University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Lattice-Based Cryptography for Privacy Preserving Machine Learning

Master's Thesis in Computer Science
by

Ivar Walskaar and Minh Christian Tran

Internal Supervisors

Supervisor 1

Supervisor 2

External Supervisors

External Supervisor 1

External Supervisor 2

Reviewers

Reviewer1

Reviewer2

July 12, 2023

'Everything is possible in your life when you believe. When you believe, everything is possible. You have two hands like me. Everything is possible. Go. Go. And take it, whatever you want to do'

Yoel Romero

Abstract

The digitization of healthcare data has presented a pressing need to address privacy concerns within the realm of machine learning for healthcare institutions. One promising solution is Federated Learning (FL), which enables collaborative training of deep machine learning models among medical institutions by sharing model parameters instead of raw data. This study focuses on enhancing an existing privacy-preserving federated learning algorithm for medical data through the utilization of homomorphic encryption, building upon prior research.

In contrast to the previous paper this work is based upon by Wibawa, using a single key for homomorphic encryption, our proposed solution is a practical implementation of a preprint by Ma Jing et. al. with a proposed encryption scheme (xMK-CKKS) for implementing multi-key homomorphic encryption. For this, our work first involves modifying a simple "ring learning with error" RLWE scheme. We then fork a popular FL framework for python where we integrate our own communication process with protocol buffers before we locate and modify the library's existing training loop in order to further enhance the security of model updates with the multi-key homomorphic encryption scheme. Our experimental evaluations validate that despite these modifications, our proposed framework maintains robust model performance, as demonstrated by consistent metrics including validation accuracy, precision, f1-score, and recall.

Acknowledgements

First we would like to extend our gratitude to Sander Låstad Olsen, a fellow student and friend, whose knowledge of Flower gave us a solid foundation to work on, and the tools he introduced us to significantly helped our debugging efforts in modifying Flower's source code. Furthermore, we extend deep thanks to our supervising professor Ferhat Özgur Catak. His guidance during our weekly meetings helped shape our work and navigate us through challenging paths. We are deeply thankful for their invaluable contributions.

Contents

List of Figures

List of Tables

Abbreviations

CCPA	**C**alifornia **C**onsumer **P**rivacy **A**ct
CNN	**C**onvolutional **N**eural **N**etwork
FL	**F**ederated **L**earning
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
gRPC	**g**oogle **R**emote **P**rocedure **C**alls
HE	**H**omomorphic **E**ncryption
MK-HE	**M**ulti-**K**ey **H**omomorphic **E**ncryption
SHE	**S**omewhat **H**omomorphic **E**ncryption
RLWE	**R**ing **L**earning **W**ith **E**rror

Chapter 1

Introduction

1.1 Motivation

The ongoing COVID-19 pandemic has highlighted the importance of the data privacy and security in medical research. As governments and healthcare organizations around the world race to collect and analyze data on the virus, concerns about the misuse and unauthorized access of sensitive patient information have become increasingly pressing. [1]

One promising approach to this problem is federated learning, which allows multiple parties to collaboratively train a machine learning model without sharing their raw data. However, even in federated learning, there is still a risk of data leakage and privacy violations, particularly when the data is transmitted between parties.

To address this challenge, multi-key homomorphic encryption (MK-HE) as been proposed as a potential solution. Unlike conventional HE, MK-HE allows computation on encrypted data with multiple private keys.

This work expands upon a previous endeavor to incorporate homomorphic encryption into federated learning. Initially, a master's thesis [2] attempted to evaluate the feasibility of preserving privacy in federated learning by employing a single public/private key pair in homomorphic encryption for all clients. In this study, we introduce a multi-key homomorphic encryption approach, which is more practical in real-world scenarios, considering that clients may not be willing to share the same private key due to mutual distrust.

In this paper, we will study a state of the art encryption technique in a federated learning system. Furthermore, we develop support for our encryption implementation on the Flower library as an alternative option to their current Transport Layer Security encryption. We then run experiments that compares the precision performance and runtime using the same CNN model trained locally, on unencrypted FL and encrypted FL.

1.2 Problem Definition

The use of sensitive data, such as medical data, for machine learning algorithms has become increasingly common. However, the disclosure of such data poses a significant threat to the privacy and security of individuals and organizations. Therefore, there is a critical need for privacy-preserving machine learning methods that can enable the analysis of sensitive data while protecting the privacy of the data subjects. [3]

Ring Learning with Errors (RLWE) based cryptography is a promising approach for achieving privacy-preserving machine learning. However, the adoption of this technique in real-world applications faces several challenges. For example, the computational and communication overheads of RLWE-based encryption can be significant, leading to slow training times. Additionally, the security guarantees of RLWE-based cryptography have not been thoroughly analyzed in the context of machine learning applications. [4]

RLWE together with Federated Learning (FL) creates an environment where individual clients never need to give up their data to the server for a machine learning model to function. This method makes it possible to "transfer data" across border by instead have the server train the ML model on weights from the clients as opposed to data which traditional models use. Using this method we have a way to retrieve data from clients without violating data protection regulations such as General Data Protection Regulation (GDPR), California Consumer Privacy Act (CCPA) or etc. [5]

1.3 Objective

This thesis aims to examine the various time, memory and security trade-offs between training a machine learning model locally compared to in an unencrypted or encrypted federated learning environment. To do this we fork a Ring Learning with Errors (RLWE) based encryption and integrate it into forked federated learning library. To simulate a real world scenario for privacy-preserving machine learning for health institutions we will be using COVID-19 data.

1.4 Contributions

In summary, in this thesis we make the following contributions:

- We create a baseline CNN model in Python using the Keras/Tensorflow library and learn the model on medical COVID-19 X-ray lung scan data. We do all the training locally to simulate a centralized training environment and measure the precision performance.
- We develop an unencrypted federated learning system with the help of the Flower library and train the baseline CNN model in the Flower federated learning system.
- We then implement an RLWE-based multi-key homomorphic encryption scheme in the same FL environment and run the identical precision performance tests.

1.5 Outline

The remainder of this thesis is structured as follows:

Chapter 2 introduces relevant background material including Ring learning with error and how it relates to privacy-preserving Federated learning.

Chapter 3 goes over the methodology and process of our solution approach

Chapter 4 provides and experimental evaluation of the Federated learning implementation with and without our proposed encryption and compares the precision performances.

Chapter 5 discusses the results as well as the pros and cons.

Chapter 6 concludes and presents suggestions for further work.

Chapter 2

Background

This chapter will provide the necessary background information for different topics addressed in this thesis. First we will explain Convolutional Neural Networks (CNN), a type of artificial neural network for image classification tasks that we will use to train models on COVID-19 lung scans. We will also cover the basics of privacy-preserving federated learning, a relatively new paradigm that allows distributed devices to collaborate and train a shared prediction model while keeping local data private. For the implementation of federated learning we have chosen to use the open source python library Flower, mostly due to its modular architecture and it's relative simplicity to be modified to our specific needs. Another subject we will briefly touch on is the built-in communication in Flower due to the necessity to make changes to it to allow for custom communication. We then compare traditional and lattice-based encryption, to discuss how the latter can offer possible solutions to vulnerabilities introduced by the former, particularly using Ring Learning With Errors (RLWE), a more momory- and time-efficient encryption scheme than Learning With Errors (LWE), a foundational cryptographic problem. We will furthermore address some more encryption terms like multi-key homomorphic encryption before laying down the conceptual and mathematical theory of xMK-CKKS, a proposed multi-key homomorphic encryption (MKHE) based on the CKKS scheme.

2.1 Convolutional Neural Network

Convolutional Neural networks (CNNs) are a type of artificial neural network utilized in deep learning applications for classification and computer vision tasks. As opposed to prior, time-consuming feature extraction methods, CNN provides a more scalable apporach to image classification and object recognition tasks with the use of matrix multiplication and leveraging principles from linear algebra to identify patterns within images. [6, 7]

In this thesis, a CNN will be trained on COVID-19 lung scans in both a centralized and a decentralized setting. In both cases, the same parameters will be used as a way to measure the difference between locally training a model and using federated learning with and without post-quantum secured model updates.

2.2 Privacy-preserving Federated learning

Federated learning is a machine learning approach that enables training of models across distributed devices or servers, without requiring the transfer of data to a centralized location. This method was introduced in 2016 by Google researchers to address the challenges of training machine learning models on large and sensitive datasets while preserving data privacy and security. [8]

In federated learning, instead of centralizing the data in one location, the training process is performed locally on each device, using the data available on that device. The local models are then aggregated to create a global model, which is deployed back to each device for further refinement. This iterative process continues until the model converges to an acceptable level of accuracy. [9]

Federated learning has several advantages over traditional centralized machine learning. One of the main benefits is the preservation of privacy, as the data remains on the device and is not shared with others. Federated learning also allows for the training of models on large and diverse datasets, which can improve model accuracy and generalization. [10, 11]

However, despite the big net positives of using federated learning over traditional learning, there are still several adversarial threats present such as model poisoning attacks [12], inference attacks [13], sybil attacks [14], model inversion attacks [15], deep leakage and more. [16] Broadly these are different types of attacks that exploit various weaknesses to either leak or reconstruct sensitive data, or introduce backdoor functionality. Still, federated learning is an active area of research where the are constantly being developed new solutions to counter or mitigate these threats.

Federated learning split into three different types [17]:

1. **Centralized federated learning** is a variant of federated learning where there is a centralized server that coordinates the training process across multiple devices.

The data remains on the devices, but the updates to the model parameters are sent to the server, which aggregates them and updates the global model. [18]

2. **Decentralized federated learning** is a variant of the federated learning where there is no centralized server, and the coordination of the training provess is done in a decentralized manner across multiple devices or nodes. Each node communicates with a subset of other nodes to exchange model updates, and the global model is computed by aggregating the local models at each node. [18]

3. **Heterogeneous federated learning** is a variant of federated learning where the devices or nodes have different capabilities, such as processing power, storage capacity, or network bandwidth. The training process is designed to take into account these differences, and the computation and communication tasks are allocated to each device or node based on its capabilities. [19]

While federated learning already provide some level of data privacy, we will be implementing privacy-preserving federated learning, as shown in Figure 2.1, which implements additional measures such as differential privacy to add noise to the model updates and multi-key homomorphic encryption to allow for encrypted aggregation of model updates. In this project, centralized learning is implemented to demonstrate the model aggregation by single centralized server, and allows better control over the encryption keys which are typically managed by the central server. We will also integrate a proposed solution that can mitigate model inversion attacks.

2.3 Flower: A Federated Learning Framework

Flwr, short for Flower, is an open source python library for federated learning. It was created to provide an easy and scalable infrastructure to move machine learning models back and forth, from being trained and evaluated on clients with local data to being sent to servers to aggregate the model updates and send back the updated model. Flower presents a unified approach for the training, analytics and evaluation of models under any workload, with any ML framework (such as tensorflow, torch, etc) and any programming language. [20] Another framework to keep an eye on is PySyft developed by OpenMined, which once fully developed and documented may be a potential candidate for the future industry standard for privacy-preserving federated learning. [21]

A big reason why we chose this framework due to it's very modular and open architecture allowing us to easily subclass core components such as the clients, server and aggregation method to more easily tailor things to our specific needs. Unfortunately the feature to

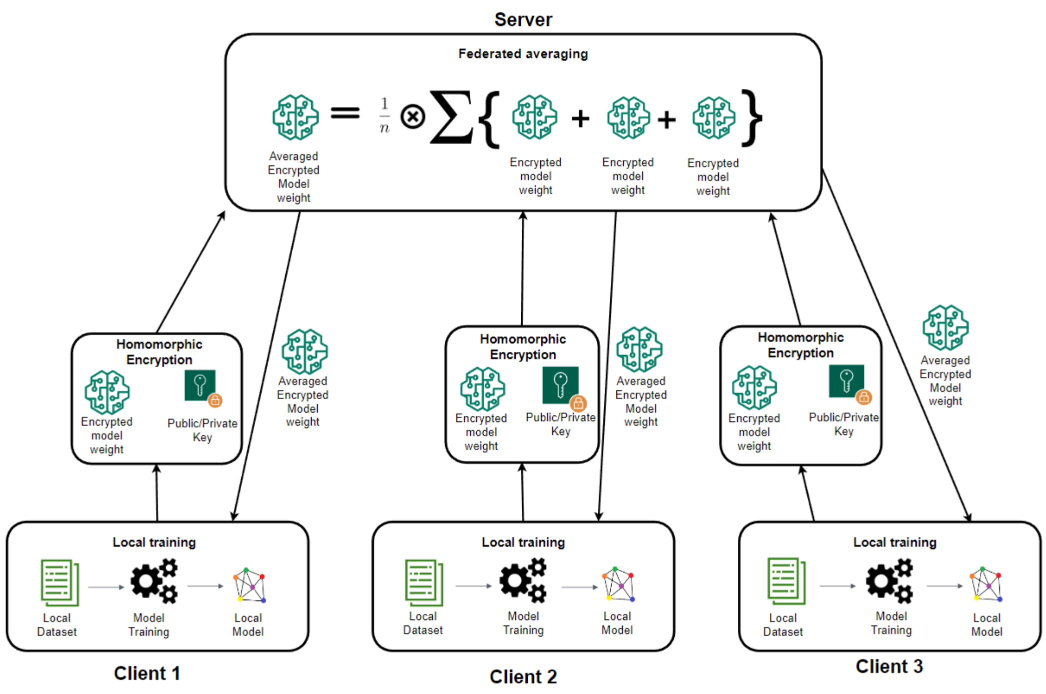


Figure 2.1: Centralized federated learning with privacy preserving encryption

create custom messages between server and clients is currently missing in Flower, but we will fork the source code to create our own messages that will be needed to implement our lattice encryption. [22] Due to flower's third party documentation on creating custom messages includes errors and lacks many of the necessary changes needed, we will in the solution approach chapter show through an example how to successfully modify the code.

2.4 Homomorphic Encryption

Unlike todays traditional encryption techniques, homomorphic encryption (HE) allows for the ability to do computation with encrypted data. This type of encryption can be split into three categories: partial, somewhat and fully homomorphic encryption. [23]

1. Partially homomorphic encryption (PHE) schemes only allow for a single operation such as addition or multiplication to be done over encrypted data an arbitrary number of times.
2. Somewhat homomorphic encryption (SHE) schemes allow for bounded computation of a set of operations. An example is the BGN encryption schemes that allows for an arbitrary number of additions and a single multiplication operation.
3. Fully homomorphic encryption (FHE) scheme allows for an arbitrary number of both addition and multiplication [24]

As with all good things there are also drawbacks to homomorphic encryption, but since HE was first introduced by Rivest, Adleman and Dertouzos in 1978 and FHE by Craig Gentry in 2009, there has been constant improvements in the field. One of the biggest challenges with using HE today is the increased computational overhead needed for encryption, decryption and the operations themselves in comparison to traditional methods. Another challenge includes data expansion, with the encrypted ciphertext usually being significantly larger in size than the plaintext. A last challenge I will include is noise management. Each homomorphic evaluation will add noise to the ciphertext and the eventual accumulation of too much noise will result in an undecryptable ciphertext, unless measures like bootstrapping, modulus reduction or dimension reduction techniques are applied. [[23], [25, p. 63-68]]

While federated learning provides an added security by allowing multiple clients train on local sensitive data without sharing any of their training data, homomorphic encryption can add an even greater level of security. The added security comes having all the clients encrypt their model updates and have the server aggregate all of them while still encrypted. The server can then decrypt the total sum without revealing any information from the individual client's training data. [26] This is called private summation and is what we will implement in our thesis by practically implementing the theory of the paper summarized in Section: 2.9

2.5 Traditional vs. Lattice-Based Encryption

Traditional encryption methods like RSA and ECC rely heavily on the computational difficulty of factoring large integers or calculating discrete logarithms. With the approach of quantum computers, these encryption methods are increasingly in risk of becoming vulnerable as quantum computers will be able to use quantum algorithms like Shor's Algorithm to solve these mathematical problems exponentially faster than classical computers. [27]

As a response to this increasing threat, there has been significant development in many different fields that are believed to be resistant to quantum attacks. One of these fields is lattice-based cryptography and is based on complex mathematical problems in lattice theory. [27] Most lattice-based key establishment algorithms are relatively simple, efficient, and highly parallelizable, while the security of these are also provably secure under a worst-case assumption. [[27, p. 3-4] [25, p. 30-33]] As a part of this project we will be modifying a basic Ring Learning With Errors (RLWE) scheme to work within our federated learning architecture with the goal of implement multi-key homomorphic encryption. RLWE is a mathematical problem that utilizes lattices over polynomial rings to construct an encryption scheme and will be more thoroughly discussed in section: 2.8

2.6 Multi-key Homomorphic encryption

In a plain homomorphic encryption system, operations can only be done on data that has been encrypted using the same key. [28] This approach is not very suitable for a federated learning environment as having all the clients use the same private key is a massive security concern and directly defeats the privacy-preserving aspect of letting clients keep their local data private. If all the clients share the same private key a malicious client could potentially decrypt the learning updates from other clients and then further do a model inversion attack to access information about other client's local data. [15] This is especially concerning if the data is of sensitive nature such as healthcare or financial information.

Multi-Key Homomorphic Encryption (MKHE) improves the security of normal homomorphic encryption and would allow a server to perform computations with ciphertexts that have been encrypted by clients using different private keys. [29] Very briefly, this is done by all clients generating a public key based on their own private key, the server will then aggregate the public keys and distribute a shared public key to the clients. This key can then be used to encrypt plaintext, but the ciphertext cannot be decrypted until the server has received partial decryption shares from all the participating clients.

2.7 Protobuf and gRPC

Flower utilizes gRPC and Protocol Buffers (protobuf) for efficient communication between server and clients. gRPC is an open-source, high-performance framework by Google that enables robust RPC communication. Remote Procedure Call (RPC) allows a program on a machine to to execute a function on another network as if it was a local call, without having to know the details of the network communication. [30] Protobufs, also by Google, is a language-agnostic and platform-neutral mechanism to serialize and deserialize structured data. In our case we need both the server and clients to serialize python objects into bytes before transferring the data, and for the recipient to deserialize the data from bytes back into python objects. [31] An important part of our project involved with creating custom protobuf messages by altering the original flower source-code to be able to implement our multi-key homomorphic encryption scheme within flower's federated learning architecture.

2.8 Ring Learning With Errors (RLWE)

Ring Learning With Errors (RLWE) is a variant of the Learning With Errors (LWE) and they are both mathematical problems in the field of lattice-theory believed to be resistant against being efficiently solved by quantum computers. The difficulty of these problems lies in trying to solve a specific equation but with the presence of small random noise or errors added to the equation results. The errors are small but are enough to obfuscate the solution results enough to make the problem computationally challenging, and thus ensure cryptographic security. [[25, p. 30-33], [32], [33]]

The main difference between RWLE and LWE is in the algebraic structure used. While LWE use vector spaces (finite field), RLWE uses polynomial rings. The use of ring strunctures allows for a more efficient implementation compared to LWE as ring operations are often much quicker and require less memory than their counterparts in vector spaces. This advantage does come with additional complexity as polynomial rings require some knowledge of number theory and abstract algebra. [25, 32]

Simplified, the rings can be understood as a collection of polynomials with a single unknown variable. To encrypt a plaintext, it must first be converted into a list of integers. This conversion allows us to represent the plaintext as a polynomial, where each plaintext value becomes a coefficient in the polynomial. The coefficients are integers within a modular arithmetic system. Typically a prime number q is chosen as the modulus and provides a convenient and symmetrical representation of data. Selecting q must be done appropriately, as the plaintext values must fall within the range of -q/2 to q/2. When adding polynomials modulo q, we simply add their coefficients and apply the modulus q. However, multiplying two polynomials results in a polynomial of higher degree, which needs to be reduced back to the original degree using a process called modulus reduction.[25] The symmetric value range is also advantageous when performing modulus reduction, as any noise or errors introduced, usually from zero-mean Gaussian distributions, are generated within this range (typically with a small standard deviation). The following chapter we will elaborate on how we fork a simple RLWE implementation to modify the setup, encryption and decryption process to successfully implement MKHE.

2.9 xMK-CKKS

2.9.1 Conceptual Theory:

CKKS (Cheon Kim-Kim Song) is a fully homomorphic encryption scheme for efficient handling of approximate arithmetic on encrypted data and MK-CKKS is a multi-key homomorphic encryption based on the CKKS scheme. xMK-CKKS however, is a newer variant of MK-CKKS proposed by a preprint study from 2021 by Jing Ma, Si-Ahmed Naas, Stephan Sigg and Xixiang Lyu. [26]

Their study presents the privacy-preserving federated learning scheme xMK-CKKS to address privacy concerns in mobile services and networks where the need to transfer data to a central unit violates privacy and also increase bandwidth demands. They claim their scheme will prevent privacy leakage from publicly shared information and protect against collusion between clients and server. Their experimental evaluations also demonstrates that their scheme preserves model accuracy.

In their proposed scheme, a key aspect of increasing the privacy of federated learning is the encryption of model updates using an aggregated public key. The aggregated public key is just the sum of individual public keys that are generated by each participating client. This collaborative approach of combining public keys adds an additional layer of security, as the model update from any individual client will be obfuscated and protected from being decrypted or analyzed by neither the server nor any of the other clients.

However, to decrypt the aggregated ciphertexts requires collaboration between all the participating clients. Each client holds a piece of the puzzle needed to decrypt the aggregated ciphertext and once the server has collected all of the pieces, the plaintext for the combined model updates can be retrieved. The server can then average the combined model updates based on the number of participating clients and send back the finalized model updates back to all the clients before a new training round occurs.

2.9.2 Mathematical Approach:

In xMK-CKKS, clients collectively generate a public key for encryption, and decryption only occurs server-side after receiving partial decryption shares from all the clients. The setup is based on the methodology outlined in the referenced study. [26]

Setup: Define the RLWE parameters n, ciphertext modulus q, three Gaussian distributions χ, ψ and ϕ for key distribution, noise for encryption, and noise for partial decryption, respectively. Generate a shared polynomial a drawn from a uniform distribution.

Key Generation (n, q, χ, ψ, ϕ, a) : Each client i, generate its secret key s_i drawn from χ and error term e_i drawn from ψ. Client can then compute it's public key $b_i = -s_i \times a + e_i$ (mod) q. The aggregated public key $\widetilde{b}$ can then be defined as:

$$\widetilde{b} = \sum_{i=1}^{N} b_i = \sum_{i=1}^{N} (-s_i) \times a + \sum_{i=1}^{N} e_i \mod q \tag{2.1}$$

Encryption (m_i, $\widetilde{b}$, a): The plaintext m_i of client i, is encrypted as follows: (See Figure: 2.2)

$$ct_i = (c_0^{d_i}, c_1^{d_i}) = (v^{d_i} \times b + m_i + e_0^{d_i}, v^{d_i} \times a + e_1^{p_i}) \mod q \tag{2.2}$$

Homomorphic Addition (ct_1, ..., ct_n): The summation of ciphertexts is as follows:

$$\begin{aligned} C_{sum} &= \sum_{i=1}^{N} ct_i = (C_{sum_0}, C_{sum_1}) \\ &= (\sum_{i=1}^{N} c_0^{d_i}, \sum_{i=1}^{N} c_1^{d_i}) \\ &= (\sum_{i=1}^{N}(v^{d_i} \times b + m_i + e_0^{d_i}), \sum_{i=1}^{N}(v^{d_i} \times a + e_1^{d_i})) \mod q \end{aligned} \tag{2.3}$$

Partial Decryption ($Csum_1$, s_i, ..., s_n): Using the provided $Csum_1$ and error term e_i^* drawn from ϕ (using a higher standard deviation than ψ). Each client i computes its decryption share D_i. (See Figure: 2.3)

$$\begin{aligned} D_i &= s_i \times C_{sum_1} + e_i^* \\ &= s_i \times \sum_{i=1}^{N}(v_i \times a + e_1^{d_i}) + e_i^*(\mod q) \end{aligned} \tag{2.4}$$

Sum of all plaintexts can be recovered as follows: (See Figure: 2.4)

$$\begin{aligned} C_{sum_0} + \sum_{i=1}^{N} D_i \mod q &= C_{sum_0} + \sum_{i=1}^{N} s_i \times C_{sum_1} + \sum_{i=1}^{N} e_i^* \mod q \\ &= \sum_{i=1}^{N}(v^{d_i} \times b + m_i + e_0^{d_i}) + \sum_{i=1}^{N} s_i \times \sum_{i=1}^{N}(v^{d_i} a + e_1^{d_i}) + \sum_{i=1}^{N} e_i^* \mod q \\ &= -\sum_{i=1}^{N} v^{d_i} s_i a + \sum_{i=1}^{N} v^{d_i} \sum_{i=1}^{N} e_i + \sum_{i=1}^{N} e_0^{d_i} + \sum_{i=1}^{N} v^{d_i} s_i a + \sum_{i=1}^{N}(s_i e_1^{d_i} + e_i^*) \mod q \\ &= \sum_{i=1}^{N} m_i + \sum_{i=1}^{N} v^{d_i} \times \sum_{i=1}^{N} e_i + \sum_{i=1}^{N} e_0^{d_i} + \sum_{i=1}^{N}(s_i e_1^{d_i} + e_i^*) \mod q \\ &\approx \sum_{i=1}^{N} m_i \end{aligned} \tag{2.5}$$

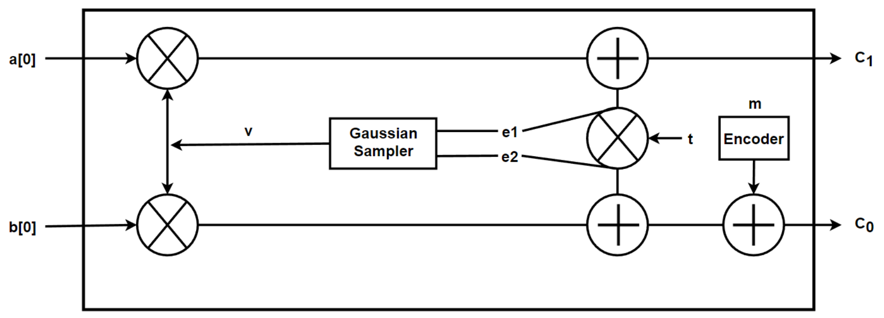


Figure 2.2: Utilizing aggregated public key b[0], shared polynomial a[0] and noise polynomials to encrypt plaintext m into ciphertext tuple (c_0, c_1). Homomorphic addition and multiplication are represented by "+" and "x" respectively.

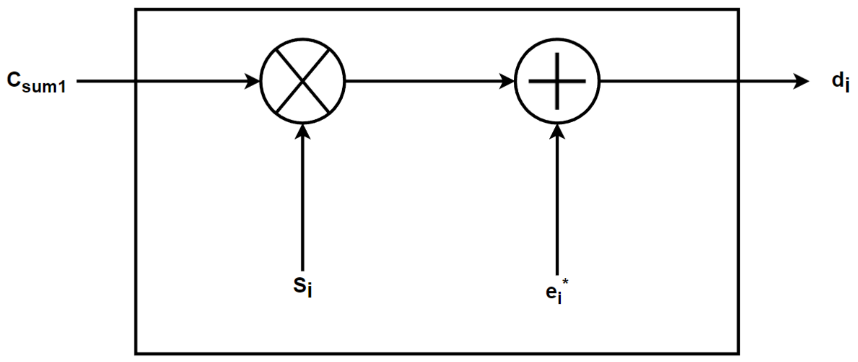


Figure 2.3: How the aggregated c_{sum_1} is used to compute a partial decryption share d_i using a client's private key and some added noise. Performed by all clients.

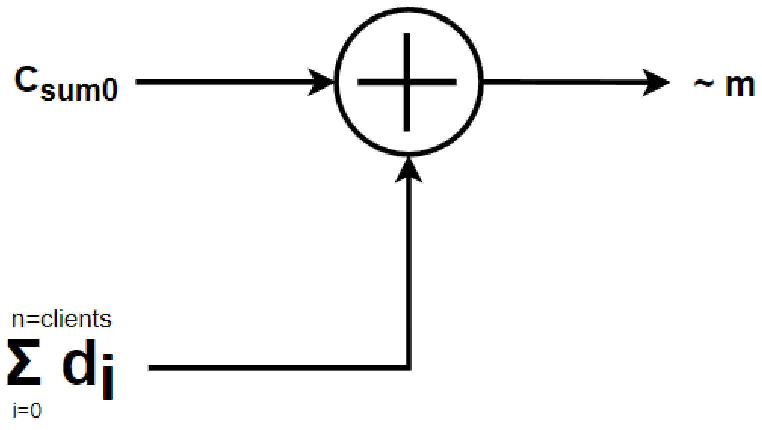


Figure 2.4: An approximate recovery of the of all plaintexts, performed by the server.

Chapter 3

Solution Approach

3.1 Introduction

The privacy of individuals in health institutes is at risk due to the sharing of sensitive data between different institutes, presenting a significant challenge for machine learning in the medical field. To address this issue, our thesis proposes a solution approach that utilizes a centralized federated learning framework, incorporating an RLWE based multi-key homomorphic encryption scheme for secure communication between the server and clients. By leveraging this approach, we aim to enhance privacy-preserving machine learning in health institutes and mitigate the risks associated with data sharing.

3.2 Existing Approaches/Baselines

Currently, Wibawa et al. [2] have proposed a model that takes use of Somewhat Homomorphic Encryption (SHE) where each client train their copy of global model with their own local dataset before updating the global model by sending the weights from the trained model from each client. Like our implementation, her implementation does not convert the ciphertext precisely back to its original plaintext value, but rather an approximation with a small noise due to the nature of encryption. A big difference between Wibawas proposed solution and our proposed solution is the form of encryption. Whereas her SHE uses a shared key across all clients, which translates poorly into a real-life scenario as clients might not trust each other, our proposed model utilizes private summation using a multi-key encryption scheme. This way no client nor the server will be able to decrypt any individual client's sensitive training data and thus vastly improves the security of the proposed model for a real-life scenario.

Ma et al. [26] introduced an improved privacy-preserving federated learning scheme called xMK-CKKS. This proposed scheme incorporates multi-key homomorphic encryption to update the model securely. It leverages an aggregated public key to encrypt the model before sharing it with a server for aggregation. Collaboration among all participating devices is necessary for decryption. In our work, we took use of their mathematical encryption and decryption approach and implemented it to our work together with our custom created communication in the Flower FL library.

3.3 RLWE: Original and Improved

3.3.1 Overview of Original RLWE Implementation

The original RLWE implementation was retrieved from a Github repository [34], and will serve as the foundation for our modification. The simple RWLE implementation consists of three main Python files. A utils.py file for utility functions, an Rq.py file that creates ring polynomial objects, and an RLWE.py to create an RLWE instance for key generation, encryption, decryption, and modular operations used for homomorphic encryption

Key Generation, Encryption, and Decryption in Original RLWE

The original RLWE.py file contains a class to instantiate an RLWE instance that takes four variables: n, t, q and std. The variables represent the degrees (length) of the polynomial plaintext that needs to be encrypted (required to be a power of 2), two large prime numbers that defines the value ranges for the plaintext and ciphertext coefficients, and the standard deviation of a zero-mean Guassian distribution used to generate both the private keys and errors.

For both the simple RLWE implementation and our modified RLWE we introduce three Guassian distributions: χ, ψ, ϕ, that are used extensively for generating keys and errors. All of them are zero-mean Guassian distributions, but with different standard deviations. χ is used to generate private keys, ψ is used to generate error polynomials during the encryption process, and ϕ is introduced in our modified RLWE scheme for the partial decryption where it is required to generate error polynomials with a slightly higher standard deviation than the error polynomials used during encryption. For our project we are using the standard deviation of 3 for χ and ψ (making them equivalent), and a standard deviation of 5 for ϕ.

When defining the variable t, it is important to make sure the possible min and max coefficients of the polynomial plaintext are within the range of -t/2 and t/2. This is crucial because in RLWE, the coefficients get wrapped around the ring, meaning a value exceeding t/2 will get interpreted as a negative value, and vice versa.

Similarly, the ratio between the modulus for the ciphertext q and the modulus for the plaintext t is also significant. As detailed in Peikert's discussion on hardness of LWE and its variants (Section 4.2), and Ring-LWE (Section 4.4), the appropriate choice of must take into account the desired security level and the specific parameters of the scheme. [25] As our work is mostly proof-of-concept we arbitrarily set the modulus q to be a prime number larger than twenty times the modulus t to hopefully balance between computational efficiency and security.

As for the choice of standard deviations, our choice is also somewhat arbitrary, selecting values of 3 and 5. As outlined in Peikert's papers [25, 32], the standard deviation also plays an important role in determining the security level along other parameters. A larger ratio between the modulus q and t will allow for a higher standard deviation that will add more noise, which will make it harder to decrypt and thereby further increase the security. In reality, these choices should be carefully balanced with considerations of security and efficiency, and for more precise details on choosing these parameters, we recommend referring to Peikert's work.

Key generation, encryption, and decryption processes in the original simple RLWE implementation are as follows:

- **Key Generation:**
    1. Generate a private key s (noise) from key distribution χ.
    2. Generate a public key a comprising two polynomials a_0 and a_1. a_1 is a polynomial over modulus q with coefficients randomly sampled from a uniform distribution, while a_0 is a polynomial over modulus t computed using the formula $a_0 = -1 \times (a_1 \times s + t \times e)$, where e is an error polynomial of modulus q drawn from distribution ψ.

```
def generate_keys(self):
    s = discrete_gaussian(self.n, self.q, std=self.std)
    e = discrete_gaussian(self.n, self.q, std=self.std)

    a1 = discrete_uniform(self.n, self.q)
    a0 = -1 * (a1 * s + self.t * e)

    return (s, (a0, a1))  # (secret, public)
```

- **Encryption:**

  1. Given a python list of integers as a plaintext, convert the list to an Rq object for the python plaintext to represent and behave as a polynomial over modulus q.
  2. Encrypt the polynomial plaintext into two polynomial ciphertexts c using the formula $(m + a_0 \times e[0] + t \times e[2], a_1 \times e[0] + t \times e[1])$, where m is the plaintext and e are error polynomials sampled from distribution ψ. c is a tuple consisting of the polynomial ciphertexts c0 and c1.

```
def encrypt(self, m, a):
    a0, a1 = a
    e = [discrete_gaussian(self.n, self.p, std=self.std)
         for _ in range(3)]
    m = Rq(m.poly.coeffs, self.p)
    return (m + a0 * e[0] + self.t * e[2], a1 * e[0] + self.t * e[1])
```

- **Decryption:**

  1. Provided the private key s we can decrypt the ciphertext c using the following function:

```
def decrypt(self, c, s):
    c = [ci * s**i for i, ci in enumerate(c)] # c = (c0, c1)
    m = c[0]
    for i in range(1, len(c)):
        m += c[i]
    m = Rq(m.poly.coeffs, self.t)
    return m
```

3.3.2 Modifications to RLWE.py for xMK-CKKS Integration

With our goal to implement secure federated learning using the lattice-based xMK-CKKS scheme, certain modifications to the original RLWE were necessary. These alterations mostly revolved around the functions for key generation, encryption, decryption, as well as adding some additional utility functions. While the explanation structure will follow a similar pattern as the previous sub-section, we will here include formulas to justify the changes we made to the original RLWE implementation. All formulas included are drawn from Chapter 4.3 [26], which explains the setup of the xMK-CKKS scheme for the key sharing process, and details step 1 through 5 for the encryption and decryption process.

Changes in Key Generation for xMK-CKKS Integration

In the context of federated learning, all clients must have unique private keys but use a shared public key. To achieve this, we use formulas proposed in the xMK-CKKS paper. Each client generates a unique public key b using a shared polynomial a and their private key s. The server then aggregates these public keys b through modular addition and distributes the shared public key back to the clients.

For each client i, let the secret key $\boldsymbol{s_i}$ be drawn from distribution $\boldsymbol{\chi}$ and let the error polynomial $\boldsymbol{e_i}$ be drawn from distribution $\boldsymbol{\psi}$. Unlike to the original RLWE implementation where both the secret key and error polynomials will be of the same degree as the plaintext, xMK-CKKS requires the secret key $\boldsymbol{s_i}$ to be a polynomial of a single degree. The polynomial a will have been shared to all the clients by the server prior to clients generating keys. A client's public key $\boldsymbol{b_i}$ can then be computed using the formula: $\boldsymbol{b_i = -s_i \times a + e_i}$

The modified key generation function looks like this:

```
def generate_keys(self):
    self.s = discrete_gaussian(1, self.p, std=self.std)
    self.e = discrete_gaussian(self.n, self.p, std=self.std)
    self.b = -1 * (self.a * self.s + self.t * self.e)
    return (self.s, (self.b, self.a))  # (secret, public)
```

Changes in Encryption for xMK-CKKS Integration

The encryption process also had to be modified slightly from the original RLWE implementation. The overall outcome is still the same, we use a public key and three error polynomials to encrypt a polynomial plaintext into a tuple of two ciphertexts: ct = ($\boldsymbol{c_0}$, $\boldsymbol{c_1}$). The main difference is this modified scheme requires only the first value of the aggregated public for the encryption instead of the full public key.

For each client i, let $\boldsymbol{m_i}$ be a client's polynomial plaintext to be encrypted to ciphertext $\boldsymbol{ct_i}$. Let b = $\mathbf{b[0]}$ and a = $\mathbf{a[0]}$ be the same for all clients. For each client generate three error polynomials: $\boldsymbol{v^i}$, $\boldsymbol{e[1]^i}$ and $\boldsymbol{e[2]^i}$. Polynomial $\boldsymbol{v^i}$ is drawn from the key distribution $\boldsymbol{\chi}$ and used for the computation of both $\boldsymbol{c_0^i}$ and $\boldsymbol{c_1^i}$, while $\mathbf{e[1]}$ and $\mathbf{e[2]}$ are unique to either $\boldsymbol{c_0^i}$ and $\boldsymbol{c_1^i}$ and are drawn from error distribution $\boldsymbol{\psi}$. The computed ciphertext can then be computed using the formula:

$$\boldsymbol{ct_i = (c_0^i, c_1^i) = (v^i \times b + m_i + e_0^i, v^i \times a + e_1^i)(modq)} \tag{3.1}$$

The updated encryption function is:

```
def encrypt(self, m, pub):
    b, a = pub
    e = [discrete_gaussian(self.n, self.p, std=self.std)
         for _ in range(3)]
    self.v = e[0]
    m = Rq(m.poly.coeffs, self.p)
    # From the math in the study e[0] = v_i <- chi
    return (m + b.poly.coeffs.tolist()[0] * self.v + self.t * e[2],
            a.poly.coeffs.tolist()[0] * self.v + self.t * e[1])
```

Changes in Decryption for xMK-CKKS Integration

In the case of federated learning with xMK-CKKS, clients cannot fully decrypt ciphertexts that have undergone homomorphic operations. Instead, clients will compute a partial decryption share and once the server has successfully retrieved all the partial results it will be able to decrypt the aggregated ciphertexts from all the clients. The partial decryption

For each client i, $\boldsymbol{s_i}$ is the client's secret key and let $\boldsymbol{e_i^*}$ be an error polynomial sampled from distribution $\boldsymbol{\phi}$ with the assumption the distribution has a larger variance than distribution $\boldsymbol{\chi}$ from the basic scheme. The server has collected all the ciphertexts $\boldsymbol{c_0^i}$ and $\boldsymbol{c_1^i}$ from each client and let $\boldsymbol{C_{sum_1}}$ be the aggregated result of the modular addition of all the $\boldsymbol{c_1^i}$ ciphertexts. Provided these variables, a client can then compute their partial decryption share $\boldsymbol{d_i}$ using this formula:

$$\boldsymbol{D_i = s_i \times C_{sum_1} + e_i^* = s_i \sum_{i=1}^{N} (v_i \times a + e_1^i) + e_i^*(modq)} \tag{3.2}$$

The updated (now only partial) decryption function is:

```
def decrypt(self, c, s, e):
    partial_decryption = c * s + e
    partial_decryption = Rq(partial_decryption.poly.coeffs, self.t)
    return partial_decryption
```

Additional modifications to the original RLWE implementation include the ability to set and get a shared polynomial a such that the server can generate a polynomial and distribute it for the clients to store in their own rlwe instance. Another simple modification is a function to convert a python list into a polynomial Rq object over a given modulus.

3.3.3 Alterations to Utility.py

While our work involved no changes to the Rq file, we added some additional functions to the utils file. Notably, we created functions to:

1. Get model weights and flatten them: This function retrieves the weights of a CNN model, which consist of nested tensors, and flattens them into a single Python list. The function also returns the original shape for future use. This function will be used by clients at the encryption stage as we want the weights in the format of a long python list to be able to represent the plaintext as a polynomial.

```
def get_flat_weights(model: CNN) -> Tuple[List[int], List[int]]:
    params = model.get_weights()
    original_shapes = [tensor.shape for tensor in params]
    flattened_weights = []
    for weight_tensor in params:
        # Convert the tensor to a numpy array
        weight_array = weight_tensor.numpy()
        # Flatten the numpy array and add it to the flattened_weights list
        flattened_weights.extend(weight_array.flatten())
    return flattened_weights, original_shapes
```

2. Revert flattened weights back to original nested tensors: This function is the reversal of the previous function. It takes a long list of flattened weights and an original shape such that the function can reconstruct the weights back into their original nested tensor format. This will be used by the clients at the end of each training round, after the decryption process. When all the clients have sent their model updates to the server, the server will compute an average and send back the updated models weights to all the clients before the next training round occurs. This is where the clients will need to update their local cnn model by retrieving the updated model weights and converting it back into the nested tensor format the cnn model requires.

```
def unflatten_weights(flattened_weights, original_shapes):
    unflattened_weights = []
    current_index = 0
    for shape in original_shapes:
        # Calculate the number of elements in the current shape
        num_elements = np.prod(shape)
        # Slice the flattened_weights list to get the elements for the current
    shape
        current_elements = flattened_weights[current_index : current_index +
    num_elements]
        # Reshape the elements to the original shape and append them to the
    unflattened_weights list
        reshaped_elements = np.reshape(current_elements, shape)
        unflattened_weights.append(tf.convert_to_tensor(reshaped_elements, dtype=
    tf.float64))
```

```
        # Update the index for the next iteration
        current_index += num_elements
    return unflattened_weights
```

3. Pad the flattened weights to the nearest 2^n: The xMK-CKKS scheme requires that a plaintext has a length of 2^n. To fulfill this condition, we provide the clients with a function to pad their plaintext to reach the required length. The target length will be the shortest power of 2 larger than the length of the plaintext.

Additionally, CNN model weights are originally floating numbers between -1 to 1 with eight decimal places. As the RLWE scheme cannot directly handle floating numbers we scale them up to large integers in our CNN class. They will by default be fully scaled up by 10^8 to ensure near full precision of the model as all the decimals will present during the encryption process. We did however make this a static variable that can be modified to test the accuracy and speed of the model should one decide to use less of the decimals present in the model weights.

```
def pad_to_power_of_2(flat_params, target_length=2**20, weight_decimals=8):
    pad_length = target_length - len(flat_params)
    if pad_length < 0:
        raise ValueError("The given target_length is smaller than the current
    parameter list length.")
    random_padding = np.random.randint(-10**weight_decimals, 10**weight_decimals
    + 1, pad_length).tolist()
    padded_params = flat_params + random_padding
    return padded_params, len(flat_params)
```

3.3.4 Summary

This section provides an analysis of the original RLWE implementation, which was sourced from a GitHub repository that served as the basis for our modifications. The initial RLWE scheme consisted of Python files for creating ring polynomial objects, utility functions, and an RLWE instance for encryption, decryption, key generation, and homomorphic encryption operations. This original version utilized uniform and Gaussian distributions for key and error generation and introduced considerations for parameters such as length of polynomial plaintext, modulus for plaintext and ciphertext, and standard deviation.

For the integration of the xMK-CKKS modifications to the original RLWE implementation were essential. These alterations mostly revolves around the key generation, encryption, and decryption functions, in addition to several new utility functions. The server and each client will have access to their own RLWE instance.

In terms of key generation, each client will generate a unique public key using a shared polynomial but their own private key. The server aggregates these public keys through modular addition to create a shared public key. The encryption process has been adjusted to only require the first value of the aggregated public key for encryption. To enable the transmission of the shared polynomial, the RLWE instance were given functions to store and retrieve this shared polynomial. During decryption, each client will only calculate a partial decryption share which are all sent to the server for it to decrypt the encrypted sum of all the model updates.

Additional utility functions were created for model weight management, including functions to flatten model weights, revert flattened weights back to the original structure, and to pad the plaintext to reach a required length. Additionally, the model weights will be scaled to large integers to allow for the RLWE scheme to process them. These changes eases the necessary processing of the model weights during the encryption and decryption process.

3.4 Flower Implementation of xMK-CKKS

The Flower federated learning library provides a flexible and relatively user-friendly interface to initialize and run federated learning prototypes. It is designed to be compatible with any machine learning framework and programming language, but with a primary focus on Python as discussed in 2.3

In a standard Flower setup, a user will implement a server and clients in separate scripts, typically named [server.py] and [client.py]. The [server.py] script is mainly responsible for starting the federating learning process. Typically this involves creating a server instance and providing it various optional parameters, like a strategy object. A strategy is a set of instructions that the server needs to determine how to aggregate and calculate the model updates provided by the clients. Users can either select one of the many existing Strategies, or subclass the pre-defined Strategy to your own needs.

The [client.py] script on the other hand, defines the logic of a client. Broadly their task is to perform computations, like training a local model on its local dataset, then send the results (model updates) back to the server. To implement a client a user has to subclass the Client or NumpyClient class provided by Flower and define the required methods: get_parameters, set_parameters and fit. The get_parameters is responsible for retrieving the parameters of it's local model. The set_paremeters updates the model with new parameters received from the server, while fit is in charge of the training process of it's current model.

This architecture allows for a flexible, scalable and very adaptable environment to implement federated learning for various specifications. Users are able to use any preferred machine learning framework while maintaining the standard level of the privacy and security benefits that federated learning provides.

Despite the flexibility of the standard Flower setup, one notable limitation is its lack of support for custom communication messages between the server and client. Our task of implementing the proposed xMK-CKKS sheme necessitates exchange of data that goes beyond the pre-defined message types in the original Flower library. This required us to fork the library's repository to make modifications to the source code. [22] Several related files had to be modified and compiled to successfully create our own messages but the changes made helped us effectively implement the scheme. To help other create their own message types we will provide a detailed guide on the necessary steps in the subsequent section. However, we will first discuss how we implemented the xMK-CKKS scheme into the Flower source code, assuming all the required message types have already been created.

3.4.1 RLWE Instance Initialization

At the heart of our xMK-CKKS implementation are our custom Ring Learning with Errors (RLWE) instances. These are integral to our work and each server and client instance needs their own rlwe instance. In our custom client script client.py, we subclass the NumpyClient and pass an rlwe instance. Similarly, in server.py, we provide the server with an rlwe instance by subclassing the pre-defined strategy FedAvg. The code below is the same for both the client.py and server.py script and dynamically defines the various parameters needed to instantiate an rlwe instance:

```
# dynamic settings
    # cnn model precision
    WEIGHT_DECIMALS = 8
    model = cnn.cnn.CNN(WEIGHT_DECIMALS)
    utils.set_initial_params(model)
    params, _ = utils.get_flat_weights(model)

    # find closest 2^x larger than number of weights
    num_weights = len(params)
    n = 2 ** math.ceil(math.log2(num_weights))

    # decide the polynomial ring range of the plaintext m
    max_weight_value = 10**WEIGHT_DECIMALS # 100_000_000 if full weights
    num_clients = 10
    t = utils.next_prime(num_clients * max_weight_value * 2) # 2_000_000_011

    # decide the polynomial ring range of the ciphertext (c0, c1)
    q = utils.next_prime(t * 20)

    # create rlwe instance
    std = 3  # standard deviation of Gaussian distribution
    rlwe = RLWE(n, q, t, std)
```

3.4.2 Key Sharing Process

Up until this point when we have referenced the server.py file, we have been talking about the user-created server.py file, the one typically created by a user to start a Flower server. However, from this section forward, when we mention server.py, we are specifically referring to the server.py file located within the Flower library's source code itself. This server.py is the backbone of the library's operations and our forked version is what enables us to make drastic changes to the communication and encryption procedures between server and clients. An important thing to note is that our python code directly involved with the encryption and decryption are peforming computations on polynonmial objects, while the transmission of polynomials between server and clients requires the polynomials to be converted to python lists of integers.

Step 1: Polynomial Generation and Key Exchange

The key sharing process begins in the forked server.py script. Right before the flower's built in training loop occurs we utilize our custom communication message types to ensure that all the participating clients end up with the same aggregated public key that will be used during the training loop to encrypt and decrypt model updates. The forked server.py script will first make use of it's local rlwe instance to generate a uniformly generated polynomial a. The server will then send this polynomial "a" to all the clients which will use it to generate their own private key and a corresponding public key "b". The clients will then send back this public key "b" as a response to the server request.

(Forked) Server.py:

```
# Step 1) Server sends shared vector_a to clients and they all send back vector_b
    self.strategy.rlwe.generate_vector_a()
    vector_a = self.strategy.rlwe.get_vector_a()
    vector_a = vector_a.poly_to_list()
    vector_b_list = []

    # Wait for clients
    sample_size, min_num_clients = self.strategy.num_fit_clients(
        self._client_manager.num_available()
    )
    clients = self._client_manager.sample(
        num_clients=sample_size, min_num_clients=min_num_clients
    )

    # Loop through clients, send vector A, retrieve vector B
    for c in clients:
        client_vector_b = c.request_vec_b(vector_a, timeout=timeout)
        vector_b_list.append(self.strategy.rlwe.list_to_poly(client_vector_b, "q"
    ))
```

Client.py:

```
# Step 1) Clients retrieve shared vector_a from server and send back vector_b
    def generate_pubkey(self, vector_a: List[int]) -> List[int]:
        vector_a = self.rlwe.list_to_poly(vector_a, "q")
        self.rlwe.set_vector_a(vector_a)
        (_, pub) = rlwe.generate_keys()
        print(f"client pub: {pub}")
        return pub[0].poly_to_list()
```

Step 2: Public Key Aggregation

Once the server has gathered all the public keys from the participating clients it will perform modular additions to aggregate all the keys. The result will be a shared public key to be sent to all the clients through a message request from the server. Each client will store this public key in their rlwe instance to be used for encrypting local model weights post-training. The clients will respond with a confirmation to the server.

(Forked) Server.py:

```
# Step 2) Server sends aggregated publickey allpub to clients and receive boolean
     confirmation
    allpub = vector_b_list[0]
    for poly in vector_b_list[1:]:
        allpub = allpub + poly
    allpub = allpub.poly_to_list()

    for c in clients:
        confirmed = c.request_allpub_confirmation(allpub, timeout=timeout)
```

Client.py:

```
# Step 2) Server sends aggregated publickey allpub to clients and receive boolean
     confirmation
    def store_aggregated_pubkey(self, allpub: List[int]) -> bool:
        aggregated_pubkey = self.rlwe.list_to_poly(allpub, "q")
        self.allpub = (aggregated_pubkey, self.rlwe.get_vector_a())
        print(f"client allpub: {self.allpub}")
        return True
```

3.4.3 Training Loop and Weight Encryption Process

Upon completion of the key sharing process, the server initiates the federated training loop, where each client simultaneously begins the training of its local model using its local dataset. After the models have been trained the server needs to aggregate and average all the results and redistribute an updated model for all the clients to use for the next training round.

```
for current_round in range(1, num_rounds + 1):
  # Simultaneously train a local model on all the participating clients
  res_fit = self.fit_round(server_round=current_round, timeout=timeout)
```

In a standard Flower library setup the clients will send their full model weights unencrypted to the server. This is of great security concern as if any clients are training their models on sensitive data, an untrusted server or malicious client could peform an inversion attack to generate a client's training data based on their updated model weights. By contrast, our implementation of the xMK-CKKS scheme greatly enhances the security and privacy of this process. The clients will encrypt their model weights and the server will homomorphically aggregate all of the updates, leaving neither the server or any client the ability to read any client's individual model updates.

Furthermore, the standard Flower setup also has the clients send the full updated weights of their local model. Contrary to this approach, our choice was to instead only send the gradient of the updated weights, meaning each client will locally compare their new weights with the old ones and only send the change of weights to the server. In a similar way, the server will homomorphically aggregate all the gradients and send back the weighted average back to the clients. This approach excludes the server from ever knowing the full weights of any models and thus can require less trust from the participating clients.

Step 3: Weight Encryption and Transmission

At the end of each round, when the clients have trained their local models, the server sends a request for the updated weights. The clients will convert the nested tensor structure of the weights into a long python list. This will be the plaintext and the follwoing encryption will result in two ciphertexts c0, c1. The server server will then homomorphically aggregate all the c0 and c1 received from the clients to a c0sum and c1sum variable. The encryption of plaintexts is performed by the client's RLWE instance and is based on Equation 3.3, while the server aggregates the ciphertexts according to Equation 3.4:

$$ct^{d_i} = (c_0^{d_i}, c_1^{d_i}) = (v^{d_i} \times b + m_i + e_0^{d_i}, v^{d_i} \times a + e_1^{d_i}) (\mod q) \tag{3.3}$$

$$C_{sum} = \sum_{i=1}^{N} ct^{d_i} = (\sum_{i=1}^{N} c_0^{d_i} \sum_{i=1}^{N} c_1^{d_i}) = (C_{sum_0}, C_{sum_1}) (\mod q) \tag{3.4}$$

(Forked) Server.py:

```
# Step 3) (~20% Less Memory)
# Let all clients encrypt plaintext p -> (c0, c1) and here we sum up all c0 and
    c1 polynomials
      clients = list(self._client_manager.clients.values())
      c0, c1 = clients[0].request_encrypted_parameters(request, timeout=timeout)
      c0sum = self.strategy.rlwe.list_to_poly(c0, "q")
      c1sum = self.strategy.rlwe.list_to_poly(c1, "q")
      for c in clients[1:]:
          c0, c1 = c.request_encrypted_parameters(request, timeout=timeout)
          c0sum = c0sum + self.strategy.rlwe.list_to_poly(c0, "q")
          c1sum = c1sum + self.strategy.rlwe.list_to_poly(c1, "q")
```

Client.py:

```
# Step 3) After round, encrypt flat list of parameters into two lists (c0, c1)
    def encrypt_parameters(self, request) -> Tuple[List[int], List[int]]:
        # Get nested model parameters and turn into long list
        flattened_weights, self.model_shape = utils.get_flat_weights(self.model)

        # Pad list until length 2**20 with random numbers that mimic the weights
        flattened_weights, self.model_length = utils.pad_to_power_of_2(
    flattened_weights, self.rlwe.n, self.WEIGHT_DECIMALS)
        # Turn list into polynomial
        poly_weights = Rq(np.array(flattened_weights), self.rlwe.t)

        # get gradient of weights if round > 1
        if request == "gradient":
            gradient = list(np.array(flattened_weights) - np.array(self.
    flat_params))
            poly_weights = Rq(np.array(gradient), self.rlwe.t)

        # Encrypt the polynomial
        c0, c1 = self.rlwe.encrypt(poly_weights, self.allpub)
        c0 = list(c0.poly.coeffs)
        c1 = list(c1.poly.coeffs)
        return c0, c1
```

Step 4: Aggregation and Partial Decryption

After receiving all the encrypted ciphertexts from the clients, the server must now request all of them to send a partial decryption share based on the c1sum computed in the previous step. The clients compute the partial decryption share "d" by modular operations with c1sum, their private key, and an error polynomial. The server will homomorphically aggregate all of the partial decryption shares as a variable dsum. The partial decryption is performed by the client's RLWE instance and is based on the following Equation 3.5:

$$\boldsymbol{D_i = s_i \times C_{sum_1} + e_i^* = s_i \sum_{i=1}^{N}(v_i \times a + e_1^{d_i}) + e_i^* (\mod q)} \tag{3.5}$$

(Forked) Server.py:

```
# Step 4) (~20% Less Memory)
    # Send c1sum to clients and retrieve all decryption shares d_i
    c1sum = c1sum.poly_to_list()
    d = clients[0].request_decryption_share(c1sum, timeout=timeout)
    dsum = self.strategy.rlwe.list_to_poly(d, "t")
    for c in clients[1:]:
        d = c.request_decryption_share(c1sum, timeout=timeout)
        dsum = dsum + self.strategy.rlwe.list_to_poly(d, "t")
```

Client.py:

```
# Step 4) Use csum1 to calculate partial decryption share di
    def compute_decryption_share(self, csum1) -> List[int]:
        std = 5
        csum1_poly = self.rlwe.list_to_poly(csum1, "q")
        error = Rq(np.round(std * np.random.randn(n)), q)
        d1 = self.rlwe.decrypt2(csum1_poly, self.rlwe.s, error)
        d1 = list(d1.poly.coeffs) #d1 is poly_t not poly_q
        return d1
```

Step 5: Weight Updates and Model Evaluation

The final step of the xMK-CKKS scheme is for the server to peform modular addition with the c0sum and dsum to retrieve a very close approximation of what the sum of all the plaintexts would have been if performed unencrypted with normal addition. The server will then average the aggregated updates based on the number of participating clients. The result will be the final model update for the next training round and will then be sent by the server to all the clients. The clients will overwrite its current weights with the new ones and this is also where evaluation of the new model weights are done. The modular addition is based on the following formula:

$$\boldsymbol{\sum_{i=1}^{N} m_1 \approx C_{sum_0} + \sum_{i=1}^{N} D_i (\mod q)} \tag{3.6}$$

(Forked) Server.py:

```
# Step 5) Use c0sum and decryption shares to retrieve plaintext, find avg and
    send back final weights to clients
    plaintext = c0sum + dsum #c0sum is poly_q and dsum is poly_t
    plaintext = plaintext.testing(self.strategy.rlwe.t)
    plaintext = plaintext.poly_to_list()
    avg_weights = [round(weight/len(clients)) for weight in plaintext]
    for c in clients:
        confirmed = c.request_modelupdate_confirmation(avg_weights, timeout=
    timeout)
    del c0sum, c1sum, dsum
```

Client.py:

```
# Step 5) Retrieve approximated model weights from server and set the new weights
def receive_updated_weights(self, server_flat_weights) -> bool:
    # Convert list of python integers into list of np.float64
    server_flat_weights = list(np.array(server_flat_weights, dtype=np.float64))

    if self.flat_params is None:
        # first round (server gives full weights)
        self.flat_params = server_flat_weights
    else:
        # next rounds (server gives only gradient)
        self.flat_params = list(np.array(self.flat_params) +
    np.array(server_flat_weights))

    # Remove padding and return weights to original tensor structure and set
    model weights
    server_flat_weights = self.flat_params[:self.model_length]
    # Restore the long list of weights into the neural network's original
    structure
    server_weights = utils.unflatten_weights(server_flat_weights,
    self.model_shape)

    utils.set_model_params(self.model, server_weights)
    return True
```

3.4.4 Summary

This section outlined the successful integration of xMK-CKKS within Flower's architecture, enabling privacy-preserving and secure machine learning. Doing so required integration of our custom RLWE instances into the source code of Flower, allow us the ability to implement multi-key homomorphic encryption into the federated learning training loop. The key sharing process that happens before the training loop is also a pivotal feature necessary for the encryption process.

Our modified federated training loop now allows for aggregation of encrypted model updates, while preventing any individual client or the server to decrypt any client's local data, thus enhancing the security of the process. The privacy is further improved by shifting from clients sending full model weights to only sharing the gradient of the updated weights, and in consequence requiring less trust in the server. All of the steps outlined in this section were dependent on custom message types that we integrated into the source code of the library. The following section will elaborate on the creation of these messages.

3.5 Flower Guide: Create Custom Messages

This chapter is a guide on an important aspect of our ability to integrate the xMK-CKKS into the Flower architecturehe: the implementation of custom messages. This added functionality is what allows us to send the necessary data between server and clients that lies outside the library's pre-defined messages.

Unfortunately the Flower library does not include any simple way to create your own custom messages. For this reason we are required to fork the library and modify the source code to manually add the necessary changes needed to implement this functionality. Doing so is not a trivial task and making incorrect changes can cause trouble to the existing grpc communication and other parts that will cause the federated learning process to crash.

Flower's own documentation site includes a contributor section where a 3rd party has made an attempt at on guide to successfully modify the source code, but sadly, even excluding the typos made in the guide itself, it's lacking many of the files and changes necessary. [35] By providing extensive code snippets and explanation, this section aims to serve as a comprehensive guide to implementing custom messages in the Flower library, a feature that is largely undocumented. Through our contribution, we aspire to support and help future projects in their quest to modify and improve upon federated learning frameworks.

We will now go through a guided example of creating a custom communication with similarities to how we made our protobuf messages to transfer list-converted polynomials. The first step is to fork the library itself and once you have all the files of the source code available, these are the files that needs to be modified: 'client.py', 'transport.proto', 'serde.py', 'message_handler.py', 'numpy_client.py', 'app.py', 'grpc_client_proxy.py' and lastly 'server.py'.

Now assume we want the server to send a string request to clients to have them respond with a list of integers to represent the weights ot their local model. Here's an example function from client.py and now we will show you the necessary steps to use this function:

```
def get_model_weights(self, request: str) -> Tuple[str, List[int]]
    if request == "full_weights"
        # retrieve and return the full updated weights
    elif request == "gradient"
        # retrieve and return the gradient of the updated weights
```

3.5.1 transport.proto (Defining Message Types):

Fundamentally, the reason we're able to do these modifications is mainly due to Flower's use of Google's Protocol Buffets (protobuf), a language-agnostic, platform-neutral, extensible mechanism for serializing structured data. Protobuf is extremely flexible, allowing you to define a wide range of message types that includes integers, floats, booleans, strings or even other message types. The first thing we need to do is define our message types for the RPC communication system. For details on proto3 syntax, see the official documentation. [31]

A protobuf file consists various message types laregely divided into two main blocks, one is called ServerMessage for the types of message the server will be able to send, and the other block is the ClientMessage with a similar structure of message types. Typically, the usual practice in designing protobuf messages is to pair the server request and client response messages with similar names to indicate that they are a part of the same operation. [31] The official proto3 documentation use pairing names like ExampleRequest and ExampleResponse, while Flower's protobuf file use ExampleIns (Ins for Instruction) and ExampleRes (Res for Response). Fundamentally the choice can be arbitrary.

Within the "ServerMessage" block:

```
message GetWeightsIns {
    string request = 1;
}
oneof msg {
    ReconnectIns reconnect_ins = 1;
    GetPropertiesIns get_properties_ins = 2;
    GetParametersIns get_parameters_ins = 3;
    FitIns fit_ins = 4;
    EvaluateIns evaluate_ins = 5;
    ExampleIns get_weights_ins = 6;
}
```

Within the "ClientMessage" block:

```
message GetWeightsRes {
                string response = 1;
    repeated int64 l= 2;
}
```

```
oneof msg {
    DisconnectRes disconnect_res = 1;
    GetPropertiesRes get_properties_res = 2;
    GetParametersRes get_parameters_res = 3;
    FitRes fit_res = 4;
    EvaluateRes evaluate_res = 5;
    ExampleRes get_weights_res = 6;
}
```

After defining protobuf messages, you need to compile them using the protobuf compiler (protoc) for generate the corresponding code for your chosen programming language (in our case, Python). To compile we navigate to the correct folder and use the command:

```
$ python -m flwr_tool.protoc
```

If it compiles succesfully, you should see the following messages:

```
Writing mypy to flwr/proto/transport_pb2.pyi
Writing mypy to flwr/proto/transport_pb2_grpc.pyi
```

3.5.2 Serde.py (Serialization and Deserialization):

The ability to convert complex data types into a form that can be easily transmitted and stored, and then convert back is a vital part of distributed systems. Serialization and deserialization, often referred to as SerDe, are the processes that enable this conversion. Our next step is to add functions in the serde.py file for our defined RPC message types. The file will handle conversion of python objects into byte streams (serialization) and vice versa (deserialization).

Each server and client request-response pair requires 4 functions to be added to the serde.py file. One function to serialize our data on the server side from python data to bytes to be transmitted to the server. A second function client-side to deserialize the bytes received into python data. A third function client-side to serialize the python data back into bytes to be transmitted to the server. And lastly the fourth function for the server to deserialize the received bytes back into python data:

```
# on server serialize from python to bytes
def request_weights_to_proto(request: str) -> ServerMessage.GetWeightsIns:
    return ServerMessage.GetWeightsIns(request=request)

# on client deserialize from bytes to python
def request_weights_from_proto(msg: ServerMessage.GetWeightsIns) -> str:
    return msg.request
```

```
# on client serialize from python to bytes
def reply_weights_to_proto(response: str, l: List[int]) -> ClientMessage.
    GetWeightsRes:
    return ClientMessage.GetWeightsRes(response=response, l=l)

# on server deserialize from bytes to python
def reply_weights_from_proto(res: ClientMessage.GetWeightsRes) -> Tuple[str, List
    [int]]:
    return res.response, res.l
```

3.5.3 grpc_client_proxy.py (Sending the Message from the Server):

This file is responsible for the low-level interaction between client and server. The function will serializes messages from the server and deserialize the response from the clients.

```
def request_vec_b(self, weightformat: str, timeout: Optional[float]) -> List[int
    ]:
    request_msg = serde.request_weights_to_proto(weightformat) # serialize
    res_wrapper: ResWrapper = self.bridge.request(
        ins_wrapper=InsWrapper(
            server_message=ServerMessage(get_weights_ins=request_msg),
            timeout=timeout,
        )
    )
    client_msg: ClientMessage = res_wrapper.client_message
    client_weights = serde.reply_weights_from_proto(client_msg.get_weights_res) #
     deserialize
    return client_weights
```

3.5.4 message_handler.py (Receiving Message by Client):

This file determines how received messages are processed. These functions takes the message payload from the server, deserialize it and then pass to the appropriate function in your clients, then serialize the results again before sending it back to the server.

Within the handle function:

```
if server_msg.HasField("get_weights_ins"):
    return _request_local_weights(client, server_msg.get_wights_ins), 0, True
```

Add a new function:

```
def _request_local_weights(client: Client, msg: ServerMessage.GetWeightsIns) ->
    ClientMessage:
    weightformat = serde.request_weights_from_proto(msg) # deserialize server msg
    weights = client.get_model_weights(weightformat) # pass to client and get
    result
    client_weights = serde.reply_weights_to_proto(weights) # serialize client
    result
    return ClientMessage(get_weights_res=client_weights)
```

3.5.5 numpy_client.py:

This file is responsible for defining how the pre-defined numpyclient interacts with the server. If your client.py subclasses NumpyClient, you will need to add new methods for sending and receiving your custom messages.

```
def has_get_model_weights(client: NumPyClient) -> bool:
    return callable(getattr(client, "get_model_weights", None)) # client.py func
    name
```

3.5.6 app.py:

Here you implement the functionality for handling your custom messages. If your message was a request to perform some computation, you would write the function that performs this computation here.

Import your numpy_client.py function:

```
from .numpy_client import has_example_response as
    numpyclient_has_get_model_weights
```

Define this function above the _wrap_numpy_client function:

```
def _get_model_weights(self, weightformat: str) -> List[int]:
    return self.numpy_client.get_model_weights(weightformat)
```

Add wrapper type method inside the _wrap_numpy_client function

```
if numpyclient_has_get_model_weights(client=client):
    member_dict["get_model_weights"] = _get_model_weights
```

3.5.7 Summary

The implementation of custom messages is an important aspect of integrating the xMK-CKKS into the Flower framework. It enables sending required data between the server and clients, which lies outside of the library's pre-defined messages. The Flower library doesn't provide a simple way to create custom messages, hence our need to fork the library and manually modify the source code.

The task is non-trivial and requires careful modification to avoid disturbing the existing gRPC communication and potentially crashing the federated learning process. Flower's documentation does have a contributor's guide for modifying the source code, but it contains errors and is missing necessary files. This section provided a detailed guide with code snippets for implementing custom messages in the Flower library.

3.5.8 Chapter Summary

This chapter introduced and explored the custom modification of the Ring Learning With Errors (RLWE) encryption scheme to modify the source code of Flower for a full integration of xMK-CKKS scheme into Flower's federated learning framework.

Initially, the analysis of the RLWE implementation was conducted, which revolved around key generation, encryption, decryption, and homomorphic encryption operations. With xMK-CKKS modifications, a unique public key was generated by each client, which were aggregated by the server to create a shared public key and allow for multi-key homomorphic encryption. A set of utility functions were also created to manage model weights.

This RLWE instance with xMK-CKKS modifications was integrated into the Flower framework, allowing for a more privacy-preserving and secure federated learning process. First a key sharing process was introduced prior to the built-in training loop to initialize each client with a shared public key. Next the federated training loop was modified to allow for aggregation of encrypted model gradients, thus improving privacy and reducing the trust required in the server.

Finally, a detailed guide was provided for implementing custom messages in Flower. This was crucial for sending necessary data between server and clients that did not fall under the pre-defined messages by Flower. This process involved careful forking, debugging and manual modification of the library's source code.

The subsequent chapters will discuss the comparisons between Flower's standard federated learning process with and without our integrated lattice-based encryption. We will compare performance metrics, time execution, memory utilization, and finally discuss potential areas for further optimization.

Chapter 4

Experimental Evaluation

4.1 Experimental Setup and Data Set

4.1.1 Setup

Our project was conducted in an Amazon Web Services (AWS) environment, utilizing several different EC2 instances to fit our varying requirements throughout the project. We leveraged the capabilities of the t3.large, c5a.8xlarge, and g4dn.4xlarge instances, each with theor own balance between computational power and cost.

The t3.large instance offered an affordable starting point, while the c5a.8xlarge and g4dn.4xlarge instances provided significantly greater computational capacity, allowing us to train our models much faster for experimental evaluations. Between the c5a.8xlarge and g4dn.4xlarge instances, the latter provided GPU-accelerated processing, which significantly sped up the training of models. It's important to note though that our encryption process, involving transmission of large polynomials and CPU-based homomorphic operations, didn't benefit much from GPU acceleration.

For development, we used Visual Studio Code's remote development features, allowing us to SSH into the EC2 instances from any device. This enabled us to work collaboratively and in real-time, negating the need for a GitHub repository for constant pushes and pulls. Since we were a team of two, this setup was more efficient and did not require any version control or merging processes. This flexible and collaborative setup contributed greatly to the efficiency of our work. Our finalized code can be found on GitHub[1] or Zenodo[2].

[1]https://github.com/MetisPrometheus/MSc-thesis-xmkckks
[2]https://zenodo.org/record/8135902

4.1.2 Dataset

In total, we have gathered 7,593 COVID-19 images from 466 patients, 6,893 normal images from 604 patients, and 2,618 CAP images from 60 patients.

In this thesis, a COVID-19 x-ray lung scan dataset from a study [36] posted on Kaggle was used as training, validation and test dataset. The original image file size is 512 x 512 pixels for all images which was then resized to 128 x 128 pixels. The dataset was split into two classifications: COVID and NON-COVID, where 7593 images was gathered from 466 COVID-19 infected patients and 6893 images was gathered from 604 normal patients. In total, we had 14486 images with close to a 50/50 split for the classification.

Below figures show samples of the dataset.

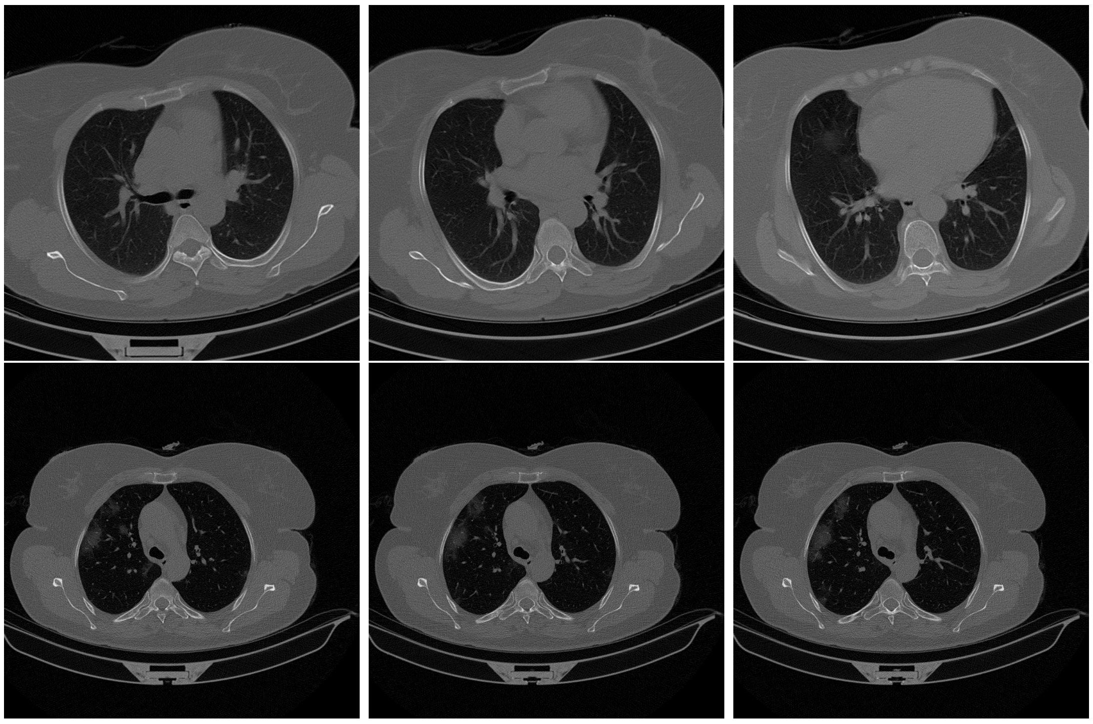
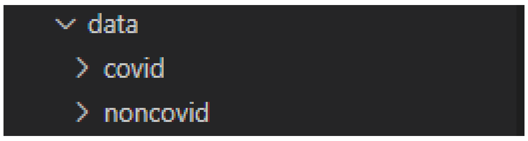


Figure 4.1: An example of an X-ray lung scan image taken from the dataset. The first row is COVID-19 negative, the second row is COVID-19 positive

The X-ray lung scan images were placed into a folder directly which was divided into "covid" and "noncovid" folders. Figure 4.2 below shows dataset folder structure.



```
⌄ data
  › covid
  › noncovid
```

Figure 4.2: Dataset folder structure

Such folder structure was required as we wanted to limit the amount of data we used but at the same time keep a classification balance when training the model. We obtained 2000 random images from each classification and then converted the dataset over to ".npy" format before splitting them into train and test datasets. The dataset was split with 75% of the sample used for training where 10% of the samples was again split into validation dataset and the remaining 25% was used as test dataset. In other words, 300 records was used as validation, 2700 records was used for training and the remaining 1000 was used as test dataset.

Table 4.1 shows the data segregation.

Dataset	**Rows**	**Label**
Training	1500	Positive
	1500	Negative
Test	500	Positive
	500	Negative

Table 4.1: Dataset description

4.1.3 Preprocessing

The data preprocessing performed in this implementation consists of both grayscaling and labeling. Grayscaling was applied to simplify algorithms and as well eliminate the complexities related to computational requirements. We also created corresponding labels for the images and used the "progressbar" library to visualize the conversion process of both the covid and non-covid datasets. The preprocessed data is then split into train and test sets for use in a machine learning model.

4.1.4 Experimental setup

In this thesis, we experimented with various number of clients ranging from 2-10 clients. We ran the federated learning process in a total of 5 rounds per client and observed the runtime and evaluation metrics to analyze the model prediction performance of the encrypted federated learning model. All the models were trained and evaluated on different AWS EC2 instances. The final tests were run on a g4dn.x4large instance where server and clients were simulated by using running multiple scripts in separate terminals.

4.1.5 Python libraries

The implementation was developed using Python 3.10.6 and libraries such as Keras, TensorFlow, NumPy, Flower 1.3.0, time and Scikit.

We used Keras and TensorFlow to create the Convolutional Neural Network model we used for prediction as well as in the preprocessing phase. Keras and TensorFlow are open-source software libraries for machine learning and can be used for a wide range of tasks, but has a particular focus on training and inference of deep neural networks.

We took use of the Numpy library to perform array operations such as concatenating the COVID and NON-COVID arrays or calculate precision and accuracy of the predicted values of a model. We used Scikit, sklearn metrics module to generate evalutation metrics on the prediction results.

The Flower library, is a FL library that originated from a research project at the University of Oxford. It's known for having components that are relatively simple to customize, extend or override and was used to create our federated learning environment. Its very modular architecture gave us a good opportunity to implement our multi-key homomorphic encryption scheme for communication between server and client.

4.1.6 Model infrastructure

Table 4.2 below shows the model summary of the CNN model used in the thesis:

Layer(type)	Output Shape	Param #
conv2d (Conv2D)	(None, 124, 124, 32)	832
batch_normalization (BatchNormalization)	(None, 124, 124, 32)	128
max_pooling2d (MaxPooling2D)	(None, 62, 62, 32)	0
conv2d_1 (Conv2D)	(None, 58, 58, 16)	12816
batch_normalization_1 (BatchNormalization)	(None, 58, 58, 16)	64
max_pooling2d_1 (MaxPooling2D)	(None, 29, 29, 16)	0
conv2d_2 (Conv2D)	(None, 25, 25, 32)	12832
batch_normalization_2 (BatchNormalization)	(None, 25, 25, 32)	128
max_pooling2d_2 (MaxPooling2D)	(None, 12, 12, 32)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 200)	921800
dropout (Dropout)	(None, 200)	0
dense_1 (dense)	(None, 2)	402
Total params: 949,002		
Trainable params: 948,842		
Non-trainable params: 160		

Table 4.2: CNN classification model summary

4.2 Evaluation metrics

We chose to measure the performances of the different execution methods through the following metrics:

4.2.1 Accuracy

Accuracy is a performance metric used in classification tasks to evaluate how well a model correctly predicts the classes of the instances in the dataset. It measures the proportion of correctly classified instances (both true positives and true negatives) among all instances in the dataset. Since we have a binary classification in this case (Positive or Negative), the accuracy is calculated as follows:

$$\boldsymbol{Accuracy = \frac{TP + TN}{TP + TN + FP + FN}}$$

Where:

- TP = True Positive
- TN = True Negative
- FP = False Positives
- FN = False Negatives

4.2.2 Precision

Precision is a performance metric used in classification taskt to evaluate the accuracy of the positive predictions made by a model. It measures the proportion of TP among the instancves that the model predicted positive (TP + FP). In other words, it measures the model's ability to avoid false positive. The precision is calculated as follows:

$$\boldsymbol{Precision = \frac{TP}{TP + FP}}$$

4.2.3 Recall

Recall is a performance metric used in classification tasks to evaluate the ability of a model to correctly identify positive instances. It measures the proportion of TP among all actual positive instances (TP + FN) in other words, it measures the model's ability to avoid false negatives. The recall is calculated as follows:

$$\boldsymbol{Recall = \frac{TP}{TP + FN}}$$

4.2.4 F1-score

The F1-score is a performance metric used in classification tasks that combines both precision and recall into a single measure. It is the harmonic mean of precision and recall, and provides a balanced measure of the model's performance on both positive and negative instances. The F1-score is calculated as follows:

$$\boldsymbol{F1 = 2 \times \frac{precision \times recall}{precision + recall}}$$

4.2.5 Time

Time is the amount of time it takes for a program or algorithm to complete a specific task or operation. In this case, the time was measured in seconds. The time calculated as follows:

$$\boldsymbol{Time = T_{end} - T_0}$$

Where $\boldsymbol{T_0}$ is the start time and $\boldsymbol{T_{end}}$ is the time the program completed.

The total execution time consists of running the following tasks:

- Open connection between server and client
- Initialize CNN model on both ends
- Preprocess test and training data
- Train the CNN prediction model at the client-side
- Encrypt weights and export encrypted weights for all clients
- Aggregate weights
- Encrypt aggregated weights and export weights back to clients
- Update model and run prediction for client
- Generate model evaluation

4.2.6 Memory usage

Memory usage is the usage the code uses to execute commands. We took use of the "memory_profiler" package which is a module that gets the memory consumption by querying the operating system kernel about the amount of memory the current process has allocated.

4.3 Experimental Results

4.3.1 Experimentation on locally trained model

Initially, we ran the COVID-19 labeling model on a plain centralized environment, with only one client involved in the training. This is typical of machine learning models, where usually only one client participates in the training process. Table 4.3 below shows the results of the precision evaluation of a locally trained model.

Metric	**Value**
Accuracy	0.927
Precision	0.925
Recall	0.925
F1-score	0.925
Time	185s
Memory	1850MB

Table 4.3: Initial results of a locally trained CNN model

4.3.2 Experimentation on plain and encrypted model with FL

We then applied federated learning framework together with 2, 3, 5 and 10 number of clients and ran the same tests we did for the locally trained model. Table 4.4 shows the results of the precision evaluation of a model trained with federated learning.

Client	**Accuracy**		**Precision**		**Recall**		**F1-score**		**Time**		**Server memory**		**Client memory**	
	Plain	HE	Plain	HE	Plain	HE	Plain	HE	Plain	HE	Plain	HE	Plain	HE
2	0.93	0.923	0.93	0.94	0.93	0.91	0.93	0.92	184s	237s	1314MB	1488MB	1040MB	1182MB
3	0.92	0.912	0.92	0.92	0.92	0.91	0.91	0.92	185s	248s	1592MB	1638MB	936MB	1128MB
5	0.93	0.904	0.93	0.96	0.93	0.86	0.92	0.90	191s	317s	1797MB	1887MB	796MB	1008MB
10	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	225s	417s	2322MB	2546MB	662MB	814MB

Table 4.4: Results from plain FL and homomorphic encrypted FL

4.3.3 Experimentation with a bigger dataset

Later on, we also tried to measure the performance of the framework with a bigger dataset. This time however, we took use of a total of 10000 images split 80/20 between training and testing. Table 4.5 shows description of the bigger dataset.

Dataset	**Rows**	**Label**
Training	4000	Positive
	4000	Negative
Test	1000	Positive
	1000	Negative

Table 4.5: Description of the bigger dataset

We still have the same split for training/validation data being at a 90/10 split, meaning that 7200 of the 8000 records are used for training, while the remaining 800 are used for validation. Table 4.6 shows the results of the precision evaluation of a model trained with federated learning. Table 4.7 shows how much of the additional time required by our encryption scheme is required for a single polynomial transmission between server and client compared to a single homomorphic operation.

Client	**Accuracy**		**Precision**		**Recall**		**F1-score**		**Time**		**Server memory**		**Client memory**	
	Plain	HE	Plain	HE	Plain	HE	Plain	HE	Plain	HE	Plain	HE	Plain	HE
2	0.973	0.967	0.98	0.98	0.97	0.97	0.97	0.97	542s	612s	1398MB	1601MB	1504MB	1906MB
3	0.968	0.964	0.98	0.98	0.97	0.97	0.97	0.96	564s	633s	1596MB	1710MB	1322MB	1730MB
5	0.962	0.933	0.96	0.98	0.97	0.89	0.97	0.93	667s	841s	1965MB	1977MB	1001MB	1175MB
10	0.938	0.927	0.94	0.98	0.94	0.89	0.94	0.92	765s	940s	2528MB	2659MB	901MB	1088MB

Table 4.6: Results with bigger dataset

Time Execution	**Data Transmission**	**Homomorphic Operation**
t3.large	1.07s (81%)	0.25s (19%)
c5a.8xlarge	0.58s (82%)	0.13s (18%)

Table 4.7: xMK-CKKS: Cause of additional time required

4.3.4 Result comparison

The figures below shows the comparison of the different results from both the small dataset and the big dataset.

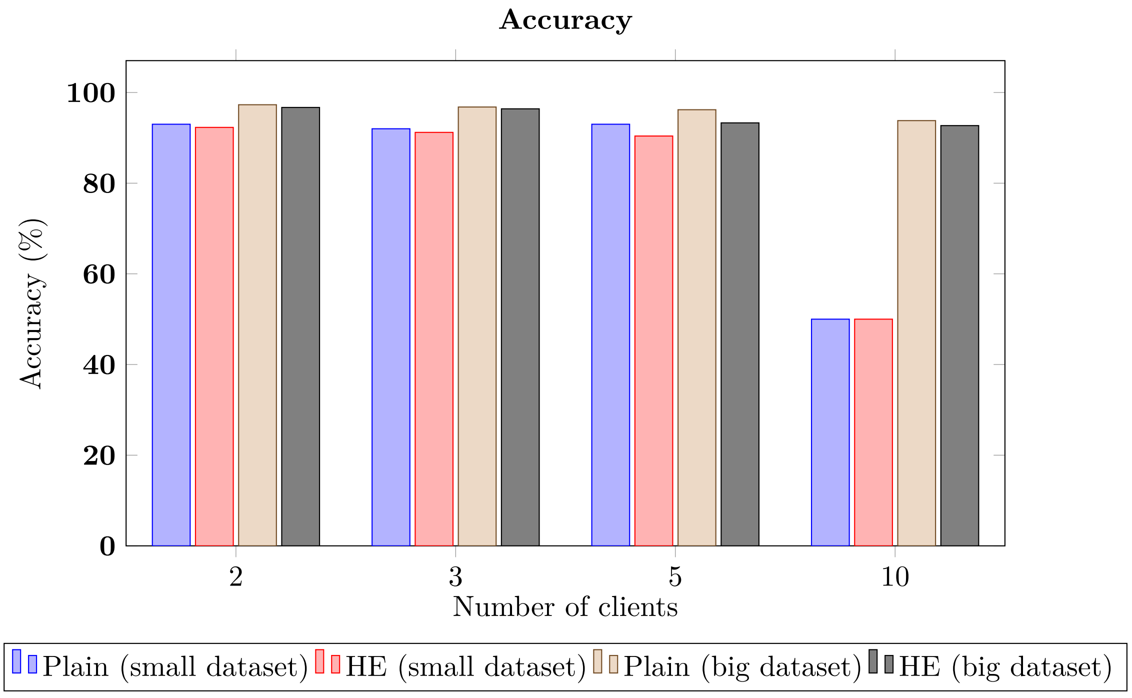


Figure 4.3: Accuracy for different numbers of clients

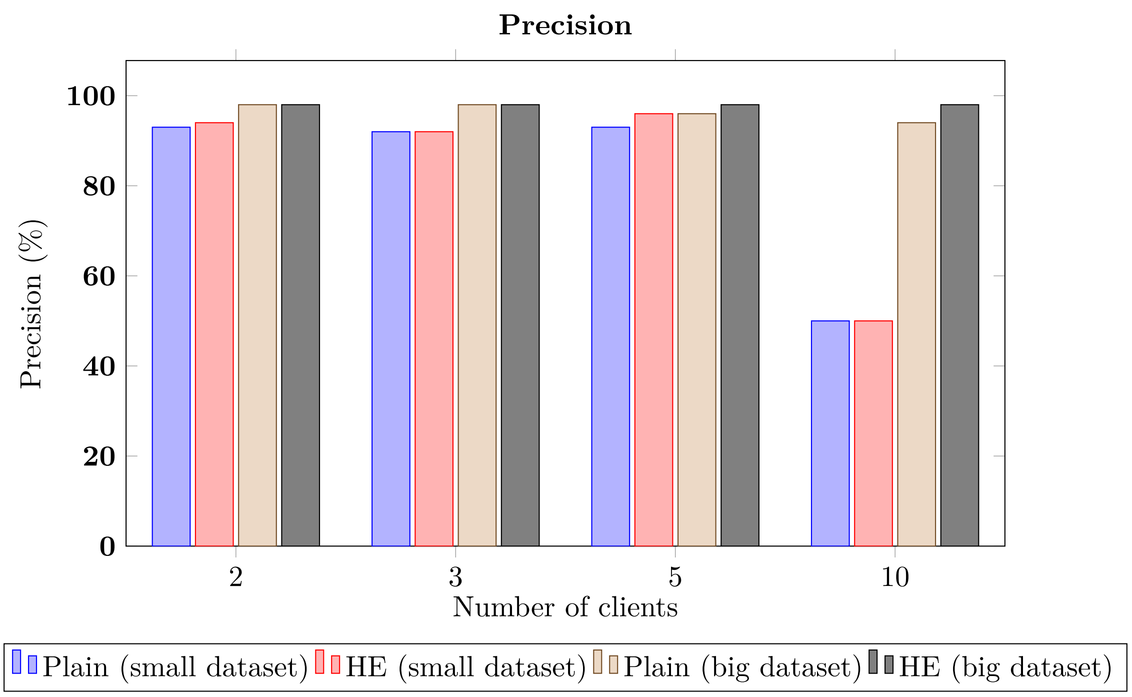


Figure 4.4: Precision for different numbers of clients

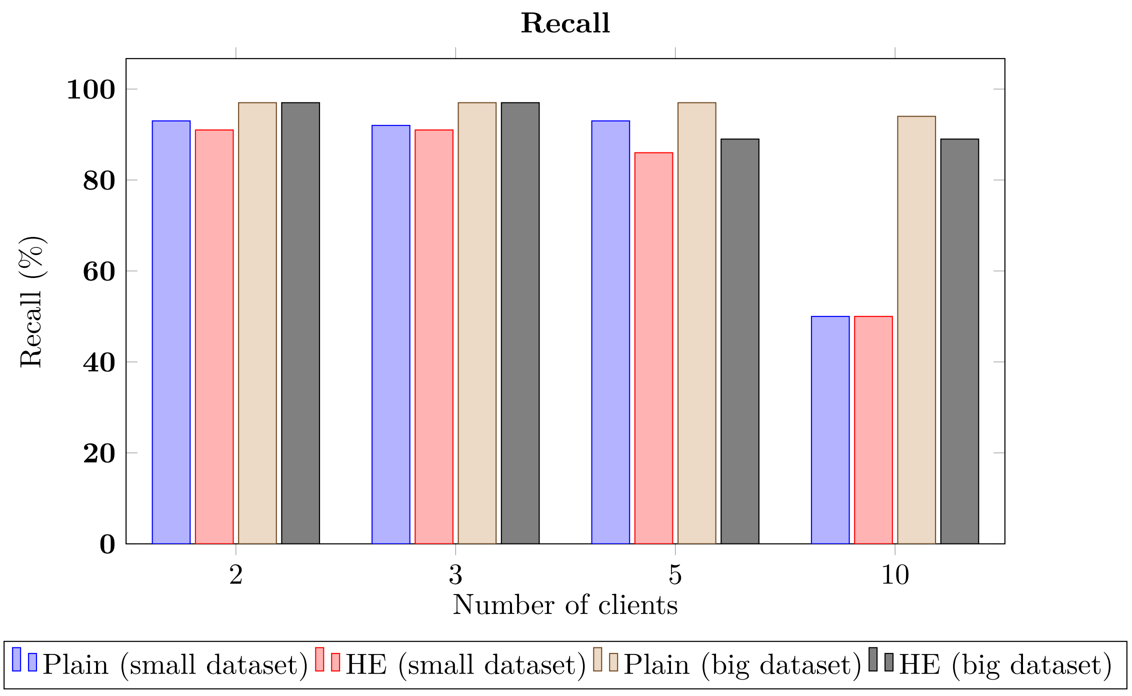


Figure 4.5: Recall for different numbers of clients

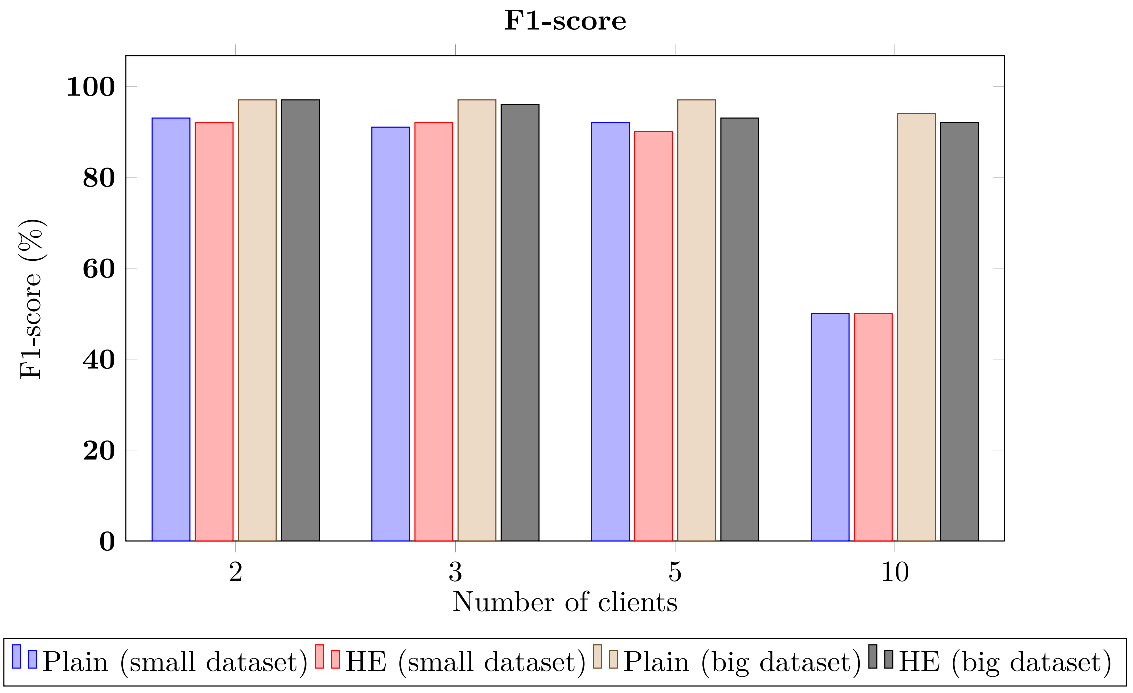


Figure 4.6: F1-score for different numbers of clients

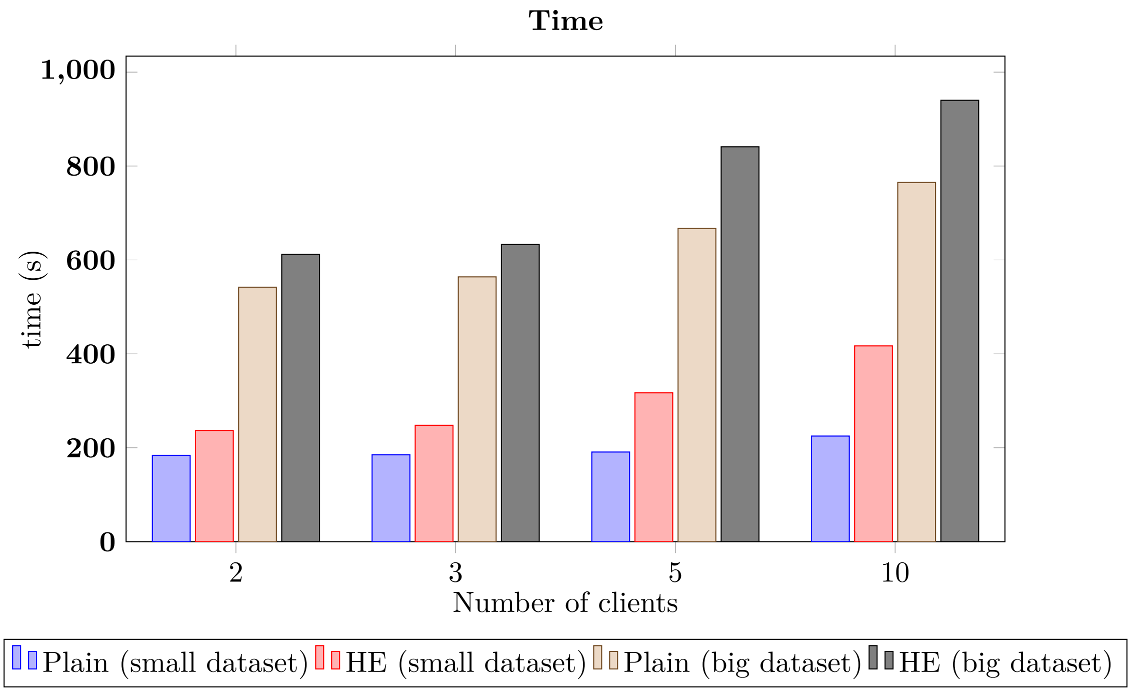


Figure 4.7: Time taken for different numbers of clients

Chapter 5

Discussion

The performance between encrypted and unencrypted models proved to be similar as demonstrated in Figures [4.3, 4.4, 4.5, 4.6]. Tests with 2, 3, and 5 clients yielded comparable scores (89%-97%), but the performance varied significantly with 10 clients. Larger datasets provided satisfactory results across all clients, but smaller datasets caused a lack of diverse data for each client, making the models unable to effectively learn.

In terms of execution time, represented in Figure 4.7, encrypted federated learning (FL) naturally took longer than its plain FL counterpart, aligning with our expectations. Furthermore while the number of clients had significant impact on execution time, the reason is primarily due to our sequential design, which aggregates data from each client one by one, resulting in an additional 4 seconds per client, per training round.

Interestingly, as demonstrated with a t3 and c5 instance in Table 4.7, more than 80% of this additional time is due to transmission of large polynomials, while less than 20% is allocated to the actual homomorphic operations. Notably, these operations maintained similar efficiency levels regardless of whether we used 3 or 9-digit coefficients, demonstrating that the operation time is mainly influenced by the length of the polynomials, not the size of its coefficient. This finding shows that the homomorphic operations are very efficient, even with over one million 9-digit coefficients.

Examining Table 4.4 reveals a notable observation. The precision measurements for both plain and encrypted data scored 50% when training on 10 clients. This can be attributed to the limited and divided dataset causing a clients' inability to discern significant patterns or correlations. The 50% score mirrors the distribution of Covid-affected and non-Covid-affected patients' data, with each client training on 300 datasets and validating on 30.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

We have proposed a privacy-preserving federated learning shceme based on homomorphic multi-key encryption with the help of RLWE and the use of Flower to protect private data. In particular, we improved the proposed solution by Wibawa et al. [2] by implementing a more secure homomorphic encryption to a federated learning framework as the previous paper only utilized one public key for all clients participating in the training. Wibawa's approach however, is not applicable in a real-world setting as not all clients are trustworthy. To fix this issue, we implemented a xMK-CKKS from Ma et al. [26] and created custom communications between client and server for the Flower framework. This approach ensures that the confidentiality of model updates are preserved by applying multi-key homomorphic encryption.

We conducted a comprehensive evaluation of a homomorphic encryption-based federated learning scheme, considering metrics such as accuracy, precision, recall, F1-score, time, and memory usage. Our evaluation involved comparing this scheme against a plain federated learning approach and a locally trained model. All tests were performed on an Amazon Web Services EC2 instance, simulating scenarios with up to 10 clients.

After a thorough evaluation and comparison of the different schemes, we concluded that the xMK-CKKS homomorphic encryption scheme effectively preserves privacy while maintaining comparable levels of accuracy, precision, recall, and F1-score compared to the other schemes. However, it is important to note that the homomorphic encryption scheme exhibits longer execution times between each round and higher memory usage per IoT device and server. These trade-offs should be carefully considered based on the project's specific needs, particularly when user privacy is of utmost importance.

6.2 Future Direction

Moving forward, there are enhancements that could further improve the privacy, efficiency, and effectiveness of our implementation. Firstly, to ensure that all clients start with identical initial weights which could provide added privacy by negating the need to share full model weights during the first round. Next, by leveraging Flower's built-in code for parallel processing for our encrypted weight sharing process. Depending on various factors, this could significantly reduce the overall execution time. Also, a more nuanced model updating process could be established by a weighted averaging approach that takes into account the size of each client's training data. Finally, some memory constraints can be mitigated by reducing the precision of model weights, allowing us to use smaller integer types. However, it is worth noting that excessively reducing the precision might hinder the model's ability to converge, resulting in a model that performs no better than majority class guessing. Therefore, a balance must be struck to ensure that memory optimization does not compromise the model's learning capacity. We think this is an area where some further research could be conducted to determine the optimal level of precision reduction that maximizes memory savings without significantly impacting model performance.

Bibliography

[1] Karim Abouelmehdi, Abderrahim beni hssane, Hayat Khaloufi, and Mostafa Saadi. Big data security and privacy in healthcare: A review. *Procedia Computer Science*, 113:73–80, 12 2017. doi: 10.1016/j.procs.2017.08.292.

[2] Febrianti Wibawa. Privacy-preserving machine learning for health institutes. UIS, 2022.

[3] Georgios Kaissis, Marcus Makowski, Daniel Rückert, and Rickmer Braren. Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence*, 2, 06 2020. doi: 10.1038/s42256-020-0186-1.

[4] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13190-5.

[5] Nguyen Truong, Kai Sun, Siyao Wang, Florian Guitton, and Yike Guo. Privacy preservation in federated learning: An insightful survey from the gdpr perspective, 2021.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL https://doi.org/10.1145/3065386.

[7] What are Convolutional Neural Networks? | IBM, . URL https://www.ibm.com/topics/convolutional-neural-networks.

[8] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2023.

[9] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated Learning: Challenges, Methods, and Future Directions, August 2019. URL https://arxiv.org/abs/1908.07873v1.

[10] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet Talwalkar. Federated multi-task learning, 2018.

[11] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet Talwalkar. Federated Multi-Task Learning, February 2018. URL `http://arxiv.org/abs/1705.10467`. arXiv:1705.10467 [cs, stat].

[12] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning, 2019.

[13] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 691–706, San Francisco, CA, USA, May 2019. IEEE. ISBN 978-1-5386-6660-9. doi: 10.1109/SP.2019.00029. URL `https://ieeexplore.ieee.org/document/8835269/`.

[14] Clement Fung, Chris J. M. Yoon, and Ivan Beschastnikh. Mitigating sybils in federated learning poisoning, 2020.

[15] 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*, pages 1322–1333, Denver Colorado USA, October 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813677. URL `https://dl.acm.org/doi/10.1145/2810103.2813677`.

[16] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients, 2019.

[17] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletarì, Holger R. Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N. Galtier, Bennett A. Landman, Klaus Maier-Hein, Sébastien Ourselin, Micah Sheller, Ronald M. Summers, Andrew Trask, Daguang Xu, Maximilian Baust, and M. Jorge Cardoso. The future of digital health with federated learning. *npj Digital Medicine*, 3(1), sep 2020. doi: 10.1038/s41746-020-00323-1. URL `https://doi.org/10.1038%2Fs41746-020-00323-1`.

[18] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür,

Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning, 2021.

[19] Enmao Diao, Jie Ding, and Vahid Tarokh. Heterofl: Computation and communication efficient federated learning for heterogeneous clients, 2021.

[20] The Flower Authors. Flower: A Friendly Federated Learning Framework. URL https://flower.dev/.

[21] Quickstart, July 2023. URL https://github.com/OpenMined/PySyft. original-date: 2017-07-18T20:41:16Z.

[22] Flower: A Friendly Federated Learning Framework, July 2023. URL https://github.com/adap/flower. original-date: 2020-02-17T11:51:29Z.

[23] Alexander Wood, Kayvan Najarian, and Delaram Kahrobaei. Homomorphic encryption for machine learning in medicine and bioinformatics. *ACM Comput. Surv.*, 53(4), aug 2020. ISSN 0360-0300. doi: 10.1145/3394658. URL https://doi.org/10.1145/3394658.

[24] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, mar 2010. ISSN 0001-0782. doi: 10.1145/1666420.1666444. URL https://doi.org/10.1145/1666420.1666444.

[25] Chris Peikert. A Decade of Lattice Cryptography. pages 1–88, 2016. URL https://eprint.iacr.org/2015/939.pdf.

[26] Jing Ma, Si-Ahmed Naas, Stephan Sigg, and Xixiang Lyu. Privacy-preserving federated learning based on multi-key homomorphic encryption, 2021.

[27] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on Post-Quantum Cryptography. Technical Report NIST IR 8105, National Institute of Standards and Technology, April 2016. URL https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf.

[28] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for Using Homomorphic Encryption for Bioinformatics. *Proceedings of the IEEE*, pages 1–16;, 2017. ISSN 0018-9219, 1558-2256. doi: 10.1109/JPROC.2016.2622218. URL http://ieeexplore.ieee.org/document/7843616/.

[29] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–412, London United Kingdom, November 2019. ACM. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3363207. URL https://dl.acm.org/doi/10.1145/3319535.3363207.

[30] Introduction to gRPC. URL https://grpc.io/docs/what-is-grpc/introduction/.

[31] Protocol Buffer Basics: Python. URL https://protobuf.dev/getting-started/pythontutorial/.

[32] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors Over Rings. URL https://eprint.iacr.org/2012/230.pdf.

[33] Nicklas Körtge. The Idea behind Lattice-Based Cryptography, June 2021. URL https://medium.com/nerd-for-tech/the-idea-behind-lattice-based-cryptography-5e623fa2532b.

[34] GitHub - yusugomori/rlwe-simple: Simple RLWE (ring learning with errors) implementation with python, . URL https://github.com/yusugomori/rlwe-simple.

[35] Creating New Messages - Flower 1.5.0. URL https://flower.dev/docs/creating-new-messages.html.

[36] Maede Maftouni, Andrew Chung Chee Law, Bo Shen, Yangze Zhou, Niloofar Ayoobi Yazdi, and Zhenyu Kong. A robust ensemble-deep learning model for covid-19 diagnosis based on an integrated ct scan images database. 06 2021.