zero Knowledge Proof. 15, 29, 34, 51

Chapter 1

Introduction

Computation and communications have undergone a remarkable transformation. Where once the Internet was the domain for a few researchers, the advent of modern technology has brought complex cloud solutions to the fingertips of everyday people. The structured classical algorithms have evolved into complex large scale models [Sha+17] that provide instant answers to a wide range of questions and achieve proficiency in different domains. However, with these new technologies, new risks arise. Deep Learning (DL) models are massive statistical models that indiscriminately consume vast amounts of information, which is later probabilistically sent back to third parties [Sho+17]. Moreover, cloud computing involves using third-party servers where data and computation remain under limited control [Sen15].

Also, despite the impressive improvement of DL in areas such as natural language processing or computer vision, their applications to sensitive environments have remained limited [RRK19]. A clear example is medical research. Several promising testbeds demonstrate the potential of DL in healthcare applications, including improved remote diagnosis, enhanced medical imaging, and health monitoring outside hospitals. Nevertheless, large-scale model training requires substantial infrastructure and datasets [Jul+17; SS19; Tan+20; Top19]. Healthcare data is predominantly sensitive, and accessing large-scale datasets required for DL models is often challenging due to privacy and legal restrictions. Furthermore, cloud providers offer the infrastructure needed for large-scale model training, but this approach raises ethical and legal concerns regarding data privacy and security. Given the ethical and legal restrictions imposed on data exchanges and working on third-party infrastructure, privacy-sensitive environments have suffered reduced permeability of the latest technological advances. Privacy-sensitive applications [Ber+19; HPW17] imperatively ask for the use of these technologies under responsible, ethical, and privacy-compliant environments where the privacy and security of individuals must be granted.

Cryptographers have addressed these concerns with significant progress in the realm of secure data exchanges and private computation through Privacy-Preserving Computation Techniques (PPCTs). The traditional trust model, where two trusted parties (i.e., Alice and Bob) sought to establish a secure communication channel, has undergone a significant change. In this new context, Alice seeks a service from Bob, yet Bob is not a trusted party, and Alice might be reluctant to transmit information to him, e.g., due to potential legal violations. This scenario is particularly relevant in sensitive domains, such as healthcare, where Alice and Bob may wish to exchange information but clear access to data is no longer tenable.

PPCTs provide reliable solutions that enable privacy-preserving computation, with Homomorphic Encryption (HE) and Secure Multiparty Computation (SMPC) being the approaches with the most potential. They provide reliable solutions for safeguarding data privacy while processing information. However, PPCTs incur a high computational and communication overhead to the already-demanding DL operations. Due to the impact of DL, PPCTs have attracted the attention of researchers with an increasing number of proposals published at a fast rate. Despite the high volume of research on this topic, the technology remains immature, and few actual deployments are being used in privacy-preserving scenarios. The limited adoption of PPCTs can be attributed mainly to issues related to efficiency and usability [Che+20; Kum+20b].

HE is particularly noteworthy due to its unique property of enabling operations over encrypted data while maintaining the confidentiality of the data. Consequently, HE has gained recognition for its ability to facilitate privacy-preserving deep learning inference, enabling private services to be hosted in the cloud. Nevertheless, it still introduces various challenges due to the complexity of operations, efficiency, and usability issues in their application to complex computations. In this thesis, we focus on lowering the entry barrier for inexperienced users to Homomorphic Encryption in its application to DL inference.

The remainder of this Chapter is structured as follows: Section 1.1 outlines the main motivations behind this thesis, and Section 1.2 introduces its goals. Finally, the structure of the remaining chapters of this manuscript is defined in Section 1.3.

1.1 Motivation

As detailed previously, introducing sensitive data in a combination of DL and cloud computing introduces previously unforeseen risks to computation and privacy concerns with regulatory implications. The application of DL in sensitive environments, such as medical research, has remained limited due to privacy and legal restrictions [Sin+22; VB17].

In 1978, with the birth of the RSA [RSA78], the authors pointed out a curious property of the cryptosystem. RSA allowed for modifications of the encrypted information without the need to decrypt, which they called Homomorphic Encryption (HE). Ever since then, the field of cryptography has evolved to elaborate more complex and compelling constructions, which were able to emulate arbitrary computation through sums and multiplications accurately, but ever so slightly missed by only providing a limited amount of consecutive operations. Their limitations came from using the Learning with Errors (LWE) problem, whose strength comes from introducing noise to a system of linear equations. The noise in the cryptosystems grew with operations, and after a certain amount, it yielded undecryptable results. More than 30 years after the RSA proposal, in 2009, Craig Gentry [Gen09] proved the first construction where unlimited and arbitrary computation was possible, referred to as Fully Homomorphic Encryption (FHE). Gentry's construction profited from a clever mechanism called bootstrapping that homomorphically evaluated the decryption circuit, reducing the noise and allowing for theoretically unlimited computation.

From then on, FHE has become a holy grail of cryptography, with numerous researchers seeking to make it equivalent to classical computation efficiency-wise. The potential implications and the existing Homomorphic Encryption (HE) applications impact numerous areas such as aerospace, blockchain, and, more importantly for our study, Deep Learning (DL). Not only this technology improves security, but it also defines legally compliant mechanisms that qualify for performing data analytics and DL on sensitive data, even on potentially hostile third-party infrastructure.

Therefore, on the one hand, privacy-sensitive applications require the use of DL under responsible, ethical, and privacy-compliant environments. On the other hand, FHE addresses legal concerns and allows privacy-preserving sensitive data processing.

Unfortunately, HE is still a relatively immature solution that presents efficiency and usability flaws. Improving the efficiency of HE has been a longstanding challenge in the field of cryptography. One improvement to this challenge is ciphertext packing [BGH13] which allows for the encryption of a whole plaintext vector within a ciphertext. Together with rotations, which enable cyclic shifting of the position of entries in the ciphertext, ciphertext packing allows for the vectorization of algorithms, permitting the processing of spatially related information and operations such as matrix multiplication [DMY16] in reduced time. Although ciphertext packing and vectorization of algorithms significantly reduce the number of operations on ciphertexts, designing algorithms for vectorized execution is far from straightforward, especially for untrained users. While some algorithms for vectorized DL have been briefly described in the literature, their implications and usability with DL have not been thoroughly discussed.

Gentry initially proposed bootstrapping to enable FHE, but it was computationally inefficient. To address this limitation, Brakerski et al. [BGV14] coined the concept of Leveled Homomorphic Encryption (LHE), where a ciphertext is encrypted with certain levels, and by reducing the levels, the noise is also reduced. While LHE is more efficient than bootstrapping, it still requires careful parameter selection. Selecting the correct parameters for HE is challenging, as it is an NP-Complete problem with complex interrelations between the computation and the parameters. Additionally, ciphertext packing introduces complexity to parameter selection, as the size of the polynomial ring constrains the maximum number of entries of the polynomial. The usability of HE poses a steep learning curve for non-expert users, who need to gain experience before using these techniques optimally. As a result, there is a need for efficient and user-friendly tools that can assist with parameter selection and facilitate the adoption of homomorphic encryption in practical applications.

Due to the considerable domain expertise and knowledge required to use HE efficiently, recent research works have striven to provide abstractions and simplifications to these techniques for non-practitioners to be able to use them [Aha+23; Dat+19; Dat+20; HS14b; Mou+20; Res20]. While existing efforts tackle the problem from different sides of the problem, most of them focus on specific parts of the problem and do not fully address DL. These barriers restrict access to privacy-preserving data science to users who might lack the expertise to work in low-level languages and with complex HE tools [Chi+20b; HS14b; Res20].

Hypothesis:
By addressing specific usability challenges and integrating the solutions into tools, it is possible to enhance the accessibility of HE-enabled DL for non-expert users.

1.2 Objectives

With the different problems raised and the different inconveniences in the use of Homomorphic Encryption for Deep Learning Inference, this thesis has the following goal:

Goal: *To provide techniques that reduce the complexity of Homomorphic Encryption when applied to Deep Learning, especially for non-expert users.*

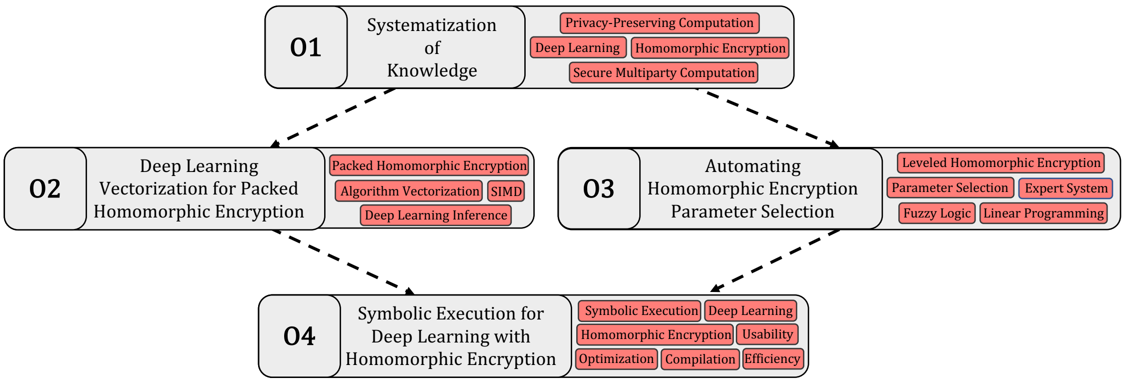
The previous goal is divided into the following objectives (Figure 1.1):

- **O1**. **Systematizing existing knowledge in the application of privacy-preserving computation techniques to Deep Learning.** The objective is to provide a systematic view of the current state of the art, together with development, identifying major trends in using PPCTs, potential research gaps, and issues with existing techniques.

- **O2**. **Analyzing and providing algorithms to adapt Deep Learning Inference with Packed Homomorphic Encryption.** The objective seeks to describe in detail and extend the algorithms required for building Convolutional Neural Network with vectorized PaHE. The objective shall provide formal analysis and guidelines that improve the use of such algorithms.

- **O3**. **Simplifying Homomorphic Encryption parameter selection for non-expert users.** It aims at designing an expert system that performs optimized parameter selection based on high-level user input parameters, reducing the overall complexity of the task.

- **O4. Integrating the previous improvements in a computer-aided tool that enables inexperienced users to use Homomorphic Encryption for Deep Learning.** The objective is to provide a compiler whose architecture simplifies the use of HE, which provides interfaces to high-level languages (e.g., Python) and integrates the possibility of using DL inference leveraging existing low-level HE frameworks.



01
Systematization of Knowledge
Privacy-Preserving Computation
Deep Learning
Homomorphic Encryption
Secure Multiparty Computation
02
Deep Learning Vectorization for Packed Homomorphic Encryption
Packed Homomorphic Encryption
Algorithm Vectorization
SIMD
Deep Learning Inference
03
Automating Homomorphic Encryption Parameter Selection
Leveled Homomorphic Encryption
Parameter Selection
Expert System
Fuzzy Logic
Linear Programming
04
Symbolic Execution for Deep Learning with Homomorphic Encryption
Symbolic Execution
Deep Learning
Homomorphic Encryption
Usability
Optimization
Compilation
Efficiency

Figure 1.1: Overview of the thesis objectives with the main keywords and techniques used to achieve each objective.

1.3 Structure of the Document

The remainder of this thesis is structured as follows:

- Chapter 1, *Introduction*, provides an introduction to the contents of the thesis.
- Chapter 2, *Background*, provides theoretical background on a series of base concepts needed for the proper understanding of the thesis contributions, including Deep Learning, Privacy-Preserving Computation Techniques and Homomorphic Encryption.
- Chapter 3, *State of the Art*, covers a detailed analysis of the current research proposals using PPCTs for DL, including a systematization of the knowledge and a discussion of the existing research gaps.
- Chapter 4, *Optimization of Deep Learning Linear Algebra Algorithms for Packed Homomorphic Encryption*, presents the analysis and description of vectorized algorithms for performing DL with packing and rotations in HE.

- Chapter 5, *Automating Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming*, details the design of an expert system based on fuzzy logic and linear programming tasks for automatic parametrization of HE schemes.

- Chapter 6, *HEFactory: A Symbolic Execution Compiler for Privacy-Preserving Deep Learning with Homomorphic Encryption*, describes the design of HEFactory a symbolic execution compiler for DL with HE.

- Chapter 7, *Conclusions and Future Work*, covers the main contributions of this thesis and outlines future lines of research.

- Appendix A, *Baseline Convolution Algorithms*, covers other generalizations of algorithms used as a baseline comparison to the improved version provided in Chapter 4.

- Appendix B, *Simplified Leveled Homomorphic Encryption Parametrization in Practice*, provides a simple non-expert introduction to practical parametrization of HE schemes.

- Appendix C, *HEFactory Internals*, provides a brief guide to the main classes and functions of the internals of *HEFactory*.

Chapter 2

Background

In this thesis, we propose various improvements to the usability of Homomorphic Encryption in its relationship with Deep Learning inference. This section provides a comprehensive background to understand the theoretical concepts. We first provide an introduction of Deep Learning to detail the primary concepts, terminology, and basis for improvement in Section 2.1. Then, in Section 2.2, we describe the different privacy goals and requirements that Deep Learning may want to achieve and the different techniques for this matter. Followingly, Section 2.3 covers a thorough description of Homomorphic Encryption, documenting its building blocks and insights for its operations, behavior, and limitations. Finally, Section 2.4 describes the set of privacy-preserving computation techniques based on Secure Multiparty Computation. While this technique is out of the scope of the main contributions of this thesis, we describe it in this Chapter since it has been proposed for privacy-preserving Deep Learning in the literature, as we study in Chapter 3.

2.1 Deep Learning for Homomorphic Encryption

Deep Learning (DL) has revolutionized artificial intelligence with outstanding solutions to previously computationally complex problems. As such, this section provides a general background to DL concepts and layers used for Privacy-Preserving Deep Learning (PPDL) and those covered in this work. While DL often comprises forward and backward propagation, it is primarily applied to the inference phase with Homomorphic Encryption (HE). As such, we only dig into the inference phase. We first cover an introduction to DL and its origins in Subsection 2.1.1. Then we cover the major Neural Network (NN) layers in Convolutional Neural Network (CNN) in Subsection 2.1.2. Finally, we cover the primary Neural Network activation functions in Subsection 2.1.3.

2.1.1 Deep Learning and Neural Networks

DL is a machine learning technique that evolved from Neural Network [Goo+16; LBH15]. NNs are biologically inspired statistical models that resemble neural connections, where neurons connect in a network and activate each other based on a stimulus [Has+95]. Learning theory states that a NN can learn any function [Csá+01; Goo+16]. However, this definition does not account for the computational complexity of such NN (i.e., the layer may be infinitely large). In practice, combining networks in deeper architectures (i.e., more layers rather than wider layers) obtains similar results in terms of accuracy while improving performance and constitute what we know as DL [Bal+17; HS18].

Classically, NNs consist of *neurons* represented by a mathematical function that combines a linear procedure and an activation function (i.e., mimicking the spiking biological behavior of neurons). NNs compose groups of one or more neurons in *layers*, and the specific arrangement of the layers is known as *architecture*.

While there are various flavors of unsupervised and self-supervised training for neural networks, NNs are generally a supervised learning technique. In supervised learning, there is an initial training phase, after which the model is available for inference. The training phase instructs the model by extracting the statistical distribution for the learning task from the data and encompassing it into the *weights* of the model.

The training phase consists of two steps. Firstly, forward propagation infers the prediction on an input training sample. Secondly, backward propagation compares the predicted result with the ground truth in the *loss function*. *Gradient descent* evaluates the particular effect of each weight on the final result and describes the modifications so that predictions are correct in further iterations.

While modern architectures exist, such as Transformers [Sha+17] with attention units, the existing limitations in Homomorphic Encryption have restricted the available options to older yet effective models, such as CNNs. The following subsections describe the building blocks of CNNs and the main existing activation functions.

2.1.2 Neural Network Layers

This section defines the main linear layers of a Neural Network, which serves as a preamble for the algorithms described in Chapter 4.

Dense Layer

The dense layer, also known as a fully connected layer, is parametrized by the number of neurons (n) it uses. The computation for these layers is defined as matrix multiplication such that:

$$z = W \cdot x \tag{2.1}$$

Where W is the weight matrix of the layer and $x \in \mathbb{R}^h$ is the input vector of the previous layers. In $W \in \mathbb{R}^{h\ timesw}$, each row represents the weight of a neuron. Fully connected layers are advantageous since they expose all connections, and during training, remote relations can be stored by the Neural Network. However, this degree of connectivity introduces the need to operate over $h \times w$ entries defined by the number of neurons in previous layers and the number of neurons in the layer. For certain kinds of information, such as images or sound, spatial or sequential information may lift such connectivity requirements.

Convolutional Layer

Convolutional layers use the concept of image filters as a foundation. Classically, image filters have provided promising results in extracting features from images. In this case, the filter weights are left to the neural network training to extract. The main advantage of convolutional layers is the reduction of connectivity, reducing the number of weights per layer, and obtaining better results by exploiting the spatial locality of data [LeC+89; LeC+98](i.e., the relations between data when they are found in close locations). It can be successfully exploited in audio and image data.

A bidimensional convolutional layer operates over a $f_x \times f_y$ pixels filter. The filter is shifted all over the pixels of the input image, resulting in a reduced representation of a feature. The operations on the filter are determined by the weights computed during the training. The formula for obtaining the bidimensional convolution of a pixel with a single dimension filter is defined by:

$$y_{h,w,c} = \sum_{k=0}^{c} \sum_{i=0}^{f_x} \sum_{j=0}^{f_y} w_{i,j,k} \cdot x_{h+i,w+j,k} \tag{2.2}$$

where x is the input c-dimensional image, $w_{i,j,k}$ are the weights, and y is the output image. In some cases, convolutions are combined with stride, which determines the displacement (i.e., for which h, w, y is computed) and padding, which helps accentuate the information values on corners of images.

Convolutional layers decrease the number of parameters and training time. They are often used in conjunction with batch normalization layers [IS15] and dropout layers [Sri+14] to improve its training convergence.

Max Pooling and Average Pooling

The max pooling and average pooling layers are sub-sampling structures that permit obtaining significant values from the results of previous layers. Bidimensional pooling layers act on $p_x \times p_y$ areas of the matrix and compute the max or average of it. Thus, they further reduce the input representation and speed up the computation. In CNN's max pooling, layers are directly coupled to the success of convolutional layers [Goo+16], and thus combined. In the case of Homomorphic Encryption, max pooling layers are minimal due to the complexity of approximating the max function. Therefore, the average pooling is often used. The formula for the bidimensional average pooling is:

$$y_{h,w,c} = \frac{1}{p_x \cdot p_y} \cdot \sum_{i=0}^{p_x} \sum_{j=0}^{p_y} x_{h+i,w+j,c} \tag{2.3}$$

where x is the input image and y is the resulting image.

2.1.3 Activation Functions

Activation functions are fundamental components that determine the intensity of the output of a neuron [Nwa+18], and they control the learning factor for the different weights. They are a vital aspect of DL training since they introduce non-linear behavior to the equations. Without activation functions, a neural network is nothing more than a complex linear regression [Goo+16]. There are several examples of it, but the most commonly used ones are the following:

Sigmoid

The sigmoid activation shifts towards values being either 0 or 1. It is used as the last layer activation. It enables a faster minimization of the cost function (i.e., values are shifted towards 0 or 1) [LeC+12]. The formula defining the activation is:

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

ReLU

This function is used in intermediate layers of the NN [NH10]. It has the advantage of allowing gradient updates to propagate correctly to the first layers. That is, for very deep architectures, ReLU is essential. The formula defining the activation is:

$$f(x) = max(0, x) \tag{2.5}$$

Softmax

The Softmax function behaves similarly to the sigmoid function, but it is especially efficient for multiclass classification, outputting a higher probability of one class and a lower for the rest. The formula for the softmax activation is:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.6}$$

In practice, the application of DL is aided by software frameworks that help in the process of producing and testing different NN architectures (e.g., Tensorflow [Aba+16b], Keras [Cho+15], or PyTorch [Pas+17]). Software frameworks for DL simplify the design and programming by providing a catalog of layers, which can be easily extended. These frameworks also provide automatic computation of backpropagation, which reduces coding complexity and speeds up the process of implementing new layers. In a way, the contributions of this thesis aim to mimic the development of DL frameworks in the field of HE.

2.2 Privacy Settings and Requirements in Deep Learning

Privacy-Preserving Deep Learning considers a scenario where one or more entities (clients) aim to perform DL training or inference privately. First, for training, a set of *input training samples* is fed to the DL pipeline producing a *model*. Then, in production time, the *model* is employed over a set of *inference samples*, producing *output predictions*. If any of these processes are outsourced to external parties, it would involve the sharing of sometimes potentially private or insecure information.

There are two main research lines related to PPDL. On the one hand, research focuses on attacks and countermeasures related to classical DL models. On the other hand, investigations looking for privacy-by-design DL architectures, where training and inference phases reveal neither the model nor the data. The former is related to an extensive research line on adversarial machine learning, which aims to disclose potential threats to classical Machine Learning (ML) algorithms and provide countermeasures [BR18; Pap+18a].

The latter is the research line covered in this thesis. It requires reformulating and adapting inner functions and algorithms to provide a complete training or inference pipeline.

In this section, we systematize and describe various properties and settings related to the privacy and security of DL models and data. It allows an understanding of the different views of the privacy of models. We first present the privacy goals in Subsection 2.2.1 and the adversarial model in Subsection 2.2.2. Then, we describe the high-level architectures and data processing phases in Subsection 2.2.3. Finally, we describe different techniques proposed to achieve such goals in Subsection 2.2.4.

2.2.1 Privacy Goals

The main goal of PPDL is to enable training and inference while preserving the privacy of the associated entities (i.e., data providers, clients, or service providers). It means the processing cannot reveal additional information about data. We identify the following privacy goals, depending on the source of information that is protected:

Input Privacy aims to preserve the data privacy of the inputs used during training or inference [WL11]. For training, we consider input privacy to all those techniques that preserve the privacy of the dataset. For inference, we consider input privacy when inference data is protected.

Output Privacy aims not to reveal private information about the data produced from training (i.e., the model) or inference (i.e., the output predictions). Output privacy is required when the model is exposed and used by non-trusted parties, which might obtain information from it [FJR15; Sho+17]. We note that input and output privacy is required when the training process is outsourced to a non-trusted party (i.e., the data must be kept secret, and the model must be processed not to reveal information about the input data).

Model Secrecy aims to preserve the monetary value of models due to the complexity and effort required to train the models [Kum+20b]. Also, gaining information about how a model works eases inference attacks against it [Tra+16]. Consequently, model secrecy is a desirable privacy feature. Such secrecy can refer to two components of the model. First, *Architectural Secrecy* keeps secret the organization of the layers and their internal hyper-parameters. Second, *Weight Secrecy* protects the values of the weights given to the internal neurons after the training phase. The former is often related to intellectual property (e.g., cloud services not willing to expose how they process the data), and the latter with the output privacy guarantees (since the weights are adjusted to represent statistical properties of the training data).

2.2.2 Adversarial Model

Security guarantees of the protocols used in distributed systems, where multiple parties interact, often rely on the *ideal-real world simulation paradigm* [GMW87; GV88; Lin20]. This paradigm considers an ideal scenario where an ideal functionality (i.e., inviolable third-party) receives data from participants, computes a function in a centralized way, and sends back the result of the execution. In the ideal scenario, an adversary can only tamper with the inputs of corrupted parties [Ode09]. A distributed protocol is considered secure if, in the real world, the information exchanged between the different parties does not reveal more information to an adversary than what the ideal world reveals. According to this paradigm, two adversarial models are defined depending on their capabilities and goals:

Honest-but-Curious (HBC) adversary is a passive opponent that complies with the protocol and does not tamper with the data for malicious purposes. However, it tries to learn as much information as possible from information exchanges.

Malicious adversary is an active opponent with stronger capabilities. In addition to having the HBC adversary capabilities, it can tamper with the protocol (e.g., dynamically changing the inputs to the computation, not executing the process, or disconnecting at any point).

2.2.3 Architectures and Processing Steps

A privacy-preserving processing pipeline generally involves preprocessing, privacy-preserving processing, and postprocessing.

In the *preprocessing* phase, the client and server transform the input data appropriate format to evaluate the cryptographic protocol. In the *privacy-preserving processing* phase, the server receives the input data from the client. It performs the blind evaluation of the circuit (i.e., without having direct access to the plaintext data). Finally, the *postprocessing* phase retrieves the final result. Depending on the technique, it may consist of a data reconstruction from the different pieces or the decryption of the ciphertexts.

When dealing with DL, only the preprocessing phase requires modifying the functions used internally in DL (i.e., in order to adapt the directives to the specific technique). The two remaining stages blindly apply the modified model to the privatized data according to the chosen protocol. Thus, most contributions focus either on improving the adaptation of DL to cryptographic protocols or creating new protocols (or combinations of them) which reduce the changes needed. Figure 2.1 shows a schematic view of the generic architecture for PPDL.

The paradigm of MLaaS assumes a distributed protocol where different entities communicate in a network. Based on that, we consider two general architectures, depending on where and how the actual computation is performed:

Centralized Architectures load the exigent processing on one party (i.e., a server with enough computational resources). As such, a single server obtains the necessary information from the client and performs most of the computation. Clients are required fewer computational resources and infrequent interaction with the server. In general, in a centralized architecture, the original model is only held on the server

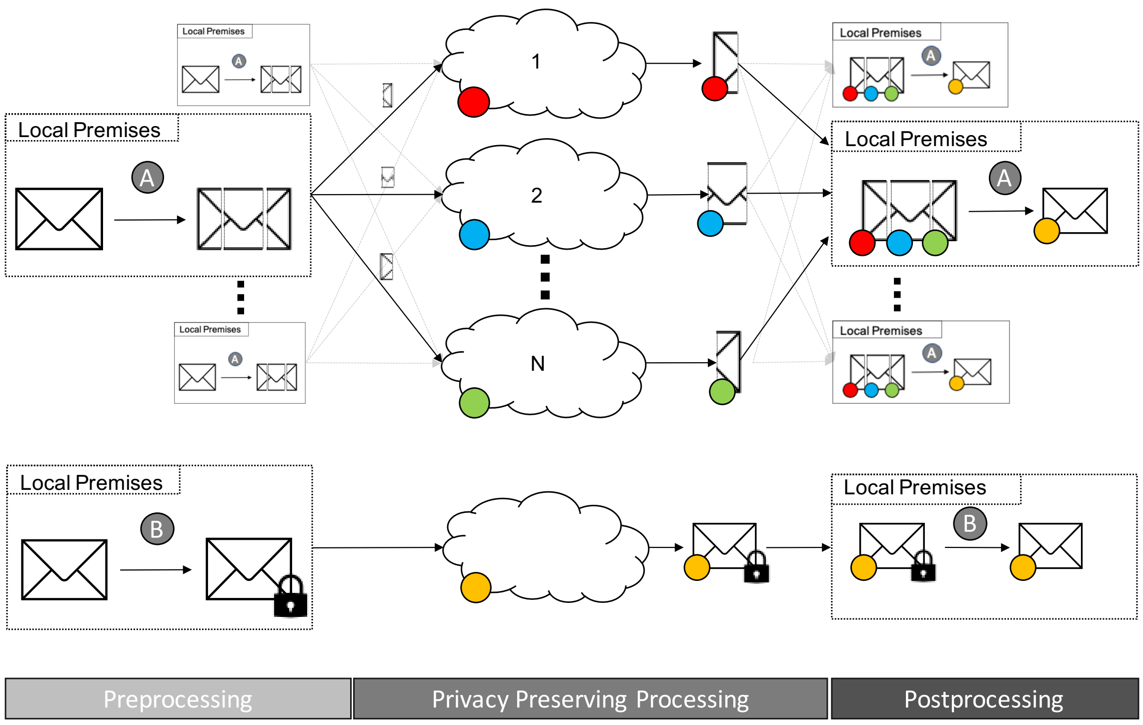


Figure 2.1: Generic Architecture for Privacy-Preserving Data Processing.

side and is often present in solutions with HE. Centralized architectures involve complex operations but less communication.

Distributed Architectures allow several parties to distributively make processing on their data without sharing the actual data. These architectures split the processing among all the participants in a distributed fashion without a central server holding the entire model. It is often done through Secure Multiparty Computation (SMPC) techniques and requires computing infrastructure on all the parties. Distributed approaches perform fewer and simpler operations at the cost of more communication and often require client interaction.

2.2.4 Privacy Techniques

Depending on privacy requirements, the adversarial model faced, and the specific settings, existing solutions often use the following different techniques (either in isolation, combined or in hybrid approaches):

Data Privacy techniques aim at reducing the amount of sensitive information that data carry. The goal is that content of the data released does not reveal private information about the entities behind it. Two main techniques are used for this: Anonymization and Differential Privacy (DP). First, *Data Anonymization* aims at the de-identification of data owners through generalization (i.e., removing identifying values from samples), omission (i.e., not including an individual in the dataset), and suppression (i.e., deleting complete identifying entries from the dataset) [LLV07; Mac+07; Swe02]. Second, *Differential Privacy* provides privacy guarantees for an individual of a larger population [DR+14] (i.e., guaranteeing the privacy of the individual while allowing the calculation of population-wide statistics). DP provides higher security than Anonymization since DP provides guarantees independently of the threat model. DP mechanisms often rely on adding noise to the data reducing its expressiveness. Data privacy techniques allow plaintext operation on the data, making them suitable for classical DL environments. However, given the information loss from dataset modifications and noise introduction, they also suffer utility reduction.

Privacy-Preserving Computation Techniques (PPCTs) rely on cryptography to hide the information while allowing computation over the underlying data. It does so through data secrecy, i.e., preserving confidentiality and, thus, privacy. There are two main approaches for this set of techniques (often used in combinations, as we discuss later), i.e., SMPC and HE. While HE often lies as a subcategory of SMPC, in general, they behave differently, and thus we separate them in this thesis. While HE relies on the strength to find a key to break the encryption, SMPC relies on sharing information among non-colluding parties for DL inference. Unlike data privacy techniques, data is encoded without causing an information loss. These techniques provide input privacy. However, while PPCTs preserve the information, the available operations are limited. Another drawback is that these techniques suffer from important performance and usability challenges when used in PPDL (see §2.4).

Trusted Execution Environment (TEE) are hardware components, often integrated into modern CPUs, that allow for the encryption of a portion of a process memory [CD16; KPW16; Win08]. TEEs keep the confidentiality and integrity of the data and operations loaded. TEEs allow for code attestation, and some approaches for DL rely on it [HCS18; Hun+18; TB18]. While TEEs preserve input privacy, their security guarantees are based on hardware and are subject to other types of attacks (e.g., side-channel attacks [Du+17; MIE17]). For example, SLALOM relies on TEEs for simple private operations and executes complex computations in an external, not necessarily trusted GPU, using ZKP to attest their correct execution [TB18]. Also, Chiron proposes a virtual server with a limited instruction set running on top of a TEE so they can be attested [Hun+18].

Federated Learning (FL) allows collaborative training of a model using local data from different entities without revealing it to the other parties [Kon+16; Ton+18]. The clients use their local data to train a local version of the model to compute the updates (i.e., gradients). Then these updates are sent back to a central server by sharing the resulting weights and parameters [SS15], which the central server aggregates onto a global model. This technique suffers from security issues since the generated model and gradients are shared and may be abused to breach privacy. Thus, FL is often combined with other techniques to preserve input and output privacy, such as HE, SMPC [Sav+20] or DP [SS15; YBS20]. We point out the reader for further information to the works by Kairouz et al. [Kai+19], and Bonawitz et al. [Bon+19]

As a summary of this section, PPDL consists of utilizing different techniques to elaborate DL training and inference in a manner that results equivalent to classic centralized training.

2.3 Homomorphic Encryption

Homomorphic Encryption (HE) is a beautiful mathematical trick whereby performing specific operations on specially-represented ciphertexts, we provoke a change in the underlying cleartext. Under specific operations over the ciphertexts, we can map the change on the cleartext to operations such as addition, multiplication, and rotation. In this section, we cover the details of HE from its conception and provide details of the schemes used in this thesis. The details of this section are based on the contents extracted from OpenMined [Huy20a; Huy20b; Huy20c; Huy20d; Huy20e] and the original papers of the authors of CKKS [Che+17].

2.3.1 Historical Background

Encryption schemes aim to provide confidentiality between two parties intending to exchange a piece of information or message without publicly releasing the content. For that, a fundamental principle is the existence of one or more keys, without which extracting the information is mathematically unfeasible. The first constructions were symmetric based, which commonly utilize the same routine for encryption and decryption, using the same secret key. Symmetric encryption assumes the existence of a previously shared key, which raises a new problem related to secret key distribution. Public key cryptography arose as a solution to this problem through trapdoor functions [DH22]. Trapdoor functions provide a robust mechanism for encryption and a mechanism for decryption using two different keys (a public and a private one), effectively enabling cryptography without using a pre-shared secret. One of the first, more popular, and still to date in using public-key algorithms is the RSA cryptosystem [RSA78]. In the RSA release paper, the authors noted an attractive property of the scheme: the ciphertext could be operated (concretely through multiplications) without decryption, and the results would apply to the hidden cleartext. As such, RSA is considered the first HE cryptosystem, also defined as Partially Homomorphic Encryption (PHE), as it allows one homomorphic operation. Other successful implementations of PHE include ElGamal [ElG85] or Paillier [Pai99] cryptosystems.

After that, research focused on finding Fully Homomorphic Encryption (FHE), or Homomorphic Encryption, that enables arbitrary computation on the ciphertext. While there were various proposals, those more promising were labeled *Somewhat Homomorphic Encryption (SHE)*, as these enabled performing addition and multiplication. However, there were limits to the amounts of computation. SHE schemes were based on the Learning with Errors (LWE) [Reg09] problem, whose strength relies on noise introduced to a system of linear equations. Operations over those ciphertexts increase the noise and introduce boundaries to the number of operations that can be performed. However, in 2009, Craig Gentry, a researcher at Standford University and IBM Watson, proved FHE feasible through bootstrapping (i.e., unbounded computation). Bootstrapping relies on the execution of the decryption routine or circuit in the encrypted domain. In that way, after the ciphertext reaches a certain noise level, it could be restored under reasonable levels for further operation. However, this initial cryptosystem proposed by Gentry suffered from very poor runtimes precisely due to the bootstrapping routine. Despite its capabilities and improvements, bootstrapping is still inefficient and suffers from a low runtime performance [DM15].

Since the work of Gentry in 2009, the research area in HE has dramatically expanded, and numerous new approaches and techniques for optimization have arisen. One of the optimizations comes in the form of Leveled Homomorphic Encryption (LHE) schemes. These introduce a number of levels, which are removed from the ciphertext after multiplications, effectively reducing its noise. The removal operation for each of the levels is called named modulo switching.

Based on the initial ideas proposed by Gentry, multiple encryption schemes have been created using different arithmetic types, like integer operations (e.g., BFV [FV12] and BGV [BGV14]) or boolean operations (e.g., GSW [GSW13], TFHE [Chi+16], or Concrete [Chi+20b]). In the context of DL, floating-point operations are available through the CKKS scheme [Che+17], which we describe next.

It is also noteworthy to mention the availability of multiple open-source libraries, such as Microsoft SEAL [Res20], IBM HElib [HS14b], Palisade [PRR17], Lattigo [Mou+20] or Concrete [Chi+20b].

2.3.2 Learning with Errors and Ring Learning with Errors

The Learning with Errors problem is a proven worst-case hard problem introduced by Regev [Reg09] and forms the basis for most HE schemes. This section introduces the variants of the Learning with Errors (LWE) and the Ring Learning with Errors (RLWE).

We use $\mathbb{Z}_Q$ to refer to the set of integers modulo Q, e.g., $3 \mod 5$. We use $\mathbb{Z}_Q^N$ to refer to vectors of N elements in $\mathbb{Z}_Q$, e.g., $(1, 2, 3) \mod 5$. We denote the inner product of two vectors $a, b \in \mathbb{Z}_Q^N$ as $\langle a, b \rangle = \sum_i^N a_i \cdot b_i \mod Q$. A system of linear equations can

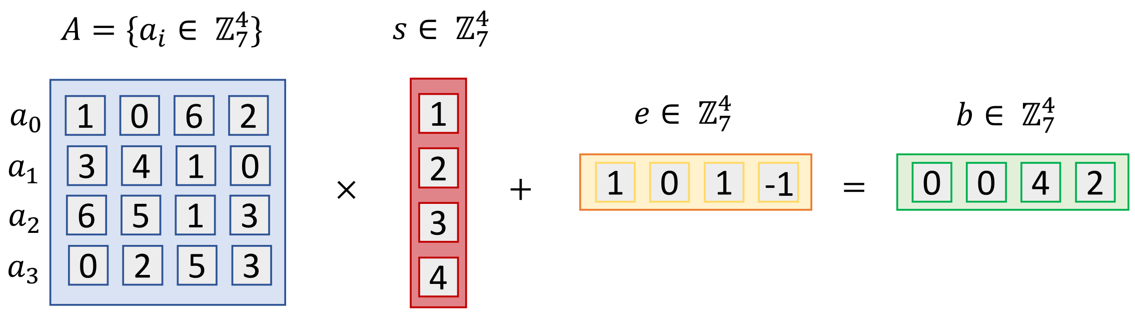


Figure 2.2: Simplification of the Learning with Errors problem. Finding s and e from A and b becomes a computationally hard problem.

be represented as $A \times s = b$. With A and b, the system can be easily solved to obtain s.

The LWE problem introduces an equation such that we generate a set of vectors $A = \{a_i \overset{\$}{\leftarrow} \mathbb{Z}_Q^N\}$ that we multiply by the secret $s \overset{\$}{\leftarrow} Z_Q^N$ and add a noise $e \overset{\$}{\leftarrow} \mathbb{Z}_Q$ to obtain $b \in Z_Q$ according to:

$$\langle a_i, s\rangle + e_i = b_i \mod Q \tag{2.7}$$

In this case, only with A and b it is computationally unfeasible to find s and e. Figure 2.2 shows an example of the procedure mentioned to generate a LWE problem.

The hardness of LWE schemes relies on two problems that try to simplify the task of solving the system of linear equations. On the one hand, the Search LWE [Bra+13] problem tries to obtain s and e from A and b. On the other hand, the Decision LWE [MP13] problem uses an oracle simulation to distinguish a legitimately created A, b tuple from a randomly extracted tuple $A, u \overset{\$}{\leftarrow} Z_Q$ to distinguish a polynomial hiding a message from a random sample.

Producing and transmitting a matrix A is inefficient because of the increased dimensionality of real-world examples. Due to that, an equivalently secure setting is produced in the RLWE problem. The RLWE problem uses polynomial quotient rings for thus generating polynomials in $R_Q = \mathbb{Z}_Q[x]/(x^N+1)$ instead of vectors (e.g., $c_0 + c_1x + c_2x^2 + ... + c_Nx^N \mod Q \mod (x^N + 1)$). In practice, we operate polynomials whose coefficients individually scale according to the coefficient modulus Q, and whose degree scales according to the polynomial degree N. In RLWE, the matrix A is produced from a cycled initial polynomial (i.e., multiplying by x and reducing $\mod Q \mod (x^N + 1)$).

2.3.3 Public Key Encryption from LWE

From the LWE and RLWE problems, we can produce both asymmetric (public-key) and symmetric encryption schemes. This thesis covers the public-key schemes, which are more widespread and useful for privacy-preserving processing. A public-key encryption scheme has two keys: public key p_k and secret key s_k. The LWE-based schemes define the public key as $p_k = (b, A) = (\langle -A, s\rangle + e, A)$ and the secret key $s_k = s$. With a message

$\mu \in \mathbb{Z}_Q^N$ for LWE and $\mu \in \mathbb{Z}_Q[x]/(x^N+1)$ for RLWE, we follow an encryption process to generate a ciphertext $c = Enc(\mu, p_k) = (c_0, c_1)$ with $u \overset{\$}{\leftarrow} \mathbb{Z}_Q^N$:

$$\begin{aligned} c_0 = \mu + p_{k0} \cdot u = \mu + b \cdot u + e = \mu - \langle A, s \rangle \cdot u - e \\ c_1 = p_{k1} \cdot u + e = A \cdot u + e \end{aligned} \tag{2.8}$$

For decryption, the procedure obtains the message μ back from the ciphertext c such that $\mu = Dec(c, s_k)$. For that, multiplying c_1 with s_k results in a close equivalence to:

$$c_1 \cdot s_k = (A \cdot u + e) \cdot s \simeq \langle A, s \rangle \cdot u + e \simeq b \cdot u + e \tag{2.9}$$

By multiplying $c_1 \cdot s$, the result approximates to $c_0 - m$, such that $c_1 \cdot s = c_0 - m$. Clearing the equation results in the following:

$$m \simeq c_0 + c_1 \cdot s \tag{2.10}$$

Generally, to guarantee that the decryption of μ is precise and does not lose significant digits, it is multiplied by a scaling factor Δ. This thesis describes the scaling in Subsection 2.3.4 and the CKKS encoding. The procedure involves working with $\mu = \Delta \cdot \mu'$ and upon decryption $\mu' = \frac{1}{\Delta} \cdot \mu$.

2.3.4 CKKS Encoding

In previous sections, we showed that some HE schemes are based on the encryption of an integer polynomial $m \in \mathbb{Z}_Q^N$. To provide an end-to-end explanation of HE, this subsection shows intuitions on the CKKS scheme, which we use in this thesis. It covers transforming a plaintext complex number vector into a plaintext integer polynomial. While this section includes the necessary knowledge for fully understanding CKKS, it is beyond the complexity of previous sections, and complete knowledge is not required to understand the contributions of this thesis. For readers interested in the high-level concepts, we provide the following explanation. In short, this section covers encoding a vector of complex numbers into a polynomial to compliant for use with RLWE through a series of isomorphisms. In a nutshell, there are two essential transformations: the scaling (Δ) and the canonical embedding. The scaling Δ aims to reduce the impact of noise and encoding on the underlying ciphertexts. The canonical embedding is one of the isomorphisms that enables translating from a vector to a polynomial. It uses a particular set of polynomials (cyclotomic polynomials) whose roots are named roots of unity, which are easy to define for specific polynomial degrees. Galois Automorphisms rely on the concept of roots of unity to perform slot rotation. Next, we start with a detailed description of the concepts.

CKKS uses a specific type of polynomial quotient rings $\mathbb{Z}_Q[x]/(x^N+1)$, named *cyclotomic polynomials* $\Phi_M(X)$. These polynomials have special properties desirable for encoding on certain degrees (e.g., primes or even degrees). In this case, the scheme profits from two of these properties. First, when the degree M is a power of 2, the cyclotomic polynomial is defined by $\Phi_M(X) = (X^N + 1)$. Second, the roots of unity of a *cyclotomic polynomial* of degree M are defined by $\xi_M = e^{\frac{2i\pi}{M}}$ (i.e., the N roots or solutions of the polynomial are $\xi, \xi^3, ..., \xi^{2N-1}$).

CKKS encodes a plaintext vector $z \in \mathbb{C}^{N/2}$, in a plaintext polynomial $m(X) \in \mathbb{Z}_Q[x]/(x^N+1)$ (i.e., CKKS allows encrypting a vector of complex values in an integer ring polynomial) through a series of isomorphisms that generate equivalent representations in various domains. We can view these isomorphisms as information transformations in different contexts. These isomorphisms have two main goals: first, they transform from the complex number domain into the integers domain, and second, they encode the vector into a polynomial (and vice-versa for decoding). Because the vector encoding establishes certain restrictions on the polynomials, we first describe the vector encoding and then the complex-integer number transformation.

In order to transform a vector from $z \in Z_Q^N$ into a polynomial $m(X) \in \mathbb{Z}_Q[x]/(x^N + 1)$, the CKKS scheme leverages the canonical embedding, a transformation defined by $\sigma\colon \mathbb{Z}_Q[x]/(x^N + 1) \rightarrow \mathbb{Z}^N$. As defined, the canonical embedding allows the extraction of the vector z by evaluating the polynomial on the roots of unity such that $\sigma(m) = (m(\xi), m(\xi^3), ..., m(\xi^{2N-1})) \in Z^N$. In summary, there exist N equations following $z_i = m(\xi^{2i-1}) = \sum_{j=0}^{N-1} \alpha_i \cdot (\xi^{2i-1})^j$, which can be transformed into a system of linear equations such that $A \cdot \alpha = z$. A is the *Vandermonde* matrix of $(\xi^{2i-1})\colon i = 1, ..., N$, where each row A_i is defined as the geometric progression of $A_{i,j} = (\xi^{2i-1})^j$. Solving the system of linear equations as $\alpha = A^{-1} \cdot z$ results in the function σ^{-1} to encode our polynomial from z (i.e., $m(X) = \sigma^{-1}(z) = \sum_{i=0}^{N-1} = \alpha_i \cdot X^i$). In a nutshell, the canonical embedding composes the decoding routine (i.e., from integer polynomial to integer vector). The canonical embedding is deduced into a vector matrix multiplication of the initial vector z and the inverse of the Vandermonde matrix (A^{-1}) to elaborate the encoding routine.

After this step, a straightforward method exists to encode an integer vector into an integer polynomial (and vice-versa). However, CKKS defines a series of isomorphisms that transform from a complex number vector $z \in \mathbb{C}^{N/2}$ to an integer vector $z \in Z^N$.

The first observation is the need to reduce from N to $N/2$. Because LWE works with integer polynomials (i.e., real numbers), we only consider the real part of complex numbers, not the imaginary part. This fact defines the number space $\mathbb{H} = z \in \mathbb{C}^N : z_j = \overline{z_{-j}}$, where complex numbers are equivalent if they are the conjugate of each other (i.e., where the real part is equal and the imaginary part is opposite). Furthermore, this implies that cyclotomic polynomials evaluation is also equivalent for conjugates such that $m(X) \in \mathbb{Z}_Q[x]/(x^N + 1), m(\xi^j) = \overline{m(\xi^{-j})} = m(\overline{\xi^{-j}})$, translating into a reduction of the dimensions of the space from N to $N/2$ (i.e., the number of polynomial roots of unity is halved). In CKKS, the first isomorphism is defined by $\pi\colon \mathbb{H} \rightarrow \mathbb{C}^{N/2}$ (i.e., used for decoding and reducing the dimensions by a factor of 2). The inverse of the isomorphism π^{-1} expands the dimensions from $\pi^{-1}\colon \mathbb{C}^{N/2} \to \mathbb{H}$ by copying the conjugate of the elements. With that, we increase the dimensions of our initial vector $z \in \mathbb{C}^{N/2}$ to $z \in \mathbb{H}$.

The transformation from $\mathbb{H}$ to $\sigma\colon \mathbb{Z}_Q[x]/(x^N + 1)$ is not straightforward as there is no isomorphism between them (i.e., the transformation is not bijective). The transformation first performs a transformation from complex numbers ($\mathbb{H}$) to real numbers ($z \in \mathbb{R}^N$) and then from real numbers into integers ($\mathbb{Z}^N$). For transforming into real numbers, CKKS profits from the fact that $\mathbb{Z}_Q[x]/(x^N + 1)$ forms an orthogonal integer basis $(1, X, ..., X^{N-1})$, and given the isomorphism given by the canonical embedding, the basis also exists in $\sigma(\mathbb{Z}_Q[x]/(x^N + 1))$ as $\beta = (\beta_1, \beta_2, ..., \beta_N) = (\sigma(1), \sigma(X), ..., sigma(X^{N-1})$. We can use this basis to project the complex numbers onto it, through Hermitian products, to obtain real coefficients such that $z = \sum_{i=1}^{N} z_i \cdot \beta_i$ with $z_i = \frac{<z,\beta_i>}{||\beta_i||^2}$, where β vectors are obtained the same as in the canonical basis (i.e., $\beta_i = \sigma_i(\xi^{2N-1})$) and $||\beta_i||$ is the norm of the vector.

Finally, to obtain the transformation from a real to an integer number, CKKS uses a technique named coordinate-wise random rounding, which takes a real number and rounds it such that $P(X = \lfloor x \rfloor) = x - \lfloor x \rfloor$ and $P(X = \lfloor x \rfloor + 1) = 1 - P(X = \lfloor x \rfloor)$. That means that, with high probability, our value z is going to be $\lfloor z \rfloor$ the closer z is to $\lfloor z \rfloor$ and vice-versa if z is closer to $\lfloor z \rfloor + 1$. Since the coordinate-wise random rounding may remove precision from the numbers, the numbers are upscaled (Δ) and downscaled (Δ^{-1}) before encoding and decoding, respectively.

At this point, the encoding has produced from an initial vector $z \in \mathbb{C}^{N/2}$ an integer cyclotomic polynomial in $\mathbb{Z}_Q[x]/(x^N + 1)$. Figure 2.3.

2.3.5 Homomorphic Operations on LWE

An essential point of HE schemes is that they allow the operation of the messages μ from the underlying ciphertext. This section covers five essential operations in these schemes: plaintext addition, ciphertext addition, plaintext multiplication, ciphertext multiplication, and ciphertext rotation. It also serves as a prelude to introducing fundamental operations, such as relinearization and rescaling.

Ciphertext-Plaintext Addition

The addition of a new message μ' to a ciphertext $c = Enc(\mu, p_k)$, denoted as $c_{add} = \mu' + c$:

$$\begin{aligned} c_{add,0} &= c_0 + \mu' = (\mu + \mu') + b \cdot u + e_0 \\ c_{add,1} &= c_1 \end{aligned} \tag{2.11}$$

The decryption of c' yields the expected result showing that the decrypted message is equivalent to the sum of the plaintext to the polynomial:

$$\mu_{add} \simeq c'_0 - c'_1 \cdot s = \mu + \mu' + e \tag{2.12}$$

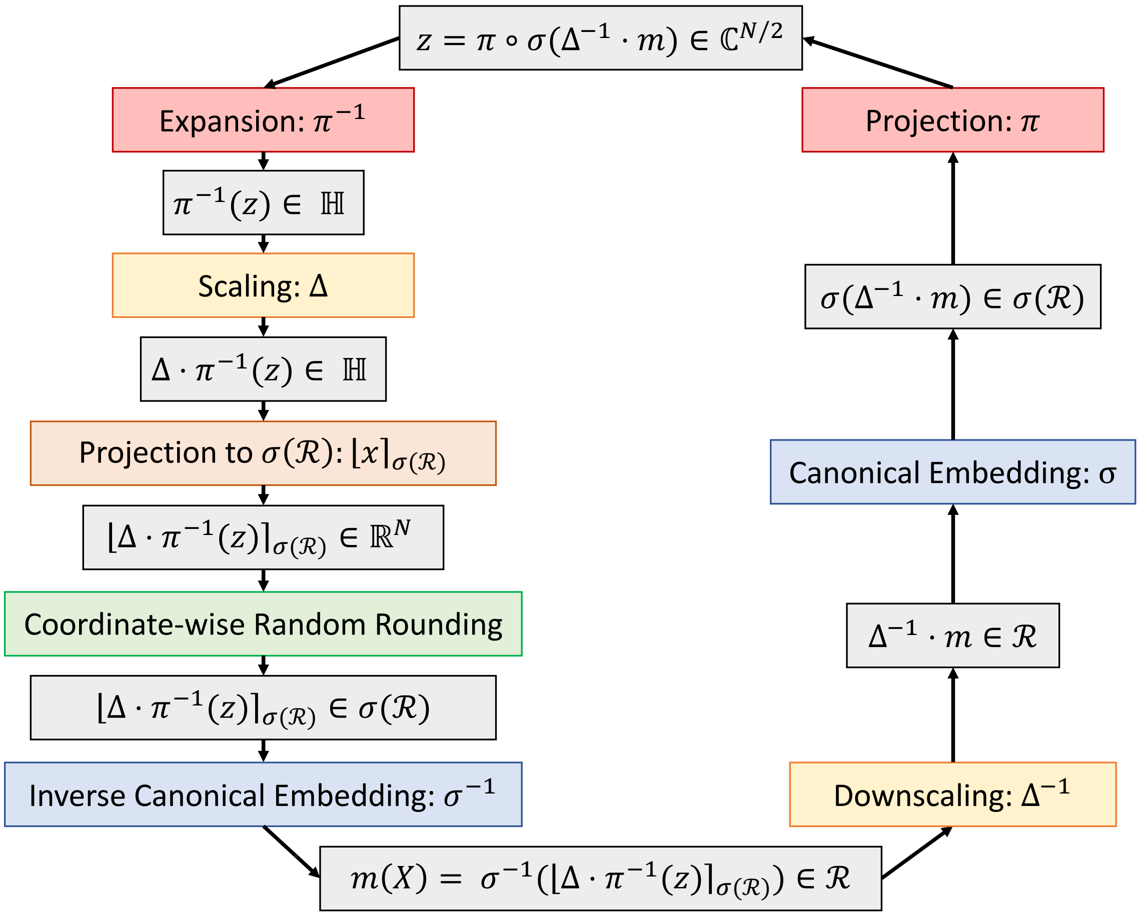

$z = \pi \circ \sigma(\Delta^{-1} \cdot m) \in \mathbb{C}^{N/2}$
Expansion: π^{-1}
$\pi^{-1}(z) \in \mathbb{H}$
Scaling: Δ
$\Delta \cdot \pi^{-1}(z) \in \mathbb{H}$
Projection to $\sigma(\mathcal{R})$: $\lfloor x \rceil_{\sigma(\mathcal{R})}$
$\lfloor \Delta \cdot \pi^{-1}(z) \rceil_{\sigma(\mathcal{R})} \in \mathbb{R}^N$
Coordinate-wise Random Rounding
$\lfloor \Delta \cdot \pi^{-1}(z) \rceil_{\sigma(\mathcal{R})} \in \sigma(\mathcal{R})$
Inverse Canonical Embedding: σ^{-1}
$m(X) = \sigma^{-1}(\lfloor \Delta \cdot \pi^{-1}(z) \rceil_{\sigma(\mathcal{R})}) \in \mathcal{R}$
Downscaling: Δ^{-1}
$\Delta^{-1} \cdot m \in \mathcal{R}$
Canonical Embedding: σ
$\sigma(\Delta^{-1} \cdot m) \in \sigma(\mathcal{R})$
Projection: π

Figure 2.3: Summarization of the encoding and decoding routines of CKKS. For notation simplicity, the figure uses $\mathcal{R} = \mathbb{Z}_Q[x]/(x^N + 1)$

Ciphertext-Ciphertext Addition

The addition of two ciphertexts $c = Enc(\mu, p_k)$, and $c' = Enc(\mu', p_k)$ can be proven similarly to plaintext addition:

$$\begin{aligned} c_{add,0} = c_0 + c'_0 = \mu + b \cdot u + e + \mu' + b \cdot u + e = (\mu + \mu') + 2 \cdot (b \cdot u + e) \\ c_{add,1} = c_1 + c'_1 = 2 \cdot (A \cdot u + e) \end{aligned} \tag{2.13}$$

The decryption yields the expected result:

$$\begin{aligned} \mu_{add} = c_{add,0} + c_{add,1} \cdot s = (\mu + \mu') - 2 \cdot (\langle A, s\rangle \cdot u + e) + 2 \cdot (A \cdot u + e) \cdot s \\ = (\mu + \mu') + e \end{aligned} \tag{2.14}$$

Ciphertext-Plaintext Multiplication

The multiplication of a plaintext value μ' with a ciphertext $c = Enc(\mu, p_k)$ can be proven as:

$$\begin{aligned} c_{mul,0} = c_0 \cdot \mu' = (\mu + b \cdot u + e) \cdot \mu' = (\mu \cdot \mu') + (\mu \cdot b \cdot u) + (\mu \cdot e) \\ c_{mul,1} = c_1 \cdot \mu' = (\mu' \cdot A \cdot u + e) \end{aligned} \tag{2.15}$$

The decryption yields the expected result:

$$\begin{aligned} \mu_{mul} = c_{mul,0} + c_{mul,1} \cdot s = (\mu \cdot \mu') - (\mu' \cdot \langle A, s\rangle \cdot u) + \mu' \cdot \langle A, s\rangle \cdot u + 2 \cdot \mu' \cdot e \\ \simeq (\mu \cdot \mu') + e \end{aligned} \tag{2.16}$$

However, we note that the noise significantly increases with the multiplication operation compared to the addition operation.

Ciphertext-Ciphertext Multiplication

The multiplication of two ciphertexts $c = Enc(\mu, p_k)$ and $c' = Enc(\mu', p_k)$ has more complexity. If we consider the decryption circuit such that:

$$\begin{aligned} \mu_{mul} = Dec(c, s_k) \cdot Dec(c', s_k) = (c_0 + c_1 \cdot s) \cdot (c'_0 + c'_1 \cdot s) \\ = (c_0 \cdot c'_0) + (c_0 \cdot c'_1 + c'_0 \cdot c_1) \cdot s + (c_1 \cdot c'_1) \cdot s^2 = (d_0, d_1, d_2) \end{aligned} \tag{2.17}$$

Equation 2.17 shows how the multiplication of two ciphertexts generates a different polynomial multiplied by s^2. This fact raises a problem: With each multiplication, the size of the ciphertext would grow exponentially in the size of polynomials. To solve this problem, after each multiplication, a *relinearization* of the polynomial is performed, which is described next.

Relinearization

As mentioned, the degree of the polynomial grows while holding a specific number of operations. Given the exponential increase in the size of the polynomials, the relinearization phase aims to find a matching pair of polynomials $c' = (c'_0, c'_1)$, whose decryption yields the same result as the increased degree polynomial:

$$Dec(c_{mul}, s_k) = (d'_0 + d'_1 \cdot s) = d_0 + d_1 \cdot s + d_2 \cdot s^2 = Dec(c, s_k) \cdot Dec(c', s_k) \tag{2.18}$$

The envisioned solution consists of obtaining an encrypted version of $d_2 \cdot s^2$ that we can use to subtract from the original polynomial to reduce its degree. This procedure is usually carried out through what is called the relinearization key ($rl_k = (b \cdot u + e + s^2, A \cdot u + e)$), which is an encryption of s^2 that allows the obtention of an encryption of $d_2 \cdot s^2$.

However, there is an essential flaw in the reasoning described by Fan and Vercauteren [FV12]. When performing $d_2 \cdot rl_k$, we obtain $d_2 \cdot s^2 + d_2 \cdot e$, and due to d_2 being a big polynomial, $d_2 \cdot e$ is not negligible after some number of multiplications. Therefore, the notion of *rescaling* or *modulus switching* is introduced, which also permits reducing the noise while simultaneously creating a Leveled Homomorphic Encryption scheme.

Rescaling

Generally, most homomorphic encryption schemes involve a rise in noise when performing multiplications (i.e., more in the case of ciphertext-ciphertext multiplications but also quite noticeably with ciphertext-plaintext multiplications).

In modulo switching, the modulus Q is defined as a multiplication of smaller lower bit co-primes q such that: $Q = \prod_{i=0}^{l} q_i$. In CKKS, Q is defined as $Q = \Delta^l \cdot q_0$, where q_0 defines the integer precision, and Δ is the scale used for the encoding which also defines the decimal precision such that $q_0 = 2^{\#bitsinteger} \cdot 2^{\#bitsdecimal}$. If we have $\Delta = 2^{20}$ and $q_0 = 2^{32}$, we would have 32 bits of precision, 12 of those for the integer part and 20 for the decimal part.

The rescaling process reduces a ciphertext c at a given level l to level $l - 1$ while reducing the noise. The modulo at level l is defined by $q_l = \Delta^l \cdot q_0$ and the rescaling operation as:

$$Rescale_{l \to (l-1)}(c) = \lfloor \frac{q_{l-1}}{q_l} c \rceil \mod q_{l-1} = \lfloor \Delta^{-1} \cdot c \rceil \mod q_{l-1} \tag{2.19}$$

In practice, the rescaling operation would involve working with huge numbers. For practicality, polynomials are operated with the Chinese Remainder Theorem, which assumes that the co-prime numbers of the remainder can be worked out independently to operate on smaller polynomials independently. The Chinese Remainder Theorem introduces the need to operate on more polynomials but under smaller numbers. This modification involves that instead of using Δ, we often have to find numbers close to Δ but being co-prime between each other p_l.

$$Rescale_{l \to (l-1)}(c) = \lfloor p_l^{-1} \cdot c \rceil \mod q_{l-1} \tag{2.20}$$

Rotations

The previous operations act on the whole encrypted and encoded vector. However, rotations are valuable to the toolset to simplify algorithms and profit from SIMD operations. Rotations permute the vector entries by operating on the polynomial [HS14b]. Rotations use what is known as Galois Automorphisms. Galois Automorphisms profit from the roots of unity (ξ) of a cyclotomic polynomial (defined in Subsection 2.3.4) in order to produce an equivalent permuted representation. By evaluating a polynomial in a different root of unity, we obtain another equivalent polynomial whose contents are the same but rotated $||m(\xi)|| = ||m(\xi^j)||$.

In order to profit from Galois Automorphisms, we generate something known as Galois Keys, which can rotate a ciphertext. The Galois Keys are generally obtained for multiples of 2 since combining those can effectively produce any rotation.

2.4 Secure Multiparty Computation Techniques

There are two main approaches for privacy-preserving computation, either computing on encrypted data (HE) or distributing the knowledge among different parties (SMPC). This section provides an overview of the techniques for implementing privacy-preserving computation through SMPC. We next describe the basic notions underpinning these techniques and advances that have enabled their applications to DL and enable us to describe Chapter 3.

Secure Multiparty Computation (SMPC) is the term used to refer to the techniques that permit a set of n parties to perform computations on input data from each party without revealing it to the other parties and to output a shared, common result. Multiple constructions support SMPC. Each has particular requirements and benefits and is often combined with others. We refer to the work by Lindell [Lin20] for a detailed description of SMPC.

2.4.1 Oblivious Transfer

Oblivious Transfer (OT) is a 2-party cryptographic protocol allowing a receiver to request k out of n pieces from a sender. The protocol ensures that the sender learns nothing about the information sent. The receiver learns nothing about the pieces of information he does not receive [Rab05]. This protocol can be extended to create a boolean SMPC protocol [NP99]. Additionally, it is used as a base for secure data exchange in other protocols.

1. The receiver chooses a bit $b \in \mathbb{Z}_2$. The sender creates public key parameters $sk, pk \overset{\$}{\leftarrow} KeyGen(1^\lambda)$ and another public key draw from random pk'. Then it chooses $pk_b = pk$ and $pk_{1-b} = pk'$ and sends those as p_0, p_1; in that way, the sender knows which key is the correct key and the random key.
2. The sender encrypts respectively with the keys $c_o = Enc_{pk_0}(x_0)$ and $c_1 = Enc_{pk_1}(x_1)$.
3. The receiver can only decrypt c_b because the other key is indistinguishable from random.

The most basic construction is based on the existence of trapdoor functions and works in the following way:

1. The sender does the following:
   - It creates a public-private keypair $sk, pk \overset{\$}{\leftarrow} KeyGen(1^\lambda)$.
   - It creates two random numbers $x_0, x_1 \overset{\$}{\leftarrow} \mathbb{Z}$.
   - It transmits pk, x_0 and x_1.
2. To answer, the receiver will issue a commitment in this phase with the following:
   - Generates a random number $k \overset{\$}{\leftarrow} \mathbb{Z}$ and $b \in \{0, 1\}$.
   - Answer back with $v = x_b + Enc_{pk}(k)$.
3. Finally, the sender creates two keys and applies the xor operation of these with the messages. Only one of the two will be able to be decrypted by the receiver, and the other will be random information. The procedure is the following:
   - The sender computes $k_0 = (v - x_0)^{sk}$ and $k_1 = (v - x_1)^{sk}$.
   - The sender answers back with $m'_0 = m_0 \oplus k_0$ and $m'_1 = m_1 \oplus k_1$
4. The receiver is only able to decrypt one of the two messages since he committed, so it decrypts the corresponding by $m_b = m'_b \oplus k$

The receiver can only commit to one value, and it will determine what it receives, and since k is private sender learns nothing about the choice. The advantage is that the commitment reveals no information since k is kept secret, and there is no way for the receiver to modify the commitment so that it can reveal information from the other secret. Although it is not the most efficient option, OT can be used as a means for Secure Multiparty Computation by secretly obtaining the results of each phase.

2.4.2 Yao's Garbled Circuits

Yao's Garbled Circuits is a 2-party secure computation cryptographic protocol for boolean circuits [Yao86] in the presence of HBC adversaries. It allows two parties to compute their private inputs x, y without knowing each others' input or the circuit.

In the basic protocol, a Garbler (G) owns inputs x, and the circuit C; the Evaluator owns some input x'. The protocol works as follows:

1. G garbles the circuit $g(C) = C'$. The garbling consists of generating six symmetric encryption keys per input of binary gate, four associated with the two input values and two associated with the output. Each key is given a value (0 or 1), and the output encryption keys are encrypted with the appropriate combination of input keys. For a gate with input wires a, b and output wire c would generate six keys $(k_a^0, k_a^1, k_b^0, k_b^1, k_c^0, k_c^1)$. Each output wire key is encrypted with the corresponding input keys. For example, for an AND gate:
   - For output 0, 0: $Enc(k_a^0, Enc(k_b^0, k_c^0))$
   - For output 0, 1: $Enc(k_a^0, Enc(k_b^1, k_c^0))$
   - For output 1, 0: $Enc(k_a^1, Enc(k_b^0, k_c^0))$
   - For output 1, 1: $Enc(k_a^1, Enc(k_b^1, k_c^1))$

   This procedure is recursively performed for all the circuit gates, and the order of keys is randomized. The garbling of the circuit (C') and the input values associated keys ($g(x)$) are sent to the Evaluator (E).

2. If the Evaluator has specific input values, the garbling ($g(y)$) is sent to the Evaluator using OT. The Evaluator then blindly follows the protocol based on $g(C)$, the garbling x, and its input (i.e.,y).

3. At the end of the evaluation, the garbler may introduce a lookup table that contains the decryption of the final values of the circuit. In that way, all parties can obtain the output value.

Several optimizations followed the initial Yao Garbled Circuits proposal, including the point-and-permute optimization [Pin+09], the half gates [ZRE15], the free XOR gates [KMR14; KS08] or garbled row reduction [BMR90; NPS99].

2.4.3 Secret Sharing

Secret Sharing (SS) is a cryptographic protocol that permits dividing a secret piece of information x into n different 'parts' or shares which can only be rebuilt into the original if at least k parties agree. SS is one of the bases for creating the most modern SMPC protocols with more than two parties. More formally, a (k, n)-secret sharing scheme comprises a pair of algorithms. First, $Share(x)$ produces a tuple of n different shares $(s_1, s_2, ..., s_n)$. Then, $Reconstruction(s_1, s_2, ..., s_k)$ computes and produces the secret x out of k shares. Next, we describe three main protocols used for SS.

Shamir Secret Sharing

Shamir Secret Sharing is based on the Lagrange Interpolation, which states that a polynomial $P(x)$ of degree n can be built from $n+1$ points [Sha79]. The sharing consists of generating a polynomial whose independent term is the secret to be shared. Shamir's secret sharing is efficient since it does not need strong preprocessing. Moreover, it is homomorphic for addition and multiplication. The main inconvenience is that the multiplication of two degree-t shares generates a share of degree $2t+1$. Therefore, it requires degree reduction after each multiplication [GRR98], which requires communications, thus incurring an overhead.

Additive Secret Sharing

Additive Secret Sharing is based on the concept that a given secret x can be decomposed in the sum of n random numbers [Bla79]. The share generation process involves selecting $n-1$ random numbers and computing the n-th share as the sum of the rest. While additive secret sharing is only homomorphic on the addition, Beaver Multiplication Triplets [Bea91] extend it to perform multiplication. This approach is more efficient for computation because it moves the computational delay onto a preprocessing phase (i.e., the multiplication can be performed offline). The protocol for additive secret sharing is defined as follows:

- $Share(x)$. Choose $n-1$ random numbers in $s_i \overset{\$}{\leftarrow} \mathbb{Z}_Q$ which will be $(s_1, s_2, ..., s_{n-1})$. To compute the nth share, we compute $s_n = x - \sum_{i=0}^{n-1} s_i \mod Q$.
- $Reconstruction(s_{i1}, s_{i2}, ..., s_{ik})$. It adds the random numbers in $\mathbb{Z}_Q$. $\sum i = 0^n s_i$ mod Q.

This flavor of secret sharing provides a simple yet efficient alternative both when sharing and on the reconstruction. Additionally, it is homomorphic to addition but not multiplication (i.e., the addition of the shares is equivalent to the regular addition). However, compared to Shamir's scheme [Sha79], it is not homomorphic regarding multiplication. Beaver Triplets are a construction that solves the multiplication need.

Beaver Multiplication Triplets

Beaver multiplication triplets [Bea91] are a construction that permits performing multiplication on secretly shared data. The construction relies on exchanging a secretly shared multiplication of random numbers such as $c = a \cdot b$. There are two assumptions for this process; first, none of the parties knows the real values of the triplets, which then turns into the second assumption, which is the existence of a trusted dealer for the triplets. Assuming we want to compute the n party multiplication of secretly shared $x = (x_1, x_2, ..., x_n)$ and $y = (y_1, y_2, ..., y_n)$; given $a = (a_1, a_2, ..., a_n)$, $b = (b_1, b_2, ..., b_n)$ and $c = (c_1, c_2, ..., c_n)$, such that $c = a \cdot b$, the computation continues as follows:

1. Each party computes $x_i - a_i$, publish and reconstruct $x - a$.
2. Each party computes $y_i - b_i$, publish and reconstruct $y - b$.
3. All the parties can perform the offline computation: $z_i = c_i + x_i \cdot (x - a) + y_i \cdot (y - b) - (x - a) \cdot (y - b)$

After the different z_i are reconstructed in a single element, the different intermediate values are removed from the equation.

2.4.4 Zero-Knowledge Proofs

Zero Knowledge Proof (ZKP) are cryptographic constructions in which a *prover P* verifies that a statement x is part of a language L to a *verifier V*, satisfying completeness (i.e., if $x \in L$, V cannot reject the statement), soundness (i.e., if $x \notin L$, then V only accepts it with 50% probability) and zero-knowledge (i.e., V learns nothing from the statement rather than the commitment and the truth of it) [GMR89]. The probability of accepting a false ZKP can be reduced from 50% by repeating the procedure multiple times.

In the scope of private computation techniques, ZKPs are commonly used to protect against malicious adversaries by requiring the different parties to prove the correct execution of operations. In recent years, there have been multiple works combining ZKP with flavors of SMPC to provide fully verifiable computation [GNS21; Par+13].

2.4.5 Verifiable Secret Sharing

Verifiable Secret Sharing schemes use homomorphic operations considering a malicious adversary [BGW88; Cho+85; Fel87; GMW91]. The main changes to the previous approaches are the sharing of commitments to ensure the order does not interfere; the verification of the correctness of the computations with zero knowledge proofs [BFM88]; the use of agreement schemes [FM89] and distributed coin-flipping protocols [Blu83].

Chapter 3

State of the Art

Recent advances in Privacy-Preserving Computation Techniques (PPCTs) (i.e., Homomorphic Encryption and Secure Multiparty Computation) have enabled Deep Learning (DL) training and inference over protected data, attracting the attention of cryptographers and computer scientists in recent years, with an increasing number of proposals published in a fast rate. However, these techniques still need to mature and be easier to deploy in practical scenarios. PPCTs incur a high computational and communication overhead to the already-demanding DL operations. Due to this, few actual deployments of these technologies exist in privacy-preserving scenarios [Che+20; Kum+20b].

In this chapter, we study the past and present literature on PPCTs applied to DL. For that sake, we provide a systematization of knowledge of the current state of the art. To this end, this chapter reviews the evolution of privacy-preserving computation techniques with DL to understand the gap between research proposals and practical applications. We highlight the relative advantages and disadvantages, considering aspects such as efficiency shortcomings, reproducibility issues due to the lack of standard tools and programming interfaces, or integration with DL frameworks commonly used by the data science community. For interested readers, other works have surveyed various areas of PPCTs [Azr+19; Che+20; Kai+20; Kum+20b; RRK19; Tan+20].

The remainder of this chapter continues as follows. First, we describe the scope and methodology used to select and analyze the articles in Section 3.1. Then, we systematically describe the literature on PPCTs. Next, we describe a set of advanced techniques of SMPC whose impact may be meaningful for DL in Section 3.2. We describe the origins of the research area, with various applications of the initial primitives in Section 3.3. Then, we describe state of the art divided into three broad sections: DL inference in Section 3.4, DL training in Section 3.5, and programming interfaces and compilers in Section 3.6. Finally, we analyze the current limitations that prevent the deployment of existing solutions in real-world settings, primarily due to deficiencies related to the efficiency and usability of the proposals, and discuss research lines that the community should address in the following years in Section 3.7. Some of these research directions impact the scope of this thesis. Specifically, those that refer to the complexity and usability of techniques in the context of usability of PPCTs for DL.

3.1 Scope and Methodology

Privacy-Preserving Computation Techniques (PPCTs) ensure the input data's privacy and secrecy, relieving the client endpoint from heavy workloads and allowing deployments in collaborative settings (e.g., various hospitals privately sharing medical information to investigate rare diseases). The main goal of this section is to provide an understanding of the landscape of MLaaS in data-sensitive contexts through privacy-preserving cryptographic computation (i.e. when the data sent to third parties for processing is never decryptable). As a secondary goal, we explore the adaptability of current DL techniques to cryptographic constructions. Privacy-Preserving Computation Techniques are often combined with other techniques as described in Subsection 3.2.3. At those crossing points, we detail the use of the adjacent techniques and their benefit for DL. Accordingly, while Differential Privacy (DP) is a widespread technique in the field of PPDL [Aba+16a; EPK14; Goo+14; JYS18; Pap+16; Pap+18b], we only include proposals that intersect with our scope (we refer to previous work for details on DP [DP19]). Also, Federated Learning (FL) is outside the scope of the paper, given its need for computing infrastructure and the reduced use of cryptographic constructions in the aggregation phase.

PPCTs present two main challenges for proper application with Deep Learning: efficiency and usability.

Efficiency. PPCTs offer a limited operation set and suffer from performance issues when dealing with complex computations. While their deployment is more widespread in less-stringent scenarios, such as private data aggregation or statistics [MZ17; Ohr+16; SS08], their application for DL is not straightforward and introduces reasonable delays. We consider that a proposal improves efficiency by introducing modifications to previous protocols reducing their runtime on DL workloads.

Usability. The second weakness is related to the deployability of these techniques. Many frameworks and tools ease the access for data scientists (not necessarily experts in computing science) to complex DL [Aba+16b; Cho+15; Pas+17]. However, adapting these frameworks to use PPCTs is complex. In this regard, we consider improvements to usability if the proposal simplifies the solution's adaptability to existing DL frameworks (i.e., by providing tools to reduce the overall programming effort). Accordingly, our study includes works that propose Application Programming Interfaces (API), compilers, or relevant practical tools that help implement and deploy the theoretical solutions into practical applications, thus fostering their usability.

Besides allowing for further improvements on the proposal, open-source implementations allow for the reproducibility of the results. Our study analyzes whether the code matches the theoretical claims (i.e., if it fully implements the security mechanisms and features described in the paper), the source code maintenance, and its integration with existing frameworks (e.g., Tensorflow or Keras).

In summary, for each of the proposals using privacy-preserving computation techniques for DL, we study the following: (i) the problem addressed, i.e., training or inference, (ii) the architecture proposed, i.e., centralized, distributed, or hybrid, (iii) the privacy goals and adversarial model assumed, (iv) the particular techniques involved, i.e., SMPC, HE, and others, and, (v) the issues considered regarding efficiency and usability.

To select relevant proposals, we conducted queries in various research repositories and databases, looking for specific keywords (e.g., *Privacy-Preserving*, *Deep Learning*, *Secure [Multiparty] Computation* or *Homomorphic Encryption*). Then, we select those with higher impact (regarding the number of citations) and those published in top venues. We read their abstracts to check whether they fit the scope of our study, which gave us an initial set of works that we carefully analyzed. Then, we apply snowball sampling using the references from the papers in the initial set to add further relevant works. We know this process has limitations, and we might have left out good research works since sometimes quality is related to popularity. Despite this limitation, our study successfully includes all the relevant works proposing PPCTs for DL. While the study was initially carried out in the first half of 2021, the study of the area has been continuous, with particular detail on the topics related to this thesis, and the conclusions have been adapted to match the current research directions. While the study was originally carried out in 2021, the queries and searches have been frequently recreated to update this thesis's state of the art.

3.2 Advanced Privacy-Preserving Computation Techniques

From the initial protocols described in Subsection 2.4 of the previous chapter, more advanced and complex protocols have introduced efficiency and versatility features. In this section, we aim to overview these protocols and provide insights into their features as these translate to the features that DL-based protocols exhibit. Subsection 3.2.1 covers the constructions and improvements to HE. Subsection 3.2.2 aims to describe the significant improvements in SMPC technologies with a link to DL. Subsection 3.2.3 describes a family of techniques that combine different base cryptographic constructions to achieve enhanced feature sets (e.g., enabling multi-arithmetic privacy-preserving processing).

3.2.1 Advanced HE Constructions

Homomorphic Encryption ciphertexts can contain a vector of values allowing for Single Instruction Multiple Data (SIMD) operations, also known as packing. The combination of packing with rotations allows the elaboration variations of algorithms, often resulting in improved efficiency respecting classic Single Instruction Single Data (SISD) operation. Most constructions in the HE realm have relied on adapting existing algorithms to the HE realm.

Halevi and Shoup [HS14a] were the first to propose HE SIMD algorithms in their adaption to HELib [HS14b]. In their paper, they provide details on various algorithms, specifically those that allow using HE Packed vectors as usual operations. They cover various algorithms, from individual entry selection to replication and matrix multiplication.

Cheon et al. [Che+19b] provide an extended relation of algorithms to provide boolean comparison operations in the HE domain based on LHE schemes thanks to mathematical properties and elaborating operations such as absolute value, square root, or comparison operators.

3.2.2 Advanced SMPC Constructions

The basic techniques form the foundation for more advanced security, performance, versatility, and usability in protocols. These protocols are nowadays at the core of many of the proposals for PPDL. SPDZ [Dam+12] is a SMPC protocol for n parties secure against the corruption of $n-1$ parties, which highly improves the security of previous approaches. It combines the following primitives: i) additive Secret Sharing and beaver multiplication triplets for the computation, ii) SHE for data encryption and beaver triplet computation, iii) ZKP to guarantee the correctness of the information, and iv) commitments to avoid malicious inputs. It relies on a computationally expensive preprocessing phase that reduces the cost of the subsequent processing phase. Overdrive [KPR18] reduces the reliance on public-key infrastructure in the preprocessing phase and the use of Beaver Triplet distribution with HE and distributed decryption. Furthermore, it optimizes the most expensive part of the protocol, the execution of ZKP, by producing a more efficient Schnorr-like protocol [Sch89]. MASCOT [KOS16] raises as the counterpart to SPDZ. In MASCOT, authors introduce modifications and remove the complexity from the preprocessing phase, where the secure protocol results in only six times slower than its non-secure counterpart. The main drawback in MASCOT is the communication delay incurred by using OT, which they overcome by introducing an OT extension that increases throughput.

Other works have attempted to speed up the computation of these protocols. Tiny-

Garble [Son+15] presents a methodology optimizing multiple aspects of Yao's Garble Circuits and defining an option for them to execute a MIPS I processor instruction set. The optimization permits scaling the computation and utilizing more complex directives, such as those of a more complex computing architecture.

3.2.3 Hybrid Techniques

Most PPCTs have a limited instruction set efficient to operate in a specific arithmetic domain. However, many problems require operating on different arithmetic domains, often forcing one to approximate the problem to specific arithmetic types partially. These approximations cause inefficiencies in terms of performance, precision, and flexibility. For example, in the case of DL, linear functions are computed efficiently with floating-point arithmetic, whereas non-linear activation functions require boolean arithmetic. To avoid the use of approximations, some authors have proposed what we define as *Hybrid Techniques*, also referred to as *share/ciphertext conversion protocol* [WGC18]. These techniques permit switching from one PPCTs algorithm to a different one, thus adapting to use the required arithmetic type while preserving the privacy of the construction. Hybrid techniques effectively improve the solutions' flexibility and preserve the computation's efficiency and accuracy since the internal functions do not require approximation.

Hybrid techniques might combine different base cryptographic protocols from a single PPCTs, i.e., HE or SMPC, and propose conversions from one to the other [JVC18]. As we analyze in Section 3.4.3, various proposals use such hybrid techniques due to the arithmetic variety of internal functions applied in DL.

For HE, CHIMERA [Bou+20] presents a framework that allows switching between three main HE schemes without decryption. Concretely, it proposes using BFV [FV12], HEAAN (CKKS [Che+17]) and TFHE [Chi+16] for integer, floating-point and boolean arithmetic respectively. It has a strong potential for DL since linear functions can be executed in floating-point arithmetic, whereas activation functions rely on boolean arithmetic, thus not needing an approximation.

Similar to HE, SMPC suffers from using a single arithmetic type. ABY [DSZ15] (Arithmetic-Boolean and Yao's sharing) gives the programmer access to three protected data types: arithmetic secret shared, boolean secret shared, and Yao's GC. The most important contribution is that they provide efficient cryptographic bridges between the different constructions. ABY is a crucial contribution as it fosters various subsequent proposals in PPDL [Cha+17a; MR18; Ria+18].

Even though ABY offers a higher-level abstraction due to the provision of data types, it remains a complex low-level notation, requiring detailed knowledge, and optimal use remains a programmer's responsibility. EzPC [Cha+17a] partially solves this problem by adding a new layer of abstraction and generating a two-party computation protocol from the high-level description of the language. This layer hides the cryptographic details from the user and selects the parameters automatically. EzPC is a cross-compiler that translates C++ into 2-party secure code using ABY beneath.

3.3 Machine Learning Approaches

ML and DL were not designed considering privacy and security goals. While the research area of PPDL is relatively new, there are multiple precedents on private data analytics and machine learning which are the basis of proposals for PPDL inference and training. The application of PPCTs for data mining and machine learning solutions has been addressed before the growth in popularity of DL [BR18; VC04]. Indeed, the idea of secretly evaluating a neural network was first proposed in 2008 by Sadeghi and Schneider [SS08] where the authors propose a distributed 2-party computation paradigm where the security relies on the secrets each of the parties stores. The authors use SMPC based on OT transforming the NN using a generalized universal circuit (i.e., any circuit can be simulated in boolean arithmetic).

ML Confidential [GLN12] proposes an approach to use linear regression models with LHE. This paper is one of the first attempts that propose a privacy-preserving machine learning solution using encryption and covers various architectural issues and problems, such as the polynomial approximations or the fixed-depth of circuits.

In CodedPrivateML [So+19], the authors train machine learning models (i.e., linear and logistic regression) using Shamir's secret sharing and speed it up with the Lagrange coding. This paper proposes a solution based on cloud computing by distributing the workload to train the algorithms.

Next in relevance, TASTY [Hen+10] presents a distributed 2-party proposal aiming to combine the Paillier PHE cryptosystem with other structures to achieve the execution of other operations, such as multiplication. Additionally, the authors present a compiler that simplifies the translation of code written in the Keras framework [Cho+15] to the TASTY protocol. It constitutes one of the first attempts to combine theoretical concepts with actual deployments.

Collaborative data analytics have been an area of research for PPCTs. The work by Ohrimenko et al. [Ohr+16] enables different parties to interact through Trusted Execution Environment (TEE) and oblivious access to data structures. In this case, the performed task is known by all the parties, but the access patterns are hidden, so no side channel data is released.

These works are examples of the ideas that form the basis for posterior contributions in DL inference and training, which we analyze in the following sections.

3.4 Privacy Preserving Deep Learning Inference

One of the main conceptions of MLaaS was the provision of services, whereby uploading a DL model could be used for inference. This process involves transmitting information that, in some cases, may be confidential. In this section, we analyze proposals for protecting the privacy of the data sent for inference. There are three main approaches, i.e., using HE, SMPC, or hybrid techniques. We also analyze proposals that aim to ease the abstraction of these techniques for existing DL frameworks through programming interfaces and tools (e.g., compilers).

3.4.1 PPDL Inference in Centralized Architectures with Homomorphic Encryption

One of the main goals framed in this section is to adapt Neural Networks to work with Homomorphic Encryption Schemes of various types. As mentioned before, the emergence of FHE schemes implied a step forward for PPDL. Due to the computational complexity incurred by these schemes, the first approaches using FHE are designed for a centralized environment and use high-performance hardware.

We consider a client-server scenario, where a client needs to outsource the computation of DL inference to a not-necessarily trusted server. For that, the client and server rely on public-key FHE. The client performs the key generation, encrypts the data with his public key, and sends the ciphertext to the server. The server receives the public and relinearization keys and the ciphertext. The server can operate on the data through privacy-preserving processing and return the result to the client. The client owns a private key which he never released, therefore is the only person able to decrypt the information. Also, we consider that the client performs no computation except light tasks before encrypting or after decryption. For example, the client can perform padding to reduce the load of convolutions since it is a soft task. Also, after decrypting the information, the client can retrieve the relevant information entries instead of having the server post-process the result.

Cryptonets [Gil+16; Xie+14] is considered the first shot at adapting DL to work with LHE. The goal was to adapt the structures of a Neural Network to work to use only addition and multiplication. The authors propose pixel-wise encryption (i.e., a pixel per ciphertext) and aim to improve inference time by packing the same pixel of multiple images. Concretely, the authors proposed a modification of the ReLU activation function using a square function and substituting the max pooling layer with an average pooling layer. However, this work does involve a significant amount of operations which may be non-optimal. Faster Cryptonets [Cho+18] improves the original proposal using a *quantization* scheme, i.e., a pruning technique for neural networks and optimal approximations.

These optimizations reduce the number of operations performed, the width of the circuits, and the number of performed operations. One of the areas for improvement of Cryptonets [Gil+16] and its subsequent optimization in Faster Cryptonets [Cho+18] is that the activation functions could be more precise, especially in the training phase. Finding a DL model that converges with square activation functions is a rather brute-force procedure. ReLU functions are paramount for the success of DL. Thus, in CryptoDL [HTG17], authors propose different approaches to approximate the ReLU activation using low-degree polynomials. Their solution relies on using the Chebyshev Polynomial approximation of the integral of the sigmoid function. Furthermore, profiting from binary reductions, with relatively high-degree polynomials, can be approximated with only $log_2(O^N_{\mathcal{D}})$ where $O^N_{\mathcal{D}}$ is the multiplication depth of the circuit.

Low-Latency Inference, LoLa [BGE19], supposed the latest evolution to the Cryptonets protocol introducing new operation layouts. In this work, the authors cover different layouts or plaintext-ciphertext mappings for the vectorization of SIMD operations for the first time. These mappings allow for faster inference, more complex NN, and vectorized algorithms.

On a sideline, the procedure to achieve accurate predictions tackled Boolean Neural Networks and Integer Neural Networks. TAPAS proposes optimizations in the domain of Boolean arithmetic [San+18]. Authors propose using Binary Neural Network (BNN) and the TFHE library [Chi+20a]. TAPAS allows first to make the matrix multiplication faster-adapting multiplication functions to the XNOR gate and then to count the number of ones in the result for the actual summation. TAPAS BNNs achieve a reasonable speedup for Deep Neural Network (DNN) tasks but work less efficiently on Convolutional Neural Network (CNN) tasks. Interestingly, the authors released this tool open-source, which uses the general-purpose HE evaluation framework SHEEP [BS18]. In FHE-DiNN [Bou+18], the contribution shifts the focus towards using discretized neural networks (i.e., integer weights) based on boolean arithmetics on top of TFHE. The use of these primitives permits reducing the size of the computation and an increase in performance. Given the native use of boolean arithmetics, they claim a performance improvement over Cryptonets. However, due to the changes performed to the neural networks after training, there is a reduction in the accuracy of the resulting model. Although the solutions achieved by these papers show efficient and promising results, they shift the focus toward different trends compared to DL state of the art.

As a last optimization line, some works strive to optimize the execution of the linear activation part of Neural Networks. Wu et al. [WH12] provide one of the first proposals, focusing on noise-efficient implementations of linear regression, mean, and covariance with Somewhat Homomorphic Encryption. Due to the nature of SHE, their work tackles noise management to ensure the operations performed allow correct decryption of the result. The work of Duong et al. [DMY16] goes one step beyond by implementing different embeddings applied when the encoding is performed and used to speed up matrix multiplications. Specifically, they cover two different embeddings, the binary and non-binary, whose placement varies in size and efficiency. These are similar to the replication factors covered in the Matrix-Matrix multiplication algorithm; however, they only cover $m \times m$ matrix multiplication, which could often result inefficient.

More recent works regarding algorithm adaptation have exploited SIMD and ciphertext packing to combine additions, multiplications, and rotations. Specifically, Jiang et al. [Jia+18] is a framework that relies on an encoding that accelerates matrix computations. Due to the improvements in these operations, the authors show an acceleration of DL inference using a specific encrypted CNN model (not trained from encrypted data) to classify images from the well-known MNIST dataset.

3.4.2 PPDL Inference in Distributed Architectures with Secure Multiparty Computation

We consider a scenario where a client needs to outsource the computation of DL inference to a not-necessarily trusted party. For that, the client and server rely on flavors of SMPC. The client performs some sharing of the information and sends the shares to non-colluding parties. The parties perform blind computation if the DL model n on the shares, from which they extract no information. The client must keep a share or ensure the parties are honest. Finally, the client aggregates all shares obtaining the final result.

In this uncertainty of scalability and efficiency, a set of works arise, continuing the ideas proposed in TASTY [Hen+10], with SMPC. Although the protocols of this section use distributed architectures in general, these are often used to hide a two-party association.

MiniONN [Liu+17] is the first proposal using boolean arithmetics for a distributed architecture. It allows private inference (model and data) through SMPC based on Oblivious Transfer. According to the proposed protocol, any DNN can be transformed into MiniONN. The main disadvantage of this work is the inherent communication delay due to the use of OT for SMPC. With the rise in using Yao's Garbled Circuits, DeepSecure [RRK18] proposes a variation of the preprocessing phase for optimizing the DL functions. Since most overhead relies on the communication delay and the preprocessing phase of garbled circuits, the authors propose an optimized version of DL models to reduce the information exchanged. Still, using garbled circuits implies *regarbling* all the circuits for each input, which entails considerable overhead.

XONN proposes optimized circuits of XNOR gates that permit faster computations of matrix multiplications. Similarly to TAPAS [San+18], XONN [Ria+19] proposes using the boolean representation of BNNs for inference using Yao's Garbled Circuits. Additionally, it introduces a pruning algorithm for the neural network to generate a compressed representation for inference and improve the garbling to set a constant number of communication rounds for architectures of up to twenty-one layers. Finally, they provide an API that permits translating a Keras [Cho+15] model into the proposed protocol.

Dalskov et al. [DEK20] propose a quantization scheme that enhances transmission and speeds up the processing of DNN, considering malicious adversaries. The proposal is evaluated with Additive Secret Sharing, and the goal is to reduce the dependency on Beaver Triplets. If those are needed, the goal is to reduce the communication overhead using 16-bit and 8-bit floating-point numbers. This approach requires both the client and server to perform the computation or the existence of a second independent server for secret-sharing to be securely deployed.

3.4.3 PPDL Inference in Hybrid Architectures

In the previous section, we observed how HE is a good technique for computing linear and polynomial operations. However, converting non-linear functions introduces a cost of precision and efficiency. On the other hand, SMPC scales poorly to many parties due to communication delays. Moreover, adapting a DL model to a single arithmetic type (e.g., integer, fixed-point, floating-point, or boolean) restricts its operations and deteriorates accuracy. Hybrid techniques aim to bridge the best of both worlds by combining and supporting operations in different arithmetic fields and bridging some gaps in previous approaches.

Chameleon [Ria+18] is the first to propose a multi-arithmetic framework for DL. It leverages ABY [DSZ15] to switch between different arithmetic types during computation. Concretely, it uses fixed-point arithmetic for the linear activation functions and boolean (based on Yao's Garbled Circuits and GMW [GMW87; Ode09]) for the non-linear ones while profiting from the contributions presented by Cryptonets [Gil+16] and CryptoDL [HTG17]. GAZELLE [JVC18] proposes a SMPC protocol based on a hybrid approach for DL inference considering an HBC adversary. On the one hand, they use Packed Additively Homomorphic Encryption (PAHE) for the linear algebra functions. PAHE speeds up vector and matrix computations and is designed to avoid bootstrapping and complex architectural encoding decisions. On the other hand, they use Yao's Garbled Circuits for non-linear functions (e.g., ReLU or MaxPool). Due to the need to combine PAHE and GC, they create a cryptographic bridge for switching between these cryptographic structures. In this way, they present a speedup of 30x compared to Chameleon [Ria+18] and MiniONN [Liu+17]. The work of GAZELLE was a breakthrough that was followed up and compared by different works. As an improvement to GAZELLE, DELPHI [Mis+20] combines GC and quadratic polynomials for the activation functions. A fundamental aspect of DELPHI is the generation of several derived models from the original using different activation functions. Depending on particular goals, the activation functions can be substituted by either garbled circuits (i.e., accurate approach) or quadratic polynomials (i.e., efficient approach). Since either performance or accuracy is degraded, DELPHI uses a planner to calculate an optimal trade-off that minimizes the loss. They also claim that the protocol permits reducing the amount of information exchanged by 90% with respect to GAZELLE.

3.5 Privacy Preserving Deep Learning Training

While DL inference is a complex procedure when coupled with PPCTs, it ignores many of the challenges that the more complex and intense computation of DL training backpropagation involves. Among the challenges, the need to ensure unbounded computation and the approximations of not only the activation functions but also the derivatives of the activation functions are challenges that these sections below need to address.

There are two prominent use cases for DL training. On the one hand, when a party wants to outsource a model's training to a third party without disclosing the dataset or the model itself. On the other hand, in various scenarios, similar to Federated Learning [Kon+16], many parties want to interact to produce a DL model securely and preserve the privacy of their respective datasets.

3.5.1 HE for PPDL Training

The first constructions to extend HE-based inference into training implement a two-party-based computation where the client helps perform backpropagation. The use of HE for PPDL training consists of the encryption of the dataset and the model. Then, it is sent to a third party for blind computation over the encrypted data. After some epochs, the information is returned to the initial user, who decrypts it.

Hesamifard et al. [Hes+18] propose a continuation to CryptoDL [HTG17] for training with LHE. In addition to the feasibility, they test their approach on multiple datasets. Given the lack of efficient bootstrapping, they opt for sending back the data to the client, who re-encrypts it and sends it back to the server for computation. However, it does not guarantee a holistic process and requires client interaction. Nandakumar et al. [Nan+19] provide the first end-to-end proposal for privacy-enhanced training on the server side based on HE. They adapt the complete DL pipeline, including the loss and stochastic gradient descent functions. They implement optimizations, thanks to which they perform encrypted training in 40 minutes.

3.5.2 SMPC for PPDL Training

Using SMPC for PPDL training consists of a client producing shares of the model and dataset. Then the information is sent to non-colluding servers, which perform the computation blindly. After some epochs, the shares of the model are returned to the client, who combines them into a working model.

Similarly to HE, most approaches relying on SMPC perform training among two-party scenarios. Due to the peculiarities of SMPC, these protocols sometimes assume the existence of a trusted third party that acts as a randomness provider. One of the first attempts to perform PPDL training was proposed in 2017 by Chase et al. [Cha+17b]. In their work, authors combine differential privacy for data protection with SMPC through additive secret sharing for collaborative gradient descent computation. While it introduces a secure collaborative protocol for computing DL training, a reasonable amount of the protocol is protected by DP. The main drawback is the accuracy loss accounted for due to the noise introduced to guarantee a privacy budget. In a more complex approach, SecureML first proposed a semi-honest client-server distributed approach to training linear regression, logistic regression, and neural networks [MZ17]. The protocol consists of two phases: an offline preprocessing phase and the private computation online phase. The offline phase is expensive since it requires generating Beaver Triplets for the rest of the computation. Thus, SecureML optimizes this generation using vectorized LHE and OT. Then, the online phase uses Additive Secret Sharing, Beaver Triplets for the shared computation, and Yao's GC for bit-level arithmetics. They introduce a second server that acts as a non-colluding party to relieve the client from computation requirements.

Other papers introduce various approaches to improve this efficiency. For example, QUOTIENT [Agr+19] profits from OT-based SMPC to improve the adaptation techniques such as batch normalization and introduce adaptive gradient methods to improve training efficiency. Other protocols, such as FLASH [Bya+20; Cha+19], focus on improving malicious adversary framework protections. It includes Guaranteed Output Delivery, which ensures that the protocol finishes even if the parties are dishonest and that the model is delivered to all parties or none. For that, they create a bi-convey primitive that evolves from a commitment scheme and permits two honest parties to discover another party who is not honest.

3.5.3 Hybrid Techniques for PPDL Training

While some SMPC techniques are designed for three or more parties, their adaption to DL is difficult. With hybrid techniques, we observe protocols adapting better to DL training. These, however, are very varied and combine techniques in particular ways that enhance their use for privacy-preserving Machine Learning.

As an evolution to ABY and EzPC (previously used for inference), ABY3 [MR18] introduces layers for DL training and proposes a similar protocol allowing for 3-party computation. It considers both HBC and a malicious adversary and integrates the protocol in a NN compiler. In a similar fashion, SecureNN [WGC18] improved based on SecureML [MZ17] to introduce 3-party previous works allowing three independent servers to perform privacy-preserving training and prediction. The protocol considers m parties who want to share their data to train a model using three servers. Firstly, they use Secret Sharing from the m parties onto the n servers. Then, two servers use Additive Secret Sharing, which is collaboratively performed with the third server. The third server provides beaver triplets for multiplication and operates in a boolean sharing of some bits of the computation. They also propose modifications and optimizations for ReLU and max pooling, reducing up to 8 times the computation overhead. With a similar idea but generalizing to a notion of n parties in n servers, POSEIDON [Sav+20] proposes a framework for training DNNs based on Federated Learning and multi-key FHE. Concretely, multi-key FHE permits the generation of multiple private keys tied to a single public key. This way, each party performs encryption on its share, while decryption requires an agreement between all the parties. Multi-key FHE uses SMPC protocols for key generation, bootstrapping, and key-switching. POSEIDON modifies Federated Learning so that training is fully executed under HE in multiple servers, and there is a hierarchical gradient aggregation with HE.

3.6 Programming Interfaces for PPDL

While it is essential to improve the efficiency of PPDL techniques, Section 3.4 and Section 3.5 show various new protocols with particularities in their design and implementation. As a common factor, PPCTs protocols require expert knowledge and mathematical background for proper understanding and fine-tuning. Privacy and efficiency are two cornerstones for PPDL deployment, but also usability and applicability of the techniques are essential and desirable features. Authors have striven to ease the integration of PPCTs into new and existing frameworks. This thesis is framed in the techniques that aim to make PPCTs more affordable and straightforward for end-users.

The techniques proposed in the literature require expertise and mathematical background for appropriate deployment. Thus, This section analyzes proposals to bridge the gap between the theoretical proposals and their actual implementations and deployment. We cover two main types of contributions: works that propose compilers from high-level programming languages into privacy-preserving protocols (e.g., using libraries) and works that automatically transform outputs from common DL frameworks (e.g., Tensorflow, Keras, PyTorch) to PPDL protocols.

Armadillo [CDS15] was one of the first toolchains for compilation. It translated from C++ code into HE operations based on a Boolean FHE scheme. However, it had significant limitations in the capabilities since no abstraction to the boolean layer was provided.

Intel nGraph HE Tranformer [Boe+19a; Boe+19b; Boe+20] implements a compiler and runtime environment for inference with HE over Tensorflow. A backend process obtains the graph from Tensorflow, generates encrypted data, and executes the homomorphically encrypted inference. The first version [Boe+19b] relies on Intel nGraph [Cyp+18] and implements high-level operations with HE to perform PPDL inference. Additionally, nGraph HE implements optimizations for ciphertext packing that permit executing SIMD operations (i.e., to improve data parallelism) but does not benefit from vectorized linear algebra primitives (e.g., matrix multiplication or convolution). Thus, to benefit from SIMD operations, users would require to program these primitives manually. Furthermore, the activation functions only allowed for non-linear approximations with a low degree. Thus, the second version of the compiler [Boe+19a] was designed to avoid executing non-linear operations on ciphertexts by distributing the computations between the server and the clients. Concretely, for those non-linear functions, the information is sent back to the client, who decrypts it, executes the activation, and then sends back the results to the server. MP2ML [Boe+20] is a version of Intel nGraph that supports Yao's GC for the computation of the activation functions (i.e., in a similar way as GAZELLE [JVC18]).

PlaidML-HE and TinyGarble2 present similar approaches, transforming high-level representations into specific machine instructions relatable to cryptographic constructions. PlaidML-HE [Che+19a] proposes a new HE compiler for PlaidML (a library for speeding machine learning workloads using heterogeneous hardware backends). This way, it provides an intermediate abstraction layer agnostic to the machine learning framework that can be HE-interpreted. TinyGarble2 [Hus+20] presents an evolution of TinyGarble [Son+15] that permits having an efficient representation of GCs, enabling the execution of Neural Networks in shorter times. For that, they implement an interface to C++ and a DL library containing primitives to build CNNs for inference.

CrypTFlow is an end-to-end approach that permits translating Tensorflow code to different SMPC protocols [Kum+20a]. CrypTFlow has three components: i) Athos is an end-to-end compiler from TensorFlow to a variety of semi-honest SMPC protocols, ii) Porthos is an improved semi-honest 3-party protocol that is allegedly faster than State of the Art tools, especially for the convolution operation iii) Aramis is a tool that converts any semi-honest SMPC protocol into a SMPC protocol that provides malicious security based on the use of TEEs.

Despite all the previous works, CHET [Dat+19] is the first compiler that emerged as the first full-fledged tool to create homomorphically encrypted code from a high-level representation. CHET aims at C++ code executed on SEAL [Res20] and integrates Deep Learning capabilities. CHET introduces various improvements, such as efficient and automated parameter selection. EVA [Dat+20] improves CHET by introducing a Python interface and reduces the complexity of various operations at the expense of removing certain functionalities, such as complex vectorized operations. CHET [Dat+19] describes naive descriptions of new vectorized Convolution and Matrix Multiplication algorithms. These suppose the basis of some of the algorithms detailed in Chapter 4. Additionally, CHET [Dat+19] and EVA [Dat+20] provide a relatively automated parameter selection, but the behavior is simplistic and does not take into consideration certain classes of algorithms (i.e., parameter-dependent algorithms). The process chooses a default scale and replicates it as often as needed by the user-provided input-output precision.

In parallel to CHET, RAMPARTS proposes a compiler-oriented view to HE. RAMPARTS [Arc+19] relies on Palisade [PRR17] HE library and provides an environment to develop applications (in the Julia language), simplifying the use of HE directives based on existing public libraries. RAMPARTS combines Crucible, a tool for symbolic execution, and the PALISADE [PRR17] HE library. Crucible unrolls the code and transforms it into a simple arithmetic circuit by Crucible that the PALISADE backend then interprets. It also supports base vectorized operations and parameter selection supported by the PALISADE library. Ver, it lacks support for floating-point arithmetic and relies on general-purpose tools, which sometimes may not follow the efficient guidelines for Homomorphic Encryption.

Since CHET, there have been multiple works aimed at improving its performance and usability on different fronts, namely HELayers [Aha+23], HECO [Via+23], and Concrete [Chi+20b]. HELayers [Aha+23] is inspired by the findings of LoLa [BGE19] and introduces the concept of tiling (i.e., making plaintext-ciphertext layout mappings with different variations). While the mappings are standardized according to the LoLa findings and some further development, the main contribution is the automatic setup of those divisions of tiles. Their work supports variations of CNN algorithms, and they support automatic tiling on those. Efficiency-wise, the use of tiling supposes a significant improvement with respect to what is proposed by CHET and nGraph.

HECO [Via+23] is an automatic vectorization compiler. It aims to produce vectorized SIMD algorithms versions from the initial non-vectorized version. For that, they use multiple optimizations on the compiler. Unlike other tools, they produce LLVM MLIR [Lat+21], enabling other high-level programming language optimizations (e.g., constant folding). After that, they produce HE-specific optimizations, often carried out by other compilers. Next, they deal with ciphertext vectors for SIMD vectorization using analyzing insert and extract operations from vectors. The compiler assumes vectors and matrices are encoded flattened in ciphertexts. Then, after any insert and extract operation is performed on the plaintext, it stores an AST-like representation preserving the information on the source operands and the shape of the input operands. Thanks to this introspection, whenever an operation is performed on the *i*th element of a vector, the compiler can analyze whether the operation has previously been performed. Although HECO provides high-level programming language optimizations, using lower layers for HE-specific optimizations results in semantically abstract notations often insufficient for HE-like code.

Finally, Concrete [Chi+20b] is an evolution of the TFHE scheme [Chi+20a] turned into a framework that focuses on usability and efficiency. Concrete supports various improvements to HE, such as programmable bootstrapping [CJP21], which introduces the ability to perform certain operations as the ciphertext noise is reduced with bootstrapping. Furthermore, it introduces high-level libraries that empower inexperienced users, such as Concrete-Numpy. Since TFHE is a boolean scheme, the constraints apply differently than in floating-point schemes such as CKKS, thus the ability to perform such tasks.

Finally, the PySyft initiative aims to provide end-user tools for inference and training [Ryf+18]. Although some tools progressively develop secure protocols, their research value is still reduced. Some of the existing protocols combine PyTorch and SPDZ [Dam+12]. Tensors built can be shared among the parties executing the computation, making it seamless. Similarly, TFEncrypted [Dah+18] is an open-source library that applies changes to Tensorflow to make it usable with SecureNN and ABY3 as compiler and runtime. As of March'23, PySyft and TFEncrypted have evolved into infrastructural initiatives where secure deployments can be made.

3.7 Current Challenges and Research Directions

The analysis of the current state-of-the-art of PPCTs for DL provided by previous sections helps outline the current challenges that need to be addressed by the community. Table 3.1 and Table 3.2 summarizes the analysis of the proposals concerning the attributes presented in Section 3.1. We group works depending on whether they address DL training or inference (Table 3.1) or if they propose an API or tool to ease the use of the PPDL techniques (Table 3.2). We first provide an analysis of these proposals and then summarize some exciting remarks and takeaways from our study. For reference, the links to the open-source repositories, when available, are provided in Table 3.3.

3.7.1 Analysis

Most PPCTs present common challenges in their application. Most proposals are focused on theoretical contributions, whose translation into real-world implementations requires expertise and remains lengthy. Furthermore, most proposals involve intricate procedures for their parametrization and algorithmics.

HE evolution has brought multiple efficiency improvements, such as algorithm vectorization and SIMD. However, these present challenges in their application by introducing a gap between classical algorithm development and vectorized algorithm development. Furthermore, the already intricate parameter selection becomes more complex due to the interrelations of vectorization with the circuit parameters.

SMPC proposals are often easier to implement, as their complexity relies on communication delays. However, their security often relies on a third party, which requires extended SMPC protocols to remove the third party if it becomes unavailable. Furthermore, minimizing the number of communications and the overall computation is essential.

As we observe in the following subsections, existing problems in PPCTs often increase when these are applied to Deep Learning.

									Adv.		Arith.			SMPC						HE										
Privacy Technique	Name	Reference	Year	Number of Parties	Input Privacy	Output Privacy	Architectural Secrecy	Weights Secrecy	Honest-But-Curious	Malicious	Integer	Boolean	Floating Point	Oblivious Transfer	Yao's Garbled Circuits	Additive SS & Beaver	Shamir's SS	Zero-Knowledge Proofs	Commitment Schemes	Partially Homomorphic	Levelled Homomorphic	Fully-Homomorphic	TEE	Differential Privacy	Efficiency Improvement	Usability Improvement	Open Source Implementation	Paper-Code Matching	Code Maintenance	Tensorflow/Keras/PyTorch
DL Inference																														
HE	TASTY†	[Hen+10]	2010	2	✓	✗	✓	✓	⊖		●								●					▲	▲	✓	✓	✗	✓	
	Cryptonets	[Gil+16]	2014	2	✓	−	✓	✓	⊖		◇		◆								●				▲	▲	✓	✓	✗	✗
	CryptoDL	[HTG17]	2017	2	✓	−	✓	✓	⊖		◇		◆								●				▲	−	✗	?	?	?
	TAPAS	[San+18]	2018	2	✓	−	✓	✓	⊖			●										●			▲	▲	✓	✓	✗	✗
	Faster Cryptonets	[Cho+18]	2018	2	✓	✓	✓	✓	⊖		◇		◆									●		■	▲	−	✗	?	?	?
	FHE DiNN	[Bou+18]	2018	2	✓	−	✓	✓	⊖		◆	◇										●			▲	▲	✓	✓	✗	✗
	Jiang et al.	[Jia+18]	2018	2	✓	−	✓	✓	⊖		◇		◆									●			▲	▲	✓	✓	✗	✗
	LoLa	[BGE19]	2019	2	✓	−	✓	✓	⊖				●								●				▲	▲	✓	✓	✗	✗
Hybrid	Chameleon	[Ria+18]	2018	2	✓	−	✓	✓	⊖		●	●		●	●	●									▲	−	✗	?	?	?
	GAZELLE	[JVC18]	2018	2	✓	−	✓	✓	⊖		◇	●	◆		●					●					▲	▲	✓	✓	✗	✗
	Delphi	[Mis+20]	2020	2	✓	−	✓	✓	⊖		◇	●	◆		●						●				▲	▲	✓	✓	✗	✗
SMPC	Reza Sadeghi et al.	[SS08]	2008	2	✓	−	✓	✓	⊖			●		●											▲	−	✗	?	?	?
	MiniONN	[Liu+17]	2017	2	✓	−	✓	✓	⊖			●		●											▲	−	✗	?	?	?
	Deep Secure	[RRK18]	2018	2	✓	−	✓	✓	⊖			●		●	●										▲	−	✗	?	?	?
	XONN†	[Ria+19]	2019	2	✓	−	✗	✓	⊖			●		●	●										▲	▲	✗	?	?	✓
	Dalskov et al.	[DEK20]	2020	2	✓	−	✓	✓		●	◇		◆			●			●				■		▲	▲	✓	✓	✓	✗
DL Training																														
HE	Hesamifard et al.	[Hes+18]	2018	2	✓	−	✗	✓	⊖		◇		◆								●				▲	−	✗	?	?	?
	Nandakumar et al.	[Nan+19]	2019	2	✓	−	✗	✓	⊖		◇		◆								●				▲	−	✗	?	?	?
Hybrid	ABY3	[MR18]	2018	3	✓	✗	✓	✓		●	●	●	◆	●	●	●			●						▲	▲	✓	✗	✗	✗
	SecureNN	[WGC18]	2018	3	✓	✗	✓	✓	⊖		◇	●	◆	●	●	●									▲	▲	✓	✓	✗	✗
	Poseidon	[Sav+20]	2020	n	✓	✗	✗	✓	⊖		●		●			●						●			▲	−	✗	?	?	?
SMPC	Chase et al.	[Cha+17b]	2017	n	✓	✓	✗	✓	⊖		●					●								■	▲	−	✗	?	?	?
	SecureML	[MZ17]	2017	2	✓	✗	✓	✓	⊖		◇	●	◆	●	●	●					●				▲	▲	✓	✓	✗	✗
	QUOTIENT	[Agr+19]	2019	2	✓	−	✗	✓	⊖			●		●											▲	−	✗	?	?	?
	Coded Private ML	[So+19]	2019	n	✓	✗	✓	✓	⊖		●						●								▲	−	✗	?	?	?
	FLASH	[Bya+20]	2020	4	✓	✗	✓	✓		●		◇	◆			●		●	●						▲	−	✗	?	?	?

✓ Provisioned | ✗ Not Provisioned | − Feature Dependent Provision | ● Secure Use | ⊖ Partially Secure Use
◇ Emulation Arithmetic | ◆ Approximated Arithmetic | ▲ Improvement | − Not Improved
■ Not Covered Technique | † Repeated Entry

Table 3.1: Summary of the papers covered in the State of the Art Section with Characteristics for Privacy-Preserving Deep Learning.

PPDL Inference

Privacy-preserving inference is one of the main pillars of PPDL. HE-based approaches are more widespread than SMPC-based ones. This prevalence is mainly due to the matching of HE-based architectures with the model of MLaaS cloud providers. While HE-based approaches suppose an increased overhead due to the complexity of routines, the compu-

Privacy Technique	Name	Reference	Year	Number of Parties	Input Privacy	Output Privacy	Architectural Secrecy	Weights Secrecy	Adv.		Arith.			SMPC				Zero-Knowledge Proofs	Commitment Schemes	HE			TEE	Differential Privacy	Efficiency Improvement	Usability Improvement	Open Source Implementation	Paper-Code Matching	Code Maintenance	Tensorflow/Keras/PyTorch
									Honest-But-Curious	Malicious	Integer	Boolean	Floating Point	Oblivious Transfer	Yao's Garbled Circuits	Additive SS & Beaver	Shamir's SS			Partially Homomorphic	Levelled Homomorphic	Fully-Homomorphic								
APIs & Compilers																														
HE	TASTY[†]	[Hen+10]	2010	2	✓	−	✓	✓	◓		●									●					▲	▲	✓	✓	✗	✓
	Armadillo	[CDS15]	2015	2	✓	✗	✗	✓	◓		●		●								●				−	▲	✓	✓	✓	✓
	PySyft	[Ryf+18]	2018	n	✓	✗	✗	✓		●	●			●		●		●	●			●			−	▲	✓	✗	✓	✓
	TFEncrypted	[Dah+18]	2018	n	✓	✗	✗	✓		●	●			●		●		●	●			●			−	▲	✓	✗	✗	✓
	nGraph HE	[Boe+19b]	2018	2	✓	✗	✗	✓	◓		●		●								●				−	▲	✓	✓	✓	✓
	XONN[†]	[Ria+19]	2019	2	✓	−	✗	✓	◓			●		●	●										▲	▲	✗	?	?	✓
	CHET	[Dat+19]	2019	2	✓	−	✓	✓	◓		●		●								●				−	▲	✓	✓	✓	✗
	nGraph HE 2	[Boe+19a]	2019	2	✓	✗	✗	✗	◓		●		●								●				−	▲	✓	✓	✓	✓
	MP2ML	[Boe+20]	2019	2	✓	−	✓	✓	◓		●		●	●	●						●				−	▲	✓	✓	✓	✓
	PlaidML HE	[Che+19a]	2019	2	✓	−	✓	✓	◓		●		●									●			−	▲	✓	✓	✗	✓
	RAMPARTS	[Arc+19]	2019	2	✓	−	✓	✓	◓		●											●			−	▲	✗	?	?	✗
	CrypTFlow	[Kum+20a]	2020	3	✓	✗	✓	✓		●	●			●	●	●			●				■		−	▲	✓	✓	✓	✓
	TinyGarble2	[Hus+20]	2020	2	✓	−	✓	✓		●		●		●	●			●	●						−	▲	✓	✓	✗	✗
	EVA	[Dat+20]	2020	2	✓	−	✓	✓	◓				●								●				−	▲	✓	✓	✓	✗
	Concrete	[Chi+20b]	2021	2	✓	−	✓	✓	◓			◇	◆										●		▲	▲	✓	✓	✓	✗
	HELayers	[Aha+23]	2022	2	✓	−	✓	✓	◓				●								●				▲	▲	✓	?	?	✗
	HECO	[Via+23]	2022	2	✓	−	✓	✓	◓				●							●					▲	▲	✓	✓	✓	✗

✓ Provisioned | ✗ Not Provisioned | − Feature Dependent Provision | ● Secure Use | ◓ Partially Secure Use
◇ Emulation Arithmetic | ◆ Approximated Arithmetic | ▲ Improvement | − Not Improved
■ Not Covered Technique | † Repeated Entry

Table 3.2: Summary of the API and Compiler proposals covered in the State of the Art Section.

tation left on the clients is minimal. On SMPC-based infrastructure, more parties participate, often relying on trusted third parties to issue parameters (e.g., beaver triplets). Also, sometimes, these require some interaction with the client, which distances them from the current MLaaS model.

As a common flaw, HE and SMPC proposals require modification of the DL model to match the corresponding cryptographic protocols, affecting accuracy and hindering efficiency with existing frameworks. Thus, the current trend shifts from HE-dependent and SMPC-dependent architectures into Hybrid Architectures, where specific challenges are less impactful due to working in multi-arithmetic environments. As an example, GAZELLE [JVC18] combines efficient constructions for linear computations (e.g., efficient HE schemes) and boolean SMPC for non-linear functions.

Proposal	Ref.	URL
Dalskov et al.	[DEK20]	github.com/data61/MP-SPDZ
PySyft	[Ryf+18]	github.com/OpenMined/PySyft
CrypTFlow	[Kum+20a]	github.com/mpc-msri/EzPC
CHET/EVA	[Dat+19; Dat+20]	github.com/microsoft/EVA
ABY3	[MR18]	github.com/ladnir/aby3
PlaidML HE	[Che+19a]	github.com/plaidml/plaidml
TinyGarble2	[Hus+20]	github.com/IntelLabs/TinyGarble2.0
MP2ML	[Boe+20]	github.com/IntelAI/he-transformer
nGraph HE 2	[Boe+19a]	github.com/IntelAI/he-transformer
nGraph HE	[Boe+19b]	github.com/IntelAI/he-transformer
SecureNN	[WGC18]	github.com/snwagh/securenn-public
Dalskov et al.	[DEK20]	github.com/anderspkd/SecureQ8
TFEncrypted	[Dah+18]	github.com/tf-encrypted/tf-encrypted
Delphi	[Mis+20]	github.com/mc2-project/delphi
SecureML	[MZ17]	github.com/shreya-28/Secure-ML
Cryptonets	[Gil+16; Xie+14]	github.com/microsoft/CryptoNets
Jiang et al.	[Jia+18]	github.com/K-miran/HEMat
TAPAS	[San+18]	github.com/amartya18x/tapas
GAZELLE	[JVC18]	github.com/chiraag/gazelle_mpc
FHE DiNN	[Bou+18]	github.com/mminelli/dinn
TASTY	[Hen+10]	github.com/tastyproject/tasty
HELayers	[Aha+23]	github.com/IBM/helayers
HECO	[Via+23]	github.com/MarbleHE/HECO
Concrete	[Chi+20b]	github.com/zama-ai/concrete

Table 3.3: URLs for the open-source repositories of the different contributions analyzed.

PPDL Training

While works intended for PPDL inference are progressively improving performance and usability, HE-based or SMPC-based solutions for training are still immature. Like PPDL inference, the major bottleneck for PPDL training is its high computational overhead, preventing its application to complex, real-world use cases. Indeed, we observe that to lower the computation; the proposed test neural network architectures are often of reduced complexity (i.e., low number of layers and dimensions). Furthermore, PPDL training in distributed settings requires higher network communications. Comparing the performance over Wide Area Networks (WAN), where the network bandwidth is lower than Local Area Networks, shows improvable delays, as shown in SecureNN [WGC18]. Finally, most PPDL inference contributions provide specific implementation details of the different DL components (e.g., linear layers and polynomial approximations strategy). In training, we have observed that some proposals omit details such as loss function or non-linear derivative approximation.

Due to the inefficiency of using such techniques for training, most deployments integrate Federated Learning in cleartext for improved training efficiency with PPCTs for the secure exchange of the model weights, lining up with proposals such as POSEIDON [Sav+20].

Programming Interfaces and Compilers

An active area of research aligned which this thesis is the provision of *programming interfaces and compilers* that allow for smooth integration of existing frameworks and DL projects using PPCTs. For end-users to become familiar with these technologies, it is essential to enable tools in native languages. In the data science domain, interfaces adapted to DL frameworks such as Tensorflow [Aba+16b], Keras [Cho+15], or Pytorch [Pas+17] are fundamental. We highlight Intel nGraph HE, where proposals from various papers were integrated into an open-source tool [Boe+19a; Boe+19b; Boe+20]. Additionally, we highlight proposals that elaborate conversion routines (e.g., TASTY [Hen+10] or CrypTFlow [Kum+20a]) to adapt existing trained models to PPDL protocols quickly.

While most tools already present improvements to perform such techniques, recent proposals have explored various fronts (e.g., the reduction of algorithmic complexity [Aha+23; BGE19]). While reducing and standardizing ciphertext formats is essential, the seamless adaptation of algorithms for HE is also a critical research direction [Via+23].

Other Challenges and Considerations of PPDL

Most PPCTs rely on a single arithmetic type. Classic computation has typically been composed of multiple datatypes in mind and efficient ways to combine them. DL in particular needs boolean and floating-point arithmetic types for linear and non-linear activations. In this line, hybrid approaches have shown a strong impact, and solutions such as CHIMERA [Bou+20] open new paths for optimization. While authors have used integer and boolean arithmetic for discretized and Binary Neural Networks, these are rare and difficult to adapt to modern DL solutions for complex problems. Accordingly, CKKS [Che+17], which natively supports floating-point arithmetic, has been used in various proposals that propose different approximations from integer arithmetic. Indeed, this is the main scheme adopted in the contributions of this thesis. We refer the reader to Section 2.3.4 of the previous chapter for a description of CKKS.

As explained in Section 2.2, we consider that a proposal provides input privacy if the data processing reveals no user information. All the covered works provide input privacy (indeed, this is a key feature of PPCTs). Output privacy guarantees that the result of a DL algorithm does not reveal additional information from the data. It protects DL models against attacks such as membership inference or model inversion. While this belongs to a broader and active area of research (i.e., adversarial machine learning [BR18]), only two works propose mechanisms to incorporate output privacy guarantees, i.e., Faster Cryptonets [Cho+18] and the work by Chase et al. [Cha+17b]. Most works on privacy-preserving inference assume that DL models are securely pre-trained (i.e., they do not contain or reveal private information). Additionally, they consider the cloud provider as the model owner or a trusted third party for the training (i.e., it establishes security measures and does not attack the model). Accordingly, while the proposals might not address output privacy, providing such a guarantee would involve combining it with other techniques, such as Differential Privacy. Regarding model secrecy, most of the works address both weight secrecy and architectural secrecy. However, the latter is more difficult to guarantee, given the patterns in common DL layers and activations. Furthermore, these do not prevent other attacks of retraining other models [Tao+23].

Most of the literature usually assumes an HBC adversary who is limited in offensive capabilities. Considering this kind of adversary lowers the performance requirements. Multiparty computation protocols can be transformed into the malicious adversary setting through commitment schemes, ZKP, and distributed randomness protocols [BGW88; Cho+85; Fel87; GMW91; GRR98]. However, it is unclear whether cryptographic bridges such as the ones implemented in GAZELLE [JVC18] or ABY [DSZ15] can be easily attested and what their performance would be on the malicious setting. Indeed, some contributions tackling the malicious adversary setting rely on trusted hardware (i.e., TEE) for code verification and confidentiality [DEK20]. Exploring the translation of hybrid protocols to malicious adversary models and efficient ZKP for distributed protocols is paramount for the real-world deployment of these solutions.

3.7.2 Takeaways

We enumerate key takeaways from the study of the current state of the art, highlighting lessons learned and points that deserve further research effort. We also highlight the contributions of this thesis that address these points:

1. While classical cryptography is relatively easy to use (e.g., due to easy-to-use and curated libraries), modern cryptography, such as HE, requires more tuning and expertise. It calls for schemes (e.g., CKKS [Che+17]) and solutions (e.g., ABY [DSZ15]) that allow for automatic analysis and adaptation to specific circuits and the election of optimal parameters [VJH21]. For efficient use of PPDL, these solutions should implement hybrid techniques to offer protocol conversion mechanisms, improving the overall performance of multi-arithmetic systems. **In**

Chapter 4 we describe algorithms that automatically adapt the internal linear algebra operation used in common DL layer so they can be adapted to HE.

2. Parametrization of PPCTs remains a complex problem that requires particular knowledge of each technique. Lowering the entry barrier supposes introducing automatic tools for parametrization that permit users to develop code with the specific PPCTs without digging down into the particularities of each technique. Furthermore, using each PPCTs effectively is an ongoing effort. Providing tools and guidelines for using each technique remains a desirable goal. **In Chapter 5, we provide a solution for automatic parameter selection in CKKS, relying on Linear Programming and Fuzzy Logic to provide an optimal solution in line with the user needs in terms of efficiency, security, and precision.**

3. Even if proposals release their source code openly for reproducibility, comparing efficiency and accuracy among contributions is complex due to different benchmarking across articles. Different works claim different runtimes and accuracy across DL models and computing architectures. Additionally, due to ad-hoc optimizations applied to each proposal, we cannot easily measure the overall impact of each optimization on the protocol. It is thus necessary to provide standardized trained models and benchmarks for an accurate and fair comparison of the proposals. Again, this would need to ease the integration of new works with said standard benchmarks. Contributions like Gouert et al. [GMT23] show the importance of providing open benchmarking of solutions to understand each solution's best settings and applications.) It also t shows that the different low-level libraries for HE has pros and cons under different settings. Thus that is desirable to create high-level compilers that are not tied to a specific library. **In Chapter 5 we describe the design of a software compiler that abstracts the high-level requirements for the vectorization, circuit adaptation, and automatic parameter selection from the lower-end HE library, thus making it inter-operable with different HE frameworks and libraries.**

4. To lower the entry barrier for data scientists, it is essential to provide privacy-preserving extensions for existing DL frameworks, e.g., Tensorflow or PyTorch. This adaptation requires scientists to i) select a privacy-preserving solution or protocol, ii) implement adaptions to the protocol, and iii) provide an interface or bridge with the used DL framework. That is, implementing a complete software stack. It is a challenging yet desirable goal to build standard solutions to adapt existing PPDL solutions by minimally changing the original code. **The compiler presented in Chapter 6 provides a high-level and easy-to-use API for data scientists that allows them to seamlessly and easily adapt existing projects in Python for their use with low-level HE frameworks.**

The following takeaways are raised, but these are not explicitly addressed by this thesis:

1. Efficiency and usability are confronting goals. Most works focused on efficiency improvements modify the original protocols to a great extent, which is detrimental to their integration into other frameworks. Meanwhile, works focusing on usability (i.e., APIs and compilers) involve few adjustments from basic techniques and original protocols but have significantly worse runtime performance than those focused on efficiency. Additionally, as analyzed before, current APIs and compilers are unsuitable for PPDL training, an area deserving more attention.

2. Except for classic Yao's Garbled Circuits, most PPCTs reveal the performed operations. As pointed out in Section 2.2, architectural secrecy ensures the non-revelation of the architecture to untrusted parties. In contexts where the cloud server is not a trusted third party, the common patterns in neural network operations allow inferring the neural network architecture to the party performing the processing. Therefore, to better protect the privacy and security of the DL models, works must hide data and processing (e.g., using whenever available Indistinguishable Obfuscation [JLS20]).

3. In general, protocols assume participating entities in these protocols are honest but curious. However, previous experience shows that honest-but-curious settings are limited, e.g., due to internal security breaches or insiders. We note a lack of proposals considering a malicious setting, limiting the deployment of PPCTs to scenarios where honest-but-curious adversaries cannot be guaranteed.

4. Similar to other Privacy-Enhancing Technologies, there is the potential misuse of PPCTs by malicious adversaries. Indeed, since PPCTs ensure the input privacy of users, this protection can conceal malicious behavior. For example, a user trying to extract model parameters will carefully craft input data to create inference samples that yield knowledge about the model. Detecting these adversarial samples might require monitoring the inputs. In such cases, the detection mechanisms become unusable due to the use of PPCTs. Thus, exploring how to integrate this technology in adversarial settings would be desirable.

In the following chapters, we address some of the issues in this analysis, precisely concerns regarding the usability and simplicity of Homomorphic Encryption. Concretely, we propose new and simplified vectorized algorithms for Deep Learning with HE, a simplified parametrization based on simple parameters, and a symbolic compiler unifying all these concepts.

Chapter 4

Optimization of Deep Learning Linear Algebra Algorithms for Packed Homomorphic Encryption

Chapter 3 outlined the main lines and problems in the different Privacy-Preserving Computation Techniques. One of the challenges regarded the usability and complexity of Homomorphic Encryption. This chapter addresses the creation of vectorized algorithms with Packed Homomorphic Encryption for Deep Learning.

The rest of this chapter is organized as follows. First, we introduce the problem and the motivation in Section 4.1. Second, we provide background around Packed Homomorphic Encryption and outline the adversarial model in Section 4.2. Then, we describe the different algorithms in Section 4.3. Next, we conduct a formal analysis of the performance of the algorithms in Section 4.4. Consecutively, we empirically evaluate a working prototype in well-defined tests in Section 4.5. Finally, in Section 4.6, we summarize the contributions of this chapter.

4.1 Introduction

Homomorphic Encryption (HE) schemes enable performing operations over the encrypted *ciphertext* without decrypting. After the key milestone of Gentry [Gen09], HE has undergone significant development to improve its efficiency [DM15]. One such fundamental improvement is *Ciphertext Packing*, which allows the encoding of various entries of plaintext data (encoded as a vector) within a single ciphertext [BGH13]. Packing improves efficiency through Packed Homomorphic Encryption (PaHE) since the individual operations to the ciphertext affect all the individual entries of the underlying plaintext

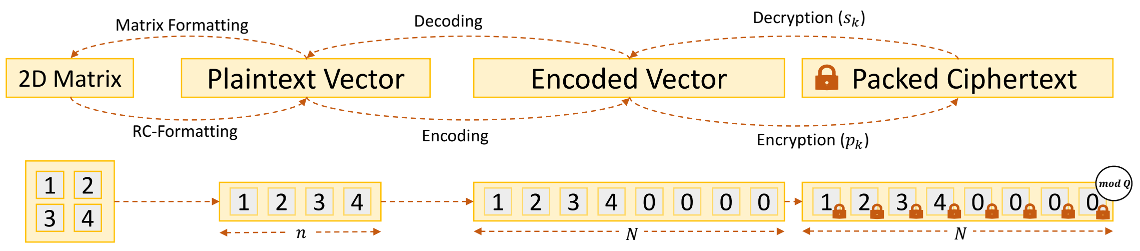


Figure 4.1: Encryption and layout mapping procedure for Homomorphic Encryption Schemes based on Learning with Errors (LWE).

vector. Figure 4.1 shows how HE packing works for the logical encoding of a 2D Matrix. First, we transform the matrix into a vector using a Row-Column format. Then, we logically encode the vector in a plaintext polynomial (packed) and encrypt it. Next, we can operate the ciphertext using Single Instruction Multiple Data (SIMD), where we modify all the encrypted vector (matrix) elements with each instruction.

Unfortunately, applying ciphertext packing in Deep Learning (DL) is not straightforward. Indeed, the linear algebra operations used in the internal structures highly affect the performance [Dat+19]. Understanding the impact of the ordering and election of internal operations on global performance is critical for designing efficient algorithms. However, this complex task requires a proper understanding of cryptographic protocols.

4.1.1 Precedents

Two main works have addressed the operation of PaHE with DL or linear algebra operations. Halevi and Shoup [HS14a] were the first to propose HE SIMD algorithms in their adaption to HELib [HS14b]. In their work, they provide details on various algorithms, specifically those that allow using HE Packed vectors as usual operations. They cover multiple algorithms, from individual entry selection to replication and matrix multiplication. In contrast to the work covered in this chapter, Halevi and Shoup [HS14a] provide more general algorithms that consider the minimization of HE parameters but do not strictly relate them to Deep Learning. Additionally, in our paper, we specifically cover the execution of algorithms for matrix multiplication on arbitrary matrix dimensions, while they only cover square matrices.

CHET [Dat+19] is a compiler for DL inference that considers an analysis of the HEcircuits to generate efficient HE code. In their work, the authors propose algorithms for convolution and matrix multiplication. However, CHET algorithms suffer from limited applicability due to lacking any initial representation or result transformation guidelines.

While other works have briefly described techniques to perform linear algebra operations on packed ciphertexts, how to fully take advantage of it for DL is still to be demonstrated.

4.1.2 Motivation

Over the last few years, there has been significant progress on the cryptographic protocols and theories related to ciphertext packing, which promises substantial improvements in the application of HE in complex, distributed applications such as DL for healthcare remote analysis and monitoring. However, adapting existing operations for SIMD over ciphertext packing is non-trivial.

Existing works have attempted to automatically transform linear algebra algorithms so they can be applied using SIMD over packed ciphertexts [Dat+19; HS14a]. However, these works provide simplistic views of the required algorithms, which limit their reproducibility and implementation in real-world architectures.

Moreover, the algorithms are described and tested for isolated computations, far from the complex workflow and interconnections from the internal layers of Deep Learning. Since the input and output encoding of ciphertext packing differs from regular operations, it is necessary to account for the different formats of the inputs and outputs in each layer of the DL architecture and how these affect the overall performance.

For example, when using SIMD operations, it is required to transform the output of a convolutional layer to be a valid input for subsequent dense layers. These transformations have not been described or accounted for the overall overhead in previous works. Also, this transformation requires different adaptions depending on specific details of the DL layer, e.g., stride or padding. Furthermore, the last entries overflow to the first positions in rotations with packed ciphertexts. Our work shows how some algorithms profit from this overflow to perform computation. Overall, existing proposals either leave the adaptation to HE on the user [HS14a] or provide algorithms that are not general for inputs of any size [Dat+19].

4.1.3 Contribution

This chapter provides a comprehensive solution to the research challenge of efficiently adapting the operations of Convolutional Neural Network (CNN) for their use on packed vectors. A key aspect in the design of the algorithms is that they work on arbitrary-sized input matrices and vectors, and thus they can adapt to images of any size. We provide a holistic view of the DL inference process by not only understanding the individual linear operations required by the algorithms but also the collective relationship of the different layers. During the design process, it is imperative to note that the connections required to link the layers in a DL architecture can be a substantial burden. This is a recent finding that cannot be overlooked. Accordingly, we conduct a thorough analysis of the implication of the algorithms and provide recommendations according to these insights for the integration in DL projects.

4.2 Background

This section provides a basis for the notation and operations used in the remainder of this chapter. We first describe the adversarial model and the privacy requirements assumed in Subsection 4.2.1. Then, we cover the more essential concepts regarding Packed Homomorphic Encryption in Subsection 4.2.2.

4.2.1 Adversarial Model

This work considers an Honest-but-Curious adversary [GMW87; GV88; Lin20], a passive adversary that complies with the protocol and does not tamper with the data for malicious purposes. However, it tries to learn as much information from data exchanges. HE inherently guarantees input privacy (i.e., the confidentiality of the data sent to the cloud during the inference process is guaranteed). However, HE does not account for the output privacy of the model. Thus, in our work, we assume that the service provider either is proprietary or has access to the model.[1].

Furthermore, for all of our use cases, we consider the set of parameters established for the Learning with Errors problem following the guidelines described in the Homomorphic Encryption Security Standard [Alb+18]. Thus, these parameters provide a secure environment for the execution of the algorithms.

4.2.2 Packed Homomorphic Encryption

Homomorphic Encryption (HE) is a property of an encryption scheme that permits operating with the ciphertext while translating those changes to the underlying plaintext. This work focuses on widely used HE schemes based on variations of the Ring Learning with Errors (RLWE) problem [Reg09].

Concretely, we focus on Packed Homomorphic Encryption (PaHE) schemes [BGH13], which enable the introduction of more than one plaintext element per ciphertext [BGH13]. In RLWE-based schemes, we parametrize them based on a tuple (N, Q, σ). We represent the ring polynomial coefficient modulus Q with a Chinese Remainder Theorem (CRT) coefficient moduli chain q_i, allowing rescaling. The polynomial degree N defines the maximum number of plaintext elements n that a ciphertext can accommodate, i.e., the maximum length of a vector that can be encoded (see Figure 4.1). Schemes such as BFV [FV12] allow packing as many elements as the length of the polynomial (i.e., $n \leq N$). In the scheme CKKS [Che+17], though, due to the complex number packing, it is only possible to pack half of the vector size (i.e.,

[1]Output privacy, e.g., preventing model inversion or membership inference attacks, requires mechanisms during training, such as Differential Privacy, which provides privacy protection against adversarial attacks. These are out of this thesis's scope, focusing on inference and assuming a pre-existing trained model.

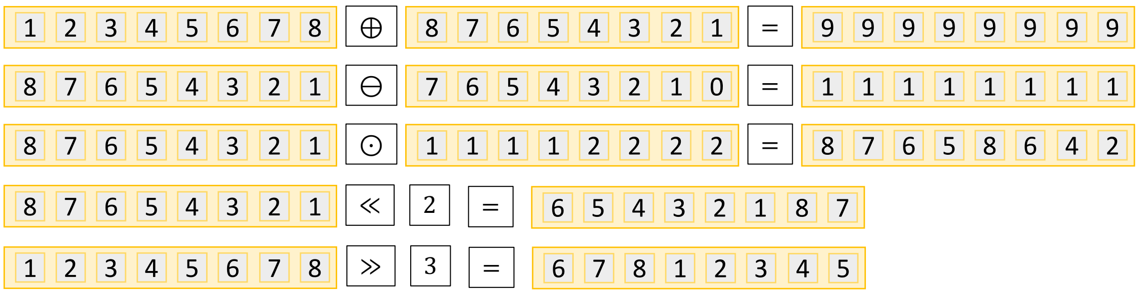


Figure 4.2: SIMD Operations allowed by CKKS Packed Homomorphic Encryption Scheme.

$n \leq N/2$). This thesis treats parameter N independently of the scheme (i.e., $n = N$). However, all the conclusions are valid to CKKS by substituting by $n = N/2$. Finally, σ represents the error distribution often fixed for HE schemes.

The routines of an HE scheme are the following:

- A **key generation** routine produces a public key p_k and its corresponding private key s_k, defined by $Keygen(N, Q, \sigma) \rightarrow p_k, s_k$, where N and Q are the homomorphic encryption parameters.

- The **encryption** routine takes a public key p_k and a plaintext vector v to generate a ciphertext vector $c_t \in \mathbb{Z}_Q[x]/(x^N+1)$ in a ring $\mathbb{Z}_Q[x]/(x^N+1)$ such that $Enc(v, p_k) \rightarrow c_t$. The inverse **decryption** routine takes a ciphertext c_t and uses the private key s_k to obtain the plaintext vector such that $Dec(c_t, s_k) \rightarrow v$.

- The different **evaluation** routines compute over the ciphertext c_t one of the following operations: element-wise sum ($\oplus$), element-wise subtraction ($\ominus$), Element-wise multiplication ($\odot$), and left/right cyclic rotation ($\ll$ and $\gg$ respectively). Note we represent operations differently to highlight the element-wise nature of those. We depict the SIMD behavior of operations in Figure 4.2.

For simplicity, in the algorithms, we assume that a vector's encryption routine also comprises the previous encoding. Similarly, the decryption routine includes the decoding after decryption. We refer the reader to Section 2.3 for more details on packing, encoding, and encryption.

4.3 SIMD Algorithms for Deep Learning

As detailed in Chapter 2, most DL building blocks rely on standard linear algebra operations (e.g., matrix or vector multiplication). Some of these operations are not available in the encrypted domain (e.g., accessing an arbitrary entry of an array). Furthermore, existing optimizations for running classical linear algebra on computers, such as tiling memory accesses in matrix multiplications [VS02], are not possible when the smallest unit considered is a packed HE ciphertext (i.e., a polynomial in $\mathbb{Z}_Q[x]/(x^N + 1)$). Thus, it is necessary to develop focused algorithmic optimizations for these ciphertexts.

In this section, we provide general algorithms to adapt linear algebra operations so they can exploit the potential of SIMD operations in DL while working on HE ciphertexts. Concretely, we propose descriptions of algorithms that operate on arbitrary matrices $\mathcal{M} \in \mathbb{R}^{h \times w}$ (i.e., of height h and width w).

Additionally, since DL architectures consist of connected layers of different natures, the representations between these layers need to be compatible. This compatibility means that the output of SIMD operations resulting from a given layer needs to be usable for the input of the following layer. While previous works have proposed SIMD functions for these layers, they have only provided partial examples, not giving a holistic view of the DL pipeline and not considering the interconnections of the layers [Dat+19; HS14a]. Indeed, as we show in Section 4.5, the data transformations have a considerable overhead which the DL design phase should account for.

The ciphertext encodes vectors of size N when dealing with PaHE. However, CNNs operate over matrices which require transforming matrices into vectors (see Figure 4.1). We consider a standard representation that we refer to as Row-Column (RC) format, where the 2D matrix is flattened row by row (Equation 4.1). This format allows the representation of information with the smallest N. Accordingly, in this work, we provide algorithms to transform the RC format to the appropriate Initial Representation for the algorithm and Result Transformation algorithms for returning to the RC format.

$$RC(\mathcal{M}) = \{\mathcal{M}_{0,0}, \mathcal{M}_{0,1}, ..., \mathcal{M}_{0,w-1}, \mathcal{M}_{1,0}, ..., \mathcal{M}_{h-1,w-1}\} \tag{4.1}$$

In a nutshell, the processing of each layer requires the following algorithms (executed before, during, and after the actual data processing):

1. **Initial Representation (IR)** algorithms provide the corresponding layer with an appropriate representation of the data for executing the algorithm (i.e., according to the requirements of the layer).

2. **Algorithm Execution (ALG)** algorithms are the actual execution of the internal operations over the data. We next describe the convolution blocks (i.e., convolutional layer, pooling layer, and activation function) and dense blocks (i.e., dense layer and activation function).

3. **Result Transformation (RT)**. Due to the nature of SIMD operations, the algorithms usually introduce some extra, irrelevant data in the result (e.g., redundant or padded). Also, different SIMD operations produce different output formats. Thus, we elaborate dedicated algorithms to extract the relevant output information and

turn it back into a format suitable for the next layer. We note that the RT process can often be combined with the IR of the following layer.

4.3.1 Notation

We use $\mathcal{M} \in \mathbb{R}^{h \times w}$ to represent a matrix $\mathcal{M}$ of dimensions $h \times w$. Also, we use $\mathcal{M}_{i,j}$ to refer to the entry on row i and column j of the matrix.

Many of the algorithms rely on binary bitmasks to obtain relevant information. These are composed of binary values. In the description of the algorithms, we assume bitmasks are initially filled with 0, and we express a condition to get the positions (indexes) where entries are set to 1. We use the parameter t ($bitmask[t] \in \mathbb{Z}N_2$), which defines such indexes. For example, for a bitmask where even indexes are 1, we denote the mask as $bitmask[t] \leftarrow \{t \mod 2 = 0\}$.

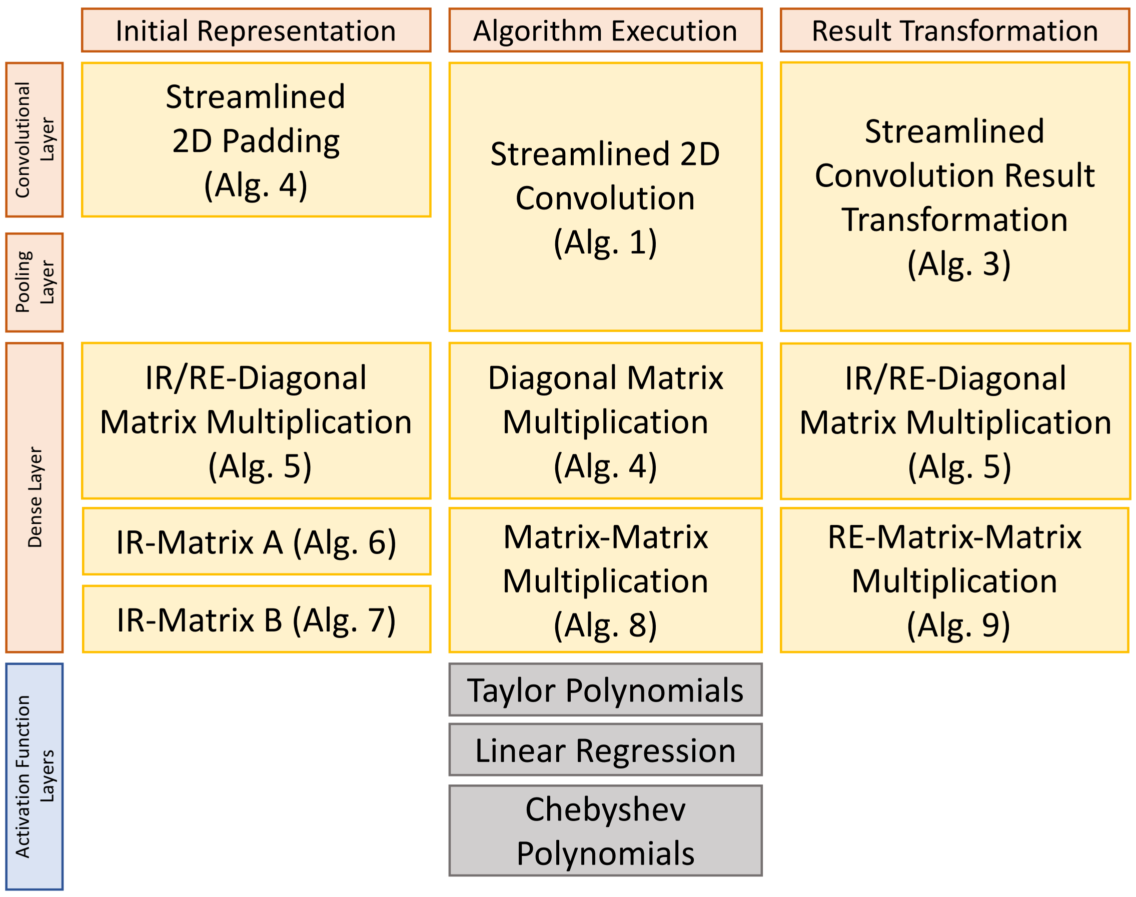
In HE, the ciphertext representation uses integer ring polynomial representation, unlike DL floating point representation. This problem has multiple approaches, such as fixed point representations or CKKS encoding [Che+17]. The algorithms provided here are represented generically without discussing the specific representation (i.e., the only requirement is the availability of the operations defined in Section 4.2.2).

In the following sections, we provide the general algorithms that allow the adaptation of common layers in CNNs, i.e., convolutional and dense blocks. Then, we provide a discussion on how to use activation functions (which are non-linear) in the context of PaHE Figure 4.3 provides a summary overview of the application of the different algorithms in a CNN pipeline.

4.3.2 SIMD Convolutional Layer

Computing a classical 2D convolutional layer involves a relationship between an input bi-dimensional matrix region and a bi-dimensional filter. The convolution can combine structures such as padding, stride, or pooling layers. Accordingly, the algorithms defined for PaHE should also account for using these variants.

This section presents a new algorithm for convolution, dubbed the Streamlined Convolution Algorithm. It allows combining convolutional, pooling, and activation layers in subsequent blocks, neglecting the cost of initial representation and executing a single result transformation algorithm (i.e., we can use the output representation of the algorithm arbitrarily). We first describe the convolution algorithm with stride and the result transformation algorithm. Then, we provide its integration with padding and average pooling layers. For reference, Appendix A details the convolution algorithms that extend and generalize previous work [Dat+19], which we use as a baseline comparison in Section 4.5.



Initial Representation
Algorithm Execution
Result Transformation
Convolutional Layer
Pooling Layer
Dense Layer
Activation Function Layers
Streamlined 2D Padding (Alg. 4)
Streamlined 2D Convolution (Alg. 1)
Streamlined Convolution Result Transformation (Alg. 3)
IR/RE-Diagonal Matrix Multiplication (Alg. 5)
Diagonal Matrix Multiplication (Alg. 4)
IR/RE-Diagonal Matrix Multiplication (Alg. 5)
IR-Matrix A (Alg. 6)
IR-Matrix B (Alg. 7)
Matrix-Matrix Multiplication (Alg. 8)
RE-Matrix-Matrix Multiplication (Alg. 9)
Taylor Polynomials
Linear Regression
Chebyshev Polynomials

Figure 4.3: Overview of the different algorithms needed to perform a Convolutional Neural Network with Homomorphic Encryption.

Streamlined Convolution Algorithm

The convolution algorithm takes as input a plaintext filter $\mathcal{F} \in \mathbb{R}^{f_x \times f_y}$ of dimensions $f_x \times f_y$. The filter is applied to a ciphertext vector $c_t \in \mathbb{Z}_Q[x]/(x^N + 1)$ that corresponds to an encrypted input matrix, i.e., $c_t = Enc(\mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$, with p_k being the encryption key and $\mathcal{M}$ the plaintext input matrix linearized. The algorithm leverages that the dimensions of filters are shorter than input matrices, and we have plaintext access to those. Thus, it computes the convolution between each filter pixel and the input matrix (i.e., represented by a ciphertext) and adds the partial results for each pixel. The algorithm is described in Algorithm 1.

We denote the linearized format of the matrix as Streamlined Backward Convolution Format (SCBF) since it depends on the information of previous layers. For multiple consecutive convolutions, the algorithm requires some information to determine the layout of the input vector. Concretely, in the execution of layer l, the algorithm receives as input the product of strides from previous layers, i.e., $\mathcal{S}_x = \prod_{i=0}^{l-1} s_x^i$ and $\mathcal{S}_y = \prod_{i=0}^{l-1} s_y^i$, where s_x^i, s_y^i are the strides on x and y axis of the i-th consecutive convolutional layer. Furthermore, the algorithm also requires the dimensions of the first encoded matrix, i.e., (h_0, w_0). These define the capacity of the algorithm to perform operations on the information. Whenever we execute a convolution, the result format depends on these values.

For the first layer, we use the Row-Column format, a subset of the SCBF format where $\mathcal{S}_x = 1$ and $h, w = h_0, w_0$.

Finally, if the convolution uses padding, the algorithm relies on the `PADDING` function, which we describe in the next section.

Algorithm 1 Streamlined 2D Convolution

Input: $c_t^{SCBF} \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(\mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$, $\mathcal{F} \in \mathbb{R}^{f_x \times f_y}$, (s_x^l, s_y^l), p, $(\mathcal{S}_x, \mathcal{S}_y)$, h_0, w_0

Output: $conv \in \mathbb{Z}_Q[x]/(x^N + 1)$ in SCBF format

```
function CONVOLUTION($c_t$, $\mathcal{F}$, $s_x$, $s_y$, $p$, $\mathcal{S}_x$, $\mathcal{S}_y$, $h_0$, $w_0$)
    $h_{out} \leftarrow \lfloor \frac{h - f_x + 2 \cdot p}{s_x^l} \rfloor + 1$
    $w_{out} \leftarrow \lfloor \frac{w - f_y + 2 \cdot p}{s_y^l} \rfloor + 1$
    $c_t^{pad} \leftarrow$ PADDING($c_t^{SCBF}, p, (\mathcal{S}_x, \mathcal{S}_y), (h_0, w_0)$)
    for $i \leftarrow 0, f_x$ do
        for $j \leftarrow 0, f_y$ do
            $rot \leftarrow c_t^{pad} \ll (i \cdot w_0 \cdot \mathcal{S}_x + j \cdot \mathcal{S}_y)$
            $conv = conv \oplus rot \odot \mathcal{F}_{i,j}$
        end for
    end for
    return $conv$
end function
```

Streamlined Padding Algorithm

The insertion of padding is common in convolutional layers to ensure the preservation of details in the corners of matrices when using filters. In general, if the padding is added on the first layer of the CNN, we can apply it on the cleartext matrix (i.e., applied by the client) and encrypt afterward. However, if the padding is not present on the initial layer, it is the responsibility of the server to execute it. For that, we propose Algorithm 2, where based on the structure of the SCBF format, it performs padding with little computational effort, as opposed to the base algorithm described in Appendix A. The algorithm has two main tasks. First, it rotates the relevant information according to the padding needed for the first row. Then, using a bitmask, it introduces zeroes in the appropriate positions. Note that, for notation clarity, we perform an inversion of the bitmask (i.e., zeroes become ones and vice versa) denoted as $1 - bitmask[t]$.

Streamlined Average Pooling Layers

Similar to the usage of stride during the convolution operation, pooling layers allow for reducing the overall complexity required to process the information. Additionally, they

Algorithm 2 Streamlined 2D Padding

```
Input: $c_t^{SCBF} \in \mathbb{Z}_Q[x]/(x^N+1) = Enc(\mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$, $p$, $(\mathcal{S}_x, \mathcal{S}_y)$, $h_0, w_0$
Output: $c_t^{pad} \in \mathbb{Z}_Q[x]/(x^N+1) = Enc(\mathcal{M}' \in \mathbb{R}^{h_{pad} \times w_{pad}}, p_k)$, in SCBF format
  function PADDING($c_t$, $p$, $\mathcal{S}_x$, $\mathcal{S}_y$, $h_0$, $w_0$ )
      $h_{pad} \leftarrow h + 2 \cdot p$
      $w_{pad} \leftarrow w + 2 \cdot p$
      $c_t^{pad} \leftarrow c_t^{SCBF} \gg (w_0 + 1) \cdot \mathcal{S}_x \cdot p$
      $bitmask[t] \leftarrow \{t = i \cdot w_0 \cdot \mathcal{S}_x + j \cdot \mathcal{S}_y \mid i = \lfloor t/w_0 \rfloor, j = t \mod w_0 \mid i < p \lor i \geq (h_{pad} - p) \lor j < p \lor p \geq (w_{pad} - p)\}$
      $c_t^{pad} \leftarrow c_t^{pad} \odot (1 - bitmask[t])$          ▷ Inverted bitmask
      return $c_t^{pad}$
  end function
```

highlight the most relevant features for the classification, extracting higher informative areas and discarding less informative ones. While max pooling is a popular choice for DL since it allows the extraction of pronounced and sharp changes [Goo+16] (e.g., edges in pictures for image segmentation), the *max* function is non-linear. Thus, its usage with current Homomorphic Encryption schemes remains complex and inefficient. In our work, we cover linear average pooling, which results in less inefficiency [2] , but extracts smoother changes from pictures [Goo+16].

The encrypted version takes an input ciphertext vector $c_t \in \mathbb{Z}_Q[x]/(x^N+1) = \mathcal{M} \in \mathbb{R}^{h \times w}$, and a pool $\mathcal{P} \in \mathbb{R}^{p_x \times p_y}$. The adaptation to SIMD HE can be obtained by using the convolution algorithm presented in §4.3.2, but using a dedicated filter for the pooling, defined as:

$$\mathcal{P} = \{\mathcal{P}_{i,j} = \frac{1}{n} \mid 0 \leq i < p_x, 0 \leq j < p_y, n = p_x \cdot p_y\} \tag{4.2}$$

Similar to the convolution algorithm, the pooling layer also requires meta-information from previous layers.

Streamlined Convolution Result Transformation

The previously presented algorithms can be arbitrarily combined between themselves and activation functions, incurring a minimal multiplication depth and the number of multiplications per layer. The only drawback is its combination with other types of layers (e.g., the dense layer). For that, we provide a function allowing the user to return to RC format to become compatible with other layers. The process depicted in Algorithm 3 may not be the most efficient depending on the subsequent layer. If that is the case, the lines detailed as formatting in blue and with the formatting comment may swap for a more appropriate layout. This algorithm receives a ciphertext vector $c_t \in \mathbb{Z}_Q[x]/(x^N+1) = \mathcal{M} \in \mathbb{R}^{h_{out} \times w_{out}}$,

[2]We note that so-called hybrid approaches, such as GAZELLE [JVC18], are potential options for evaluating boolean non-linearities. However, they rely on other privacy-preserving computation techniques.

being the encrypted result of the convolutions of which we stored the output dimensions $h_{out} \times w_{out}$. Additionally, as in the previous examples, it is necessary to keep track of the product of strides in previous layers $(\mathcal{S}_x, \mathcal{S}_y)$ and the initial dimensions (h_0, w_0).

Algorithm 3 Streamlined Convolution Result Transformation

```
Input: $c_t^{SCBF} \in \mathbb{Z}_Q[x]/(x^N + 1)$ in format, $\mathbb{R}^{h_{out} \times w_{out}}$, $h_0 \times w_0$, $(\mathcal{S}_x, \mathcal{S}_y)$
Output: $c_t^{RC}$ in RC format
  function RT-SCR-RC($c_t, w_0, \mathcal{S}_x, \mathcal{S}_y$)
      for $i \leftarrow 0, h_{out}$ do
          for $j \leftarrow 0, w_{out}$ do
              $bitmask[t]_i \leftarrow \{t = j \cdot \mathcal{S}_y + (i \cdot w_0 \cdot \mathcal{S}_x)\}$
              $shift_i \leftarrow t - (i * h_{out} + j)$                    ▷ Formatting
              $c_t^{RC} = c_t^{RC} \oplus (c_t^{SCBF} \odot bitmask[t]_i) \ll shift_i$
          end for
      end for
      return $c_t^{RC}$
  end function
```

4.3.3 SIMD Dense Layer

The Dense or Fully-Connected Layer of a Neural Network performs a weighted connection of all the inputs to all outputs. It is a linear transformation that a matrix-vector multiplication can represent, where a weight matrix $\mathcal{W} \in \mathbb{R}^{h \times w}$ is multiplied by the vector $x \in \mathbb{R}^n$. The weights matrix contains parameters fine-tuned during the training. The input vector comprises all the outputs from the n neurons in the previous layer.

This section proposes algorithms for efficiently applying SIMD operations over encrypted data on Dense Layers. These algorithms generalize two previously proposed algorithms for DL inference: a Diagonal Matrix-Vector multiplication [HS14a] and a Matrix-Matrix multiplication [Dat+19]. While these matrix multiplication algorithms report simplistic examples, in our work, we describe a generalization together with all the transformation algorithms required for the internal connections.

Diagonal Matrix-Vector Multiplication

This algorithm is based on the multiplication of a ciphertext vector $x \in \mathbb{Z}_Q[x]/(x^N + 1)$ (i.e., data is encrypted, thus providing input-privacy) by a cleartext matrix $\mathcal{W} \in \mathbb{R}^{h \times w}$ (i.e., the weights are not encrypted, thus not providing weight-secrecy). The algorithm decomposes the matrix in the extended diagonals (i.e., diagonal vectors of length h). Then the input vector x is rotated, multiplied by the diagonal matrix, and added. In this way, the algorithm ensures that each entry of the ciphertext vector multiplies each matrix value. Halevi and Shoup were the first to describe this approach in HElib [HS14a]. In their algorithm, authors do not cover the application to HE where the number of elements in the ciphertext n is fewer than the size of the ciphertext N. We provide an explanation and an IR algorithm to solve this obstacle. We also introduce a generalization of the algorithm that enables its application to arbitrary size matrices in Algorithm 4. We discuss the practical implications of its application to HE and its relation to other layers in Section 4.4.

Algorithm 4 Diagonal Matrix Multiplication

```
Input: $c_t \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(v \in \mathbb{R}^n), \mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$
Output: $result$
  function DiagonalMatMul($c_t, \mathcal{W}$)
      $h_{ext} \leftarrow 2^{\lceil log_2(h) \rceil}$
      $w_{ext} \leftarrow 2^{\lceil log_2(w) \rceil}$
      $\alpha \leftarrow max(h_{ext}, w_{ext})$
      $spacing \leftarrow \frac{N}{\alpha}$
      $c_t^{ext} \leftarrow$ SwitchSpacing($c_t$)
      for $i \leftarrow 0, \alpha$ do
          $d_i \leftarrow \{\mathcal{W}_{j,(i+j \mod w)} \mid 0 \leq j < h\}$
          $c_{ti}^{ext} \leftarrow c_t^{ext} \ll (i \cdot spacing)$
          $d_i^{ext} \leftarrow$ Enc($d_i, N$)
          $result \leftarrow result \oplus (d_i^{ext} \odot c_{ti}^{ext})$
      end for
      return $result$
  end function
```

The algorithm takes advantage of the overflow of the input vector during rotations, i.e., when the last values move to the first positions. In HE, the size of the underlying slots is determined by N, which is a power of 2. Introducing a small plaintext vector within a larger ciphertext would prevent us from preserving the overflow behavior. Thus we propose a preprocessing step to keep the overflow happening.

The algorithm relies on two preprocessing steps for correctness. First, to enforce an overflow in longer vectors, the matrix $\mathcal{W} \in \mathbb{R}^{h \times w}$ is extended to the closest power of 2 in both dimensions, resulting in ($h_{ext} = 2^{\lceil log_2(h) \rceil}, w_{ext} = 2^{\lceil log_2(w) \rceil}$). Second, based on these new dimensions, a spacing (Δ) is computed to move and split each entry of the plaintext vector uniformly within the ciphertext vector. This way, the shifts can be weighted by the spacing, and the overflow is kept. Concretely, the spacing is computed as follows: $\Delta = \frac{N}{max(h_{ext}, w_{ext})} \in \mathbb{Z}$. The denominator takes the maximum since a matrix multiplication either reduces or increases the size of the matrix. Note that since N is a power of 2, and so are h_{ext} and w_{ext}, the result is an integer spacing value, also a power of 2. Algorithm 5 shows the algorithm for these preprocessing steps. The algorithm is adapted to switch from a Δ_i spacing between vector elements to Δ_f spacing. In this way, the algorithm can be used both as the RT of a dense layer and the IR of a subsequent dense layer.

Algorithm 5 Initial Representation and Result Transformation for Diagonal Matrix Multiplication

```
Input: $c_t \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(v \in \mathbb{R}^n, p_k)$, $\Delta_i$, $\Delta_f$
Output: $c_t^{ext} \in \mathbb{Z}_Q[x]/(x^N + 1)$
  function SwitchSpacing($c_t, \Delta_i, \Delta_f$)
      $shift = \Delta_f - \Delta_i$
      for $i \leftarrow 0, n$ do
          $bitmask[t]_i \leftarrow \{t = i * \Delta_i\}$
          $c_t^{ext} \leftarrow c_t^{ext} \oplus ((bitmask[t]_i \odot c_t) \gg (i * shift))$
      end for
      return $c_t^{ext}$
  end function
```

Matrix-Matrix Multiplication

We propose an algorithm for a matrix-to-matrix multiplication, which takes as a starting point an example provided in CHET [Dat+19] for matrices of 3x3. Besides offering a general description to apply this in arbitrary size matrices, we also provide the IR and RT algorithms so these matrix multiplications can be chained together in a DL architecture.

We assume that we want to multiply two matrices $A \in \mathbb{R}^{h_A \times w_A}$ and $B \in \mathbb{R}^{h_B \times w_B}$, which are encrypted and formatted in Row-Column format, $c_t^A \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(A \in \mathbb{R}^{h_A \times w_A}, p_k)$ and $c_t^B \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(B \in \mathbb{R}^{h_B \times w_B}, p_k)$. The key concept for the algorithm is the replication of the RC matrix representation. These special and alternative replications permit linear computation of all the necessary combinations. Thus, the main complexity of the algorithm resides in the Initial Representation algorithms. Once this is completed, the overall multiplication complexity is very low.

For matrix $A = \{A_{i,j} \mid 0 \leq i < h_A, 0 \leq j < w_A\}$, its placement over the vector is repeated alternatively according to the formula $c_t^{PA} = \{c_{t_k}^{PA} = A_{i,j} \mid 1 \leq i < h_A, 1 \leq j < w_A, 0 \leq k = \left\lceil \frac{i + j \cdot w_A}{w_B} \right\rceil < h_A \cdot w_A \cdot w_B\}$. Thus, *each element $A_{i,j}$ of matrix A is consecutively repeated w_B times in the vector representation v_A* (e.g., for $A = [1, 2, 3]$, $c_t^{PA} = [1, 1, ..., 2, 2, ..., 3, 3, ...]$). Algorithm 6, shows the process to prepare the matrix c_t^{PA} from a Row Column representation of A denoted as c_t^A.

For matrix $B = \{B_{i,j} \mid 1 < i < h_B, 1 < j < w_B\}$, its transformation involves repeating multiple times the vector according to the formula $c_t^{PB} \in \mathbb{Z}_Q[x]/(x^N + 1) = \{c_{t_k}^{PB} = B_{(k \mod w_B),(\left\lceil \frac{k}{w_B} \right\rceil \mod h_A)} \mid 0 \leq k < h_B \cdot w_B \cdot h_A\}$. Thus, *the Row-Column representation of the matrix B is repeated h_A times* (e.g., for $B = [1, 2, 3]$, $c_t^{PB} = [1, 2, 3, ..., 1, 2, 3]$. Algorithm 7 shows the algorithm to prepare the matrix c_t^{PB}.

Once both matrices are transformed into the specified layout, Algorithm 8 performs element-wise multiplications of the vectors (obtaining a vector $c_t^C = c_t^{PA} \odot c_t^{PB}$) and applies w_A rotations and sums to the product, obtaining: $c_t^{C'} = \sum_{i=0}^{h_A} c_t^C \ll N - (i \cdot w_B)$.

Algorithm 6 Initial Representation for Matrix A in Matrix Multiplication

```
Input: $c_t^A \in \mathbb{Z}_Q[x]/(x^N+1) = Enc(A \in \mathbb{R}^{h_A \times w_A}, p_k)$, $w_B$
Output: $c_t^{PA}$
  function PrepareMatrixA($c_t^A, w_B$)
      for $i \leftarrow 0, h_A \cdot w_A$ do
          $bitmask[t]_i \leftarrow \{t = i\}$
          $c_{ti}^A = bitmask[t]_i \odot c_t^A$
          $partial \leftarrow partial \oplus (c_{ti}^A \gg (i \cdot (w_B - 1))$
      end for
      for $i \leftarrow 0, w_B$ do
          $c_t^{PA} \leftarrow c_t^{PA} \oplus (partial \gg i)$
      end for
      return $c_t^{PA}$
  end function
```

Algorithm 7 Initial Representation for Matrix B in Matrix Multiplication

```
Input: $c_t^B \in \mathbb{Z}_Q[x]/(x^N+1) = Enc(B \in \mathbb{R}^{h_B \times w_B}, p_k)$, $h_A$
Output: $c_t^{PB}$
  function PrepareMatrixB($c_t^B, h_A$)
      for $i \leftarrow 0, h_A$ do
          $result \leftarrow result \oplus (c_t^B \gg (i \cdot h_B \cdot w_B))$
      end for
      return $result$
  end function
```

Algorithm 8 Matrix-Matrix Multiplication

```
Input: $c_t^{PA} \in \mathbb{Z}_Q[x]/(x^N+1)$, $c_t^{PB} \in \mathbb{Z}_Q[x]/(x^N+1)$, $\mathbb{R}^{h_A \times w_A}$, $\mathbb{R}^{h_B \times w_B}$
Output: $c_t^{C'} \in \mathbb{Z}_Q[x]/(x^N+1)$
  function MatrixMatrixMul($c_t^{PA}, c_t^{PB}$)
      $c_t^C \leftarrow c_t^{PA} \odot c_t^{PB}$
      for $i \leftarrow 0, w_A = h_B$ do
          $c_t^{C'} \leftarrow c_t^{C'} \oplus (c_t^C \gg (N - (i \cdot w_B))$
      end for
      return $c_t^{C'}$
  end function
```

However, due to the nature of the algorithm, this result contains extra spacing that needs to be discarded. Concretely, w_B relevant items in the vector (i.e., items from the actual outcome of the multiplication) are followed by w_B non-relevant ones (i.e., irrelevant artifacts). These are discarded in a Result Extraction algorithm to finally get the Row-Column representation of the multiplication (see Algorithm 9, where $c_t^{C'}$ is the result with spacing that needs to be transformed, and $c_t^{A \times B}$ is the output of the transformation).

Algorithm 9 Result Extraction Matrix-Matrix Multiplication

```
Input: $c_t^{C'} \in \mathbb{Z}_Q[x]/(x^N + 1)$, $\mathbb{R}^{h_A \times w_B}$
Output: $c_t^{A\times B} \in \mathbb{Z}_Q[x]/(x^N + 1)$ in RC format:
  function RE-MatrixMatrixMul($c_t^{C'}$)
      $bitmask[t] \leftarrow \{0 \leq t < w_B\}$
      for $i \leftarrow 0, h_A$ do
          $bitmask[t]_i \leftarrow bitmask[t] \gg w_B$
          $c_{ti}^{C'} \leftarrow (c_t^{C'} \ll (i \cdot w_B)) \odot bitmask[t]_i$
          $c_t^{A\times B} \leftarrow c_t^{A\times B} \oplus c_{ti}^{C'}$
      end for
      return $c_t^{A\times B}$
  end function
```

4.3.4 Activation Functions

Neural Networks have excelled at classification and regression tasks because they can map non-linear distributions. Activation functions are a crucial component of such success, and their integration within FHE schemes is a hot research topic in the literature. Linear approximations of various kinds are among the most successful yet straightforward proposals to introduce activation functions in HE-based DL. Authors have proposed solutions ranging from alternative polynomials [Gil+16; Xie+14], Taylor and Chebyshev Polynomials [HTG17] or simple linear regressions [ISY20; WH12]. The common goal of these works is to obtain a low-degree polynomial to approximate the non-linear behavior as accurately as possible. This chapter assumes that the activation functions have been approximated to polynomials using existing proposals. Thus, computing an activation function to Packed Homomorphic Encryption does not require any particular format or construction as it applies the transformation to all the slots of the ciphertext. We can often insert the activation layers with other layers or building blocks of the layers (i.e., algorithm execution and result transformation) without changing the final result. In Section 4.4.3, we provide insights on where introducing activation functions for the convolutional and dense blocks would be more or less desirable.

4.4 Efficiency Analysis of Algorithms

The previous section presented algorithms for SIMD execution of CNN inference. This section formally analyzes the algorithms' efficiency and performance impact. Indeed, efficiency is one of the biggest challenges for applying HE for DL. We first define the metrics used to measure efficiency. Second, we provide some insights regarding applying rotations and large ciphertext vectors. Finally, we analyze all the algorithms in terms of the proposed metrics. That enables us to provide a series of guidelines for their application.

4.4.1 Efficiency Metrics

Generally, HE operations are performance-wise heavy to execute over ciphertexts. Our analysis focuses on the transformations applied to ciphertexts. Plaintext operations have a negligible impact on the computation; thus, we do not account for them in the analysis. For evaluating these algorithms, we rely on four metrics that define the efficiency of a circuit C:

Multiplication Depth ($O_{\mathfrak{D}}$) defines the maximum number of consecutive products that an HE ciphertext needs to apply in a given circuit C. The multiplication depth directly impacts the parametrization of HE schemes, specifically in N and Q. In terms of cleartext operations, N defines the polynomial degree. Thus, a bigger N would involve operating over larger degree polynomials (i.e., more coefficients to compute per ciphertext operation). Also, working with bigger Q involves computing more remainders. In Leveled Homomorphic Encryption (LHE) Schemes, each multiplication usually requires a rescaling operation to reduce the underlying noise. Therefore, we consider the need for one rescaling per $O_{\mathfrak{D}}$ (i.e., per multiplication). In this case, we need $O_{\mathfrak{D}}$ different moduli (q_i) in a polynomial coefficient modulus Q. In direct relation with Q, N often defines a maximum capacity for a Q (i.e., increasing the $O_{\mathfrak{D}}$ may not only involve increasing Q but also N). The optimization of these parameters is of paramount importance to obtain better runtimes. For all these reasons, keeping a minimal depth of the circuit is very important for achieving efficiency in the desired computation, which justifies why $O_{\mathfrak{D}}$ is one of the metrics analyzed for the efficiency of the algorithms.

Operation Cost differs across the different available computations in HE. Multiplication is the most costly since it requires multiplication and is paired with a relinearization phase (i.e., preventing the polynomial degree from growing) and a rescaling phase (i.e., reducing the noise scale). The next more costly operation is rotation, which involves generating different Galois Keys. In CKKS, if the encoding scale s is chosen the same as the smallest modulus prime q_i, we can neglect the noise and depth cost. Element-wise additions are the lowest cost operation and are considered linear in computation and noise growth. For the rest of the analysis, we denote the addition and subtraction complexity (O_{sum}) as the number of sums (and subtractions) required by a circuit. Likewise, we consider the multiplication complexity (O_{mul}) and rotation complexity (O_{rot}) as the number of multiplications and rotations in the circuit.

Memory Complexity (O_{mem}) accounts for the number of ciphertexts needed in memory to execute one of the algorithms. Given the large memory size of ciphertexts, minimizing the number of ciphertexts simultaneously residing in the main memory is essential.

Memory Constraints (O_{con}) determines the constraints that an algorithm imposes on the size of plaintext vectors n it operates with, so these can fit in ciphertext with N slots (i.e., $n < N$). If the plaintext vectors do not fit in the ciphertext, the circuit would require an extended vector representation (i.e., the plaintext vectors are packed within multiple ciphertexts). In general, for most algorithms, we consider that for an input matrix $\mathcal{M} \in$

Metric	$O_{\mathfrak{D}}$	O_{sum}	O_{mul}	O_{rot}	O_{mem}	O_{con}
Value	1	$2 \cdot r$	r	r	$2 \cdot r$	-

Table 4.1: Efficiency analysis of Rotation of a big cleartext vector when it is packed over multiple ciphertexts (Algorithm 10)

$\mathbb{R}^{h \times w}$, we can compute the algorithm if $h \cdot w \leq N$ (i.e., if the full matrix size fits in a ciphertext vector slot). However, some specific algorithm representations of information may define harder or softer limits for the execution. As we detail in the following section, the memory constraints O_{con} directly impact the operation cost. In the following sections, we use r to define the number of ciphertext vectors of size N needed to host a plaintext vector of length n.

4.4.2 Rotations on Large Ciphertexts

The effect of the 'Memory Constraints' O_{con} is essential for the Rotation operation. In HE programming, most algebra circuits present memory constraints, given the difficulty of packing all the information within the same ciphertext. In parallelism with classical (non-HE) programming, we could consider when a program does not fit into the available memory and uses swap space. At that point, computation becomes a constraint and is more expensive. In HE circuits with packing, when we need multiple ciphertexts to represent the plaintext data, rotations have a worse impact on the efficiency of algorithms. Indeed, many algorithms rely on rotations to benefit from SIMD operations (e.g., to transform output layouts). Previous works assume rotations as 'cost-free' operations and thus use them arbitrarily [Dat+19]. We observe, however, that when an algorithm is generalized to work on arbitrary-size plaintext inputs (often large scale), the assumption does not hold anymore. Suppose the plaintext vector entries n extend over the available slots N. In that case, multiple ciphertexts are required, and the plaintext vectors and rotation cost are no longer neglectable since at least $r = \lceil n/N \rceil$ ciphertext vectors are needed.

To demonstrate this performance decrease, we depict in Figure 4.4 the rotation procedure for $n > N$, i.e., when more than one ciphertext is needed. In the example, we represent one input vector with two ciphertexts. Considering a 2-left rotation ($\ll 2$), we observe that individual rotations of the vectors are partial. Thus, this involves further modifications, such as an additional multiplication of the vectors by a mask, which increases $O_{\mathfrak{D}}$. Table 4.1 shows the overall complexity of this process. We provide the details in Algorithm 10.

4.4.3 Analysis and Takeaways for Application to Deep Learning

This section provides a detailed analysis of the algorithms presented in Section 4.3 concerning the proposed efficiency metrics. Table 4.2 presents the formal performance

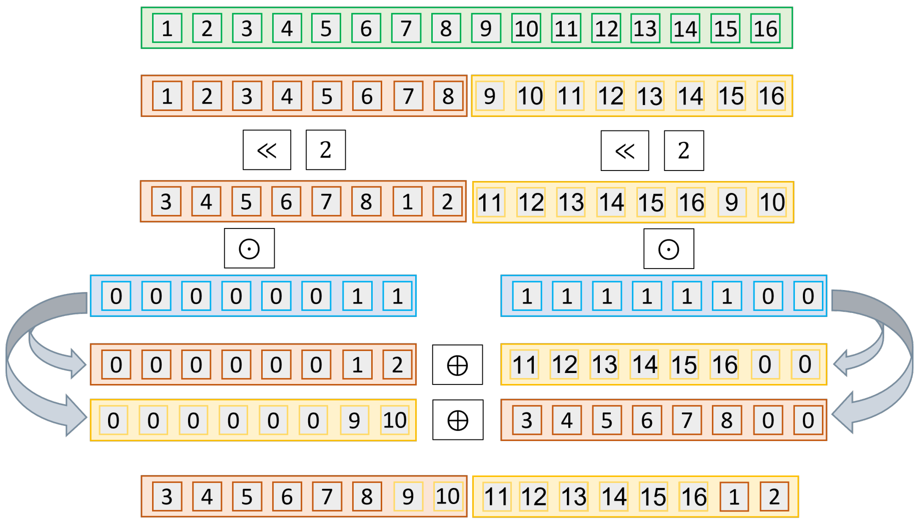


Figure 4.4: Operations performed for privately rotating twice ($ct \ll 2$) a vector of dimension $n = 16$ in with homomorphic encryption ciphertexts of size $N = 8$ and $r = 2$.The vector is encoded in two ciphertexts. The ciphertexts need to be rotated, masked and reorganized to obtain the same result as in the plaintext rotation.

Algorithm 10 Rotation r times of $\mathcal{V}$ cleartext vector encoded in multiple ciphertexts $v_0, v_1, ..., v_n$.

```
function ROTATE($\{c_{t0}, c_{t1}, ..., c_{tr} \in \mathbb{Z}_Q[x]/(x^N + 1)\} = Enc(\mathcal{M}, p_k), rot$)
    $q, rot \leftarrow \lceil rot/r \rceil, rot \mod r$
    $bitmask[t] \leftarrow \{N - r \leq t < N\}$
    for $i \leftarrow 0, r$ do
        $c'_{ti} \leftarrow c_{ti} \gg rot$
        $c^0_{ti} \leftarrow c'_{ti} \odot bitmask[t]$
        $c^1_{ti} \leftarrow c'_{ti} \ominus c^0_{ti}$
    end for
    for $i \leftarrow 0, r$ do
        $c^{rot}_{ti} \leftarrow c^0_{t(i-q \mod r)} \oplus c^1_{t(i-1-q \mod r)}$
    end for
    return $\{c^{rot}_{t0}, c^{rot}_{t1}, ..., c^{rot}_{tr}\}$
end function
```

complexity extracted from the different algorithms. Next, we give key insights extracted from the analysis and discuss future directions and best practices to apply the algorithms for DL Inference.

The Streamlined Convolutional Blocks reduce the multiplication depth. The improved version of the algorithm we propose in Section 4.3.2 introduces many efficiency improvements to the base algorithm. The streamlined version of the algorithms allows to insert l_{conv} convolutions, l_{stride} strided convolutions, l_{pad} padding layers, l_{pool} average

Formal Analysis of Algorithms

Algorithm	Metrics					
Name	$\mathcal{O}_{con}$	$\mathcal{O}_{\mathfrak{D}}$	$\mathcal{O}_{sum}$	$\mathcal{O}_{mul}$	$\mathcal{O}_{rot}$	$\mathcal{O}_{mem}$
ALG (1) Str. Convolution	$h{\cdot}w \leq N$	1	$f_x{\cdot}f_y$	$f_x{\cdot}f_y$	$f_x{\cdot}f_y$	3
	$h{\cdot}w > N$	2	$3r{\cdot}f_x{\cdot}f_y$	$2r{\cdot}f_x{\cdot}f_y$	$r{\cdot}f_x{\cdot}f_y$	$4r$
ALG (11) Convolution	$h{\cdot}w \leq N$	1	$f_x{\cdot}f_y$	$f_x{\cdot}f_y$	$f_x{\cdot}f_y$	3
	$h{\cdot}w > N$	2	$3r{\cdot}f_x{\cdot}f_y$	$2r{\cdot}f_x{\cdot}f_y$	$r{\cdot}f_x{\cdot}f_y$	$4r$
ALG (2) Str. Padding	$h_0{\cdot}w_0 \leq N$	1	0	1	1	2
	$h_0{\cdot}w_0 \leq N$	2	$2r$	$2r$	r	$3r$
ALG (14) Private Padding	$h{\cdot}w \leq N$	1	h	h	h	3
	$h{\cdot}w > N$	2	$3r{\cdot}h$	$2r{\cdot}h$	$r{\cdot}h$	$4r$
RT (3) SCBF to RC	$h{\cdot}w \leq N$	1	$h_{out}{\cdot}w_{out}$	$h_{out}{\cdot}w_{out}$	$h_{out}{\cdot}w_{out}$	3
	$h{\cdot}w > N$	2	$3r{\cdot}h_{out}{\cdot}w_{out}$	$2r{\cdot}h_{out}{\cdot}w_{out}$	$r{\cdot}h_{out}{\cdot}w_{out}$	$4r$
RT (12) CRF to RC	$h{\cdot}w \leq N$	1	h_{out}	h_{out}	h_{out}	3
	$h{\cdot}w > N$	2	$3r{\cdot}h_{out}$	$2r{\cdot}h_{out}$	$r{\cdot}h_{out}$	$4r$
RT (13) SCRF to RC	$h{\cdot}w \leq N$	1	$h_{out}{\cdot}w_{out}$	$h_{out}{\cdot}w_{out}$	$h_{out}{\cdot}w_{out}$	3
	$h{\cdot}w > N$	2	$3r{\cdot}h_{out}{\cdot}w_{out}$	$2r{\cdot}h_{out}{\cdot}w_{out}$	$r{\cdot}h_{out}{\cdot}w_{out}$	$4r$
IR/RT (5) Diag. Mat. Mult.	$n{\cdot}(\Delta_f - \Delta_i + 1) \leq N$	1	n	n	n	3
	$n{\cdot}(\Delta_f - \Delta_i + 1) > N$	2	$3r{\cdot}n$	$2r{\cdot}n$	$n{\cdot}r$	$4r$
ALG (4) Diag. Mat. Mult.	$max(\alpha, n) \leq N$	1	α	α	α	4
	$max(\alpha, n) > N$	2	$3r{\cdot}\alpha$	$2r{\cdot}\alpha$	$r{\cdot}\alpha$	$5r$
IR (6) Prepare Matrix A	$h_A{\cdot}w_A{\cdot}w_B \leq N$	1	$h_A{\cdot}w_A + w_B$	$h_A{\cdot}w_A$	$h_A{\cdot}w_A + w_B$	3
	$h_A{\cdot}w_A{\cdot}w_B > N$	3	$3r{\cdot}h_A{\cdot}w_A + w_B$	$r(2{\cdot}h_A{\cdot}w_A + w_B)$	$r{\cdot}(h_A{\cdot}w_A + w_B)$	$4r$
IR (7) Prepare Matrix B	$h_B{\cdot}w_B{\cdot}h_A \leq N$	0	h_A	0	h_A	2
	$h_B{\cdot}w_B{\cdot}h_A > N$	1	$3r{\cdot}h_A$	$r{\cdot}h_A$	$r{\cdot}h_A$	$3r$
ALG (8) Mat-Mat. Mult.	$max(h_A{\cdot}w_A{\cdot}w_B, h_B{\cdot}w_B{\cdot}h_A) \leq N$	1	$w_A = h_B$	1	$w_A = h_B$	4
	$max(h_A{\cdot}w_A{\cdot}w_B, h_B{\cdot}w_B{\cdot}h_A) > N$	2	$3r{\cdot}w_A$	$r(w_A + 1)$	$r{\cdot}w_A$	$5r$
RE (9) Mat-Mat. Mult.	$n \leq N$	1	h_A	h_A	h_A	4
	$n > N$	2	$3r{\cdot}h_A$	$2r{\cdot}h_A$	$r{\cdot}h_A$	$3r$

Table 4.2: Detailed analysis of the different metrics proposed in Section 4.4.1 ($\mathcal{O}_{con}$, $\mathcal{O}_{\mathfrak{D}}$, $\mathcal{O}_{sum}$, $\mathcal{O}_{mul}$, $\mathcal{O}_{rot}$ and $\mathcal{O}_{mem}$). Additionally, it shows the performance variation if Rotations are as in Algorithm 10.

pooling layers or l_{act} activation functions (with cost $\mathcal{O}_{\mathfrak{D}}^{act}$). This results in a depth cost of:

$$\mathcal{O}_{\mathfrak{D}} = l_{conv} + l_{stride} + l_{pad} + l_{pool} + l_{act} \cdot \mathcal{O}_{\mathfrak{D}}^{act} + 1$$

The cost is one operation per layer and the RT algorithm. Also, it does not make any difference in using stridden or non-stridden convolutions. On the other hand, the base version proposed in previous work requires applying a RT function after each convolution, and the stride makes it more expensive in the $\mathcal{O}_{mul}$. The overall cost of the base version is:

$$\mathcal{O}_{\mathfrak{D}} = 2l_{conv} + 2l_{stride} + l_{pad} + 2l_{pool} + l_{act} \cdot \mathcal{O}_{\mathfrak{D}}^{act}$$

In summary, the base version heavily affects the depth because it requires RT algorithms.

The Streamlined Convolutional Blocks reduce the overall cost of auxiliary convolution routines. If we analyze the cost of operations, we can see how the overall cost of the streamlined convolution algorithm does not change concerning the base algorithm. However, looking at the rest of the streamlined routines (i.e., padding or stride), we can observe how the cost is highly reduced. Although the padding occupies the same multiplication depth slot, it reduces its cost to a single multiplication and rotation. Furthermore, the reduction in the cost of stride permits using it freely, allowing for faster training algorithms over higher dimensionality data. If we used the baseline algorithm, it would be preferable not to use padding and stride to 1 as much as possible to keep efficiency.

Matrix Multiplication Algorithm Comparison on l-Layer Dense Neural Network

Algorithm		Metrics					
Name	Alg.	O_{con}	$O_{\mathfrak{D}}$	O_{sum}	O_{mul}	O_{rot}	O_{mem}
Diagonal Matrix Multiplication	5, 4	$max(\alpha, n) \leq N$	$2l + 1$	$\alpha(2l + 1)$	$\alpha(2l + 1)$	$\alpha(2l + 1)$	4
		$max(\alpha, n) > N$	$4l + 2$	$3r{\cdot}\alpha(2l + 1)$	$2r{\cdot}\alpha(2l + 1)$	$\alpha{\cdot}r{\cdot}(2l + 1)$	$5n$
Matrix-Matrix Multiplication	6, 7, 8, 9	$max(h_A \cdot w_A, h_B^2) \leq N$	$2l$	$l(2h_A + w_A)$	$l(h_A + 1)$	$l(2h_A + w_A)$	5
		$max(h_A \cdot w_A, h_B^2) > N$	$5l$	$3r{\cdot}l(2h_A + w_A)$	$2r{\cdot}l(3h_A + w_A + 1)$	$l{\cdot}r{\cdot}(2h_A + w_A)$	$6n$

Table 4.3: Detailed comparison of the different complete matrix multiplication algorithms described in the thesis according to the different metrics proposed in Section 4.4.1 (O_{con}, $O_{\mathfrak{D}}$, O_{sum}, O_{mul}, O_{rot} and O_{mem}) for a generic l-layer Neural Network. Additionally, it shows the performance variation if rotations are computed as in Algorithm 10. Note that, for Algorithms 4 and 5, we can consider $\alpha = n$. In Algorithms 7, 8, and 9, we consider the optimization of not using Preprocessing A and considering B is a one-dimensional vector.

Prioritize IR Prepare Matrix B over IR Prepare Matrix A. Comparing both algorithms, we observe a clear advantage in the algorithm used to prepare Matrix B in the Matrix-Matrix multiplication. Indeed, both $O_{\mathfrak{D}}$ and O_{mul} are smaller than in Prepare Matrix A. Furthermore, we consider the weights matrix to be provided in cleartext for DL Inference. In such a case, we recommend prioritizing IR Prepare Matrix A over the cleartext matrix (i.e., executing the heavy algorithm over the plaintext matrix); thus, the IR Prepare Matrix B algorithm on the ciphertext space. It allows for improving the overall performance of the multiplication routine while maintaining input privacy as a constraint.

Avoid using the Matrix-Matrix Algorithm for large input matrices. We observe that the Matrix-Matrix Algorithm (Alg. 8) imposes significant limits on the size of matrices that can be multiplied (O_{con}). For example, for a latent vector size N of 1024, the maximum length of two square matrices A, B that we can privately multiply is around 10×10. Introducing a larger size of N (e.g., 16,384 or 32,768) would improve this factor slightly (e.g., 25×25 and 32×32, respectively). This problem occurs due to the replication factor introduced by the algorithm, i.e., it requires the replication of A's RC format w_B times and B's RC format w_A times. In a real-world setting, Neural Networks often involve larger matrices. Encoding those input matrices involves using multiple ciphertexts to represent the plaintext vector. The performance would be less in the encoding and execution time (as rotations demand). It also increases the memory requirements and the number of required operations (as described in Table 4.2).

The Matrix-Matrix multiplication Algorithm improves when B is a one-dimensional vector. This optimization partially overcomes certain of the previously presented weaknesses of this algorithm. Indeed, if we consider a real use case, often dense layers are flattened, representing information as a one-dimensional vector. If matrix B is a vector (i.e., $w_B = 1$), the constraint O_{con} is reduced to $max(h_A{\cdot}w_A, h_B{\cdot}w_A) = max(h_A{\cdot}w_A, h_B^2) \leq N$. Furthermore, for reducing dense layers, where the size of the output vector is smaller than the size of the input vector (i.e., $h_A \leq w_A = h_B$), the constraint would be just on the shape of the underlying vector to the ciphertext. This constraint still imposes hard constraints for the underlying vector size (e.g., around 180 elements for $N = 32,768$ or 128 for $N = 16,384$).

Choosing between Matrix-Matrix or Diagonal Matrix Multiplication mostly depends on O_{con}. Table 4.3 shows both algorithms' overall cost of an arbitrary l-layer dense architecture. First, it is essential to consider the memory constraints of ciphertexts O_{con}. The Matrix-Matrix Multiplication generally remains more efficient for small underlying plaintext vectors n. The improvement is due to having a lower $O_{\mathfrak{D}}$ and half the number of multiplications O_{mul} than the Diagonal Matrix Multiplication. However, this only holds under the O_{con} assumption, where we can represent the plaintext information under a single ciphertext. If not, the diagonal matrix multiplication becomes more efficient (i.e., the algorithm accepts larger matrix dimensions). At the same time, we must consider that increasing N to permit using more underlying plaintext elements n and working with the Matrix-Matrix Multiplication may be counterproductive. This deficiency is due to the cleartext operations performed to execute a ciphertext operation. When we increase N, so does the number of cleartext operations to compute on each polynomial. Therefore, keeping a minimal N becomes likewise critical for efficiency. Before increasing N, using the Diagonal Matrix Multiplication would be better for performance. Finally, if the constraint O_{con} requires multiple vectors in both instances, it would be needed a trade-off between $O_{\mathfrak{D}}$ and O_{mul} based on the number of layers l. While the overall complexity remains similar for both algorithms, it is important to note two things. First, O_{mul} grows double as the number of layers l grows. Indeed, the Diagonal Matrix multiplication is less efficient for the same number of layers. Second, the increase of $O_{\mathfrak{D}}$ of l reduces in the Diagonal Matrix Multiplication with a comparison $4l + 2 \leq 5l$. For neural network architectures with one dense layer $l = 1$, the $O_{\mathfrak{D}}$ is better with the Matrix-Matrix multiplication algorithm. In the rest of the cases $l > 1$, the overall $O_{\mathfrak{D}}$ of the Diagonal Matrix is smaller.

4.5 Performance Evaluation of Guidelines

We conduct different experiments to study the algorithms' impact on real-world inference and corroborate the formal analysis and the critical findings from Section 4.4. These

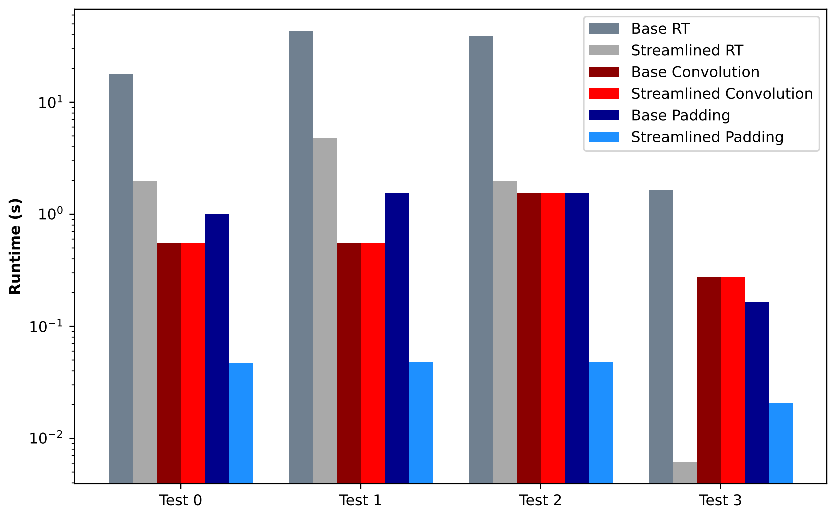


Figure 4.5: Convolution performance evaluation between streamlined and baseline approaches to algorithms.

experiments implement variations from the base use case described in A[Dat+19]. In each experiment, we conduct various tests varying the architectures and parameters to examine the performance impact of the different designed routines.

The first experiment compares the baseline and streamlined convolution algorithms. The use case executes a set of c convolutions in a row. All the convolutions have the same properties, with the parameter values described in Table 4.4. The results are depicted in Figure 4.5. As expected by the formal analysis, the streamlined convolution algorithm does not impact the convolution operation since the execution times are similar. However, the algorithm produces an output format where the placement of the elements allows for efficient integration with the following layers. It impacts the execution time of the padding and the results transformation algorithms achieving a speedup of 8 times faster on average. Also, we can observe that in the baseline algorithm, the result transformation involves a substantial part of the computational effort, with a high impact on the overall performance.

In the second experiment, we evaluate the Matrix-Matrix Multiplication algorithm. We compute a sequence of Prepare Matrix Multiplications (for both A and B), the matrix multiplication algorithm, and the result transformation. We show the test cases evaluated in the second section of Table 4.4 and the results in Figure 4.6. As demonstrated in Section 4.4, the IR Algorithm executed for Matrix A is highly inefficient. However, the IR for Matrix B is much more efficient, supposing a relatively minor difference. Therefore, if we consider one of the matrices to be cleartext (e.g., the weights of a Dense Layer are not private), we should always choose it to be matrix A. Another conclusion drawn from Figure 4.6 is the relevance of the IR and RT algorithms. Even in the most optimal use case, the matrix multiplication itself only supposes 40% of the total amount of processing. Therefore, in other works that omit the IR or RT, the overall performance is only partially

Convolution Test Case						
Test	Num.	Initial Shape	Kernel Size	Stride	Padding	Speedup
0	10	20×20	3×3	$(1,1)$	1	7.51
1	10	30×30	3×3	$(1,1)$	1	8.37
2	10	40×40	5×5	$(1,1)$	1	11.75
3	5	30×30	3×3	$(2,2)$	1	6.86

Matrix-Matrix Multiplication Test Case		
Test	Matrix Shape	N
0	2×2	16
1	3×3	32
2	4×4	2048
3	5×5	2048
4	20×20	8192

Diagonal Dot Multiplication Test Case		
Test	Matrix Shape	N
0	3×3	16
1	3×24	64
2	24×3	64
3	4×50	2048
4	50×4	2048
5	50×50	64
6	50×50	128
7	50×75	2048
8	75×100	2048

Matrix-Matrix vs Diagonal Matrix Multiplication Test Case		
Test	Matrix Shape	N
1	5×5	512
2	10×10	1024
3	20×20	8192
4	40×40	65536
5	50×50	65536

Table 4.4: Parametrization of the different tests performed for each of the four takeaways considered in Subsection 4.4.3.

shown, as the transformations involve a significant portion of the overhead.

The third experiment analyses the Diagonal Matrix Multiplication. For comparison purposes, we provide a detailed analysis of the diagonal matrix multiplication algorithm in Figure 4.7. This algorithm generally shows the lesser O_{con} requirements of the Diagonal Matrix multiplications. For bigger matrix sizes, the underlying ciphertext vector needs a smaller size of N than the Matrix-Matrix Multiplication Algorithm. The tests executed on these algorithms can be found in the third subdivision of Table 4.4. As we can observe, the ordering of dimensions influences the preprocessing algorithm's time. In general, the maximum size of the matrices defines the length of the extended diagonal. Therefore, the test with matrices of 50×4 (test 4) obtains similar performance times to the test with matrices of 50×50 (tests 5 and 6). Also, given the small dimensions, the differences between the two tests with matrices of 50×50 (tests 5 and 6) are negligible.

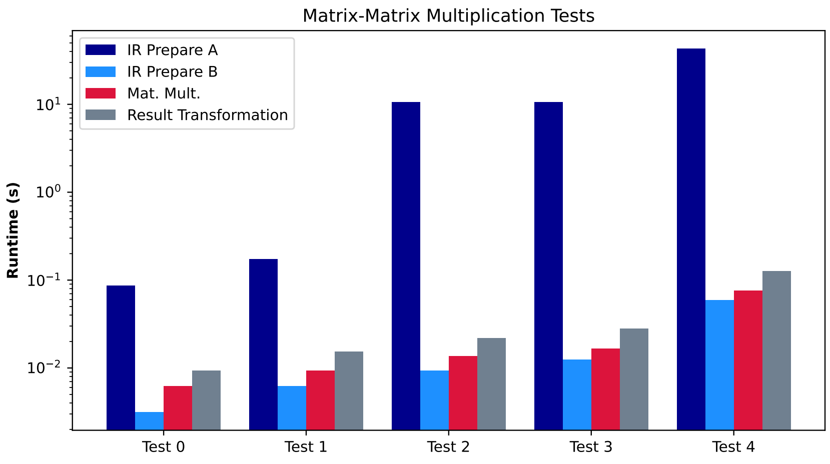


Figure 4.6: Performance evaluation of Alg. 8. The graph compares the two preprocessing algorithms together with the multiplication and RT algorithms in absolute terms (execution time).

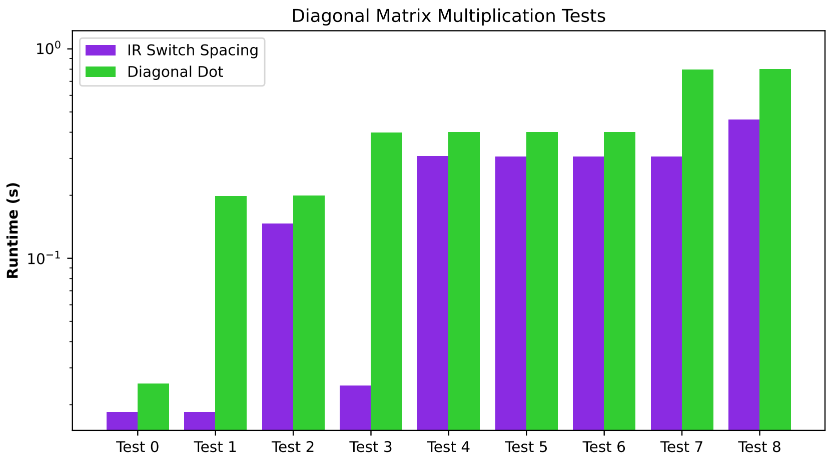


Figure 4.7: Performance evaluation of the diagonal matrix multiplication of two matrices. The graph shows the overall cost of the two main routines for IR/RT and the algorithm in absolute terms and relative to the execution time.

The fourth experiment compares the different matrix multiplication algorithms. We perform the fourth experiment with the exact dimensions of the Matrix-Matrix and the Diagonal Dot multiplication algorithms. It enables us to compare the algorithms accurately. We provide the results in Figure 4.8 and the executed tests at the bottom of Table 4.4. Overall, we can observe how for smaller sizes of matrices, the lower execution time of the Matrix-Matrix Multiplication Algorithm imposes better runtimes. However, once the dimensions grow, the Diagonal Matrix Multiplication provides better runtimes. However, we note that the tests may give misleading information since, for the same N, the Diagonal Matrix Multiplication enables working with more significant matrices and generally involves less computation.

Finally, our fifth experiment combines the different algorithms in a Neural Network

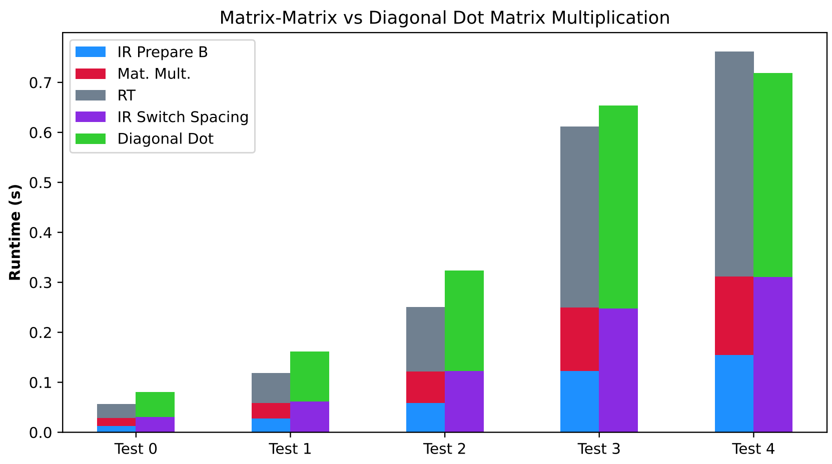


Figure 4.8: Performance comparison of the Matrix-Matrix Multiplication Algorithm and the Diagonal Matrix Multiplication Algorithms.

use case for cardiology and healthcare. This test analyzes the implications of putting together the algorithm in an actual use case. For that, we develop a CNN model based on the CheXpert dataset [Irv+19] with the typical Homomorphic Encryption-based architecture of Cryptonets [Gil+16]. We perform inference on 250 samples and obtain the average runtime of the layers for the different proposed algorithms. The results of such tests are depicted in Figure 4.9. First, we observe a noticeable difference between the first layers of the Neural Network and the last layers. As we showed in Section 4.4, the complexity of the algorithms is often determined by the dimensionality of the treated matrices. The first layers deal with larger dimensions; thus, the computation is more affected by such dimensionality. This fact is especially noticeable with the Convolution and Average Pooling layers, where the RT is affected by such high dimensionality. Furthermore, when using Homomorphic Encryption, this behavior is emphasized with the existence of LHE, which introduces the concept of levels. Levels are treated with the Chinese Remainder Theorem and operate with more levels before dropping them with each rescaling. On the first layers, the efficiency is worse before rescaling, as HE works on more remainders than in the latest layers, where most of the moduli have been dropped. Intermediate layers introduce a reduced delay due to being fundamentally a processing-based layer requiring no internal reorganization of the vectors. As a last factor, we analyze the algorithms' precision compared to classic algorithms and obtain an equivalent absolute precision difference of $3.79 \cdot 10^{-6}$, which we consider negligible for this application.

4.6 Summary

Efficient Homomorphic Encryption (HE) requires structural optimizations to perform the complex computation of the internal layers. One such optimization is Packed Homomorphic Encryption (PaHE), which encodes multiple elements on a single ciphertext, allow-

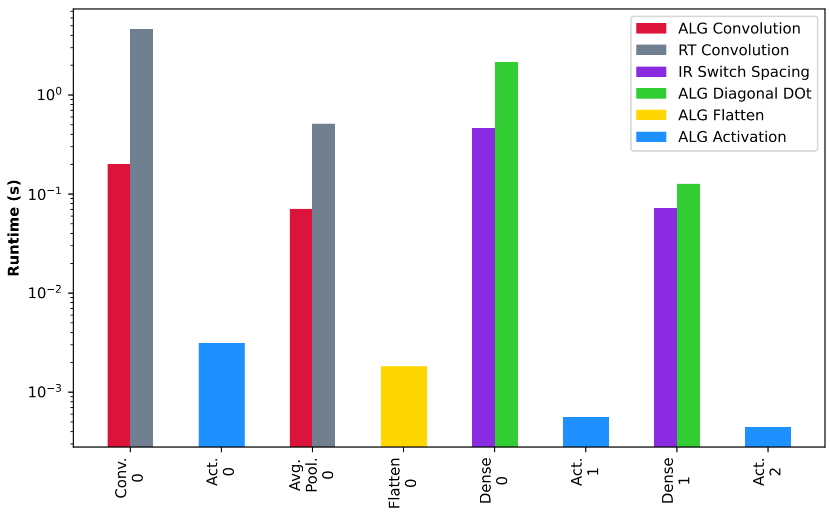


Figure 4.9: Performance evaluation Cryptonets Neural Network based on the CheXpert dataset. Average runtime of the execution of 250 random samples of the dataset. The ordering of the x-axis corresponds to the layer execution order (i.e., Conv. 0 is the first layer, and Act. 2 is the last layer).

ing for efficient SIMD operations. However, using PaHE in DL circuits is not straightforward, and it demands new algorithms and plaintext-ciphertext data mapping, which existing literature has not adequately addressed. To fill this gap, in this chapter, we elaborate on novel algorithms to adapt the linear algebra operations of DL layers to PaHE, focusing on Convolutional Neural Network (CNN). We provide detailed descriptions and insights into the different algorithms and efficient inter-layer data format conversion mechanisms. We formally analyze the complexity of the algorithms in terms of performance metrics and provide guidelines and recommendations for adapting architectures that deal with private data. Furthermore, we confirm the theoretical analysis with practical experimentation. Among other conclusions, we prove that our new algorithms speed up the processing of convolutional layers compared to the existing proposals.

Chapter 5

Automating Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming

In Chapter 4, we discussed the complexity of performing efficient vectorization of Packed Homomorphic Encryption algorithms. Parametrization of Homomorphic Encryption schemes is another long-studied challenge that introduces a multi-faced problem with numerous implications regarding precision, performance, and security. To solve this issue, in this chapter, we propose an expert system that allows the selection of an optimal set of parameters for HE based on the combination of Fuzzy Logic and Linear Programming from high-level user parameters. The resulting solution reduces the expertise needed to deal with HE schemes and yields parametrizations that preserve the high-level features expressed by the user.

The remainder of this chapter is organized as follows. Section 5.1 motivates the main problem dealt with in the chapter. Then, Section 5.2 provides a background to the different techniques used in this chapter and concretely details insights into the parametrization of HE schemes. Next, Section 5.3 introduces the design of the expert system, which is then evaluated in Section 5.4. Finally, Section 5.5 summarizes the main contributions of this chapter. For interested readers, in Appendix B, a detailed practical description of parameter selection is overviewed.

5.1 Introduction

Selecting critical parameters for parametrization in Homomorphic Encryption (HE) frameworks is complex due to various issues (e.g., circular relations between parameters, low-level knowledge requirements, and different parameter value combinations tradeoffs). This section describes the problem and the other solutions provided in previous works.

5.1.1 Motivation

One of the main unaddressed challenges of current HE frameworks is the selection of a set of critical parameters, i.e., the parametrization. In a nutshell, parameter selection is complex since there are circular relations between parameters and the underlying circuit. Also, the combination of different values for these parameters results in a tradeoff between three essential features of privacy-preserving scenarios, i.e., security (having robust encryption), performance (requiring a reasonable amount of computational resources), and precision (limiting to an affordable amount of noise in the final result). Furthermore, to understand the consequences, the user needs detailed knowledge about the low-level primitives.

Also, the Homomorphic Encryption Standard defines minimum settings to establish secure parameters in HE [Alb+18]. Like classical cryptography, where the selection of critical parameters (such as RSA key size and AES block size) is not left to end-users or programmers, for widespread adoption of HE, solutions shall not leave these decisions to users. Still, it is needed to provide degrees of freedom for the user to determine parameters such as security, performance, or precision, since these might depend on the scenario of the application of the circuit.

5.1.2 Precedents

In recent years, some proposals have looked at the affordability of HE techniques for end-users, presenting HE protocols as application programming interfaces (API) or compilers [VJH21]. Compilers and APIs are software pieces that aim to provide a simple functionality for end programmers while hiding the complexity employed in the logic of programs. In HE, compilers and APIs allow the users to access abstracted representations of low-level instructions with reduced complexity and within standard programming languages. The software generates a logical coupling between the high-level model and the underlying HE circuit, concealing the particularities of the technique. However, as described in Section 3.6, most compilers still do not provide an automated parameter selection, requiring users to select them manually. The most effective options are CHET [Dat+19] and EVA [Dat+20], which provide a relatively automatic parameter selection. However, their behavior uses naive approaches and, thus, is suboptimal. As a difference to our proposal, EVA requires the user to input the maximum precision for the integer scale. It chooses a default scale, uses it as many times as needed by the multiplication depth, and then sets the rest of the parameters based on the total $logQ$. In our case, we decide it through a user-defined value for precision.

Takeaway. Overall, a proper setup of FHE schemes such as CKKS [Che+17] highly depends on the optimal selection of parameters. Unfortunately, such optimal selection is complex since parameters depend on each other and the underlying circuit. Moreover, the selection seriously impacts security, efficiency, and precision. Balancing the tradeoff of these features depends on the scenario of the application. Existing FHE frameworks usually leave the selection of parameters to the user, which is time-consuming and requires expertise in cryptography. Thus, this chapter proposes an expert system that leverages Linear Programming and Fuzzy Logic to assist in parameter selection and overcomes these challenges.

5.1.3 Contribution

In this chapter, we propose an expert system that allows the selection of an optimal set of parameters for HE within user preference. Given an input circuit, our system leverages Linear Programming to select a close-to-optimal solution given a set of intrinsic rules to the HE scheme. The characteristics and constraints of HE parameter selection enable their representation as a Linear Programming model. Moreover, due to the potential parameter choices and their confronting impact on security, efficiency, and performance, the system allows users to guide the selection around these constraints. Concretely, the system uses Fuzzy Logic to select the best options based on the criteria defined by the user. Fuzzy Logic employs language variables to capture the intricacies of verbal language and model it into actual values that a computer can treat. We leverage Fuzzy Logic to couple the implications of parameter selections with the specific parameters that govern it.

5.2 Background

This section provides a theoretical background of the concepts tackled in this chapter to properly understand the complete presented expert system.

5.2.1 Linear Programming

Linear Programming (LP) is an optimization technique that obtains -if existent- the optimal solution to a model, defined by an objective function, the decision variables, and a set of constraints [Mur20]. Concretely, LP attempts to maximize or minimize the objective function. The objective function shows the overall benefit or cost of decision variables.

More formally, an LP task is defined as:

$$\begin{aligned} max\ z = c^T x \mid min\ z = c^T x \\ Ax \leq b \\ x \geq 0 \end{aligned} \tag{5.1}$$

where z is the objective function, x represents the decision variables, c are the coefficients, A are the constraints the rule the objective function and b is the vector of resources.

In LP tasks, the constraints define a region of all feasible solutions (i.e., the solutions to the objective function that fulfill the constraints). Optimal solutions usually lie in the extreme points of the feasible region. The Looseness Fuzzy Logic Module, described in the following section, generates coefficients that modify the constraints of the feasible region so that extreme points vary.

LP tasks are solved with methodologies widely falling under two categories, i.e., Simplex-based methods and Interior Point Methods. Simplex-based methods work on the polyhedron constructed with the decision variables and bounded by the constraints [BG69]. In Interior Point Methods, the algorithms work with the feasible region and travel the extreme points to find the optimal solutions [PW00]. Although a priori the methods have different computational complexities, their practical efficiencies are similar nowadays.

5.2.2 Fuzzy Logic

Fuzzy Logic (FL) is a control system technique aimed at systems working with a degree of uncertainty. In particular, FL profits from language rules and uses them to express control system values. The benefit of FL is the ability to represent a value within a threshold. In FL, there exist two predominant solvers, Mandiani and Sugeno. For this proposal, we use Mandiani as it delivers more detailed explanations of the decision variable relations.

During the fuzzy inference process, a set of crisp antecedents (i.e., crisp values we know as truth) are transformed into fuzzy sets to relate them logically and obtain a crisp consequent. The complete process is the following: first, the antecedents are *fuzzified*. The *fuzzification* analyzes the degree of membership of the crisp value to membership functions. In the second phase, the fuzzy inference process relates the different fuzzy values (i.e., belonging to the membership functions) through boolean logic statements. The result of the fuzzy inference process is a fuzzy consequent, which is *deffuzified* with some predefined metric (i.e., usually the area of the centroid) to obtain the final value.

5.2.3 Leveled Homomorphic Encryption Parametrization

An RLWE-based HE scheme allows encrypting a message μ using a secret s in a ring polynomial of degree N with modulus Q (with $N, Q \in \mathbb{Z}^+$). It also requires setting a parameter σ for the standard deviation of a random distribution $\chi(\mu, \sigma)$. As we detail later, the parametrization consists of choosing the more suitable values for the tuple (N,Q, and σ).

In LHE schemes, two special operations, relinearization, and modulo switching, follow every multiplication operation. The multiplication of two polynomials of degree N increases the degree (e.g., $x^N * x^N = x^{2N}$). It introduces the need for exponentially more polynomials to obtain the solution and effectively grow the ciphertext size. The relinearization uses the encrypted secret s to reduce the degree homomorphically. Still, multiplications introduce a potentially non-negligible error. The modulo-switching operation aims to reduce the noise introduced with each multiplication. For this process, the polynomial modulus Q is defined according to the Chinese Remainder Theorem as the multiplication of $O_{\mathfrak{D}}$ smaller moduli q_i, i.e., $Q = \prod_{i=0}^{O_{\mathfrak{D}}} q_i$.

We note that, in practice, modulo switching exhibits a particular behavior in specific libraries. If the noise scale has not grown significantly, there is no need to rescale and perform modulo switching. In the same way, if the noise scale has increased substantially, the rescaling may be performed twice. In our experience, these cases only occur with small modulus and are very rare, so in this contribution, we assume only the case where each rescaling uses one modulo and is always performed.

The term $O_{\mathfrak{D}}$, which corresponds with the multiplication depth of the circuit, defines the minimum number of moduli q_i needed to perform modulo switching. For simplicity (and consistency with HE libraries), in the remainder, we use $logQ$, $logN$, and $logq_i$ to refer to the bit counts of these numbers (e.g., $logQ = \lceil log_2(Q) \rceil$).

In general, HE schemes aim at reducing the noise introduced by operations, so circuits allow more operations [Bra+13; MP13]. In this regard, the Homomorphic Encryption Security Standard [Alb+18] provides a set of guidelines for the secure selection of these parameters. In this chapter, we adhere to those guidelines and adopt them strictly in the design of the proposed system.

Assigning a parametrization (N, Q, σ) to a circuit C for its execution under LHE poses many challenges. In most cases, there is a unidirectional dependence of the parametrization (N, Q, σ) on the circuit C.

In such cases, the first step is to extract the circuit constraints, precision p, multiplication depth $O_{\mathfrak{D}}$, and maximum vector length $|v|_{max}$. The precision p is the number of bits needed to represent the integer part of the most significant floating point number in the computation $p = \lceil log_2(d_{max}) \rceil$. Note that it does not only affect input variables but also intermediate and resulting values. The multiplication depth $O_{\mathfrak{D}}$ is the maximum number of consecutive multiplications that occur over a ciphertext in the circuit C. The maximum vector length $|v|_{max}$ is the length of the most significant vector we aim to encode in a single ciphertext. Most HE libraries limit the slots to 2^{15} for CKKS, and thus it may be necessary to represent a vector in multiple ciphertexts. This step may introduce additional computation rounds and multiplication depth [CP22].

After obtaining these values (p,$O_{\mathcal{D}}$ and $|v|_{max}$), the second step is to select the parameters (N, Q, σ).

In most cases, to reduce the amount of noise, and according to the HE Standard, $\sigma \approx 3.2$. Increasing σ improves the security but reduces the precision during decryption. That is due to the introduction of more noise per operation.

Next, the chain of moduli q_i is selected, which includes at least $O_{\mathcal{D}}$ $logq_i$-bit prime numbers, and whose product defines the polynomial modulo Q. Finally, according to the guidelines of the Homomorphic Encryption Security Standard [Alb+18], each N allows for a maximum budget or amount of moduli bits, represented as $\mathcal{B}_{N,\lambda,\mathcal{T}}$. More formally, for each value of N, for a security type $\mathcal{T} \in \{classical, quantum\}$ and for a security parameter $\lambda \in \{128, 192, 256\}$ the standard defines a maximum budget that must be lower than the sum of the selected $logq_i$ ($\mathcal{B}_{N,\lambda,\mathcal{T}} >= \sum_{i=0}^{O_{\mathcal{D}}} logq_i$). Therefore, for a circuit parametrization to be secure, one would need to compute the sum of all $logq_i$ and then compare it to the budget $\mathcal{B}_{N,\lambda,\mathcal{T}}$ established to obtain a N.

At the same time, N shall compare with the maximum vector length $|v|_{max}$. If $\frac{N}{2} >= |v|_{max}$, then the largest vector fits within the ciphertext slots of the chosen parameter set. Furthermore, to ensure that the maximum precision does not overflow, the distance between q_0 (known as the *special prime*) and the rest of the moduli q_i has to be p following $logq_0 = logq_i + p \; \forall \; i > 0$. In practice, only $logq_1$ needs to preserve this property but to maintain the cost of rotations negligible, all $logq_i$ must share the same bit count.

However, the above reasoning misses two crucial factors in selecting parameters. First, suppose $|v|_{max}$ does not fit within the largest vector. In that case, the plaintext vector is represented by multiple smaller ciphertexts (i.e., involving additional multiplication depth for each rotation, thus modifying the circuit). Also, some algorithms rely on the parameter N for their execution. For example, vector aggregation algorithms (i.e., sums all the entries of a vector) perform $logN$ consecutive multiplications. Then, this behavior involves a different multiplication depth $O_{\mathcal{D}}$ per each possible N value (in this chapter, we denote this relation between the multiplication depth and N as $O_{\mathcal{D}}^{N}$). Using such algorithms requires an iterative parametrization process since the different choices of (N, Q, σ) may introduce changes in the circuit C, which in turn might require re-definitions of the parameters as represented by Figure 5.1. One potential solution is to keep fixed one or two of the parameters. However, this can bias the parametrization and make it less efficient.

The second factor not mentioned before is related to the impact of the parameter selection on the output. Indeed, each parameter selection involves a tradeoff between precision, security, and performance. For example, increasing N increases the security since a larger polynomial hides the secret, but it becomes less efficient since it requires operations over a higher degree polynomial. Similarly, higher values of Q allow for larger multiplication depth and better precision. However, the smaller the value Q, the more secure and efficient the encryption scheme is.

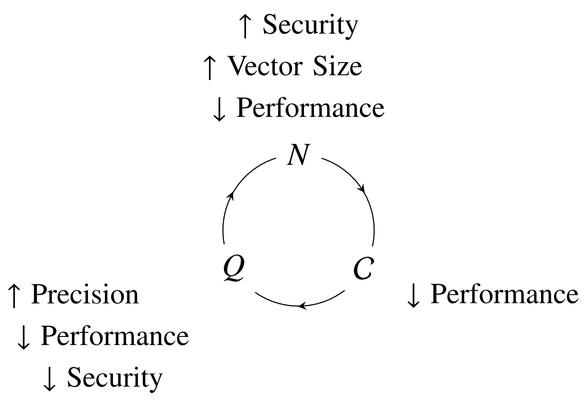


Figure 5.1: Circular dependencies in HE parametrization between polynomial degree N, polynomial modulus Q and circuit programming C. The diagram also depicts the implications of each parameter in the growing (↑) or decreasing (↓) of each variable.

5.3 System Model

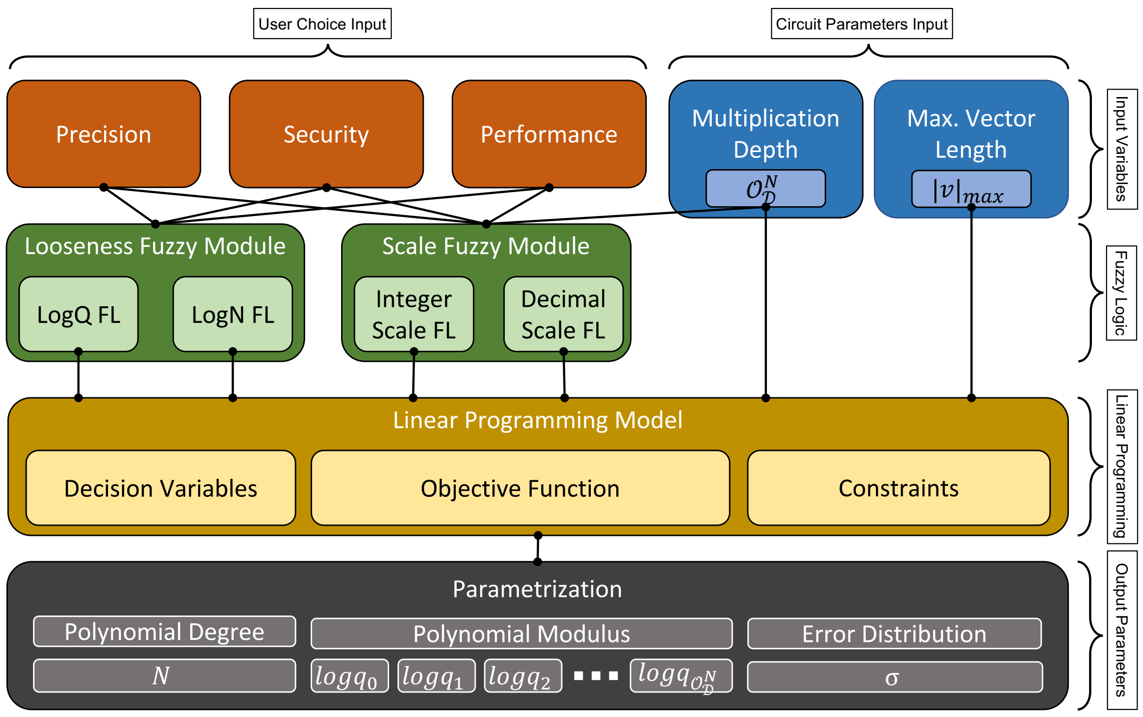


Figure 5.2: General architecture of the FL-based and LP-based system proposed.

In the previous section, we described the process for parameter setup in HE. We also discussed the difficulty of manually establishing appropriate values in the absence of expertise on the inner cryptographic workings of the scheme. Also, the complexity of the LWE scheme makes that even a human with expertise might fail to provide an optimal set of parameters due to circular dependencies among them and with the circuit being operated. Also, as discussed before, the selection of parameters has an essential influence on the performance, security, and precision of the scheme. These are confronting goals, and their setup depends on the application scenario (e.g., one would prefer a more secure scheme, even at the cost of losing precision or performance, and vice-versa).

Accordingly, we propose a system that automatically finds the optimal parameters and, to make the system more flexible, asks the user to choose a priority level for these confronting features.

The methodology is depicted in Figure 5.2. It receives as input the level of priority for each of the features mentioned above, i.e., security, precision, and performance, in a score from 0 to 10. We note, however, that stating the priority of security of 0 means putting *the lowest priority* on secure parameters (i.e., reducing N and increasing Q). However, in no way will the system generate an insecure set of parameters, as established by the HE Standard [Alb+18]. The system applies Fuzzy Logic, which permits making fuzzy decisions on parameters while establishing fuzzy rules on the input values. The output of the FL, together with information regarding the circuit (i.e., the multiplication depth and the maximum vector length), are used to define the Linear Programming (LP) model. The LP model allows for balancing the tradeoff of each of the value choices for the selected parameters. Concretely, it defines the LP constraints, decision variables, and objective function. The remainder of this section describes further the Fuzzy Logic and the Linear Programming processes.

5.3.1 Fuzzy Logic Initialization

This section elaborates on the two different Fuzzy Logic Modules used to generate the coefficients to design the Linear Programming model, i.e., the *Scale FL Module* and the *Looseness FL Module*. The Scale Fuzzy Module generates the maximum and minimum values of the polynomial modulus $logQ$, fulfilling the user input requirements regarding security, performance, and precision. Then, to make the LP task more flexible, the Looseness Fuzzy Module expands the interval of valid values for the LP task to adjust the final parameters in an unconstrained interval. The combination of these two models outputs the coefficients used to define the actual LP model values that later define the constraints and objective function used in Linear Programming. We next explain these two models.

Scale Fuzzy Module

The Scale Fuzzy Module produces two coefficients k_{int} and k_{dec} using two independent processes, i.e., the Integer Scale Fuzzy Logic and the Decimal Scale Fuzzy Logic. The LP tasks use these coefficients to determine the bit length of each polynomial moduli $logq_i$. On the one hand, the Decimal Scale represents the total bit length value of $logq_i$. On the other hand, the Integer Scale represents the value of precision p needed (i.e., the number of bits to represent the integer part of a decimal number).

The Integer and Decimal Scale FL consist of two consecutive Fuzzy Inference Process (FIP), Initial FIP and Final FIP. The initial FIP only considers the users' security, performance, and accuracy priorities. The final FIP refines the previous output by considering information about the circuit (i.e., the multiplication depth). Figure 5.3 depicts how the chaining of the different FIPs within each Fuzzy Logic occurs, detailed next.

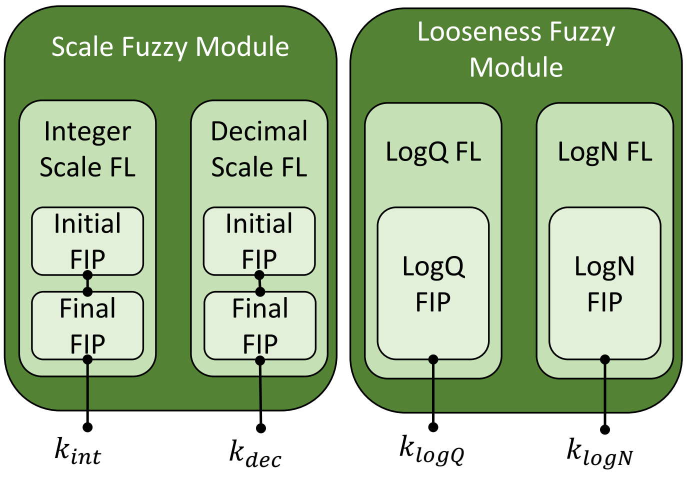


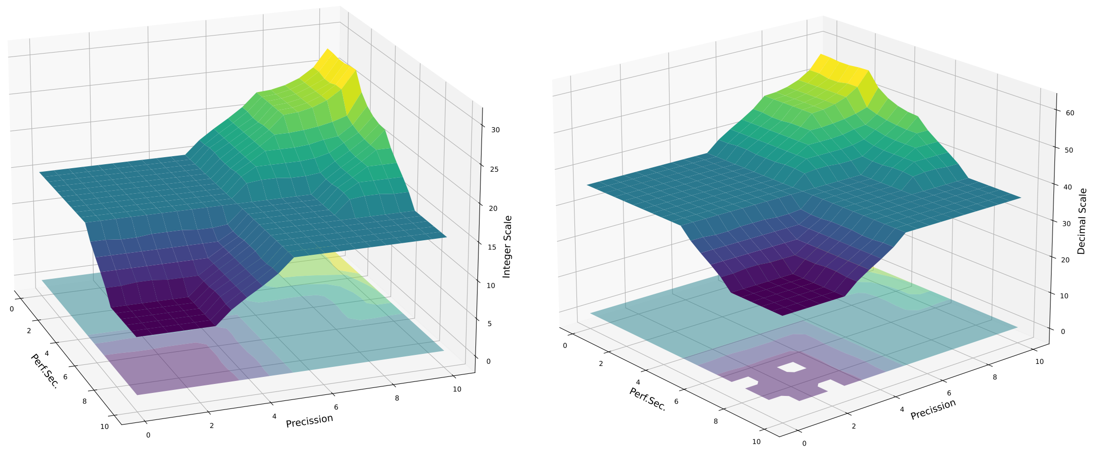
Figure 5.3: Overall hierarchization of the different Fuzzy Inference Process (FIP) carried in the Fuzzy Logic phase. For the Scale Fuzzy Module, there are two Fuzzy Logic Modules, each of which has two separate FIP (i.e., Initial FIP and Final FIP. In the case of the Looseness Fuzzy Module, there is two Fuzzy Logic with a single FIP.

The Integer and Decimal Scale FL share four antecedents and one consequent. The first three antecedents are the inputs from the user, i.e., the valuation (from 0 to 10) of the importance given to precision, performance, and security. The fourth antecedent is the *multiplication depth* $O^N_{\mathfrak{D}}$ that shall be obtained from the circuit. Each FIP produces one consequent (coefficient), i.e., the integer and decimal scale (k_{int} and k_{dec} respectively).

As mentioned before, for each of the FL processes, two consecutive FIPs are applied. These use five membership functions to describe the antecedents and consequent as *very low*, *low*, *medium*, *high*, and *very high*. For more details on the intervals for each membership function, we refer the reader to Table 5.1. Splitting the Fuzzy Logic into two separate FIP significantly reduces the number of rules $5^2 \ll 5^3$. Also, it produces a more straightforward understanding and relations of antecedents based on the boolean rules.

The first FIP for both Integer and Decimal Scale FL uses as antecedents the *precision*, *performance*, and *security* to compute an initial estimation of the Integer and Decimal Scale as consequents. We note that, for the computation of $logq_i$, *performance* and *security* are aligning goals. Thus we compute the maximum of these two to compete with the *precision*[3]. Figure 5.4 shows the overall Integer and Decimal Scale estimations as surface plots according to the user inputs. The surface plots show the variations in the resulting Fuzzy Logic value based on the modifications to the antecedent. The main variations observed on the plots among the Integer and Decimal Scales are in the resulting

[3]We use the max value since our preliminary experimentation showed that using other metrics, such as the average, leads to less consistent results with the user inputs.

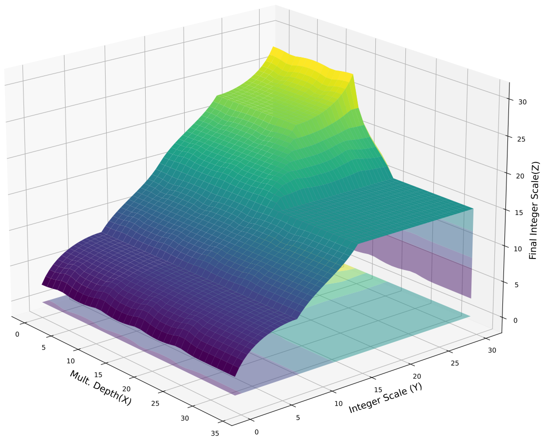


(a) Integer Scale (b) Decimal Scale

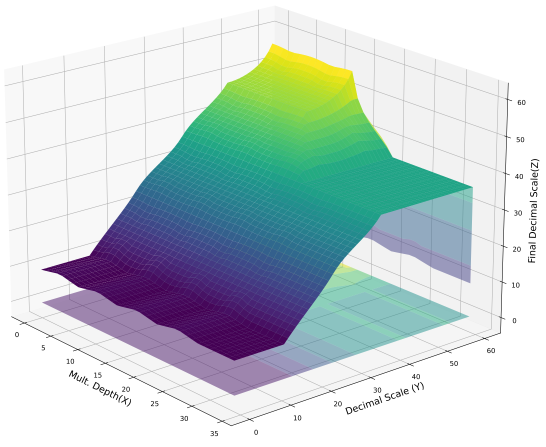
Figure 5.4: Initial Fuzzy Inference Process (FIP) for Integer and Decimal Scale estimation obtention based on the antecedents on the x and y axis (*precision*, *performance* and *security*), the surface plots determines the value of the consequents on the z axis. Given the aligning goals of *performance* and *security*, these are combined within the same metric. The surface plots show the variations in the resulting Fuzzy Logic value based on the modifications to the antecedent. This surface plot aims at having a slow decrease based on *high* and *very-high* precision values as well as avoiding very-low values of Integer and Decimal Scale (those are not beneficial for the precision of results). The main differences lay in the large precision values as the Decimal Scale is less affected by user parameters to prevent extreme inaccuracy due to Homomorphic Encryption noise.

ranges (i.e., the integer scale follows a shorter range than the decimal scale). The surface plot aims at having a slow decrease based on *high* and *very-high* precision values. It also avoids very-low values of the Integer and Decimal Scale, as those are not beneficial for the precision of results. The main differences lay in the large precision values since the Decimal Scale must be less affected by reduced precision preventing excessive precision loss due to HE noise. The graphs show how, for *medium* values of precision, performance, and security (i.e., between 4 and 6), the FIP averages the result obtaining medium values of scale (i.e., around 15 for the scale and 30 for the decimal coefficient). However, the resulting decimal scale values are furthest in contrary cases (e.g., high precision and low performance/security). We note that, in our implementation, we favor precision over performance and security as it may yield useless results otherwise. Therefore, we see a lower decrease in precision suffered by the surface plot but a more significant reduction in performance and precision.

The circuit restrains the permitted values for the integer and decimal scales despite the user's choices. For example, if the user chooses precision as its goal, the Integer and Decimal Scales (k_{int} and k_{dec}) will grow significantly. If the circuit also has a high multiplication depth, the maximum budget allowed within $N = 15$ & $\lambda = 128$ may need to fit more bits of $logq_i$ and thus result in a non-solvable LP task. Accordingly, the second FIP rationalizes the integer and decimal scale coefficients by either maintaining or reducing them based on the *multiplication depth* of the circuit. If the values of the multiplication depth are very big, the Integer and Decimal scales are reduced accordingly to levels that make the parametrization feasible. Figure 5.5 shows the surface plots of the results for the Integer and Decimal scales. The surface plots show the two-dimensional relationship between two input variable values (on the x and y axis) and the output variable (on the z axis). We can see how the coefficients are maintained from the previous FIP for low values of Multiplication Depth (i.e., simpler circuits). However, we observe a plateau for larger values of the estimated scales and multiplication depth where the final values are rationalized and reduced. In this way, the resulting coefficients fit within the maximum security budget. The main difference between the Integer and Decimal Scale processes is for *low* predicted scales. Wherein the Integer Scale tends to increase its final value even for predicted low values (so it does not decrement precision), the decimal scale is kept constant across the *very-low* values (i.e., from 0 to 20).



(a) Integer Scale



(b) Decimal Scale

Figure 5.5: Final Fuzzy Inference Process (FIP) for Integer and Decimal Scale coefficients obtention based on *initial Integer or Decimal Scale Estimation* and *Multiplication Depth*. The surface plot presents a readjustment of the Integer and Decimal Scales based on the multiplication depth (O_D^N), reducing it when large multiplication depths are present. The main difference relies on the low values where the Integer Scale is partially corrected from drastic reductions, while on the Decimal Scale, these are directly hard-coded. Plateau values of the surface plots are designed to enable feasible parametrizations when the Integer and Decimal Scales are coupled to large multiplication depths.

	Fuzzy Logic Membership Function Interval				
Var.	Very Low	Low	Medium	High	Very High
Precision	$[-\infty, 0, 2.5]$	$[0, 2.5, 5]$	$[2.5, 5, 7.5]$	$[5, 7.5, 10]$	$[7.5, 10, \infty]$
Performance	$[-\infty, 0, 2.5]$	$[0, 2.5, 5]$	$[2.5, 5, 7.5]$	$[5, 7.5, 10]$	$[7.5, 10, \infty]$
Security	$[-\infty, 0, 2.5]$	$[0, 2.5, 5]$	$[2.5, 5, 7.5]$	$[5, 7.5, 10]$	$[7.5, 10, \infty]$
$O_{\mathcal{D}}$	$[-\infty, 0, 7]$	$[0, 7, 14]$	$[7, 14, 21]$	$[14, 21, 28]$	$[21, 28, \infty]$
k_{real}	$[-\infty, 0, 7.5]$	$[0, 7.5, 15]$	$[7.5, 15, 22.5]$	$[15, 22.5, 30]$	$[22.5, 30, \infty]$
k_{dec}	$[-\infty, 12, 24]$	$[12, 24, 36]$	$[24, 36, 48]$	$[36, 48, 60]$	$[48, 60, \infty]$
k_{logN}	$[-\infty, 0, 0.25]$	$[0, 0.25, 0.5]$	$[0.25, 0.5, 0.75]$	$[0.5, 0.75, 1.0]$	$[0.75, 1.0, \infty]$
k_{logQ}	$[-\infty, 0, 0.25]$	$[0, 0.25, 0.5]$	$[0.25, 0.5, 0.75]$	$[0.5, 0.75, 1.0]$	$[0.75, 1.0, \infty]$

Table 5.1: Fuzzy Logic Membership Function Intervals for the Triangular functions. Represented as [beginning, peak, end].

Looseness Fuzzy Logic

The Looseness FL provides margins that capture the need for flexibility of the LP task. While preserving user priorities, it permits a range where the parametrizations remain optimal. The intervals provide flexible room for the LP task to extract the more optimal values in terms of performance (i.e., choosing a more optimal *logN*) and precision (i.e., profiting from the *logQ* budget). It benefits from the properties of the centroid defuzzification (i.e., it never takes extreme values) to establish flexible intervals for the LP task to work.

This FL Module extracts two different coefficients, k_{logN} and k_{logQ}, that scale the polynomial degree N and polynomial modulus Q, respectively. Each of them uses a different FIP. The FIP for k_{logN} uses two antecedents, i.e., *performance* and *security*, since increasing N improves security but reduces performance. The FIP for k_{logQ} uses three antecedents, i.e., *precision*, *performance* and *security*. However, similar to the Scale FL, we take the maximum of *performance* and *security* as they align goals. The goal of these coefficients is to provide the Linear Programming task with guidelines on what to prioritize through the objective function as we depict in Section 5.3.2

We use five membership functions for the antecedent and consequent values *very low*, *low*, *medium*, *high*, and *very high*. Figure 5.6 represents the FIP that outputs the two coefficients k_{logN} and k_{logQ}. The surface plot shape achieved by this Fuzzy Logic allows us to prioritize medium or close-to-extreme values where we see plateaus. Anything that does not lie within the medium category generally shifts to it (i.e., we see the local maxima on those points).

As a result of the two Fuzzy Logic modules, the proposed system outputs four coefficients: $k_{int}, k_{dec}, k_{logN}$, and k_{logQ}. In the following sections, the variables are used to weigh and define various ranges of it. For reference, in Table 5.1, we show the different intervals used for membership functions.

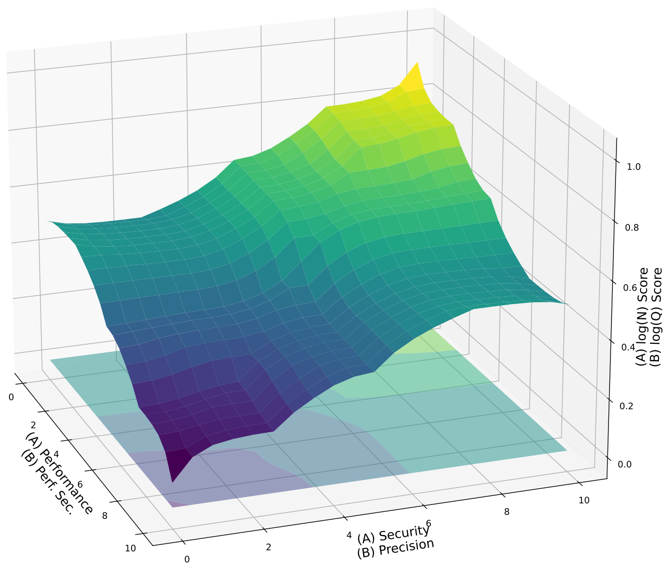


Figure 5.6: Fuzzy Logic used for estimating coefficients to weight the polynomial degree N and polynomial modulus Q (k_{logN} and k_{logN}). With (A), we represent the antecedents and consequent of the FIP to obtain k_{logN}. With (B), we represent the antecedents and consequents of k_{logQ}. With the selected k_{logN}, k_{logQ}, the objective function is parametrized to prefer specific values which match the user-defined characteristics (i.e., *performance*, *security* or *precision*). As these parameters share conflicting goals, the user-provided values serve to find meet-in-the-middle scores that parameterize the Linear Programming task ranging in $(0, 1.0)$.

5.3.2 Linear Programming Tasks

LP solvers intend to provide an optimal solution for a problem defined in an LP model. In our case, the problem is to generate a valid parametrization for N and Q. [4]. The LP model receives inputs from the FL modules, some of which are used to define Global Parameters, which are presented first. Then, we describe the different characteristics of the LP model, i.e., the decision variables, the objective function, and the constraints.

Global Parameters

As a result of the two Fuzzy Logic modules, the system outputs four coefficients $k_{int}, k_{dec}, k_{logN}$, and k_{logQ}. Using these coefficients requires some transformations to integrate into the LP model. Concretely, we define global parameters for the LP model, which represent the range of allowed values for $logQ$ and the precision p (i.e., the inte-

[4]As described in Section 5.2.3, typically, the value of σ is fixed to 3.2 due to the reduced increase in noise of that distribution. As such, we do not include this parameter in the selection.

ger scale). The system performs it according to the maximum ($max(O^N_{\mathcal{D}})$) and minimum ($min(O^N_{\mathcal{D}})$ possible multiplication depths. Concretely, the initial interval is described as follows:

$$\begin{aligned}
&[logQ_{min} = k_{dec}^{min(O^N_{\mathcal{D}})}, logQ_{max} = k_{dec}^{max(O^N_{\mathcal{D}})}] \\
&[p_{min} = k_{int}^{min(O^N_{\mathcal{D}})}, p_{max} = k_{int}^{max(O^N_{\mathcal{D}})}]
\end{aligned} \tag{5.2}$$

Limiting the range to $[logQ_{min}, logQ_{max}]$ turns into a very static model, where the LP solver has little room to optimize the parameters. As such, and based on the precision-performance-security tradeoff, we extend the interval by a factor of k_{logQ} such that:

$$\begin{aligned}
logQ^{ext}_{min} &= logQ_{min} + logQ_{min} \cdot k_{logQ} \\
logQ^{ext}_{max} &= logQ_{max} + logQ_{max} \cdot k_{logQ}
\end{aligned} \tag{5.3}$$

Decision Variables

The LP task uses five sets of variables. Some of these are auxiliary because they do not have a predefined term in the objective function. The solvers set the values for these auxiliary variables by relating them with other variables in the constraints. We model the problem with auxiliary decision variables since LP tasks cannot express quadratic relations between variables. We can model the connection between a variable in the objective function and the auxiliary variable in the constraints with the auxiliary variables.

$b^{choice}_{N,\lambda,\mathcal{T}}$ are a set of boolean decision variables that indicate the election of a given N. For a given N, two additional parameters determine the maximum budget for the Q bit length ($\mathcal{B}_{N,\lambda,\mathcal{T}}$). These parameters are, i) the type of security ($\mathcal{T} \in \{classical, quantum\}$) and, ii) the security parameter ($\lambda \in \{128, 192, 256\}$). Thus, for the model to choose any option, we generate a boolean variable for each combination of a given N under a type of security $\mathcal{T}$ and security parameter λ, which indicates the election, or not, of that $b^{choice}_{N,\lambda,\mathcal{T}}$ set. More formally, the set of variables is defined as $b^{choice}_{N,\lambda,\mathcal{T}} \in \mathbb{Z}_2 \forall N, \lambda, \mathcal{T}$.

z_{logq_i} are integer auxiliary decision variables that determine the bit length of each element in the polynomial moduli chain. Since there are different $O^N_{\mathcal{D}}$ for each N, we create as many as the maximum $max(O^N_{\mathcal{D}})$ of decision variables. Then, our LP model and other variables ensure the use of the minimum required number of $logq_i$ and that this number coincides with the particular multiplication depth $O^N_{\mathcal{D}}$ for that N.

$b^{set}_{logq_i}$ are a set of boolean variables used to define whether the variable z_{logq_i} has a value or not. This allows us to properly account for z_{logq_i} during the calculation of the specific budget $\mathcal{B}_{N,\lambda,\mathcal{T}}$ for a polynomial degree selection $b^{choice}_{N,\lambda,\mathcal{T}}$.

$b^{thr}_{logq_i}$ are a set of boolean variables used to define whether the corresponding z_{logq_i} has surpassed a certain threshold. We combine these variables with the global parameters obtained from the Looseness FL Module to enable flexibility of the parameter choice. Furthermore, it also avoids setting values far from the initial user choice.

z_p is an integer auxiliary decision variable defining the integer scale precision. The variable combines with z_{logq_i} so that at least a level of precision is guaranteed. Contrary to z_{logq_i}, with the integer precision, we want to maintain precision; thus, the intervals are narrower than in the decimal. The LP task chooses this value from the range $[p_{min}, p_{max}]$.

Objective Function

The objective function represents the metrics that need to be minimized or maximized. In this work, we define a cost for every decision variable, and the objective is to minimize it. As discussed in Section 5.2.3, there is no objective way to determine whether an HE parametrization is more suitable before execution. We instead compare them in terms of *performance*, *precision*, and *security*. Due to that, we perform a multistep quantification of each decision variable. First, we establish the metrics used to compare the different parameter selections. Second, we normalize the metrics in the $[0, 1]$ range and combine them by aggregate multiplying all in a final metric. The resulting metric scores the decision variables in the objective function. Overall, N and Q share an equivalent weight in the minimization of the objective function.

Polynomial Degree N Metrics. To properly account for the impact of the polynomial degree, we define seven different metrics. Table 5.2 shows the definition and description of these metrics. To unify the impact of a metric m, we normalize every metric in the $[0, 1]$ range with $C_{m'} = \dfrac{C_m - min(C_m)}{max(C_m) - min(C_m)}$. Finally, all metrics are combined with the multiplication and renormalized as in $C_\Pi = C_{\lambda'} \cdot C_{N'} \cdot C_{O'_{add}} \cdot C_{O'_{mul}} \cdot C_{O'_{rot}} \cdot C_{\mathcal{T}'} \cdot C_{|v|'_{max}}$. With this processing, we achieve uniformity on the impact of every metric over $b^{choice}_{N,\lambda,\mathcal{T}}$. In that way, the objective function takes a uniform and equal representation of the different possible choices, taking a value of 0 for the best and 1 for the worst.

Polynomial Modulus Q Metrics. To loosen the Linear Programming task when choosing the values for Q, we introduce only the boolean variables $b^{set}_{logq_i}$ and $b^{thr}_{logq_i}$ in the objective function. The $b^{set}_{logq_i}$ variables penalize the objective function with each use, promoting the use of less z_{logq_i} whenever possible. We ensure that the amount of $b^{set}_{logq_i}$ variables needed are chosen with the constraints. Also, with the constraints we link z_{logq_i} decision variables with $b^{set}_{logq_i}$ and $b^{thr}_{logq_i}$. In that way, we force z_{logq_i} to take a value within $[logQ_{min}, logQ^{ext}_{max}]$. Also, depending on the value of k_{logQ} we use the decision variable $b^{thr}_{logq_i}$ to reward (i.e., $k_{logQ} \leq 0.5$) or penalize (i.e., $k_{logQ} > 0.5$) the variable z_{logq_i} if it is set above the range $logQ^{ext}_{min}$. In such a way, we promote choosing higher values of z_{logq_i} when precision is the goal and lower values of z_{logq_i} when performance or security are the goals. In all cases, the weight of $b^{set}_{logq_i}$ and $b^{thr}_{logq_i}$ in the objective function is $\dfrac{1}{max(O^N_{\mathcal{D}})}$.

Polynomial Degree N Metrics			
Metric	Definition	Formula	
C_λ	Establishes the security guarantees of the current N.	$1 - \frac{\lambda_i - \lambda_{min}}{\lambda_{max} - \lambda_{min}}$	(5.4)
C_N	Establishes the tradeoff between performance and security obtained by choosing a particular N considering k_{logN}.	$a = \frac{N - N_{min}}{N_{max} - N_{min}}$ $(1 - k_{logN}) \cdot a + (1 - a) \cdot k_{logN}$	(5.5)
$C_{O_{add}}$	Defines the cost of performing additions given the specific circuit and multiplication depth $O^N_{\mathfrak{D}}$	$\frac{O_{add} - O^{min}_{add}}{O^{max}_{add} - O^{min}_{add}}$	(5.6)
$C_{O_{mul}}$	Defines the cost of performing multiplications given the specific circuit and multiplication depth $O^N_{\mathfrak{D}}$	$\frac{O_{mul} - O^{min}_{mul}}{O^{max}_{mul} - O^{min}_{mul}}$	(5.7)
$C_{O_{rot}}$	Defines the cost of performing rotations given the specific circuit and multiplication depth $O^N_{\mathfrak{D}}$	$\frac{O_{rot} - O^{min}_{rot}}{O^{max}_{rot} - O^{min}_{rot}}$	(5.8)
$C_{\mathcal{T}}$	Equalizes the cost of choosing a quantum-safe and a classical parametrization.	$\frac{\mathcal{B}_{N,\lambda,\mathcal{T}}}{\mathcal{B}_{N,\lambda,\mathcal{T}=quantum}}$	(5.9)
$C_{\lvert v\rvert_{max}}$	Measures the number of ciphertext vectors needed to represent the given maximum length vector $\lvert v\rvert_{max}$ under the current degree N.	$\lceil \frac{N}{2 \cdot \lvert v\rvert_{max}} \rceil$	(5.10)

Table 5.2: Metrics used in Linear Programming to evaluate the suitability of a polynomial degree $b^{choice}_{N,\lambda,\mathcal{T}}$ according to the different areas of influence it has. For each metric, a value from 0 to 1 is extracted, with one being the worst and zero being the best. The different values are always weighted from 0 to 1 before being multiplied in a final metric C_Π.

Constraints

Finally, after defining the decision variables and objective function, we introduce the constraints that guarantee to fulfill the requirements established in the HE Standard [Alb+18] Again, we note that the constraints ensure that the resulting parametrization of the Linear Programming task is always within the needed security standards. Although we do not note them as explicit constraints, all decision variables must take a positive value in Linear Programming. Also, we remind that z_{logq_i} and z_p are integer variables and $b^{choice}_{N,\lambda,\mathcal{T}}$, $b^{set}_{logq_i}$ and $b^{thr}_{logq_i}$ are boolean variables.

Variable Ranges. We set constraints that define the ranges for each variable. In this case, we set the variable z_{logq_i} to lie in the range $[0, 60]$. Also, the variable z_p is constrained in the interval $[p_{min}, p_{max}]$ so that the value of z_p is congruent with the precision established in the FL modules.

Choose one $N^{N,\lambda,\mathcal{T}}_{choice}$. This constraint guarantees that a single choice is made for N, λ, and $\mathcal{T}$. In that way, the LP task is forced to set to one a single $b^{choice}_{N,\lambda,\mathcal{T}}$ and the rest of $b^{choice}_{N,\lambda,\mathcal{T}}$ to 0 such as:

$$\sum_{i}^{N}\sum_{j}^{\lambda}\sum_{k}^{\mathcal{T}} b^{choice}_{i,j,k} = 1 \tag{5.11}$$

Adjust to the budget $\mathcal{B}_{N,\lambda,\mathcal{T}}$. The constraint guarantees that, for each $b^{choice}_{N,\lambda,\mathcal{T}}$, we do not surpass the associated budget with the different variables z_{logq_i}.

$$\sum_{i}^{N}\sum_{j}^{\lambda}\sum_{k}^{\mathcal{T}} b^{choice}_{i,j,k} \cdot \mathcal{B}_{i,j,k} = \sum_{i}^{O_{\mathcal{D}}} z_{logq_i} \tag{5.12}$$

Guarantee enough rescaling for the circuit. This constraint ensures the availability of enough moduli for each multiplication depth. It assures that all rescalings can take place after multiplication. More formally, we want to ensure that for an N, there are at least $O^N_{\mathcal{D}}$ moduli $b^{set}_{logq_i}$ for relinearization.

$$\sum_{i}^{N}\sum_{j}^{\lambda}\sum_{k}^{\mathcal{T}} b^{i,j,k}_{choice} \cdot O^i_{\mathcal{D}} = \sum_{i}^{O_{\mathcal{D}}} b^{set}_{logq_i} \tag{5.13}$$

Guarantee precision. This constraint ensures there are at least z_p bits of precision to represent the integer scale. We remind the reader that the previous constraint set the interval for minimum and maximum values for z_p. It performs the constraining by setting the bit distance z_p between the special prime z_{logq_0} and the rest of the moduli $(z_{logq_i} \mid 0 < i < O^N_{\mathcal{D}})$.

$$z_{logq_i} + p_{choice} \leq z_{logq_0} \forall i \in 0 < i < max(O^N_{\mathcal{D}}) \tag{5.14}$$

Precise encryption. This constraint guarantees that the encryption provides enough bits of precision. We set z_{logq_0} to be the same as $z_{logq_{O^N_{\mathfrak{D}}}}$.

$$z_{logq_0} = z_{logq_{O_N}} \tag{5.15}$$

Pair z_{logq_i} with $b^{set}_{logq_i}$ guarantees that if $b^{set}_{logq_i}$ is 1, then the related value of $logq_i$ has to be within the expected range defined by $[logQ_{min}, logQ^{ext}_{max}]$. Also, if the value of $b^{set}_{logq_i}$ is 0, then the value of z_{logq_i} is necessarily 0, thus not influencing all the previous constraint computations.

$$z_{logq_i} \geq b^{set}_{logq_i} \cdot logQ_{min} \forall i \in 0 < i < max(O^N_{\mathfrak{D}}) \tag{5.16}$$

$$z_{logq_i} \leq b^{set}_{logq_i} \cdot logQ^{ext}_{max} \forall i \in 0 < i < max(O^N_{\mathfrak{D}}) \tag{5.17}$$

Promote or discourage higher precision values. The LP task has certain flexibility to choose z_{logq_i} based on user choices. If the objective function is positive for $b^{thr}_{logq_i}$, then this constraint will try to preserve values in the range $[logQ_{min}, logQ^{ext}_{min}]$. That means it will look for performance and security rather than precision. Otherwise, if the objective function is negative for $b^{thr}_{logq_i}$ then, we look for values in $[logQ^{ext}_{min}, logQ^{ext}_{max}]$ though it is not enforced.. The constraint is formulated in any case as follows:

$$z_{logq_i} \geq b^{thr}_{logq_i} \cdot logQ^{b}_{min} \forall i \in 0 < i < max(O^N_{\mathfrak{D}}) \tag{5.18}$$

5.4 Evaluation

In this section, we evaluate the correctness of the parameter selection of the proposed system. First, we explain the experimental design conducted, and then we present the results.

5.4.1 Experimental design

The proposed expert system selects parameters based on i) the user choices for the priority of precision, performance, and security and ii) the processed circuit (i.e., multiplication depth and maximum vector length). The experimental design aims to provide insights into the correctness of the system. We create eleven tests that apply different computations and imply other aspects of HE operations. Concretely, the tests evaluate encryption and decryption, additions, multiplications, rotations, rescaling on various multiplication depths, and combinations of the previous results. Table 5.3 details the different tests executed.

To evaluate **precision**, we measure the noise introduced by HE operations. First, we use the expert system to extract parameters for different values for priorities of the precision. Second, we execute the benchmark tests in plaintext and using HE. Third, we

Benchmark Tests	
Test	Details
T1	**Encrpytion Decryption Test** performs the encryption and decryption of a packed ciphertext.
T2	**Addition Test** performs a large number of additions on packed ciphertexts.
T3	**Multiplication Test** performs a large number of multiplications on packed ciphertexts with multiplication depth $O^N_{\mathcal{D}} = 1$.
T4	**Rotation Test** performs a large number of rotations on packed ciphertexts.
T5-T10	**Large Multiplication Depth Test** performs a set of tests with a large number of multiplications with increasing multiplication depths (i.e., where the test number TX determines the multiplication depth). For example, in T5 the multiplication depth is $O^N_{\mathcal{D}} = 5$ or in T10 the multiplication depth is $O^N_{\mathcal{D}} = 10$.
T11	**Matrix Multiplication Test** performs a large number of matrix multiplications between a packed vector and a cleartext matrix which is vectorized according to the diagonal matrix multiplication [CP22].

Table 5.3: Benchmark tests performed to evaluate the different parameter selections of the expert system. The different algorithms are executed for each user choice value.

compare the expected plaintext result with the HE result to measure the error incurred (if any). In order to make the evaluation more accurate, we profit from packing, which allows us to introduce a vector of numbers per ciphertext operating over all of them in SIMD, making the evaluation more uniform. We refer the reader to Chapter 4 for more details on algorithm creation with packing.

To evaluate **performance**, we use the expert system to generate parameters for various values of priorities for the performance. Then, we execute the program measuring the encryption, processing, and decryption time. Note that, for consistency, we force the rescaling or level dropdown in the evaluation. The rationale is that if the set of chosen parameters introduces less noise, then rescaling may be unnecessary, introducing an inequality for the remainder of the test in the number of operations performed in each test and their cost.

To evaluate **security**, we assess the parameter sets for different values of priorities for the security parameter. As described in Section 5.2.3, as $logN$ increases, it provides more security, but as $logq_i$ increases, it reduces the security. Accordingly, we theoretically evaluated by analyzing the chosen parameters $logN$ and $logq_i$ and considering the security guidelines of the HE Standard [Alb+18].

We conduct two different experiments with two types of parametrizations. First, we individually analyze each variable (performance, security, and accuracy) and show different values for one metric affecting the overall system. We fix the priority value for two metrics at their lowest (i.e., 0) and set the third to values ranging from 0 to 10. This allows us to observe the output for the different values for each variable when the others do not affect the result. The second experiment attempts to evaluate how the system behaves with conflicting values of the variables, using intermediate priorities. Concretely, we take 4 and 9 as priorities for each variable and run the benchmark tests on all the possible combinations for these two values. In this last test, we also include a manual parametrization that a non-expert user may carry out when trying to use Homomorphic Encryption to evaluate the effectiveness of our compiler.

Each test is executed ten times during the experimentation, and the average of the executions is reported. We run the tests on a machine with an AMD Ryzen 3950X and 32 GB of RAM running Ubuntu 20.04. The tests are run on Golang 1.16 and the HE framework Lattigo v2 [Mou+20].

5.4.2 Results

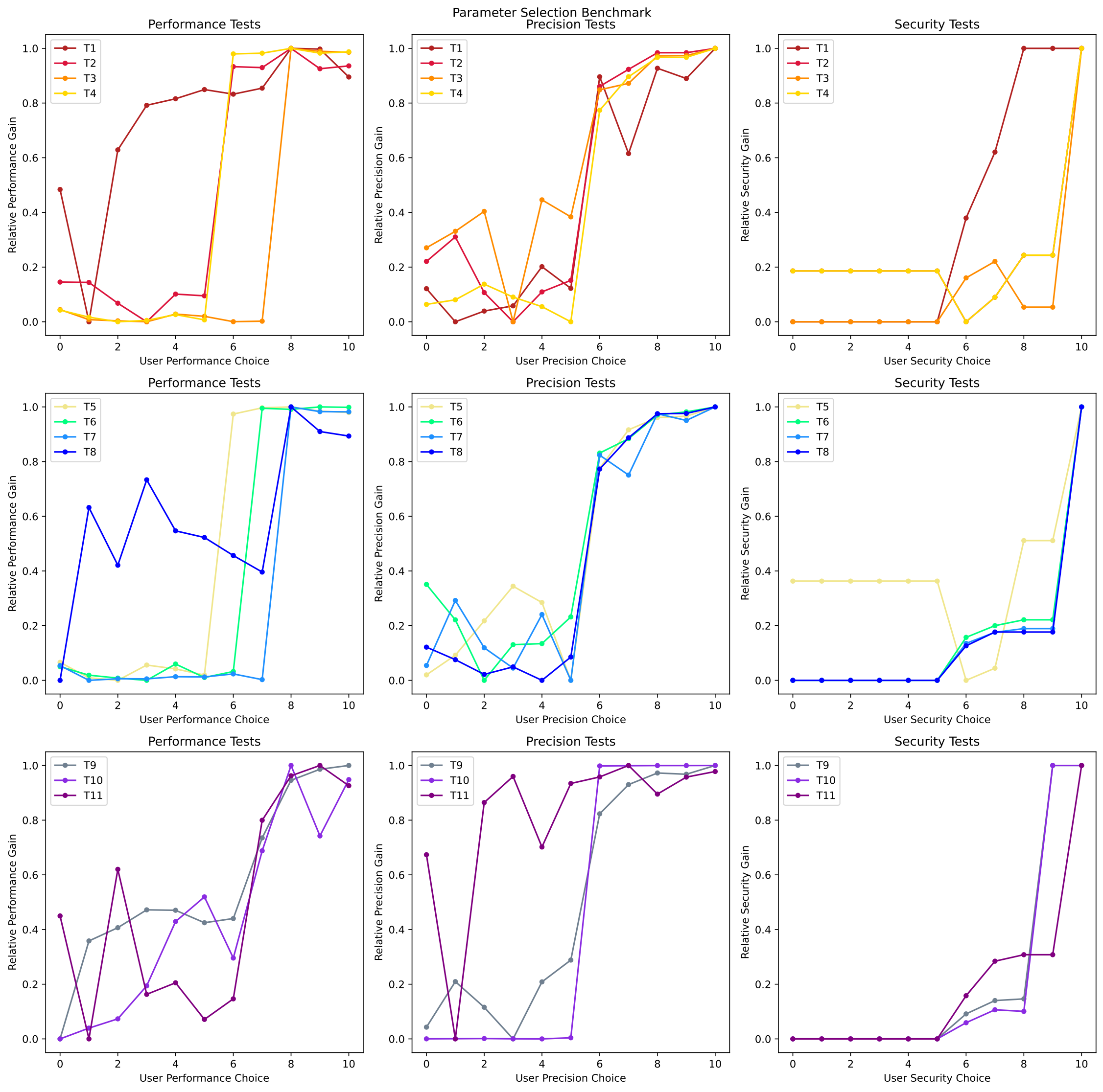


Figure 5.7: Combined representation of the results relatively aggregated. The figures show relative performance, precision, and security improvements from generating parameter selections with the expert system. In all these tests, we scale the figures so that a higher metric means a better result in all terms, performance, security, and performance.

We divide this section into the analysis of the two experiments used, one to evaluate each feature individually, using all possible priority values, and the other to test the different combinations of conflicting priorities.

Prec.	Perf.	Sec.	T1					T2					T3				
			Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale
4	4	4	1.49e-01	1.03e-03	10	40	20	4.39e-01	4.31e-01	10	91	20	1.32e+00	3.49e+00	10	110	20
4	4	9	5.32e-01	5.49e-03	12	93	21	8.94e-01	3.05e+00	13	114	21	3.42e+00	4.37e+01	13	136	21
4	9	4	5.41e-01	4.81e-03	12	82	19	8.10e-01	1.83e+00	12	100	19	3.85e+00	4.33e+01	13	120	19
4	9	9	5.83e-01	5.06e-03	12	82	19	8.71e-01	1.85e+00	12	100	19	3.47e+00	4.34e+01	13	120	19
9	4	4	3.52e-02	5.07e-03	12	82	19	1.14e-01	1.88e+00	12	100	19	1.25e-01	4.41e+01	13	120	19
9	4	9	1.33e-01	5.20e-03	12	109	23	5.48e-01	3.07e+00	13	132	23	1.09e+00	4.41e+01	13	160	24
9	9	4	1.49e-01	5.16e-03	12	93	21	3.96e-01	3.05e+00	13	114	21	9.71e-01	4.41e+01	13	136	21
9	9	9	1.26e-01	5.14e-03	12	93	21	4.05e-01	3.05e+00	13	114	21	8.73e-01	4.40e+01	13	136	21
Manual Selection			6.73e-02	5.39e-03	12	93	21	1.14e-01	3.05e+00	13	114	21	2.29e-01	4.40e+01	13	136	21
Prec.	Perf.	Sec.	T4					T5					T6				
			Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale
4	4	4	1.93e+00	7.38e+00	10	131	20	3.03e+02	1.08e+01	15	232	20	5.96e+02	1.46e+01	15	254	20
4	4	9	2.96e+00	4.46e+01	13	114	21	1.07e+04	5.40e+00	14	237	21	4.95e+03	7.23e+00	14	259	21
4	9	4	2.99e+00	2.11e+01	12	100	19	1.08e+04	2.58e+00	13	207	19	4.48e+03	7.12e+00	14	237	19
4	9	9	2.89e+00	2.12e+01	12	100	19	1.08e+04	2.56e+00	13	207	19	5.43e+03	7.20e+00	14	237	19
9	4	4	1.24e+00	2.13e+01	12	100	19	8.07e+01	2.61e+00	13	207	19	1.58e+02	7.24e+00	14	237	19
9	4	9	1.95e+00	4.50e+01	13	132	23	2.67e+02	5.48e+00	14	271	23	6.84e+02	7.39e+00	14	293	23
9	9	4	1.97e+00	4.50e+01	13	114	21	2.84e+02	5.48e+00	14	237	21	6.72e+02	7.34e+00	14	259	21
9	9	9	2.04e+00	4.50e+01	13	114	21	2.59e+02	5.46e+00	14	237	21	5.84e+02	7.37e+00	14	259	21
Manual Selection			2.99e+00	4.50e+01	13	114	21	3.91e+03	5.43e+00	14	237	21	1.98e+03	7.30e+00	14	259	21
Prec.	Perf.	Sec.	T7					T8					T9				
			Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale
4	4	4	1.41e+03	2.10e+01	15	276	20	3.00e+03	2.68e+01	15	298	20	6.85e+03	3.34e+01	15	321	20
4	4	9	8.05e+03	9.62e+00	14	278	21	1.45e+04	1.22e+01	14	300	21	1.05e+06	1.51e+01	14	317	21
4	9	4	2.15e+04	9.46e+00	14	258	19	7.48e+04	1.21e+01	14	280	19	1.05e+06	1.51e+01	14	302	19
4	9	9	5.17e+03	1.01e+01	14	258	19	1.20e+04	1.24e+01	14	280	19	1.05e+06	1.55e+01	14	302	19
9	4	4	1.63e+02	9.68e+00	14	258	19	3.29e+02	1.22e+01	14	280	19	8.98e+02	1.54e+01	14	302	19
9	4	9	1.39e+03	9.87e+00	14	328	24	3.08e+03	1.23e+01	14	352	24	8.24e+03	1.56e+01	14	376	24
9	9	4	1.39e+03	9.65e+00	14	278	21	3.04e+03	1.23e+01	14	300	21	7.42e+03	1.52e+01	14	317	21
9	9	9	1.29e+03	9.72e+00	14	278	21	3.45e+03	1.23e+01	14	300	21	6.32e+03	1.55e+01	14	317	21
Manual Selection			7.44e+03	9.60e+00	14	278	21	5.27e+05	1.23e+01	14	300	21	1.03e+06	1.52e+01	14	317	21
Prec.	Perf.	Sec.	T10					T11									
			Δ Res.	Time(s)	*logN*	*logQ*	Scale	Δ Res.	Time(s)	*logN*	*logQ*	Scale					
4	4	4	2.09e+06	4.11e+01	15	344	20	3.57e+01	3.14e+01	14	151	20					
4	4	9	2.10e+06	1.85e+01	14	333	21	6.03e+01	1.97e+01	13	177	21					
4	9	4	2.10e+06	1.84e+01	14	323	19	1.26e+02	1.97e+01	13	157	19					
4	9	9	2.10e+06	1.87e+01	14	323	19	1.43e+02	2.02e+01	13	157	19					
9	4	4	1.54e+03	1.89e+01	14	323	19	6.30e+00	2.03e+01	13	157	19					
9	4	9	2.09e+06	1.91e+01	14	401	24	6.58e+01	2.02e+01	13	208	24					
9	9	4	2.09e+06	1.88e+01	14	333	21	4.81e+01	2.01e+01	13	177	21					
9	9	9	2.09e+06	1.87e+01	14	333	21	7.46e+01	2.02e+01	13	177	21					
Manual Selection			8.19e+04	1.87e+01	14	333	21	1.76e+02	2.01e+01	13	177	21					

Benchmark Tests

Table 5.4: Numeric benchmarks for each test executed on precision, performance, and security values for all combinations of 4 and 9. The columns for each test show Δ Res. as the variation of the result with the HE noise, with the expected result (i.e., computed in cleartext); the total time in seconds (s) needed to execute the test and the security parameters used *logN*, *logQ* and scale.

The results for the individual tests are shown in Figure 5.7. Due to the nature of the tested circuits, we note that the parametrizations may not change significantly, and thus the implication of user choices in performance are more irregular. These occur because different values of the user choices may lead to equivalent parametrization and similar runtimes. Nevertheless, we observe a clear improvement in the performance gain for higher values of the user choice. For example, in Tests 5-11, which are the most complex from our benchmark set, the performance gain is reduced for low values (below 6) and then considerably increases for higher values.

In the case of precision, there is a clear improvement for values higher than 4 or 5 in all tests, which shows that the expert system successfully materializes the user choices. Furthermore, precision is a key factor in Homomorphic Encryption that the parametrization manages to land. A similar pattern can be observed with security, where in most of the tests, the improvement affects values higher than 5. However, in some circuits (i.e., Tests 3, 4, and 5), the increased security is smaller due to the requirements of the circuit. There is a particular behavior in Tests 3, 4, and 5 for values 6 and 7, where the security decreases regarding lower priorities. This is due to the performance-security tradeoff, where the LP tasks chose a lower value of *logQ*, and, in doing so, it is able to reduce the budget $\mathcal{B}_{N,\lambda,\mathcal{T}}$ thus fitting the parameters within a smaller *logN* (and in turn, improving performance). Although the security metric slightly decreases, it remains high according to the HE standard, and we do not consider it a worse parametrization in practice.

Table 5.4 shows the results for the experiment where we have run the test suite using different combinations of priorities for the three features. This experiment evaluates the behavior of the expert system when conflicting user choices appear. We observe that, as a general pattern, when there is no conflict (i.e., all features have the same value, either 4 or 9), the results show that the three features equally improve, obtaining equivalent output. It is mainly due to the Fuzzy Logic membership functions, logic rules, and defuzzification that average and smooths the changes avoiding any decision that has a more significant impact over the others. As long as the choices are not diverse, the overall optimization of parameters is shared for the three features. Indeed, when precision is the most crucial feature (with a value of 9 versus 4 for the other two variables), the *logQ* and *scale* increase, and the error in the result is considerably minimized in all tests. As for security, we observe that for higher values of security choice, the polynomial degree *logN* is kept high, and the polynomial modulus *logQ* is minimized. The best performance value is shared among parametrizations with the lowest *logN* and *logQ*.

An especially meaningful insight arises from the comparison of T5 to T10. When performance is the goal, we observe how there is a linear relation between the number of moduli and the runtime. As the multiplication depth increases, the runtime also increases. This comes from the need to operate over more moduli per operation performed. Regarding precision, noise also has an apparent influence on operations. As the multiplication depth increases, given the scales, the noise propagation increases more significantly in T10 than in T5 with $8.07 \cdot 10^1 \ll 1.54 \cdot 10^3$.

Finally, compared to the non-expert handpicked parameterizations, we observe how the parametrization achieves mixed goals. In most cases, we remark a shift in the achieved goals. This apparent difference comes from the need to perform manual circuit analysis and obtain a biased result for any characteristics. Furthermore, we observe how, due to the selection of the prime moduli (*logqi*), sometimes, the runtimes are affected significantly (T9). In general, the multi-faced evaluation of the expert system's parameters enables the automated assessment of potential consequences and minimizes those.

5.4.3 Discussion

From the results of our first experiment, we observe that the improvement in each of the variables increases with higher priority values when these are set independently (i.e., the other two variables are not considered). When different values are combined and conflict (experiment 2), the expert system behaves as expected, and the result often favors the variable with higher priority. However, we observe that in some cases, the different combination of priorities results in the same set of parameters and in similar accuracy, performance, and security. This is expected since the system is designed to guarantee a minimum level of security as defined in the HE Standard. Overall, we have shown that the expert system helps to provide optimal values for the parameters, balancing security, performance, and accuracy. Indeed, the system outputs a set of parameters congruent with what an expert initially seeks. These choices are performed from the heuristics and expert knowledge implemented within the system. Furthermore, it requires minimal effort from the user to achieve goals by setting high-level parameters. Otherwise, the user must manually understand each choice's constraints and considerations.

5.5 Summary

Homomorphic Encryption (HE) is not widespread due to limitations in terms of efficiency and usability. Among the challenges of HE, scheme parametrization (i.e., selecting appropriate parameters within the algorithms) is a relevant multi-faced problem. First, the parametrization needs to comply with a set of properties to guarantee the security of the underlying scheme. Second, parametrization requires a deep understanding of the low-level primitives since the parameters have a confronting impact on the scheme's precision, performance, and security. Finally, circular relations exist between the circuit to be executed and the parametrization. Thus, there is no general rule for the optimal selection of parameters, and this selection depends on the circuit and the scenario of the application. Currently, most existing HE frameworks require cryptographers to address these considerations manually. It requires a minimum of expertise acquired through a steep learning curve. This chapter proposes a unified solution for the previously mentioned challenges. Concretely, we present an expert system combining Fuzzy Logic and Linear Programming. The Fuzzy Logic Modules receive a user selection of high-level priorities for the cryptosystem's security, efficiency, and performance. Based on these preferences, the expert system generates a Linear Programming Model that obtains optimal combinations of parameters by considering those priorities while preserving a minimum required level of security for the cryptosystem. We conduct an extended evaluation showing that an expert system generates optimal parameter selections that maintain user preferences without undergoing the inherent complexity of analyzing the circuit.

Chapter 6

HEFactory: A Symbolic Execution Compiler for Privacy-Preserving Deep Learning with Homomorphic Encryption

In Chapter 4, we addressed the vectorization of algorithms in Packed Homomorphic Encryption for Deep Learning, which showed the difficulty of producing such operations and provided guidelines to improve the use of this. Then, in Chapter 5, we described the parametrization process of LWE-based schemes, together with a tool to automate this process. To bring these scientific improvements to the community, the logical follow-up for such contributions is to design and implement a tool that simplifies the production of Homomorphic Encryption code in the form of a compiler. In this chapter, we describe *HEFactory*, a multi-layer framework that introduces several improvements to usability and automation, easing access to interdisciplinary data scientists and enabling DL inference on sensitive data while preserving privacy.

The remainder of this chapter is distributed as follows. First, we provide the motivation and introduction of the work in Section 6.1. Section 6.2 describes the flexible and extensible system model of *HEFactory*. Then, Section 6.3 describes a multi-layered architecture where each layer provides support for the upper layers and relies on functions from the inner layers. Then, in Section 6.4, we evaluate the performance and code simplicity of *HEFactory* for general and Deep Learning applications. Finally, in Section 6.5, we summarize the main contributions of this chapter.

6.1 Introduction

Many advanced techniques in the field of Homomorphic Encryption (HE) can be complex for non-experts to understand and use. While basic libraries have been available for a long time, using them requires specialized knowledge and effort. Elaborating efficient algorithms for Deep Learning (DL) on top of these is challenging and often needs interoperability with classic algorithms. This section outlines the main approaches to address this issue and its limitations.

6.1.1 Precedents

Most modern HE techniques present technical barriers to researchers and practitioners in other areas of knowledge. While low-level libraries have remained available since very early in the field, using such libraries requires practical expertise and effort. Due to the considerable domain expertise and knowledge needed to use Homomorphic Encryption (HE) efficiently, recent works provide abstractions and simplifications to these techniques for non-practitioners to use them [Aha+23; Dat+19; Dat+20; HS14b; Mou+20; Res20; Via+23].

However, existing efforts to solve this problem either provide partial solutions to the problem, e.g., not offering direct integration for Deep Learning [Dat+19; Dat+20], not offering vectorized support and algorithms [Boe+19b], not providing high-level language support [Gil+16; HTG17; JVC18; Mou+20] or the reliance of circuit analysis for parametrization [Aha+23]. Therefore, these barriers restrict its access to data science from many disciplines, which might lack the expertise to work with languages such as C or C++ [Chi+20b; HS14b; Res20]. For a summary comparison of the features of the different frameworks, compared to *HEFactory*, we refer the reader to Table 6.1 and Section 3.6.

6.1.2 Motivation

High-level programming languages like Python or R are equipped with production-ready data science libraries [Van16]. However, the software libraries and frameworks supporting HE use efficient languages, such as C++ [Chi+20b; HS14b; Res20] or Go [Mou+20]. Thus, adapting data-science libraries is a complex task due to the numerous changes required for compatibility with Deep Learning and the particular challenges of HE. In general, deploying such scenarios requires subsequent development efforts to bridge the gap between the different programming languages and can be affected by reiterations in design.

Providing easy-to-use Privacy-Preserving Deep Learning with HE is a key require-

	nGraph	EVA	Concrete	HELayers	*HEFactory*
Vectorized Deep Learning	✓	✗	✗	✓	✓
Symbolic Execution	✗	✓	✗	✓	✓
Graph Execution	✓	✗	✗	✗	✗
Floating-Point CKKS	✓	✓	✗	✓	✓
Boolean TFHE	✗	✗	✓	✗	✗
Automated Parameter Selection	✗	■	✓	■	✓
Automated Ciphertext Vector Batching	✗	✗	✗	✗	✓
Tiling	✗	✗	✗	✓	✗
Optimization	✗	✓	✓	✓	✓
DL Frameworks Support	✓	✗	✓	✗	✓
✓: Provided | ✗: Not Provided | ■: Semi-Automatic					

Table 6.1: Framework Comparison table between *HEFactory* and other similar frameworks for Homomorphic Encryption.

ment for the spread of such solutions. Experts need to deliver non-experts frameworks that generate simple, easy-to-use language without introducing security-specific concepts. These need to relieve the user from any code adaptation and allow fast iteration while providing an optimal and functional solution.

6.1.3 Contribution

In this chapter, we describe *HEFactory*, a multi-layer framework that introduces several improvements to usability and automation, easing access to interdisciplinary data scientists and enabling DL inference on sensitive data while preserving privacy. Furthermore, it features support for vectorized DL routines, automated parameter selection, automatic ciphertext vector batching, and DL framework support, which places it as a competitive alternative among the existing proposals.

HEFactory is composed by three main modules, dubbed *Dahut*, *Tapir* and *Boar*[5]. The main scientific contributions are:

1. We present *Dahut*, a high-level API layer for Python that allows adapting existing data science projects to private execution easily. This API makes the inner workings of HE transparent to the developers and thus avoids the need to learn how it works and how to make the correct decisions for its proper use.

2. We propose *Tapir*, a compilation toolchain that applies symbolic execution on the source code in two phases. It processes the high-level functions and decomposes

[5]These names were inspired by the animal family of Cingulata [CDS15], one of the first software-aided HE proposals described in Section 3.6.

them into low-level instructions for later homomorphic evaluation on the encrypted data. Two stages of compilation allow the introduction of parameter-dependent functions (i.e., procedures that depend on the parametrization of the HE scheme but also influence it) at an early stage. Among others, *Tapir* allows for automated parameter selection, automated ciphertext batching, and the provision of vectorized techniques.

3. Finally, we describe *Boar*, a virtual machine software that interprets the output of the *Tapir* compiler (using a custom Intermediate Representation Code (IR) format) leveraging a HE backend framework for cryptographic operations (key generation, encryption, and decryption), and the homomorphic circuit evaluation.

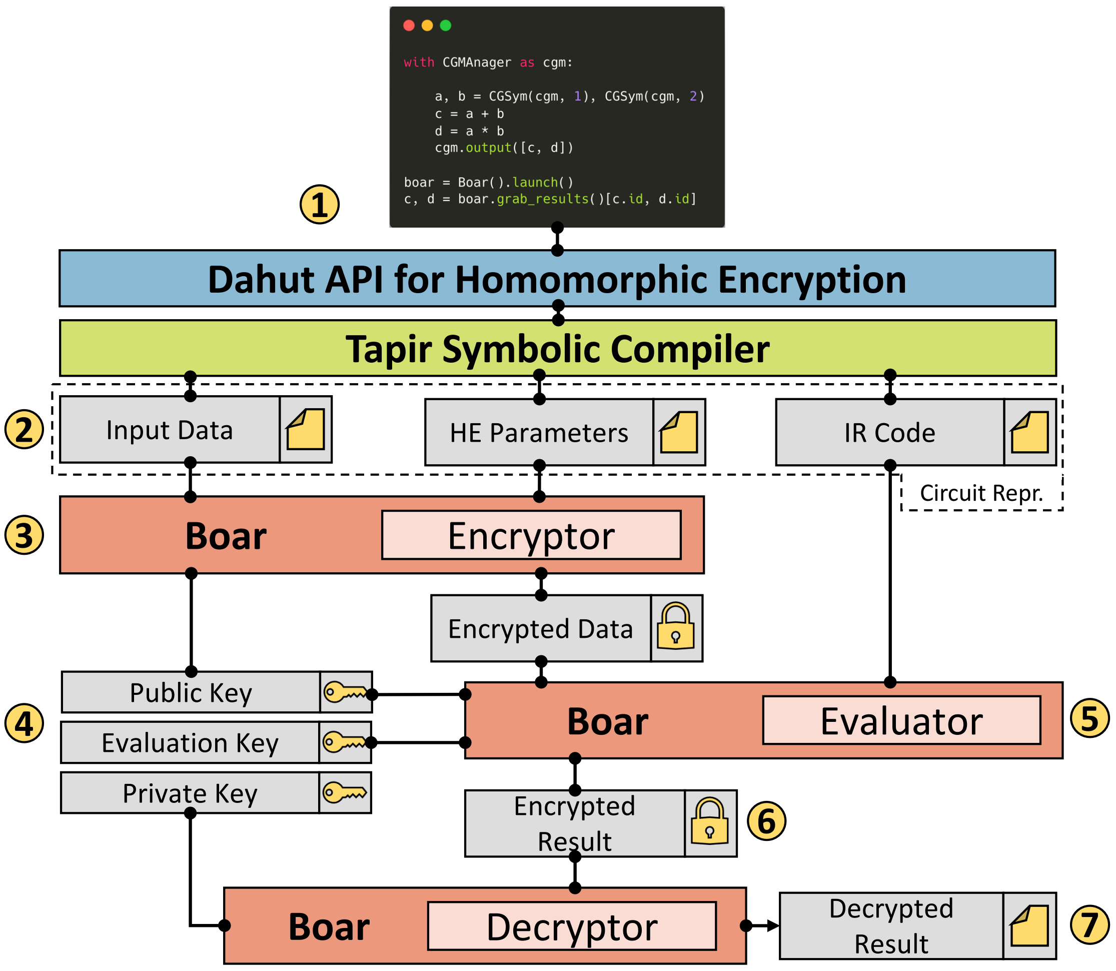
6.2 System Model

This section explains how *HEFactory* can be applied in a cloud scenario, i.e., to use Machine Learning as a Service (MLaaS). We consider two different actors, the Data Owner (DO) and the Data Processor (DP). The DO is the client willing to use cloud services for computation over sensitive data. At the same time, the Data Processor (DP) is the server hosting the high-computing infrastructure to carry out the computation. We currently assume that the Data Processor (DP) contains a pre-trained model on which it can conduct inference. However, *HEFactory* can be extended to support bootstrapping, potentially allowing training models over encrypted data.

Adversarial Model. We consider an Honest-But-Curious adversary model for the server. Thus, we assume that DP correctly executes the trained circuit on the given data without tampering with the input and output. However, it might attempt to access and learn the input and output data content, thus breaking its privacy. In our setting, we only focus on *input privacy* during inference, which preserves data privacy from an external non-trusted party [CP21b]. Thus, we do not consider model secrecy (i.e., the inner workings of the circuit are known both by the DP and the DO). Also, we do not cover output privacy regarding DL models, where the trained model might reveal internal information about the data used for training (e.g., Membership Inference Attacks [Sho+17]). Also, *HEFactory* enforces secure parametrizations according to the guidelines in the HE Standard [Alb+18].

Figure 6.1 represents the workflow of the process. First, the programmer writes code using the *Dahut* API, either for NumPy-like operations or to perform actual Deep Learning inference. This code is given as input to *Dahut* (Step 1). Depending on who is responsible for the programming task, we note that this step can be carried out entirely by DO (if the DO provides the code) or split between DO and DP (if the DP provides the code). In the latter case (i.e., when the actual circuit is programmed/designed by DP), the DO compiles the input version of the program to create a data file with custom format (see §6.3.2). The DP compiles the program into the IR code with placeholder data inputs that are later filled with the data provided by the DO.

As a result of the compilation, *Tapir* outputs the input data in plaintext with a specific format (only known by the DO), and also the HE params and IR code, which are of public knowledge for both the DO and the DP (Step 2). The client (DO) uses the *Boar* Encryptor to encrypt the input using specific HE params, resulting in the ciphertext and three keys: a public key, a private key, and an evaluation key (Step 3). These keys are then transmitted to the DP (Step 4), e.g., using a secure channel. DP then uses the *Boar* virtual machine to execute the circuit (IR code) over the encrypted data, resulting in the encrypted result (Step 5). This output is then transmitted back to the DO (Step 6), which can safely decrypt on its local premises using the *Boar* Decryptor and the private key (Step 7).


```
with CGMAnager as cgm:

    a, b = CGSym(cgm, 1), CGSym(cgm, 2)
    c = a + b
    d = a * b
    cgm.output([c, d])

boar = Boar().launch()
c, d = boar.grab_results()[c.id, d.id]
```

Figure 6.1: Workflow of *HEFactory* in an MLaaS scenario. *Tapir* generates the necessary outputs from the code to complete the execution. The *Boar* encryptor generates the keys and encrypted input from the data. Then the *Boar* evaluator performs the execution from the respective inputs. Finally, the Boar Decryptor extracts the decrypted result with the private key.

6.3 *HEFactory* Architecture

HEFactory uses a layered architecture, where upper layers transparently rely on the functionality provided by lower layers. In that way, these layers can assume specific abstractions concreted in lower layers. For example, any layer above the automatic parameter selection can conduct its functions without caring about the HE internal parameters. As such, since some expressions depend on these parameters (named Parameter-Dependent expressions), our framework provides a solution in the form of *Delayed Algorithms*, which are detailed in the following subsections. Figure 6.2 depicts the general architecture, which is composed of three main modules:

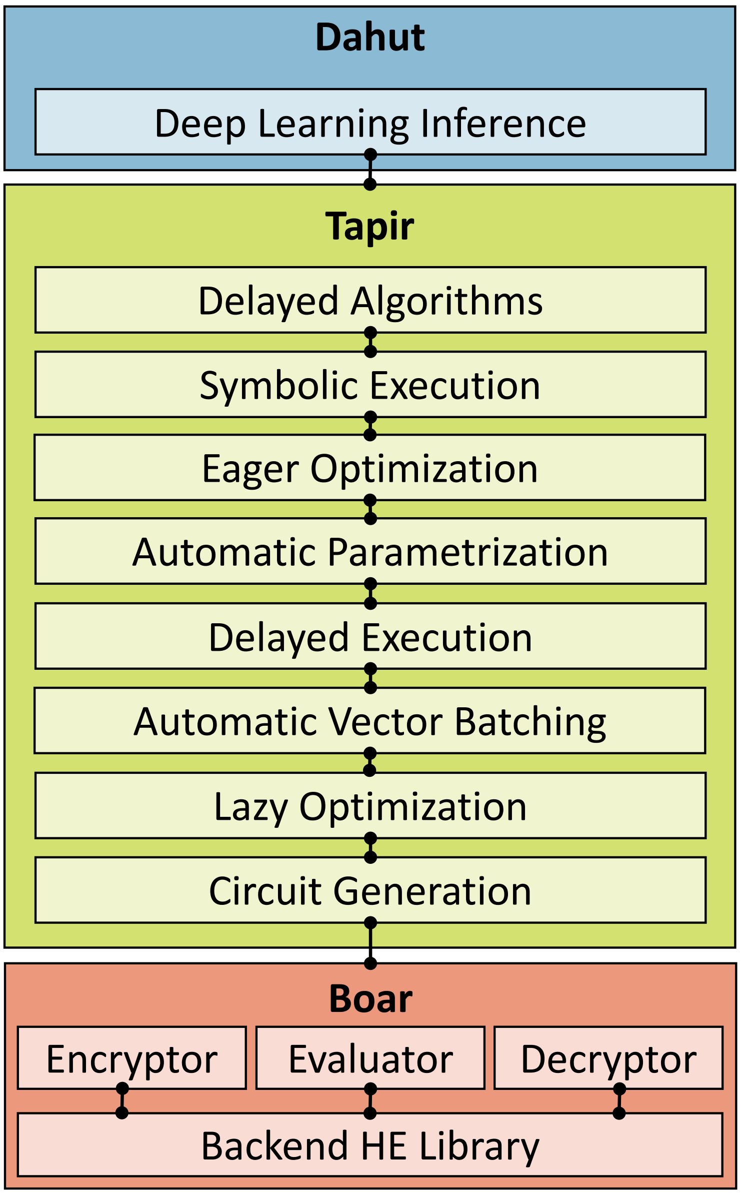


Figure 6.2: Multi-layered architecture of *HEFactory*

Dahut provides a high-level Python API that acts as a translation layer for data scientists. It encompasses two submodules. On the one hand, the code submodule allows writing high-level Python instructions to interact with NumPy (for simple arithmetic operations) and Tensorflow (for DL). *Dahut* code submodule modifies the original DL model circuit to use HE-friendly algorithms (i.e., models are decomposed into HE-supported operations). On the other hand, the data submodule defines new data types that can be used by the programmers in their code as if they were regular types (e.g., floating-point numbers and arrays). At a high level, *Dahut* allows code operations with these data types as if they were plaintext information but internally transforms (in *Tapir*) them to a format that allows for their encryption and privacy-preserving computation (in *Boar*).

Tapir is a symbolic compiler that takes as input the code and data obtained from *Dahut* and transforms it into a custom binary format that can be then interpreted for different HE frameworks in *Boar* (see Figure 6.2). In short, *Tapir* performs symbolic execution to transform the high-level circuit into simpler low-level operations. In the first phase, the compiler executes symbolic execution on the code and transforms it into internal expressions. Since some expressions (e.g., a matrix multiplication) depend on the HE parameters, *Tapir* initially marks these expressions as a *Delayed Algorithms*, so it can transform them later. These expressions are stored in internal structures (objects) that provide the required metadata for the parameter selection algorithm (e.g., the multiplication depth). Once the compiler obtains the circuit representation from the first phase, it extracts relevant information (e.g., the total number of multiplications or rotations required). It uses an expert system to optimize the parameter selection automatically. Then, the *Delayed Algorithms* are processed and symbolically executed, considering these parameters and obtaining a complete low-level representation of the circuit using a custom Intermediate Representation Code (IR) format. Finally, the compiler analyzes and compresses vectors in the circuit compliant with the HE parameters, optimizes the code, and outputs the circuit in a binary format.

Boar is a low-level virtual machine that acts as an interpreter for the binary code obtained in *Tapir*. Concretely, it conducts the actual encryption and decryption of the data (typically done by the DO) and allows the DP to run the circuit with HE operations on the encrypted data. It translates the IR code into low-level instructions from a given HE backend framework instructions (e.g., Lattigo). The IR code uses a standard format produced by *Tapir*, and that *Boar* can interpret. *HEFactory* implements serialization to divide data from code into different files, on which different access controls can be applied (e.g., to prevent the DP from accessing the data).

Both *Tapir* and *Boar* are transparent to the programmer, who only needs to interact with high-level subroutines using the *Dahut* high-level API. In that way, *HEFactory* provides a high-level NumPy-compatible interface and DL operations in a friendly Pythonic way, lowering the barrier to its use for data scientists. Next, we describe each of the modules in detail.

6.3.1 Dahut: A Python API for Homomorphic Encryption

Dahut is the uppermost layer of the *HEFactory* stack. It defines compatibility layers for two popular Python libraries used in data science projects, i.e., NumPy for simple algebra operations (e.g., multiplications of scalars or vectors) and TensorFlow for DL. Currently, it implements common layers for Convolutional Neural Network (CNN) [Gil+16].

Dahut defines two new data types for symbols (`CGSym`) and arrays (`CGArray`). These types are internally encrypted, and developers can use them as regular Python variables. Concretely, `CGSyms` are used for declaring single values (e.g., integer or floating point numbers), and `CGArrays` for declaring vectors and matrices. The `CGArray` is an abstract type designed to track shape changes on the matrices and vector shapes during HE operations. In the following section, we describe how *Tapir* internally handles the interactions of these data types during compilation.

Dahut provides a developer-friendly interface to *Tapir* through a central structure, i.e., the Context Manager or `CGManager`, which tracks the changes and operations on the declared symbols. During execution, *Dahut* invokes the execution of the different processes of *Tapir* (see Figure 6.2) through `CGManager`. As a result of this invocation, *Tapir* outputs the necessary files to encrypt (or decrypt) the data and evaluate the circuit homomorphically in *Boar*. To show its simplicity, Listing 1 depicts the code snippet required to transform a Tensorflow model in *HEFactory* and apply DL inference on an encrypted matrix of size 28x28.

Listing 1: Code snippet that adapts a TensorFlow model to *HEFactory* and applies DL inference on the encrypted data

```python
tf_model = load_model('model.h5')
input_data = load_dataset()['x_test'][0]
private_model = Model.from_tf(tf_model)
with CGManager(precision=10,
                performance=0,
                security=0,
                sec_type='classical') as cgm:
    ciphertext_vec = CGArray(cgm, input_data)
    res = private_model.forward(ciphertext_vec)
    cgm.output([res])
boar = Boar(verbose=True)
boar.launch()
results = boar.grab_results()
pt_res = results[res.get_id()]
```

For DL compatibility, *Dahut* internally adapt operations such as matrix multiplication or convolution for their use during HE evaluation. Thus, it relies on algorithms that translate the high-level TensorFlow layers to low-level HE implementation in *Tapir*. These algorithms are designed to optimize under Packed Homomorphic Encryption (PaHE) schemes which allow conducting Single Instruction Multiple Data (SIMD). As discussed in §3.6, SIMD operations substantially optimize the performance and allow for practical applications of HE schemes such as CKKS in real scenarios. We refer to our previous work for a complete description and analysis of these algorithms. These algorithms are optimized to compute a complete forward pass, i.e., *Dahut* provides a transparent compatibility layer for DL inference. We note that the design of *HEFactory* permits easy integration of new algorithms for DL training. Due to the existing challenges in the inference phase [Aha+23; CP21b; HTG17], we leave this integration for future work.

6.3.2 Tapir: A Symbolic-Execution compiler

Tapir, the keystone of *HEFactory*, is a symbolic execution compiler. Symbolic execution is a process where an interpreter evaluates a code over symbolic values without actually executing the program. By performing symbolic execution on HE, the interpreter generates a trace of the operations computed over the symbols and arrays. This trace permits streamlining the required operations and optimizing the memory layout. While *Tapir* performs symbolic execution on HE symbols (i.e., `CGSym` and `CGArray`), it operates dynamically on the remaining operations (i.e., the Python interpreter computes operations over classical variables). This way, *Tapir* simplifies the final output, which will only contain the required low-level HE operations (using a custom IR format), and thus eases the integration of different backend frameworks implemented in *Boar*.

Symbolic execution in regular programs presents challenges for its application, e.g., path explosion that generates complex layouts when multiple branches exist or aliasing when multiple names point to the same memory region. However, these challenges do not affect our framework since we only conduct Symbolic Execution to HE circuits, where only a restricted set of arithmetic operations are permitted (i.e., multiplication, addition, subtraction, and rotation).

Compilation Overview

Tapir follows a classical compiler structure made of a lexer, a syntax analyzer, and a semantic analyzer. In order to integrate *Tapir* as a Python library, it uses the same structures provided by the language. Thus, the Python programming language provides the lexing and parsing of the code. Then, through symbolic execution of the code, the expressions on symbols are extracted and introduced on the AST. In addition to the syntax analysis performed by Python, *Tapir* enforces syntactic rules on the kind of operations that can be introduced in the AST. For example, it enforces the number of rotation operations as an integer value. The semantic analysis tracks the context where operations are permitted. For example, operating on vectors enforces that vector shapes are compatible. Then, it generates an Abstract Syntax Tree (AST), on top of which optimizations and analysis are carried out. Finally, the code generator produces an IR code, which can be serialized and deserialized. It takes a Python program as input and produces output code written interpretable by low-level HE framework (e.g., Lattigo). Internally, it uses an Initial Representation (IR) language, on top of which some optimizations are conducted.

The main goal of *Tapir*'s compilation process is to build a *complete*, *functional*, and *efficient* AST. The AST is *complete* when it includes all operations performed on the symbols, and those can be translated to a binary representation (i.e., it does not include any delayed operation and can be expressed as IR Code). The AST is *functional* if its interpretation by the *Boar* VM can perform all operations, including vector encoding and rescaling. The AST is *efficient* if it allows implementing optimizations to reduce unnecessary operations and minimize the memory footprint. The AST is an essential data structure containing information on the different symbols and their operations. The AST acts as a registry, assigning each variable and expression an identifier. In that way, the internal representation optimizes compilation time.

The compilation process works in two phases. The first phase consists of the symbolic execution of the code to build an initial AST. At this stage, the parameters for the HE circuits are not established; thus, we divide the expressions of the AST into parameter-dependent and parameter-independent. When dealing with parameter-independent expressions, *Tapir* runs the syntactic and semantic analysis. The syntactic analysis enforces rules on which operations can be introduced in the AST. For example, it enforces the number of rotation operations to be an integer value. The semantic analysis tracks the context where operations are permitted. For example, operating vectors enforces that vector shapes are compatible. When dealing with parameter-dependent symbols, in this first Phase *Tapir* leaves a placeholder *Delayed Operation* object on the AST, which provides necessary data for the parameter selection, like the number of expected multiplications or rotations that will be produced.

The second phase aims to perform all the necessary tasks on the initial AST to make it *complete*, *functional*, and *efficient*. To make it *complete*, *Tapir* implements a *Delayed Execution* layer that processes and unrolls the *Delayed Operations*, guaranteeing all the resulting operations in the AST can be translated into IR code. To make the AST *functional*, *Tapir* implements a *Vector Batch Rewriting* layer that rewrites the AST so that all vector operations fit within the slots defined by the HE parametrization. Both the *Delayed Execution* and the *Vector Batching* require to know the HE parameters already. Therefore, *Tapir* implements an automatic parametrization layer. To make the AST *efficient*, *Tapir* implements two layers of optimization, the Eager and Lazy Optimizers.

For *Tapir* to traverse, analyze and rewrite the AST, it implements auxiliary structures in the form of AST Visitors and AST Rewriters. AST Visitors traverse the instructions to analyze the AST without making changes. Multiple visitors can traverse the AST simultaneously without affecting the consistency of the AST (i.e., without affecting the *completeness* or *functionality* of the AST). We use AST Visitors to compute multiplication depth ($O^N_{\mathcal{D}}$) per $logN$, the maximum vector length $|v|_{max}$ used to compute the parameter selection. AST Rewriters are structures that traverse the AST and create, modify or delete instructions. AST Rewriters affect the consistency of the AST and, thus, shall not be executed concurrently. *Tapir* uses Rewriters in the compilation for the Optimization Phase, the Vector Batch Rewriter, and the Delayed Expression Rewriter.

Finally, the circuit generation phase generates binary files that optimally represent all the information in the AST. The circuit generation also involves retrievability and deserialization of the contents by the *Boar* VM.

Parameter-Independent Expressions

Symbols (`CGSym` and `CGArray`) are the main programming entities used by *Tapir* and exposed to developers in *Dahut*. *Tapir* uses symbols to guarantee syntactic and semantic consistency of the different interactions with other programming variables and symbols.

For consistency, we consider an expression to be a complex interaction of different symbols in the source code. In its first phase, *Tapir* simplifies and traces the interactions of symbols in instructions, which are reduced traces of an expression. When *Tapir* interacts with parameter-independent expressions, it simplifies them and traces the different instructions thanks to symbols. Internally, instructions comprise three operands (i.e., two inputs and one output) and an operation. While the input and the output operand are always a `CGSym`, the other input operand or *special* operand can be an integer or floating-point number, a vector, or another symbol. Furthermore, *Tapir* supports four operations: addition, subtraction, multiplication, and rotation. Depending on the operation and the *special* operand, *Tapir* implements different syntactic and semantic rules on the instructions.

Upon these basic operations, *Tapir* supports additional functionality (e.g., optimized polynomial computation with repeated squaring and Horner's algorithm). Also, these operations are the building blocks for the inner algorithms that transform DL internal layers to HE-friendly functions.

Parameter-Dependent Expressions

In Homomorphic Encryption, some algorithms can take advantage of overflows during rotation, e.g., for vector indexing. The appropriate execution of the algorithm depends on the disposition of the real ciphertext slots, which in turn depend on the HE parameters. Thus, these are considered parameter-dependent expressions and cannot be processed before the HE parameters are selected.

In the first phase of the compilation, for these parameter-dependent expressions, *Tapir* introduces a placeholder instruction or *Delayed Operation* object in the AST. After performing the parameter selection in the second phase, the Delayed Executor rewrites the AST, replacing the delayed operation object with the analogous algorithm using the low-level instructions.

Delayed Operation objects are a simple interface that permits representing the algorithm in a compiler-friendly way. The *Delayed Operation* object functionality allows the implementation of algorithms for different parametrizations guaranteeing compliance and flexibility. They conveniently provide two main functions. On the one hand, they store, at a per-N basis, the cost of the *Delayed Operation* in terms of additions, multiplications, rotations, multiplication depth, and multiplication depth with extended rotations. These are specific metrics needed for parameter selection and enabling a forward pass of the code without actually unrolling the operation. On the other hand, after performing parameter selection, the *Delayed Operation* object contains the specific function code that, combined with the AST, is executed under specific parameters.

With the actual instructions for Parameter-independent expressions and the Delayed objects for parameter-dependent expressions, the first phase of the compilation gets an initial AST. Then, consecutive layers transform that into a *complete*, *functional*, and *efficient*, as explained next.

Eager Optimizer

This optimizer consists of an AST Rewriter, which acts on top of the initial AST built from the code written by the user. It works before parameter selection and delayed execution. The role of this optimization is to detect redundant patterns in operations before the Parameter Selection takes place. In our current prototype, the Eager Optimizer is instructed to detect and remove duplicated operations. This optimization significantly reduces the multiplication depth, thus speeding up the computation and reducing the total cost of the operations. Due to its modular design, further compiler optimizations can be further designed and applied at this stage.

Automated Parameter Selection

In *Tapir*, the parameter selection results from a combination of an AST Visitor, Fuzzy Logic, and Linear Programming. The AST Visitor is in charge of extracting various circuit parameters relevant to parameter selection, e.g., the multiplication depth at a per-$logN$ basis, the number of uses of each operation, and the maximum vector size $|v|_{max}$. We refer to Chapter 5 for further details on this automatic parameter selection process. This selection helps automate the process and simplifies the selection for non-expert users.

Delayed Execution

Once the HE parameters are defined, the *Delayed Execution* layer consists of an AST Rewriter that operates on the delayed operations (left as a placeholder during the first phase of the compilation) and unrolls them into low-level instructions. It traverses the AST looking for delayed operations, replacing the placeholder with the analogous instructions of the operation, considering the pre-existing parametrization. The code replacement is done thanks to the delayed operation object present in the instruction. The AST Rewriter relies upon a program pointer to control the program flow. Whenever this pointer targets a function marked as *Delayed*, the AST Rewriter moves the program pointer to the entry point of this function and processes it. Afterward, the algorithm is unrolled into its low-level operations, and once it finishes, the program pointer is restored to its previous position.

Vector Operation Batch Rewriter

After the *Delayed Execution* layer, the AST provides a *complete* representation of the code. However, because operations contain arbitrary length vectors, representation is not yet *functional*. The AST gives no guarantee that the vectors would fit on the maximum number of slots (i.e., the maximum size of a plaintext vector that can allocate a ciphertext). When the maximum number of slots is exceeded, typically, the programmer must represent the plaintext with multiple ciphertexts. We denote the split of a plaintext vector into multiple ciphertexts as batching and each ciphertext vector as batches. This procedure involves manually splitting every plaintext vector and guaranteeing consistency along the code. This procedure is highly inconvenient, cumbersome, and prone to errors process. Furthermore, an extended procedure is needed once rotations appear since the overflow of rotations should be preserved across batches. *Tapir*'s Vector Operation Batch Rewriter analyzes the expressions in the AST and rewrites them so that batching is performed automatically according to the maximum vector size $|v|_{max}$. To the best of our knowledge, this is the first automatic implementation of such techniques in an HE compiler.

The procedure works under a set of guarantees that the architecture of *Tapir* enforces from previous layers. Tapir assumes that the algorithms inform when they contain size-increasing operations. In this case, the output symbol shall explicitly reshape the output symbol (similar to broadcasting rules in NumPy). The assumptions are motivated by the inability of the compiler to determine how to interact with two differently-sized batched vectors (i.e., with a different number of batches) or on size-increasing operations. Due to this, we need to establish a uniform rule for the compiler to interpret these expressions. The guarantees assumed are the following:

1. Any two input operands within the same operation are always equally sized arrays. The semantic analyzer in `CGSym` guarantees that this condition holds; therefore, every input symbol operates with another symbol or vector of the same shape.

2. Size-growing operations (i.e., those that can potentially increase the size of a vector) always reflect the shape change on the output vector. Holding the previous condition means two operands share the same shape, but the output symbol shape may increase. We detail later the batching of such operations.

3. All symbols derive from a previous symbol, except input symbols. Therefore, unless specific size-growing operations are performed (i.e., matrix multiplication), the vectors shall always fit within the batch size of the previous vector.

4. Rotations are the uttermost relevant step in size-growing operations where a size-changing vector influences the control flow. Not performing a rotation on the appropriate sizes can yield unpreserved algorithm integrity, given the use of overflows in HE. Additions and multiplications on size-growing operations can be padded and do not require such attention.

With these considerations in mind, the algorithm follows a three-step process.

1. It analyzes input symbols for potential rewriting.

2. It iterates over instructions looking for i) interactions of batched input symbols and ii) size-increasing operations that would not fit within the maximum slots (i.e., those where the output symbol shape is larger than the input symbols).

3. If any of the two previous conditions occur, the output batched symbols are also cached for subsequent propagation of the batching in the remaining operations.

Batching implies changes to symbols and operations. For symbols, it is only required to replicate their representation. With operations, we differentiate between non-rotations and rotations. Non-Rotations only require replicating the operation n times on the different batches. On the other hand, rotation operations require performing combinations of frames of the different vectors. For a more detailed explanation of extended rotations, we refer the reader to Chapter 4, where we tested and elaborated these algorithms.

For vector and symbol operations, we fill the vector operator with 0 up to the maximum length and then perform the batching on the output vectors. The algorithm takes automatic care of the extension of operations. It simply replicates the operation n times acting on the different batches for additions and multiplications. For rotation, the algorithm substitutes each rotation with the equivalent extended rotation algorithm, as shown in Figure 4.4. For example, the operation $\vec{c} = \vec{a} + b$ where $\vec{a}$ is bigger than the number of slots would require replicating $\vec{a}$ into n smaller vectors $\vec{a}_0, \vec{a}_1, ..., \vec{a}_{n-1}$. The operation would be replicated such that $\{\vec{c}_0 = \vec{a}_0 + b, \vec{c}_1 = \vec{a}_1 + b, ..., \vec{c}_{n-1} = \vec{a}_{n-1} + b\}$. The batched result of the operation is stored as equivalent to the original symbol in operation. In that way, after analyzing each operation, propagation occurs on the remainder of the circuit.

Lazy Optimizer

This layer consists of an AST Rewriter, which executes right before binary generation and serves for memory optimization at execution time. The main task of the Lazy Optimizer is to introduce `free` operations. These operations identify the last operation using a symbol, after which the symbol can be removed from memory. In that way, during execution time, the compiler guarantees a minimum memory allocation for variables (i.e., reduced memory footprint).

Circuit Generation

At the end of the compilation, *Tapir* produces three different binary files through its binary generator. First, the HE Parameter File holds a binary representation of the parameters that the compiler estimates necessary for the execution of the circuit. Second, the Input Data File stores the data of all input `CGSym` in a binary format that Boar understands for encryption. Finally and more importantly, the IR code File contains a translation of the AST instruction in an assembly-like format. The IR code File contains a header including metadata, i.e., the number of instructions and the size of operands. On average, every operation occupies 13 bytes, except when we deal with vector operations, where the full vector needs to be stored. In *Tapir*, every instruction is organized as a tuple with the format: `[<op> <result> <input_a> <input_b>]`. The operands `<result>` and `<input_a>` are always a `CGSym` or `CGArray` and can be represented by a 32-bit integer. The free operand accepts three different values for `<input_b>`, a floating point value, vector, or symbol. Floating-point and integer numbers are represented in 32-bit notation. *HEFactory* supports the operations (`<op>`) that are needed for HE computation, i.e., addition, subtraction, multiplication, rotation, and `free`. Additionally, the IR code has a different representation for the operation depending on the type of the free operand. We use a byte to represent the operation code in IR. The Initial Representation details can be found in Table 6.2.

All three files generated by *Tapir*'s Circuit Generation are independent and use relative addressing to other files. That means that subsequent executions of the compiler may generate different data files compatible with the code file without requiring a recompilation.

Further compression can be effectuated on the binary representation to optimize the file size. Although *HEFactory* currently supports the CKKS scheme and parameters, the binary format is extensible and could potentially be implemented with other homomorphic encryption schemes.

6.3.3 Boar VM: A bridge to HE frameworks

Tapir outputs three different files with a specific, custom format: one for the parameters, one for the data, and one for the circuit. The serialization of the circuit and data into different files allows for enforcing access control policies for different users, i.e., only the Data Owner might get access to the data. These files are processed by *Boar*, composed of three different entities for encryption and decryption of the data and for computation of the circuit. *Boar* acts as a virtual machine since it receives the circuit represented in a custom binary format from *Tapir,* and blindly executes it using the HE backend. The proof-of-concept prototype implemented for this chapter supports the Lattigo framemework [Mou+20]. However, we note it can be adapted to any other HE framework, such as Microsoft SEAL [Res20] or IBM HELib [HS14b].

Boar Encryptor

It produces an encrypted file and the different keys needed for its processing. In order to do that, it reads the HE parameters file, from which it generates appropriate keys. Afterward, it uses those keys to encrypt the contents of the input data file. The secret key shall be kept private for Data Owner, whereas the public and evaluation keys must be shared with the Data Processor. These different keys are also serialized into different files to allow the setup of appropriate access control.

Boar Evaluator

It simulates the runtime of a standard OS process. For that, it needs data, code, and additional elements. The additional elements are the public and evaluation keys obtained from the client. In general, the Data Owner might provide the data in an encrypted format. In contrast, the Data Owner or Data Processor might provide the actual compiled code (depending on who trained the model).

It loads the information and generates an internal memory structure to store the intermediate ciphertexts, guaranteeing minimal memory usage by freeing variables when appropriate. The low-level virtual machine can automatically cache large ciphertexts on disk, introducing a reduced residing set in memory so that the memory usage of the circuit is reduced (at the cost of an increased computational time) to enable processing in memory-constrained devices. We note that further processing optimizations (e.g., parallelization or GPU acceleration) can be implemented in this module. Once the execution finishes, *Boar* Evaluator serializes and writes the remaining ciphertexts from memory into files.

Once the code finishes the execution, *Boar* Evaluator serializes and writes the remaining ciphertexts from memory into files.

IR Operations					
Op. Name	Opcode	Operand A (a)	Operand B (b)	Output (c)	Operation
addi	0	Ciphertext	Integer	Ciphertext	$c = a + int(b)$
subi	1	Ciphertext	Integer	Ciphertext	$c = a - int(b)$
muli	2	Ciphertext	Integer	Ciphertext	$c = a \cdot int(b)$
addf	3	Ciphertext	Float	Ciphertext	$c = a + float(b)$
subf	4	Ciphertext	Float	Ciphertext	$c = a - float(b)$
mulf	5	Ciphertext	Float	Ciphertext	$c = a \cdot float(b)$
addx	6	Ciphertext	Ciphertext	Ciphertext	$c = a + b$
subx	7	Ciphertext	Ciphertext	Ciphertext	$c = a - b$
mulx	8	Ciphertext	Ciphertext	Ciphertext	$c = a \cdot b$
rot	9	Ciphertext	Integer	Ciphertext	$c = a << b$
addv	10	Ciphertext	Vector	Ciphertext	$c = a + array(b)$
subv	11	Ciphertext	Vector	Ciphertext	$c = a - array(b)$
mulv	12	Ciphertext	Vector	Ciphertext	$c = a \cdot array(b)$
free	99	Ciphertext	-	-	$delete(a)$

Table 6.2: Intermediate Representation code produced by *Tapir* and interpreted by *Boar*. Every instruction is organized as a tuple with the format: `[<op> <output> <operand_a> <operand_b>]` The table details the values and datatypes allowed by each operation.

Boar Decryptor

Finally, similarly to the encryptor, the decryptor reads the encrypted result from the output file. It allows it to decrypt the ciphertexts using the secret key generated on the *Boar* Encryptor. We assume the Data Owner does this process in a controlled environment.

6.4 Framework Evaluation

This section evaluates *HEFactory* describing the experimental design and goals first and then showing the results.

6.4.1 Experimental Design

HEFactory makes HE usable for non-experts. For testing, we use a benchmark inspired in HECO [Via+23], evaluating different operations (detailed in Table 6.3). For comparison, we implement these operations using four approaches:

1. Code produced with *HEFactory* (i.e., using the *Dahut* library, in Python).

2. Code written in Lattigo with expert awareness (i.e., conducting manual optimizations based on HE expert knowledge in Go)

Benchmark Tests	
Test	Details
T1	**Box Blur** performs the 3×3 blurring of a 32×32 image.
T2	**Dot Product** performs a dot product between two 8-element ciphertext vectors.
T3	**Gx Kernel** performs the x axis Sobel filter to a 32×32 image.
T4	**Hamming Distance** computes the distance between two 8 element vectors $a, b = \{0, 1\} \in \mathbb{Z}$ using logarithmic accumulation.
T5	**L2 Distance** computes the L2-distance between two 8 element vectors $a, b \in \mathbb{Z}$ using the square root.
T6	**Linear Polynomial** performs the computation of a linear polynomial $(3x + 7)$ over 8 elements of a vector.
T7	**Matrix Multiplication** performs the diagonal matrix multiplication [CP22] of a 64×64 matrix and a 64 element vector.
T8	**Quadratic Polynomial** performs the computation of a linear polynomial $(2x^2 + 3x + 7)$ over 8 elements of a vector.
T9	**Robert Cross** combines the computation of two convolutions and the normalization of those two convolutions using the square root.
T10	**Deep Learning Emulation** combines the computation of two convolutions, followed by a result transformation, a degree 2 polynomial activation function, and matrix multiplication.

Table 6.3: Benchmark tests performed to evaluate the performance of *HEFactory*.

3. Code written as a non-expert (i.e., using a naive approach to only use basic operations of HE framework in Go).

4. Code written in EVA, a similar Python library that works on top of Microsoft SEAL.

To evaluate **performance**, we measure the execution times of the HE processing. The detailed parameters used in each test are depicted in Table 6.4. For the non-expert version, we assume the user utilizes predefined sets of parameters available in the library. The expert version uses performant and well-tested parameters. The EVA and *HEFactory* versions use automatically generated parameters by their automated circuit analysis.

We also evaluate **code simplicity** by analyzing two factors: the number of lines of code (LoC) and characters per LoC. In general, shorter code is faster to write, easier to iterate, and faster to fix.

All the experiments are run on an AMD Ryzen 9 3950x with 32 Gigabytes of RAM on Ubuntu 20.04, running Python 3.10 and Golang 1.16 with Lattigo v2.1.1. EVA is compiled according to the installation instructions provided in the Github repository with Clang for C++17 in Ubuntu 20.04.

6.4.2 Results

Performance. Figure 6.3 shows the different runtimes. We observe major differences between non-expert code and *HEFactory* code. Specifically, algorithms with little vectorization content (e.g., linear and quadratic polynomial) show a lesser degree of overhead, but vectorized algorithms show considerable differences (e.g., matrix multiplication or box-blur). Expert-written code and *HEFactory* code have less disparity between each other, as code produced by *HEFactory* resembles the operations performed by an expert.

EVA performance shows equivalent performance to *HEFactory*. However, it shows problems with certain algorithms, such as the square root algorithm (used in L2 Distance and Robert Cross), which is unable to perform because of internal crashes. Also, for certain very small programs, such as the linear and quadratic polynomial, the execution time remains bigger than *HEFactory* and Lattigo Expert. We note that given that the backend libraries are different (Microsoft SEAL vs. Lattigo), the differences might be related to issues in these libraries. Unlike EVA, however, *HEFactory* can be easily modified to work on top of SEAL or any other library by implementing new translation functions in the Boar layer (i.e., to transform our IR code to SEAL code in C++). We refer to a recent work by Gouert et al. for a detailed comparison of backend libraries [GMT23].

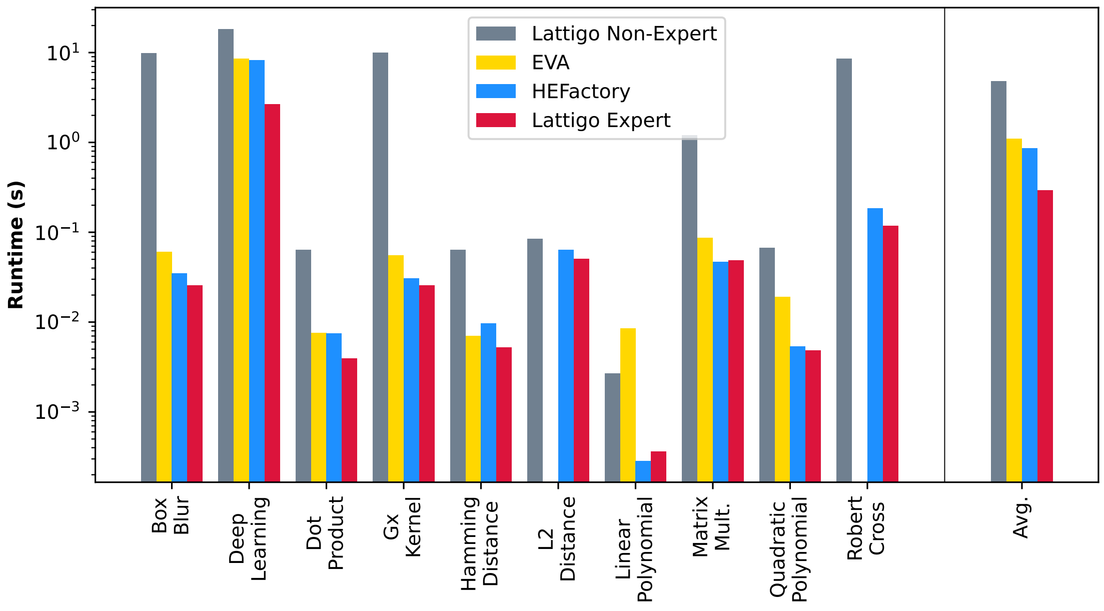


Figure 6.3: Performance comparison of *HEFactory* with code written in Lattigo.

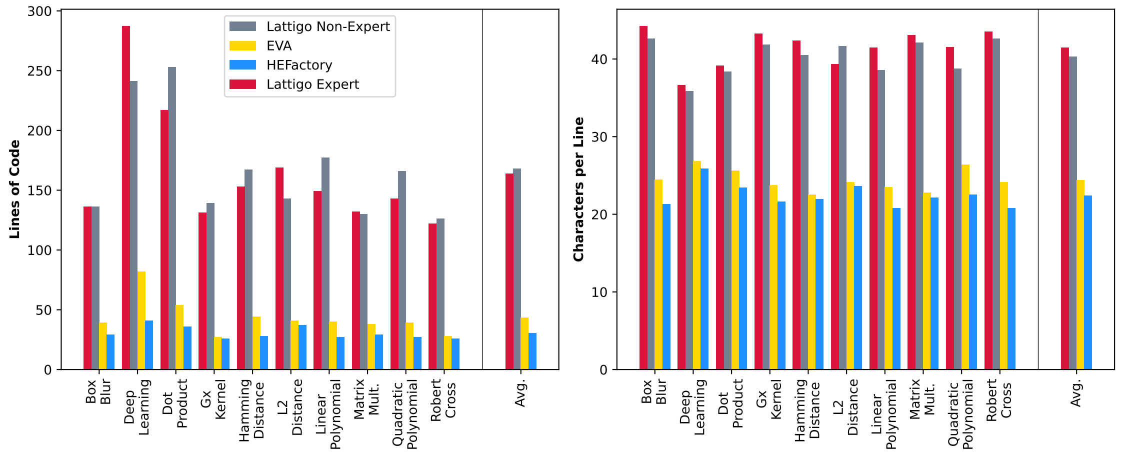


Figure 6.4: Comparison of *HEFactory* with base code written in Lattigo in terms of lines of code (left) and characters per line of code (right).

Test Benchmark Parameters												
	Box Blur				Dot Product				Gx Kernel			
	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ
HEFactory	13	150	3	30	13	150	3	30	13	150	3	30
EVA	13	150	3	30	13	150	3	30	13	150	3	30
Lattigo Expert	13	160	2	40	13	160	2	40	13	160	2	40
Lattigo Non-Expert	13	218	6	30	13	218	6	30	13	218	6	30
	Hamming Distance				L2 Distance				Linear Polynomial			
	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ
HEFactory	14	241	6	30	13	150	3	30	13	150	3	30
EVA	13	150	3	30	-	-	-	-	13	180	3	60
Lattigo Expert	13	281	5	40	13	160	2	40	13	160	2	40
Lattigo Non-Expert	13	218	6	30	13	218	6	30	13	218	6	30
	Matrix Multiplication				Quadratic Polynomial				Robert Cross			
	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ	LogN	LogQ	$O^N_{\mathcal{D}}$	Δ
HEFactory	13	150	3	30	14	241	6	30	13	180	4	30
EVA	13	150	3	30	13	150	3	30	-	-	-	-
Lattigo Expert	13	160	2	40	14	281	5	40	13	241	4	40
Lattigo Non-Expert	13	218	6	30	13	218	6	30	13	218	6	30

Table 6.4: Parameters used for the benchmark test for each framework.

Code Simplicity. Figure 6.4 shows the number of LoC (left) and the average number of characters per LoC (right) for the different benchmarks. Implementations written in Lattigo are similar (averaging over 150 LoC per benchmark) and are worse than the average 35 LoC per benchmark in *HEFactory*. *HEFactory* also provides better results regarding characters/LoC, with 25 characters (versus 50 for the Lattigo variants). On average, *HEFactory* produces 80% fewer lines of code with 55% fewer characters per line of code (i.e., making code shorter and more straightforward to read). While the differences with EVA are narrow, EVA provides a minimal interface that displaces the programming burden onto the user for complex operations. *HEFactory*, on the other hand, provides a set of essential tools for program building, such as basic vectorized functions and other primitives, which eases the work on this library. This is especially meaningful in the Deep Learning test, where the increased complexity of the use case impacts EVA more than *HEFactory*.

Takeaway. *HEFactory* provides performance-equivalent results with a simpler interface. Thus, it lowers the bar for using HE primitives for non-expert users at no cost in terms of performance.

6.4.3 Deep Learning Evaluation

A primary goal of *HEFactory* is to provide an abstraction layer for Deep Learning. We evaluate compilation time, performance, and code simplicity of the execution of DL inference in Lattigo for a Convolutional Neural Network. We train a modified LeNet 5 model, similar to the described in CryptoDL [HTG17] model. LeNet 5 [LBH15] neural network is a handwritten digit classification model. The modifications to the base neural network

Neural Network Architecture	
Layer	Details
2D Convolution	Kernels: 5, Kernel Size: 5×5, Stride: 1, Padding: 0
ReLU Activation	Chebyshev approximation of $x + log(1 + e^{-x})$, of Degree: 4 and Domain: $[-100, 100]$
2D Average Pooling	Pool Size: 3×3
Flatten	-
Dense	Units: 100
Sigmoid Activation	Chebyshev approximation of $\frac{1}{1+e^{-x}}$, of Degree: 4 and Domain: $[-100, 100]$
Dense	Units: 1
Sigmoid Activation	Chebyshev approximation of $\frac{1}{1+e^{-x}}$, of Degree: 4 and Domain: $[-100, 100]$

Table 6.5: CNN architecture used in DL inference with Homomorphic Encryption.

include using linear Chebyshev approximations of the activation functions and vectorized linear algebra primitives for the convolution and matrix multiplication [CP; CP22]. Table 6.5 depicts the architecture of the implemented neural network. As described in Listing 1, *HEFactory* can directly receive a Tensorflow model, which underneath transforms into basic instructions that *Tapir* can elaborate on and later on transmit to *Boar*. The `model` class transforms the model. It extracts the activation functions of the model to use. The only exception to this replacement is the last activation function, which only needs to be executed on the output classification and can be performed by the DO and remains a sigmoid function.

Table 6.6 summarizes the results. In total, the compilation time for the circuit is 5.4 seconds. As for the performance when running the inference, we observe that it requires around 199s. However, *HEFactory* is motivated by the computation burden being outsourced to a cloud provider, which can be assumed to have higher computing capabilities. Therefore, in Table 6.6, we split the total time into the time taken by the different tasks and analyze which of these need to be executed either locally by the client (i.e., key generation, encryption, and decryption) or remotely by the server (i.e., circuit evaluation). The execution of inference on a sample only requires 11s of client computation (10.4s for key generation and 0.06s for encryption and decryption). In contrast, the most significant computation of the evaluation (i.e., 187s, $\approx 94\%$) is offloaded to the cloud or a third-party server, which can be assumed to have higher computing capabilities. A significant result is that *HEFactory* requires only 66 lines of code to adapt DL inference from an existing CNN in TensorFlow. Most of these lines are related to API calls to the internal functions of *HEFactory* in a similar fashion as other Python-related libraries.

DL Compilation Time Results

Parameter Selection	Delayed Execution	Vector Batching	Optimizer	Binary Generation	Total
1.1 s	0.4 s	7 ms	0.01 s	3.48 s	5.4 s

DL Performance Time Results

Parameter generation	Key generation	Encryption	Evaluation	Decryption	Total
$6.9 * 10^{-4}$ s	10.4 s	0.05 s	187 s	0.01 s	199 s

Neural Network Lines Of Code

Lines of Code	Characters / LoC
66	40

Table 6.6: Evaluation results for Deep Learning inference with Homomorphic Encryption.

Takeaway. This test confirms that *HEFactory* allows for efficient execution of vectorized privacy-preserving DL inference, with a few modifications to existing data science programs. We believe that this will enable the use of HE and DL for users without previous knowledge of Privacy-Preserving Technologies, without requiring previous knowledge about the inner cryptographic techniques or the implementation details.

6.5 Summary

Homomorphic Encryption remains a complex technology with multiple application challenges. It also requires a steep learning curve for its practical usage, preventing such techniques from being successfully applied by non-experts. In this chapter, we present *HEFactory*, a software stack composed of *Dahut*, a high-level Python API; *Tapir*, a symbolic compiler; and *Boar*, a low-level virtual machine. *HEFactory* relies on a layered architecture that implements several improvements to deal with the existing challenges of HE. *Dahut* allows adapting trained Deep Learning models (and other high-level operations) using HE. It directly interacts with *Tapir*, a symbolic compiler that translates high-level expressions to simple low-level homomorphic instructions using a custom binary format. It integrates various intelligent systems and algorithms for the functional evaluation of complex operations, parameter selection, vectorized ciphertext batching, and optimizations. *Boar* is the low-level virtual machine that interprets binary files using existing HE backend frameworks. We implement a prototype and program various Python tests on top of Lattigo, a state-of-the-art HE framework. Our evaluation shows that *HEFactory* substantially reduces the complexity of programming Deep Learning applications with HE (i.e., a reduction of 80% in the number of lines of code and by 40% in the complexity of statements required with respect to its equivalent in Lattigo), with negligible compilation times and performance overhead.

Chapter 7

Conclusions and Future Work

In this thesis, we set the goal of providing techniques that reduce the complexity of Homomorphic Encryption (HE) when applied to Deep Learning (DL) techniques, especially for non-expert users. To this end, we stated a set of sub-objectives.

We next review each of these objectives and discuss the corresponding contributions proposed in this thesis:

- **O1. Systematizing existing knowledge in the application of privacy-preserving computation techniques to Deep Learning.** Chapter 2 and Chapter 3 address this objective by providing a systematic view of the current state of the art of Privacy-Preserving Deep Learning (PPDL) with PPCTs. The systematic review allows identifying major use trends, potential research gaps, and issues with existing techniques, focusing on efficiency and usability issues. The insights from this study informed the remainder of the research questions addressed in this thesis.

- **O2. Analyzing and providing algorithms to adapt Deep Learning Inference with Packed Homomorphic Encryption.** Chapter 4 provides a detailed description of new and existing vectorization algorithms for computing convolutional neural networks with PaHE. Additionally, it provides a detailed analysis of the different algorithms and concludes with guidelines for optimal use and empirical demonstrations of those.

- **O3. Simplifying Homomorphic Encryption parameter selection for non-expert users.** Chapter 5 introduces the design of a novel expert system combining fuzzy logic and linear programming tasks that perform optimized parameter selection based on high-level user input parameters. Our evaluation shows how the high-level user elections yield parametrizations that reflect user choices at execution time without requiring a low-level understanding of the cryptographic primitives.

- **O4. Integrating the previous improvements in a computer-aided tool that enables inexperienced users to use Homomorphic Encryption for Deep Learning.**

Chapter 6 describes the implementation of a symbolic execution compiler for HE. The architecture simplifies the use of HE by providing efficient vectorized algorithms, automated parameter selection, and high-level DL support in Python. Our evaluation shows that, with a reduced number of lines, the symbolic compiler can produce performance-wise equivalent code to expert-written low-level code.

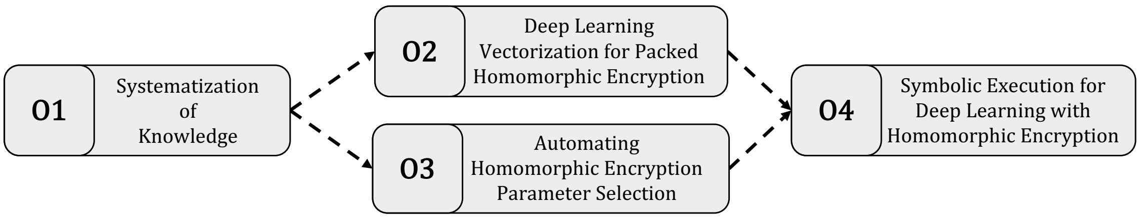

O1
Systematization of Knowledge
O2
Deep Learning Vectorization for Packed Homomorphic Encryption
O3
Automating Homomorphic Encryption Parameter Selection
O4
Symbolic Execution for Deep Learning with Homomorphic Encryption

Figure 7.1: Overview of the thesis objectives with the main keywords and techniques used to achieve each objective.

7.1 Contributions

Collectively, this thesis presents the subsequent contributions:

- **A systematization of knowledge of the field of Privacy-Preserving Deep Learning (PPDL) focused on using PPCTs** that introduces the different main goals of PPDL and the solutions provided by PPCTs. The work describes the various approaches taken by state-of-the-art contributions in applying PPCTs. It analyzes the major trends and potential research gaps to detail future lines of research.

- **A formal analysis with usage guidelines of vectorized algorithms for PPDL with HE** that addresses the generalization problem of vectorized algorithms and details the best practices to use vectorized algorithms in convolutional neural networks. Furthermore, it generalizes existing algorithms and introduces new ones.

- **An expert system combining fuzzy logic with linear programming tasks** that effectively reduces the complexity of performing HE parameter selection and provides users with high-level features which are then reflected in the parametrizations.

- **An HE symbolic compiler in Python** that enables the use of HE in data science projects for non-experts. Among other features, it implements efficient vectorized algorithms, automated parameter selection, and high-level DL support. Thus, non-expert users can use HE with expert-equivalent performance without having to learn complex mathematical formulas and without tedious circuit optimizations. The compiler leverages the scientific contributions presented at earlier stages of the thesis and provides a new software design that allows to include of additional improvements in the future. As such, the compiler is provided open-source.

7.2 Dissemination

The outcomes of this thesis have been disseminated through the following publications at diverse forums.

7.2.1 Publications

Journals

- José Cabrero-Holgueras and Sergio Pastrana. "Towards Realistic Privacy-Preserving Deep Learning over Encrypted Medical Data". In: *Frontiers in Cardiovascular Medicine* 10 (), p. 641
- José Cabrero-Holgueras and Sergio Pastrana. "Towards Automated Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming". In: *Expert Systems with Applications* (2023)
- José Cabrero-Holgueras and Sergio Pastrana. "HEFactory: A Symbolic Execution Compiler for Privacy-Preserving Deep Learning with Homomorphic Encryption". In: *Software X* (2023)

Conferences

- José Cabrero-Holgueras and Sergio Pastrana. "SoK: Privacy-Preserving Computation Techniques for Deep Learning". In: *Proceedings on Privacy Enhancing Technologies* 2021.4 (2021), pp. 139–162

7.2.2 Presentations

Conferences

- **SoK: Privacy-Preserving Computation Techniques for Deep Learning**. The 21st Privacy Enhancing Technologies Symposium. July 12–16, 2021, Online.
- **Usable Homomorphic Encryption for Private Telemedicine on the Cloud.** Italian Telemedicine Society Congress (SIT) October 22-23, 2021, Online.

Workshops

- **Privacy and Deep Learning for Healthcare Research**. CERN openlab Workshop. March 10, 2021. CERN, Geneva, Switzerland.
- **Post-Quantum Privacy-Preserving Data Analysis Pipelines**. CERN openlab Workshop. March 22, 2022. CERN, Geneva, Switzerland.

- **Quantumacy Project**. OpenQKD 5th General Assembly. June 23, 2022. Paris, France.

7.3 Future Work

From the proposals presented in this thesis, various future lines of research emerge:

- The contents presented in Chapter 4 demonstrate that developing vectorized algorithms can be a time-consuming process compared to the more verbose semantics produced by traditional code. To alleviate this issue, future proposals should incorporate internal transformations that generate equivalent vectorized algorithms from classical representations or automatically select optimal layouts based on the complexity of operations and subsequent result transformations.
- Chapter 5 showcases a novel method of automating algorithm parametrization that maintains user objectives by incorporating expert knowledge in a heuristic system. This work is premised on a static circuit that remains unchanged following its generation. Alternative strategies may involve integrating the circuit within the search space to assess options where circuits and parameters are subject to modification. Furthermore, our current tool has the potential to facilitate the creation of a training dataset for DL models that can optimize parametrization based on known outputs.
- The symbolic compiler presented in Chapter 6 is aimed at producing user-friendly code that eases the adoption of HE. While efficiency was desirable, it was not a priority in the development. Further compilation strategies remain to be tested and evaluated, such as introducing low-level programming language optimization strategies. Additionally, the analysis and application of compilation features for optimizing both the code generation process and the generated intermediate representation code.

  Second, current HE is generally a sequential process. It is desirable to analyze code to enable parallel processing and exploit it is desirable to analyze parallel patterns to exploit multiprocess execution of HE routines.

  Finally, this thesis claims an improvement in code complexity thanks to using the symbolic-compiler *HEFactory*. However, it is not the same code complexity as usability. To support the claims of usability improvement, it would be desirable to carry out a usability study where we analyze the coding efficiency of users performing. The usability study can also help understand the main challenges and complexities present in the current use of the tool.

Appendix A

Baseline Convolution Algorithms

In this appendix we overview the base algorithms upon which we improve the proposed in Section 4.3.2. First, we describe the base convolution algorithm (§A.1), its base result transformation (§A.2) and the introduction of Stride (§A.3) and padding (§A.4).

A.1 Convolution

We propose a general algorithm that permits the application of the convolution operation to arbitrary matrices using SIMD operations. As a starting point, we based the algorithm on examples proposed for matrices of 3×3 using kernel filters of 2×2 [Dat+19].

The algorithm takes as input a plaintext filter $\mathcal{F} \in \mathbb{R}^{f_x \times f_y}$ of dimensions $f_x \times f_y$. The filter is applied to a ciphertext vector $c_t \in \mathbb{Z}_Q[x]/(x^N + 1)$ (in Row-Column (RC) format) that corresponds to an encrypted input matrix, i.e., $c_t = Enc(\mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$, with p_k being the encryption key and $\mathcal{M}$ the input data in cleartext. The algorithm leverages the fact that the dimensions of filters are shorter than input matrices and that we can operate them in plaintext. Thus, it computes the convolution between each pixel of the filter and the input matrix (i.e., represented by a ciphertext) and adds the partial results for each pixel. The algorithm is described in Algorithm 11.

Depending on whether we use stride or not, we consider a different result layout. In our work, we name two, the Convolution Resulting Format (CRF) for non-stridden convolution and the Stridden Convolution Resulting Format (SCRF) for stridden convolutions. These layouts include o_{null} meaningless values between the values of the result (w_{out}) designated by the input matrix $\mathcal{M} \in \mathbb{R}^{h \times w}$ and filter size $\mathcal{F} \in \mathbb{R}^{f_x \times f_y}$. Generally, these formats are not valid for consecutive operations (layers), therefore Result Transformation (RT) algorithms are needed, and we describe them next.

Algorithm 11 2D Convolution

Input: $c_t \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(\mathcal{M} \in \mathbb{R}^{h\times w}, p_k)$, $\mathcal{F} \in \mathbb{R}^{f_x\times f_y}$
Output: $conv \in \mathbb{Z}_Q[x]/(x^N + 1)$ in CRF or SCRF format

```
function CONVOLUTION($c_t$,$\mathcal{F}$)
    for $i \leftarrow 0, f_x$ do
        for $j \leftarrow 0, f_y$ do
            $rot \leftarrow c_t \ll (i * w) + j$
            $conv = conv \oplus rot \odot \mathcal{F}_{i,j}$
        end for
    end for
    return $conv$
end function
```

A.2 Result Transformation for CRF Format

In the CRF format, the amount of o_{null} values is given by $w_{out} = w - f_y + 1$ and $o_{null} = w - w_{out} = f_y - 1$. Therefore, a Result Transformation algorithm is developed to transform the CRF format to the Row-Column format (named RT-CRF-RC). The original layout splits the useful rows w_{out} by o_{null} values, therefore, the algorithm creates bitmasks for the rows and shifts them to the appropriate position on the resulting RC) format. This processing is described in Algorithm 12.

Algorithm 12 Result Transformation CRF to RC

Input: $c_t^{CRF} \in \mathbb{Z}_Q[x]/(x^N + 1)$ in CRF format, $\mathbb{R}^{h\times w}$, $\mathbb{R}^{f_x\times f_y}$
Output: $c_t^{RC} \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(\mathcal{M} \in \mathbb{R}^{h_{out}\times w_{out}}, p_k)$, in RC format

```
function RT-CRF-RC($c_t$, $h$, $w$, $f_x$ $f_y$)
    $h_{out} \leftarrow h - f_x + 1$
    $w_{out} \leftarrow w - f_y + 1$
    $o_{null} \leftarrow w_{out} - w$
    for $i \leftarrow 0, h_{out}$ do
        $bitmask[t]_i \leftarrow \{i \cdot w_{out} \leq t < w_{out} \cdot (i + 1)\}$
        $row_i \leftarrow c_t^{CRF} \ll (i \cdot o_{null})$
        $c_t^{RC} = c_t^{RC} \oplus (row_i \odot bitmask[t]_i)$
    end for
    return $c_t^{RC}$
end function
```

A.3 Stride

Sometimes, the input matrices to a convolutional layer are high resoultion images (i.e., have long dimensions (h, w)). Processing these images demands high performance cost, since the convolutions extract features from small areas (as defined by the kernel). To avoid processing large portions of the images, the process can be optimized by skipping the result of parts of the convolutions. The amount of data to be skipped is defined by a *stride* tuple (s_x, s_y). That is, considering that in a normal convolution the output shape is defined by $h_{out} = h - f_x + 2 \cdot p + 1$ and $w_{out} = w - f_y + 2 \cdot p + 1$, stridden convolutions reduce the output shape by a factor of the stride (s_x, s_y) such that $h_{out} = \frac{h - f_x + 2 \cdot p + 1}{s_x}$ and $w_{out} = \frac{w - f_y + 2 \cdot p + 1}{s_y}$.

Given the convolution algorithm is based on the filter size, the reduction of output elements does not affect the convolution algorithm itself, but it does provide a different layout for the output format. In such a layout, the elements are more scattered than without stride. For reference, we name this layout Stridden Convolution Resulting Format. As with the CRF format, we cannot directly use the output layout in consecutive layers. Thus, we propose an algorithm that translates from this layout to the RC, dubbed RT-SCRF-RC, described in Algorithm 13. In this algorithm, we use a formula to determine where the w_{out} elements for the stride output are placed and extract them iterative addition through bitmasks.

Algorithm 13 Result Transformation SCRF to RC

```
Input: $c_t^{SCRF} \in \mathbb{Z}_Q[x]/(x^N + 1)$ in SCRF format, $\mathbb{R}^{h \times w}$, $\mathbb{R}^{f_x \times f_y}$, $(s_x, s_y)$, $p$
Output: $c_t^{RC}$ in RC format
  function RT-SCRF-RC($c_t, h, w, f_x, f_y, s_x, s_y, p$)
    $h_{out} \leftarrow \left\lfloor \frac{h - f_x + 2 \cdot p + 1}{s_x} \right\rfloor$
    $w_{out} \leftarrow \left\lfloor \frac{w - f_y + 2 \cdot p + 1}{s_y} \right\rfloor$
    for $i \leftarrow 0, h_{out}$ do
      for $j \leftarrow 0, w_{out}$ do
        $bitmask[t]_i \leftarrow \{t = j \cdot s_y + (i \cdot w \cdot s_x)\}$
        $shift_i \leftarrow t - (i * h_{out} + j)$
        $c_t^{RC} = c_t^{RC} \oplus (c_t^{SCRF} \odot bitmask[t]_i) \ll shift_i$
      end for
    end for
    return $c_t^{RC}$
  end function
```

A.4 SIMD Padding

In many modern CNN architectures, it is common to chain multiple convolutional layers. While in the first convolutional layer it is possible to introduce padding in ‘cleartext’ (i.e., the data owner can add it at the end of the ciphertext before encryption), for consecutive ones, it is required to do it privately.

The proposed algorithm takes a bidimensional linearized ciphertext array $c_t \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(\mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$ and pads it uniformly with p zeroes on each dimension. It initially assumes a Row-Column format where the remaining entries of the vector are set to zero. The algorithm extracts each row, and computes the necessary shifting for a row, defined by the formula $shift_i = (w + 1) + (p \cdot i) \mid 0 \leq i < h$. The details are described in Algorithm 14. This algorithm outputs a Row-Column format directly usable by the convolution algorithm described in §A.1. Furthermore, it does not affect the Result Transformation algorithm.

Algorithm 14 2D Padding

```
Input: $c_t \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(\mathcal{M} \in \mathbb{R}^{h \times w}, p_k)$, in RC Format, $p$
Output: $c_t^{pad} \in \mathbb{Z}_Q[x]/(x^N + 1) = Enc(\mathcal{M}' \in \mathbb{R}^{h_{pad} \times w_{pad}}, p_k)$, in RC format
  function PADDING($c_t$, $p$)
      $h_{pad} \leftarrow h + 2 \cdot p$
      $w_{pad} \leftarrow w + 2 \cdot p$
      for $i \leftarrow 0, h$ do
          $bitmask[t]_i \leftarrow \{i \cdot w \leq t < w \cdot (i + 1)\}$
          $shift_i \leftarrow (w + 1) + (p \cdot i)$
          $c_t^{pad} = c_t^{pad} \oplus ((c_t \odot bitmask[t]_i) \gg shift_i)$
      end for
      return $c_t^{pad}$
  end function
```

Appendix B

Simplified Leveled Homomorphic Encryption Parametrization in Practice

This chapter simplifies the parametrization process and shows the challenges of performing HE parameter selection in practice. In the following sections, we describe practical approaches to the problem. First, Section B.1 describes a practical approach to the parametrization of Leveled Homomorphic Encryption schemes. Then, Section B.2 covers simple static parametrization, and Section B.3 covers an example of iterative parametrization.

B.1 Practical Leveled Homomorphic Encryption Parametrization

The parameters of an Leveled Homomorphic Encryption (LHE) scheme for a circuit C are determined by a tuple (N, Q, σ), where:

The polynomial degree (N) determines the size of the polynomial used for the encryption. At the same time, the size of the polynomial determines the number of ciphertext slots or the maximum number of plaintext elements that can be encrypted within a single ciphertext. It is always a power of 2 and is often depicted as logN or the bit count (i.e., $logN = log_2(N)$).

The polynomial modulus (Q) produce ring arithmetic in the polynomial. The modulus is often a product of several smaller coprime moduli q_i, which serve for modulo switching or rescaling. In a nutshell, whenever there is a multiplication operation, the ciphertext polynomial drops a modulus q_i, and in doing so, it reduces the scale of noise. In the same way, Q and q_i can be represented as a bit count $logQ$ and $logq_i$, respectively (i.e., $logQ = log_2(Q)$ and $logq_i = log_2(q_i)$).

Error distribution (σ) determines the scale of the error e introduced in ciphertexts for the security of LWE. It is usually set to 3.2 to reduce the noise introduced while guaranteeing security [Bra+13; MP13].

Choosing these parameters for the CKKS scheme consists of analyzing the circuit and extracting some parameters from it based on the following steps:

1. Defining the integer and decimal precision values (k_{int} and k_{dec} respectively). These are the values that determine the bit precision of the encryption.
2. Compute the multiplication depth ($O_{\mathfrak{D}}$) by looking for the maximum number of consecutive multiplications present in the circuit.
3. Select $n = O_{\mathfrak{D}} + 2$ moduli bit lengths $logq_i$, whose value is determined by:
   - The first and last moduli determine the integer precision for encryption and decryption. Thus, these are set to the sum of the integer and decimal scales: $logq_0 = logq_n = k_{int} + k_{dec}$.
   - The remaining moduli are set to preserve the noise scale on rescalings. Thus, these are set to the decimal scale: $logq_i = k_{dec} : 0 < i < n$.
4. Compute the total modulus bit count $logQ$ by summing the partial moduli: $logQ = \sum_{i=0}^{O_{\mathfrak{D}}^{N}} logq_i$.
5. Based on the HE Standard [Alb+18], determine the appropriate $logN$ according to the corresponding $logQ$.

B.2 Parametrizing Static Circuits

This section describes the parametrization of a simple static circuit described in Listing 2. For that we follow the steps described in Section B.1:

1. First, we select the integer and decimal scales. In this case, we select these to be 25 bits:
   - $k_{int} = 25$ and $k_{dec} = 25$
2. Then we compute the maximum multiplication depth ($O_{\mathfrak{D}}$). The maximum multiplication depth consists of three consecutive multiplications: a first multiplication to compute x^2, a second multiplication to compute x^4, and the final multiplication to compute $3 \cdot x^4$. The remaining operations all have smaller multiplication depths.

- $O_{\mathfrak{D}} = 3$

3. Then we select the chain of $n = O_{\mathfrak{D}} + 2$ moduli ($logq_i$). The first and last moduli ($logq_0$ and $logq_n$) equal the integer sum and decimal precision. The remaining moduli are equal

   - $n = O_{\mathfrak{D}} + 2 = 5$
   - $logq_0 = logq_n = k_{int} + k_{dec} = 25 + 25$
   - $logq_i = k_{dec} = 25$

4. Then we compute the addition of the different moduli ($logq_i$) into the polynomial modulus bit count $logQ$:

   - $logq_i = \{50, 25, 25, 25, 50\}$
   - $logQ = \sum_{i=0}^{O_{\mathfrak{D}}^{N}} logq_i = 175$

5. Finally, we can look up the values for the maximum budgets according to the HE Standard [Alb+18]. The HE Standard defines the maximum $logQ$ that can be introduced per $logN$. In this case, the maximum value of $logQ$ for $logN = 12$ is 111, thus not fitting. For $logN = 13$, the maximum budget $logQ = 220$ fits our values.

   - $logN = 13$

At the end of this procedure, we have produced a valid parametrization that guarantees the initial requirements we set for it. This example circuit shows a static behavior since the parameters do not influence it.

Listing 2: Example static circuit computing $f(x) = 3x^4 + 2x + 7$

```python
def f(x):
    #Repeated squaring reduces the multiplicative depth
    x2 = x * x
    x4 = x2 * x2
    return 3 * x4 + 2 * x + 7
```

B.3 Iterative Parameter Selection

Some circuits present dependencies between the parametrization and the circuit itself. The example shown in Listing 3 presents a dependency between the multiplication depth and the size of the ciphertext vector. This fact implies that the elections made in the initial phases may have to be reviewed. The parametrization works as follows:

1. First, we select the integer and decimal scales. In this case, we select these to be 25 bits:

   - $k_{int} = 25$ and $k_{dec} = 25$

2. Then we compute the maximum multiplication depth ($O_{\mathfrak{D}}$). As stated before, there is a dual relationship between the vector length defined by $logN$ and the multiplication depth in this circuit. We start by assuming the vector length equals $n = N$ $logN = 2$:

   - $O_{\mathfrak{D}} = 2$

3. Then we select the chain of $n = O_{\mathfrak{D}} + 2$ moduli ($logq_i$). The first and last moduli ($logq_0$ and $logq_n$) equal the integer sum and decimal precision. The remaining moduli are equal

   - $n = O_{\mathfrak{D}} + 2 = 5$
   - $logq_0 = logq_n = k_{int} + k_{dec} = 25 + 25$
   - $logq_i = k_{dec} = 25$

4. Then we compute the addition of the different moduli ($logq_i$) into the polynomial modulus bit count $logQ$:

   - $logq_i = \{50, 25, 25, 50\}$
   - $logQ = \sum_{i=0}^{O_{\mathfrak{D}}^{N}} logq_i = 150$

5. Finally, we can look up the values for the maximum budgets according to the HE Standard [Alb+18]. The HE Standard defines the maximum $logQ$ that can be introduced per $logN$. In this case, the maximum value of $logQ$ for $logN = 12$ is 111, thus not fitting. For $logN = 13$, the maximum budget $logQ = 220$ fits our values.

   - $logN = 13$

However, we see how our initial assumption $n = N$ and $logN = 2$ is no longer true, as we need a $logN = 13$ for our circuit to be secure. It implies reproducing the same steps again, updating our assumption to $N = 2^{13}$ and $logN = 13$:

1. First, we select the integer and decimal scales. In this case, we select these to be 25 bits:

   - $k_{int} = 25$ and $k_{dec} = 25$

2. Then we compute the maximum multiplication depth ($O_{\mathfrak{D}}$). Now we assume $N = 2^{13}$ and $logN = 13$.

   - $O_{\mathfrak{D}} = 13$

3. Then we select the chain of $n = O_{\mathfrak{D}} + 2$ moduli ($logq_i$). The first and last moduli ($logq_0$ and $logq_n$) equal the integer sum and decimal precision. The remaining moduli are equal
   - $n = O_{\mathfrak{D}} + 2 = 5$
   - $logq_0 = logq_n = k_{int} + k_{dec} = 25 + 25$
   - $logq_i = k_{dec} = 25$
4. Then we compute the addition of the different moduli ($logq_i$) into the polynomial modulus bit count $logQ$:
   - $logq_i = \{50, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 50\}$
   - $logQ = \sum_{i=0}^{O_{\mathfrak{D}}^N} logq_i = 375$
5. Finally, we can look up the values for the maximum budgets according to the HE Standard [Alb+18]. The HE Standard defines the maximum $logQ$ that can be introduced per $logN$. Our previous guess was for $logN = 13$, where the maximum budget $logQ = 220$. Again, this does not fit our values; we need to update to $logN = 14$ where the budget is $logQ = 440$.
   - $logN = 14$

Again, we spot the same problem, the circuit has updated our requirements, and our previous assumption is no longer true. We need $logN = 14$, and thus we need to repeat the whole process again:

1. First, we select the integer and decimal scales. In this case, we select these to be 25 bits:
   - $k_{int} = 25$ and $k_{dec} = 25$
2. Then we compute the maximum multiplication depth ($O_{\mathfrak{D}}$). Now we assume $N = 2^{14}$ and $logN = 14$.
   - $O_{\mathfrak{D}} = 14$
3. Then we select the chain of $n = O_{\mathfrak{D}} + 2$ moduli ($logq_i$). The first and last moduli ($logq_0$ and $logq_n$) equal the integer sum and decimal precision. The remaining moduli are equal
   - $n = O_{\mathfrak{D}} + 2 = 5$
   - $logq_0 = logq_n = k_{int} + k_{dec} = 25 + 25$
   - $logq_i = k_{dec} = 25$
4. Then we compute the addition of the different moduli ($logq_i$) into the polynomial modulus bit count $logQ$:

- $logq_i = \{50, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 50\}$
- $logQ = \sum_{i=0}^{O^N_{\mathcal{D}}} logq_i = 400$

5. Finally, we can look up the values for the maximum budgets according to the HE Standard [Alb+18]. The HE Standard defines the maximum $logQ$ that can be introduced per $logN$. Our previous guess was for $logN = 13$, where the maximum budget $logQ = 220$. Again, this does not fit our values; we need to update to $logN = 14$ where the budget is $logQ = 440$.

   - $logN = 14$

This time, our assumptions are correct, and we can assume a correct parameterization. However, this presents multiple challenges that are difficult to sort out in more complex circuits.

Listing 3: Example static circuit computing the multiplication of entries of a vector of $n = 4$ entries.

```python
def vector_mult(x):
    y = x
    logN = math.log2(N)
    for i in range(logN):
        y *= y << i
    return y
```

Appendix C

HEFactory Internals

This section describes the overall internal and engineering workflow of the *HEFactory* described in Chapter 6. For that, we describe the main classes that describe the workflow of the compiler. First, we depict the symbolic execution phase in Section C.1. Then, we detail the interactions to work with vectorized algorithms in Section C.2. Finally, Section C.3 describes the Deep Learning interactions to adapt Tensorflow models.

C.1 Symbolic Execution

HEFactory aims to provide a fully integrated Python library environment. As a result, it eliminates the need for a separate compilation process and instead integrates it within Python. To achieve this, *HEFactory* leverages the lexing and parsing capabilities of Python. Additionally, it utilizes various language features and defines two structures for compilation, namely, the compilation unit `CGManager` and the symbols `CGSym`.

The compilation environment of *HEFactory* enables the storage of the symbolic execution trace and facilitates the production of a binary for later execution. Upon initialization, it creates an empty `AST` class. To simplify resource behavior emulation, the `CGManager` executes compilation upon resource exit using the Python `with` clause. While *HEFactory* tracks relations between symbols, only operations on `CGSym` are considered relevant, not all Python variables.

In *HEFactory*, symbols are represented by the `CGSym` class, created using a `CGManager`. The `CGSym` class overrides the behavior of all operations that can be executed on a variable in the Python programming language, using operation overriding to track operations. It also introduces syntax and semantic analysis to ensure that behaviors are permitted within Homomorphic Encryption (HE). Upon declaration of a `CGSym`, an initial value can be provided, which is used as a template for determining HE parameters.

This usage simplifies the problem of parsing operation order, and the same strict operation order provided by Python is introduced in the compiler.

After each operation, the result is added to the `AST`, a list of `Expr`. Each expression is represented by a four-element tuple that includes an operation code, an output symbol, and two operands, one of which is always a ciphertext. Upon the completion of all operations, various optimizations are performed by the compiler, including the introduction of `free` expressions, which are added after the last use of a symbol.

Once the optimizations have been performed, the compiler produces a binary using the `BinGen` class, which creates three files: a data file, a parameters file, and a circuit file. The data file has a map-like structure that assigns an identifier to each input symbol as a key and then introduces a value, which can be encrypted later. The parameters file is a JSON-like file that stores all the parameters for the circuit. In contrast, the circuit file includes a header that provides information about the circuit, such as the number of operations. Each operation is then included in the file, and the parameters are standardized after parsing the instruction type.

C.2 Delayed Execution

Delayed execution is a core component of *HEFactory*, as it allows for the inclusion of vectorized operations and Single Instruction Multiple Data (SIMD) packing. It comprises two distinct classes, namely `DelayedExpr` and `DelayedExecutor`. Delayed expressions are employed whenever an operation's execution requires input from the HE parameters. During operation execution, an instance of `DelayedExpr` is appended to the `AST` expression list. This unique instruction exposes an output `CGSym`, whose characteristics align with the expected output had the `DelayedExpr` been executed. This capability enables the selection of parameters, assuming that the `AST` expansion will not alter these parameters. Once the parameters have been selected, the `DelayedExecutor` expands the `DelayedExpr` into equivalent code.

HEFactory incorporates preprogrammed vectorized primitives such as matrix multiplication, convolution, average pooling, and vector aggregation to facilitate the implementation of Privacy-Preserving Deep Learning (PPDL).

C.3 Deep Learning Adaptation

HEFactory includes multiple classes that facilitate the adaptation of base DL models for PPDL usage. Specifically, it features a Tensorflow `Model` parser class that generates an equivalent encrypted circuit. This process is facilitated by the wrapper classes, namely `Conv2D`, `AvgPooling2D`, `Dense`, and `Activation`, which adapt the layers of the trained Neural Network (NN). To support these functions, the `Model` class of *Dahut* offers all the essential infrastructure required. Specifically, for inference purposes, the `Model` class can be invoked with a `CGArray`, which represents the `CGSym` for vectors in an abstract form. The execution process can then be performed automatically without requiring further input from the user.

Bibliography

[Aba+16a] Martin Abadi et al. "Deep learning with differential privacy". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 308–318.

[Aba+16b] Martıén Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint:1603.04467* (2016).

[Agr+19] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. "QUOTIENT: two-party secure neural network training and prediction". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1231–1247.

[Aha+23] Ehud Aharoni et al. "HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data". In: *Proceedings on Privacy Enhancing Technologies (PETS)*. 2023.

[Alb+18] Martin Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018.

[Arc+19] David W Archer et al. "Ramparts: A programmer-friendly system for building homomorphic encryption applications". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2019, pp. 57–68.

[Azr+19] Monir Azraoui et al. "SoK: Cryptography for Neural Networks". In: *IFIP International Summer School on Privacy and Identity Management*. Springer. 2019, pp. 63–81.

[Bal+17] David Balduzzi, Marcus Frean, Lennox Leary, JP Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. "The shattered gradients problem: If resnets are the answer, then what is the question?" In: *International Conference on Machine Learning*. PMLR. 2017, pp. 342–350.

[Bea91] Donald Beaver. "Efficient multiparty protocols using circuit randomization". In: *Annual International Cryptology Conference*. Springer. 1991, pp. 420–432.

[Ber+19] Daniel S Berman, Anna L Buczak, Jeffrey S Chavis, and Cherita L Corbett. "A survey of deep learning methods for cyber security". In: *Information* 10.4 (2019), p. 122.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. “Non-Interactive Zero-Knowledge and Its Applications”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC ’88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 103–112. DOI: 10.1145/62212.62222. URL: https://doi.org/10.1145/62212.62222.

[BG69] Richard H Bartels and Gene H Golub. “The simplex method of linear programming using LU decomposition”. In: *Communications of the ACM* 12.5 (1969), pp. 266–268.

[BGE19] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. “Low Latency Privacy Preserving Inference”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 812–821. URL: https://proceedings.mlr.press/v97/brutzkus19a.html.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. “Packed ciphertexts in LWE-based homomorphic encryption”. In: *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*. Springer. 2013, pp. 1–13.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC ’88. Chicago, Illinois, USA: ACM, 1988, pp. 1–10. DOI: 10.1145/62212.62213. URL: https://doi.org/10.1145/62212.62213.

[Bla79] George Robert Blakley. “Safeguarding cryptographic keys”. In: *1979 International Workshop on Managing Requirements Knowledge (MARK)*. IEEE. 1979, pp. 313–318.

[Blu83] Manuel Blum. “Coin flipping by telephone a protocol for solving impossible problems”. In: *ACM SIGACT News* 15.1 (1983), pp. 23–27.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. “The round complexity of secure protocols”. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 1990, pp. 503–513.

[Boe+19a] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. “NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data”. In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC’19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 45–56. DOI: 10.1145/3338469.3358944. URL: https://doi.org/10.1145/3338469.3358944.

[Boe+19b] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. “NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data”. In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. CF ’19. Alghero, Italy: Association for Computing Machinery, 2019, pp. 3–13. DOI: 10.1145/3310273.3323047. URL: https://doi.org/10.1145/3310273.3323047.

[Boe+20] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. “MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference”. In: *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. PPMLP’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 43–45. DOI: 10.1145/3411501.3419425. URL: https://doi.org/10.1145/3411501.3419425.

[Bon+19] Keith Bonawitz et al. “Towards federated learning at scale: System design”. In: *arXiv preprint:1902.01046* (2019).

[Bou+18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. “Fast homomorphic evaluation of deep discretized neural networks”. In: *Annual International Cryptology Conference*. Springer. 2018, pp. 483–512.

[Bou+20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. “Chimera: Combining ring-lwe-based fully homomorphic encryption schemes”. In: *Journal of Mathematical Cryptology* 14.1 (2020), pp. 316–338.

[BR18] Battista Biggio and Fabio Roli. “Wild patterns: Ten years after the rise of adversarial machine learning”. In: *Pattern Recognition* 84 (2018), pp. 317–331.

[Bra+13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. “Classical hardness of learning with errors”. In: *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 2013, pp. 575–584.

[BS18] Nick Barlow and Oliver Strickson. *SHEEP is a Homomorphic Encryption Evaluation Framework*. https://github.com/alan-turing-institute/SHEEP. 2018.

[Bya+20] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. “FLASH: fast and robust framework for privacy-preserving machine learning”. In: *Proceedings on Privacy Enhancing Technologies* 2020.2 (2020), pp. 459–480.

[Cab21] José Cabrero-Holgueras. “Usable Homomorphic Encryption for Private Telemedicine on the Cloud”. In: *Italian Telemedicine Society Conference (SIT)* (2021).

[CD16] Victor Costan and Srinivas Devadas. “Intel SGX Explained.” In: *IACR Cryptol. ePrint Arch.* 2016.86 (2016), pp. 1–118.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. “Armadillo: a compilation chain for privacy preserving applications”. In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 2015, pp. 13–19.

[Cha+17a] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. “EzPC: programmable, efficient, and scalable secure two-party computation for machine learning”. In: *ePrint Report* 1109 (2017).

[Cha+17b] Melissa Chase, Ran Gilad-Bachrach, Kim Laine, Kristin E Lauter, and Peter Rindal. “Private Collaborative Neural Network Learning.” In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 762.

[Cha+19] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. “Astra: High throughput 3pc over rings with application to secure prediction”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 2019, pp. 81–92.

[Che+17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. “Homomorphic encryption for arithmetic of approximate numbers”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 409–437.

[Che+19a] Huili Chen, Rosario Cammarota, Felipe Valencia, and Francesco Regazzoni. “PlaidML-HE: Acceleration of Deep Learning Kernels to Compute on Encrypted Data”. In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE. 2019, pp. 333–336.

[Che+19b] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. *Numerical Method for Comparison on Homomorphically Encrypted Numbers*. Cryptology ePrint Archive, Paper 2019/417. https://eprint.iacr.org/2019/417. 2019. URL: https://eprint.iacr.org/2019/417.

[Che+20] Huili Chen et al. “Developing privacy-preserving AI systems: the lessons learned”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–4.

[Chi+16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. “Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds”. In: *international conference on the theory and application of cryptology and information security*. Springer. 2016, pp. 3–33.

[Chi+20a] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. “TFHE: fast fully homomorphic encryption over the torus”. In: *Journal of Cryptology* 33.1 (2020), pp. 34–91.

[Chi+20b] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. “CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE”. In: *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. Vol. 15. 2020.

[Cho+15] François Chollet et al. *Keras*. `https://github.com/fchollet/keras`. 2015.

[Cho+18] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. “Faster cryptonets: Leveraging sparsity for real-world encrypted inference”. In: *arXiv preprint:1811.09953* (2018).

[Cho+85] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. “Verifiable secret sharing and achieving simultaneity in the presence of faults”. In: *26th Annual Symposium on Foundations of Computer Science*. IEEE. 1985, pp. 383–395.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. “Programmable bootstrapping enables efficient homomorphic inference of deep neural networks”. In: *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be’er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer. 2021, pp. 1–19.

[CP] José Cabrero-Holgueras and Sergio Pastrana. “Towards Realistic Privacy-Preserving Deep Learning over Encrypted Medical Data”. In: *Frontiers in Cardiovascular Medicine* 10 (), p. 641.

[CP21a] José Cabrero-Holgueras and Sergio Pastrana. “A methodology for large-scale identification of related accounts in underground forums”. In: *Computers & Security* 111 (2021), p. 102489.

[CP21b] José Cabrero-Holgueras and Sergio Pastrana. “SoK: Privacy-Preserving Computation Techniques for Deep Learning”. In: *Proceedings on Privacy Enhancing Technologies* 2021.4 (2021), pp. 139–162.

[CP22] José Cabrero-Holgueras and Sergio Pastrana. *Towards Realistic Privacy-Preserving Deep Learning Inference Over Encrypted Data*. http://dx.doi.org/10.2139/ssrn.4140183. 2022.

[CP23a] José Cabrero-Holgueras and Sergio Pastrana. "HEFactory: A Symbolic Execution Compiler for Privacy-Preserving Deep Learning with Homomorphic Encryption". In: *Software X* (2023).

[CP23b] José Cabrero-Holgueras and Sergio Pastrana. "Towards Automated Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming". In: *Expert Systems with Applications* (2023).

[CP23c] José Cabrero-Holgueras and Sergio Pastrana. "Towards Automated Homomorphic Encryption Parameter Selection with Fuzzy Logic and Linear Programming". In: *arXiv preprint arXiv:2302.08930* (2023).

[Csá+01] Balázs Csanád Csáji et al. "Approximation with artificial neural networks". In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24.48 (2001), p. 7.

[Cyp+18] Scott Cyphers et al. "Intel ngraph: An intermediate representation, compiler, and executor for deep learning". In: *arXiv preprint:1801.08058* (2018).

[Dah+18] Morten Dahl et al. "Private machine learning in tensorflow using secure computation". In: *arXiv preprint:1810.08130* (2018).

[Dam+12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. "Multiparty computation from somewhat homomorphic encryption". In: *Annual Cryptology Conference*. Springer. 2012, pp. 643–662.

[Dat+19] Roshan Dathathri et al. "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 142–156. DOI: 10.1145/3314221.3314628. URL: https://doi.org/10.1145/3314221.3314628.

[Dat+20] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 546–561. DOI: 10.1145/3385412.3386023. URL: https://doi.org/10.1145/3385412.3386023.

[DEK20] Anders Dalskov, Daniel Escudero, and Marcel Keller. "Secure evaluation of quantized neural networks". In: *Proceedings on Privacy Enhancing Technologies* 2020.4 (2020), pp. 355–375.

[DH22] Whitfield Diffie and Martin E Hellman. "New directions in cryptography". In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 2022, pp. 365–390.

[DM15] Léo Ducas and Daniele Micciancio. “FHEW: bootstrapping homomorphic encryption in less than a second”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 617–640.

[DMY16] Dung Hoang Duong, Pradeep Kumar Mishra, and Masaya Yasuda. “Efficient secure matrix multiplication over LWE-based homomorphic encryption”. In: *Tatra mountains mathematical publications* 67.1 (2016), pp. 69–83.

[DP19] Damien Desfontaines and Balázs Pejó. “SoK: Differential Privacies”. In: *CoRR* abs/1906.01337 (2019). arXiv: 1906.01337. URL: http://arxiv.org/abs/1906.01337.

[DR+14] Cynthia Dwork, Aaron Roth, et al. “The algorithmic foundations of differential privacy.” In: *Foundations and Trends in Theoretical Computer Science* 9.3-4 (2014), pp. 211–407.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. “ABY-A framework for efficient mixed-protocol secure two-party computation.” In: *NDSS*. 2015.

[Du+17] Zhao-Hui Du et al. “Secure encrypted virtualization is unsecure”. In: *arXiv preprint:1712.05090* (2017).

[ElG85] Taher ElGamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. “Rappor: Randomized aggregatable privacy-preserving ordinal response”. In: *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 2014, pp. 1054–1067.

[Fel87] Paul Feldman. “A practical scheme for non-interactive verifiable secret sharing”. In: *28th Annual Symposium on Foundations of Computer Science*. IEEE. 1987, pp. 427–438.

[FJR15] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. “Model inversion attacks that exploit confidence information and basic countermeasures”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 1322–1333.

[FM89] Paul Feldman and Silvio Micali. “An optimal probabilistic algorithm for synchronous byzantine agreement”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1989, pp. 341–378.

[FV12] Junfeng Fan and Frederik Vercauteren. “Somewhat Practical Fully Homomorphic Encryption.” In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[Gen09] Craig Gentry. “A fully homomorphic encryption scheme”. crypto.stanford.edu/craig. PhD thesis. Stanford University, 2009.

[Gil+16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy”. In: *International Conference on Machine Learning*. 2016, pp. 201–210.

[GLN12] Thore Graepel, Kristin Lauter, and Michael Naehrig. “ML confidential: Machine learning on encrypted data”. In: *International Conference on Information Security and Cryptology*. Springer. 2012, pp. 1–21.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The knowledge complexity of interactive proof systems”. In: *SIAM Journal on computing* 18.1 (1989), pp. 186–208.

[GMT23] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. “SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks”. In: *Proceedings on Privacy Enhancing Technologies* 2023.3 (July 2023), pp. 1–20.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. “How to Play ANY Mental Game”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC ’87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 218–229. DOI: 10.1145/28395.28420. URL: https://doi.org/10.1145/28395.28420.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. “Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems”. In: *Journal of the ACM (JACM)* 38.3 (1991), pp. 690–728.

[GNS21] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. “Rinocchio: SNARKs for ring arithmetic”. In: *Cryptology ePrint Archive* (2021).

[Goo+14] Ian Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[Goo+16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[GRR98] Rosario Gennaro, Michael O Rabin, and Tal Rabin. “Simplified VSS and fast-track multiparty computations with applications to threshold cryptography”. In: *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*. 1998, pp. 101–111.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based”. In: *Annual Cryptology Conference*. Springer. 2013, pp. 75–92.

[GV88] Oded Goldrcich and Ronen Vainish. "How to Solve any Protocol Problem - An Efficiency Improvement (Extended Abstract)". In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 73–86.

[Has+95] Mohamad H Hassoun et al. *Fundamentals of artificial neural networks*. MIT press, 1995.

[HCS18] Nick Hynes, Raymond Cheng, and Dawn Song. "Efficient deep learning on multi-source private data". In: *arXiv preprint:1807.06689* (2018).

[Hen+10] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. "TASTY: tool for automating secure two-party computations". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 451–462.

[Hes+18] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. "Privacy-preserving machine learning as a service". In: *Proceedings on Privacy Enhancing Technologies* 2018.3 (2018), pp. 123–142.

[HPW17] James B Heaton, Nick G Polson, and Jan Hendrik Witte. "Deep learning for finance: deep portfolios". In: *Applied Stochastic Models in Business and Industry* 33.1 (2017), pp. 3–12.

[HS14a] Shai Halevi and Victor Shoup. "Algorithms in helib". In: *Annual Cryptology Conference*. Springer. 2014, pp. 554–571.

[HS14b] Shai Halevi and Victor Shoup. *Bootstrapping for HElib*. Cryptology ePrint Archive, Paper 2014/873. https://eprint.iacr.org/2014/873. 2014. URL: https://eprint.iacr.org/2014/873.

[HS18] Nathan O Hodas and Panos Stinis. "Doing the impossible: Why neural networks can be trained at all". In: *Frontiers in psychology* 9 (2018), p. 1185.

[HTG17] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. "Cryptodl: Deep neural networks over encrypted data". In: *arXiv preprint:1711.05189* (2017).

[Hun+18] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. "Chiron: Privacy-preserving Machine Learning as a Service". In: *arXiv preprint:1803.05961* (2018).

[Hus+20] Siam Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. "TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit". In: *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. 2020, pp. 65–67.

[Huy20a] Daniel Huynh. *CKKS EXPLAINED, PART 2: FULL ENCODING AND DECODING*. Accessed on March, 30 2023. 2020. URL: https://blog.openmined.org/ckks-explained-part-2-ckks-encoding-and-decoding/.

[Huy20b] Daniel Huynh. *CKKS EXPLAINED, PART 3: ENCRYPTION AND DECRYPTION*. Accessed on March, 30 2023. 2020. URL: https://blog.openmined.org/ckks-explained-part-3-encryption-and-decryption/.

[Huy20c] Daniel Huynh. *CKKS EXPLAINED, PART 4: MULTIPLICATION AND RELINEARIZATION*. Accessed on March, 30 2023. 2020. URL: https://blog.openmined.org/ckks-explained-part-4-multiplication-and-relinearization/.

[Huy20d] Daniel Huynh. *CKKS EXPLAINED, PART 5: RESCALING*. Accessed on March, 30 2023. 2020. URL: https://blog.openmined.org/ckks-explained-part-5-rescaling/.

[Huy20e] Daniel Huynh. *CKKS EXPLAINED: PART 1, VANILLA ENCODING AND DECODING*. Accessed on March, 30 2023. 2020. URL: https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/.

[Irv+19] Jeremy Irvin et al. *CheXpert: A Large Chest Radiograph Dataset with Uncertainty Labels and Expert Comparison*. 2019. arXiv: 1901.07031 [cs.CV].

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[ISY20] Takumi Ishiyama, Takuya Suzuki, and Hayato Yamana. “Highly accurate CNN inference using approximate activation functions over homomorphic encryption”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 3989–3995.

[Jia+18] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. “Secure Outsourced Matrix Computation and Application to Neural Networks”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1209–1222. DOI: 10.1145/3243734.3243837. URL: https://doi.org/10.1145/3243734.3243837.

[JLS20] Aayush Jain, Huijia Lin, and Amit Sahai. “Indistinguishability obfuscation from well-founded assumptions”. In: *arXiv preprint:2008.09317* (2020).

[Jul+17] D Julkowska et al. “The importance of international collaboration for rare diseases research: a European perspective”. In: *Gene therapy* 24.9 (2017), pp. 562–571.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1651–1669. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[JYS18] James Jordon, Jinsung Yoon, and Mihaela van der Schaar. "PATE-GAN: Generating synthetic data with differential privacy guarantees". In: *International Conference on Learning Representations*. 2018.

[Kai+19] Peter Kairouz et al. "Advances and open problems in federated learning". In: *arXiv preprint:1912.04977* (2019).

[Kai+20] Georgios A Kaissis, Marcus R Makowski, Daniel Rückert, and Rickmer F Braren. "Secure, privacy-preserving and federated machine learning in medical imaging". In: *Nature Machine Intelligence* (2020), pp. 1–7.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. "FleXOR: Flexible garbling for XOR gates that beats free-XOR". In: *Annual Cryptology Conference*. Springer. 2014, pp. 440–457.

[Kon+16] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. "Federated optimization: Distributed machine learning for on-device intelligence". In: *arXiv preprint:1610.02527* (2016).

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: faster malicious arithmetic secure computation with oblivious transfer". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 830–842.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: Making SPDZ great again". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 158–189.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. "AMD memory encryption". In: *White paper* (2016).

[KS08] Vladimir Kolesnikov and Thomas Schneider. "Improved garbled circuit: Free XOR gates and applications". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2008, pp. 486–498.

[Kum+20a] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. "Cryptflow: Secure tensorflow inference". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 336–353.

[Kum+20b] Ram Shankar Siva Kumar et al. "Adversarial machine learning-industry perspectives". In: *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2020, pp. 69–75.

[Lat+21] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. doi: 10.1109/CGO51591.2021.9370308.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[LeC+12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[LeC+89] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[LeC+98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Lin20] Yehuda Lindell. “Secure Multiparty Computation”. In: *Commun. ACM* 64.1 (Dec. 2020), pp. 86–96. doi: 10.1145/3387108. url: https://doi.org/10.1145/3387108.

[Liu+17] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. “Oblivious neural network predictions via minionn transformations”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 619–631.

[LLV07] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. “t-closeness: Privacy beyond k-anonymity and l-diversity”. In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 106–115.

[Mac+07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. “l-diversity: Privacy beyond k-anonymity”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1.1 (2007), 3–es.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “Cachezoom: How SGX amplifies the power of cache attacks”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 69–90.

[Mis+20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. “DELPHI: A cryptographic inference service for neural networks”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[Mou+20] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. “Lattigo: A multiparty homomorphic encryption library in go”. In: *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. CONF. 2020, pp. 64–70.

[MP13] Daniele Micciancio and Chris Peikert. “Hardness of SIS and LWE with small parameters”. In: *Annual Cryptology Conference*. Springer. 2013, pp. 21–39.

[MR18] Payman Mohassel and Peter Rindal. “mixed protocol framework for machine learning”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 35–52.

[Mur20] Kazuo Murota. “Linear programming”. In: *Computer Vision: A Reference Guide*. Springer, 2020, pp. 1–7.

[MZ17] Payman Mohassel and Yupeng Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2017, pp. 19–38.

[Nan+19] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. “Towards deep neural network training on encrypted data”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2019.

[NH10] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *ICML*. 2010.

[NP99] Moni Naor and Benny Pinkas. “Oblivious transfer and polynomial evaluation”. In: *Proceedings of the 31st ACM Symposium on Theory of Computing*. 1999, pp. 245–254.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. “Privacy preserving auctions and mechanism design”. In: *Proceedings of the 1st ACM conference on Electronic commerce*. 1999, pp. 129–139.

[Nwa+18] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. arXiv: 1811.03378 [cs.LG].

[Ode09] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. 2009.

[Ohr+16] Olga Ohrimenko et al. “Oblivious multi-party machine learning on trusted processors”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 619–636.

[Pai99] Pascal Paillier. “Public-key cryptosystems based on composite degree residuosity classes”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1592. Springer Verlag, 1999, pp. 223–238. DOI: 10.1007/3-540-48910-X_16. URL: https://link.springer.com/chapter/10.1007/3-540-48910-X%7B%5C_%7D16.

[Pap+16] Nicolas Papernot, Martıén Abadi, Ulfar Erlingsson, Ian Goodfellow, and Kunal Talwar. “Semi-supervised knowledge transfer for deep learning from private training data”. In: *arXiv preprint:1610.05755* (2016).

[Pap+18a] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P Wellman. “SoK: Security and privacy in machine learning”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 399–414.

[Pap+18b] Nicolas Papernot, Shuang Song, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Úlfar Erlingsson. “Scalable Private Learning with PATE”. In: *arXiv preprint:1802.08908* (2018).

[Par+13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. *Pinocchio: Nearly Practical Verifiable Computation*. Cryptology ePrint Archive, Paper 2013/279. https://eprint.iacr.org/2013/279. 2013. URL: https://eprint.iacr.org/2013/279.

[Pas+17] Adam Paszke et al. “Automatic Differentiation in PyTorch”. In: *NIPS 2017 Workshop on Autodiff*. Long Beach, California, USA, 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ.

[Pin+09] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. “Secure two-party computation is practical”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2009, pp. 250–267.

[PRR17] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. “PALISADE lattice cryptography library user manual”. In: *Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep* (2017).

[PW00] Florian A Potra and Stephen J Wright. “Interior-point methods”. In: *Journal of computational and applied mathematics* 124.1-2 (2000), pp. 281–302.

[Rab05] Michael O Rabin. “How To Exchange Secrets with Oblivious Transfer.” In: *IACR Cryptol. ePrint Arch.* 2005.187 (2005).

[Reg09] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.

[Res20] Microsoft Research. *Microsoft SEAL (release 3.6)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Nov. 2020.

[Ria+18] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. “Chameleon: A hybrid secure computation framework for machine learning applications”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, pp. 707–721.

[Ria+19] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. “XONN: XNOR-based Oblivious Deep Neural Network Inference”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1501–1518.

[RRK18] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. “Deepsecure: Scalable provably-secure deep learning”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.

[RRK19] M Sadegh Riazi, Bita Darvish Rouani, and Farinaz Koushanfar. “Deep learning on private data”. In: *IEEE Security & Privacy* 17.6 (2019), pp. 54–63.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[Ryf+18] Theo Ryffel et al. “A generic framework for privacy preserving deep learning”. In: *arXiv preprint:1811.04017* (2018).

[San+18] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. “TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service”. In: *International Conference on Machine Learning*. 2018, pp. 4490–4499.

[Sav+20] Sinem Sav et al. “POSEIDON: Privacy-Preserving Federated Neural Network Learning”. In: *arXiv preprint:2009.00349* (2020).

[Sch89] Claus-Peter Schnorr. “Efficient identification and signatures for smart cards”. In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 239–252.

[Sen15] Jaydip Sen. “Security and privacy issues in cloud computing”. In: *Cloud technology: concepts, methodologies, tools, and applications*. IGI global, 2015, pp. 1585–1630.

[Sha+17] Noam Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017. DOI: 10.48550/ARXIV.1701.06538. URL: https://arxiv.org/abs/1701.06538.

[Sha79] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[Sho+17] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. “Membership inference attacks against machine learning models”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 3–18.

[Sin+22] Saurabh Singh, Shailendra Rathore, Osama Alfarraj, Amr Tolba, and Byungun Yoon. “A framework for privacy-preservation of IoT healthcare data using Federated Learning and blockchain technology”. In: *Future Generation Computer Systems* 129 (2022), pp. 380–388.

[So+19] Jinhyun So, Basak Guler, A Salman Avestimehr, and Payman Mohassel. “CodedPrivateML: A Fast and Privacy-Preserving Framework for Distributed Machine Learning”. In: *arXiv preprint:1902.00641* (2019).

[Son+15] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. “Tinygarble: Highly compressed and scalable sequential garbled circuits”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 411–428.

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[SS08] Ahmad-Reza Sadeghi and Thomas Schneider. “Generalized universal circuits for secure evaluation of private functions with application to data classification”. In: *International Conference on Information Security and Cryptology*. Springer. 2008, pp. 336–353.

[SS15] Reza Shokri and Vitaly Shmatikov. “Privacy-preserving deep learning”. In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 2015, pp. 1310–1321.

[SS19] Nils Strodthoff and Claas Strodthoff. “Detecting and interpreting myocardial infarction using fully convolutional neural networks”. In: *Physiological measurement* 40.1 (2019), p. 015001.

[Swe02] Latanya Sweeney. “k-anonymity: A model for protecting privacy”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10.05 (2002), pp. 557–570.

[Tan+20] Harry Chandra Tanuwidjaja, Rakyong Choi, Seunggeun Baek, and Kwangjo Kim. “Privacy-Preserving Deep Learning on Machine Learning as a Service—a Comprehensive Survey”. In: *IEEE Access* 8 (2020), pp. 167425–167447.

[Tao+23] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.

[TB18] Florian Tramer and Dan Boneh. “Slalom: Fast, verifiable and private execution of neural networks in trusted hardware”. In: *arXiv preprint:1806.03287* (2018).

[Ton+18] Samet Tonyali, Kemal Akkaya, Nico Saputro, A Selcuk Uluagac, and Mehrdad Nojoumian. “Privacy-preserving protocols for secure and reliable data aggregation in IoT-enabled smart metering systems”. In: *Future Generation Computer Systems* 78 (2018), pp. 547–557.

[Top19] Eric J Topol. “High-performance medicine: the convergence of human and artificial intelligence”. In: *Nature medicine* 25.1 (2019), pp. 44–56.

[Tra+16] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. “Stealing machine learning models via prediction apis”. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 601–618.

[Van16] Jake VanderPlas. *Python data science handbook: Essential tools for working with data*. " O’Reilly Media, Inc.", 2016.

[VB17] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. 1st. Springer Publishing Company, Incorporated, 2017.

[VC04] Jaideep Vaidya and Chris Clifton. “Privacy-preserving data mining: Why, how, and when”. In: *IEEE Security & Privacy* 2.6 (2004), pp. 19–27.

[Via+23] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. *HECO: Fully Homomorphic Encryption Compiler*. 2023. arXiv: 2202.01649 [cs.CR].

[VJH21] A. Viand, P. Jattke, and A. Hithnawi. “SoK: Fully Homomorphic Encryption Compilers”. In: *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1166–1182. DOI: 10.1109/SP40001.2021.00068. URL: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00068.

[VS02] Vinod Valsalam and Anthony Skjellum. “A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels”. In: *Concurrency and Computation: Practice and Experience* 14.10 (2002), pp. 805–839.

[WGC18] Sameer Wagh, Divya Gupta, and Nishanth Chandran. “SecureNN: Efficient and Private Neural Network Training.” In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 442.

[WH12] David Wu and Jacob Haven. “Using homomorphic encryption for large scale statistical analysis”. In: *FHE-SI-Report, Univ. Stanford, Tech. Rep. TR-dwu4* (2012).

[Win08] Johannes Winter. “Trusted computing building blocks for embedded linux-based ARM trustzone platforms”. In: *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. 2008, pp. 21–30.

[WL11] Ting Wang and Ling Liu. “Output privacy in data mining”. In: *ACM Transactions on Database Systems* 36.1 (2011), pp. 1–34.

[Xie+14] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig. “Crypto-nets: Neural networks over encrypted data”. In: *arXiv preprint:1412.6181* (2014).

[Yao86] Andrew Chi-Chih Yao. “How to generate and exchange secrets”. In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (1986), pp. 162–167.

[YBS20] Tao Yu, Eugene Bagdasaryan, and Vitaly Shmatikov. “Salvaging federated learning by local adaptation”. In: *arXiv preprint:2002.04758* (2020).

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. “Two halves make a whole”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 220–250.