cryptography

MDPI

Article

A Practical Implementation of Medical Privacy-Preserving Federated Learning Using Multi-Key Homomorphic Encryption and Flower Framework

Ivar Walskaar †, Minh Christian Tran † and Ferhat Ozgur Catak *,†

Department of Electrical Engineering and Computer Science, University of Stavanger, 4021 Rogaland, Norway; i.walskaar@stud.uis.no (I.W.); mc.tran@stud.uis.no (M.C.T.)

* Correspondence: f.ozgur.catak@uis.no

† These authors contributed equally to this work.

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, 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 Wibawa, using a single key for HE, our proposed solution is a practical implementation of a preprint 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 federated learning 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 a robust model performance, as demonstrated by consistent metrics including validation accuracy, precision, f1-score, and recall.

Keywords: data privacy; homomorphic encryption; multi key; medical data

check for updates

Citation: Walskaar, I.; Tran, M.C.; Catak, F.O. A Practical Implementation of Medical Privacy-Preserving Federated Learning Using Multi-Key Homomorphic Encryption and Flower Framework. *Cryptography* **2023**, *7*, 48. https://doi.org/10.3390/cryptography7040048

Academic Editor: Josef Pieprzyk

Received: 4 September 2023
Revised: 26 September 2023
Accepted: 1 October 2023
Published: 4 October 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The significance of data privacy and security in medical research has been brought to the forefront by the ongoing COVID-19 pandemic. With governments and healthcare organizations worldwide striving to gather and examine virus-related data, the apprehension regarding the improper use and unauthorized retrieval of confidential patient information has become progressively urgent [1].

Federated learning (FL) emerges as a promising solution to addressing this challenge, enabling multiple parties to collectively train a machine learning model without divulging their raw data. Nevertheless, despite implementing FL, there remains a potential risk of data leakage and privacy breaches, particularly during data transmission between the involved parties.

To tackle this issue, a potential solution that has been proposed is multi-key homomorphic encryption (MK-HE). Unlike traditional homomorphic encryption (HE), MK-HE enables computation on encrypted data using multiple private keys. This advancement offers enhanced privacy and security measures.

In our study, we present a practical implementation of a multi-key HE approach that addresses the challenges faced in real-world scenarios. Clients involved in the encryption process may have mutual distrust, leading to their unwillingness to share the same private

key. Our approach offers a more feasible and applicable solution for secure computations in such scenarios by accommodating this practical concern.

This study aims to investigate a cutting-edge encryption technique within an FL system. Additionally, we enhance the existing encryption capabilities of the Flower library by implementing our encryption approach, offering it as an alternative to the current Transport Layer Security encryption. We conduct experiments to compare the precision performance and runtime of three scenarios: local training with no encryption, FL without encryption, and FL with our encryption technique applied. These experiments enable us to assess the impact of encryption on model performance and runtime in an FL setting.

Using sensitive data, including medical data, in machine learning algorithms has risen significantly. Nevertheless, disclosing such data presents a substantial risk to individuals and organisations' privacy and security. Consequently, a crucial demand exists for privacy-preserving machine learning techniques that facilitate the analysis of sensitive data while ensuring the protection of data subjects' privacy [2].

The utilization of Ring Learning with Errors (RLWE)-based cryptography holds promise for enabling privacy-preserving machine learning. However, the practical implementation of this approach in real-world applications encounters several challenges. One such challenge is the substantial computational and communication overhead associated with RLWE-based encryption, which can result in slow training times. Furthermore, the security guarantees of RLWE-based cryptography in the specific context of machine learning applications have yet to be extensively examined [3].

Combining Ring Learning with Errors (RLWE) and FL establishes an environment where individual clients are not required to surrender their data to the server to function as a machine learning model. This approach allows data transfer across borders by enabling the server to train the ML model using weights derived from the clients rather than the actual data utilized by traditional models. By adopting this method, we can retrieve client information without infringing upon data protection regulations such as the General Data Protection Regulation (GDPR), the California Consumer Privacy Act (CCPA), and other relevant regulations [4].

This study 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 FL environment. To do this we fork a Ring Learning with Errors (RLWE)-based encryption and integrate it into a forked FL library. To simulate a real world scenario for privacy-preserving machine learning for health institutions, we will be using COVID-19 data.

This study makes several significant contributions to the field of privacy-preserving machine learning in the context of medical COVID-19 X-ray lung scan data analysis:

- We begin by constructing a baseline Convolutional Neural Network (CNN) model using the Keras/Tensorflow library in Python. This baseline model is trained on a medical COVID-19 X-ray lung scan dataset. This initial step serves as the foundation for our subsequent experiments.
- To establish a performance benchmark, we train our baseline CNN model in a traditional, centralized manner. This approach replicates the conventional training setup and allows us to evaluate the precision performance of the model under standard conditions.
- We delve into privacy-preserving machine learning by implementing an unencrypted Federated Learning (FL) system. Leveraging the Flower library, we train the same baseline CNN model within this FL environment. This step is crucial for understanding the impact of FL on model performance.
- Building on our exploration of FL, we take a significant step forward by introducing an innovative privacy-preserving technique. We implement a Ring Learning With Errors (RLWE)-based multi-key Homomorphic Encryption (HE) scheme within the same FL environment. This addition aims to protect sensitive medical data while allowing collaborative model training.
- To assess the effectiveness of our privacy-preserving technique, we conduct precision performance tests under identical conditions for both the unencrypted FL system and the

FL system enhanced with RLWE-based multi-key HE. This comparative analysis sheds light on the trade-offs and advantages of privacy preservation in medical data analysis.

The remainder of the study is structured as follows: In Section 2, we provide a comprehensive review of related works and the state-of-the-art in medical privacy-preserving federated learning, highlighting the significance of our proposed scheme and its contributions to the field. Section 3 presents a detailed explanation of the xMK-CKKS homomorphic encryption scheme, outlining how it improves the privacy and security of federated learning. The experimental setup and evaluation methodology are covered in Section 4, where we outline the datasets, hardware configurations, and performance metrics used to assess the effectiveness of our approach. We present the evaluation results, comparing the encrypted federated learning scheme against plain federated learning and locally trained models. The trade-offs, advantages, and limitations of our scheme are thoroughly discussed. In Section 5, we discuss the results. Finally, in Section 6, we conclude the study, summarizing our findings and contributions and outlining the implications for practical implementations of privacy-preserving federated learning in medical and healthcare domains.

2. Related Work

This section provides essential background information on various topics addressed in this study. Firstly, Convolutional Neural Networks (CNN) are artificial neural networks utilized for image classification tasks, which will be employed to train models on COVID-19 lung scans. Next, the basics of privacy-preserving FL are covered, elucidating the relatively recent paradigm that enables distributed devices to collaborate and train a shared prediction model while maintaining local data privacy. The open-source Python library Flower is chosen for implementing FL due to its modular architecture and ease of customization to meet specific requirements. The section also briefly touches upon the built-in communication in Flower and the necessary modifications made to enable custom communication. A comparison is drawn between traditional and lattice-based encryption to discuss how the latter offers potential solutions to vulnerabilities introduced by the former. Specifically, the Ring Learning With Errors (RLWE) encryption scheme is highlighted as a more memory- and time-efficient alternative to the Learning With Errors (LWE) cryptographic problem. The study further addresses encryption terms such as multi-key HE. Finally, the conceptual and mathematical foundations of xMK-CKKS, a proposed multi-key HE (MKHE) scheme based on the CKKS scheme, are outlined.

Li et al. [5] present PPMA, a scheme addressing smart grid data aggregation privacy concerns by allowing aggregation of users' electricity consumption data across different ranges while preserving privacy. Their analysis shows PPMA's resilience to strong adversaries and minimal computational and resource costs. In comparison, our research shares the goal of privacy-preserving data aggregation but advances the field further. We employ advanced encryption techniques for more robust privacy protection and introduce an efficient data compression algorithm, significantly reducing computational overhead.

Pu et al. [6] present PMAP, a lightweight, privacy-preserving mutual authentication and key agreement protocol for Internet of Drones (IoD) environments. PMAP employs physical unclonable functions (PUF) and chaotic systems to establish secure session keys and mutual authentication between communication entities in IoD systems. Their evaluation showcases PMAP's resilience to security attacks and superior performance in terms of computation cost, energy consumption, and communication overhead when compared to existing AKA and IBE-Lite schemes.

Sala et al. [7] introduce a re-configurable design that combines a Physical Unclonable Function (PUF) and a True Random Number Generator (TRNG) on an FPGA platform, using the Delay-Difference-Cell (DD-Cell) as an entropy source. Their work achieves a favourable trade-off between PUF and TRNG performance, showcasing competitive results in compactness and TRNG throughput compared to existing PUF+TRNG designs. In contrast, our research focuses on a different aspect of IoT security by proposing a

lightweight and privacy-preserving multi key cryptography, ensuring the confidentiality of data during communication and aggregation processes in untrusted settings.

Sun et al. [8] propose MADAR, a privacy-preserving mutual authentication framework for Vehicular Ad Hoc Networks (VANETs) that addresses the challenge of efficient anonymous authentication. MADAR combines identity-based signature schemes and distinguishes between inner-region and cross-region authentications to enhance efficiency while providing asymmetric inter-vehicle mutual authentication and resistance to computational denial-of-service attacks. Our study and their work target different network environments and security challenges, complementing them in enhancing privacy and security in distinct contexts.

2.1. Convolutional Neural Network

CNNs are a specific type of artificial neural network widely employed in deep learning for tasks such as image classification and computer vision. Unlike traditional approaches that involve laborious feature extraction methods, CNNs offer a more efficient and scalable solution. They leverage matrix multiplication and principles from linear algebra to recognize patterns within images. This enables CNNs to automatically learn and extract relevant features directly from the raw input data, reducing the need for manual feature engineering. As a result, CNNs have revolutionized image processing and become the go-to architecture for various computer vision tasks [9,10].

In this study, we aim to train a CNN-based model using COVID-19 lung scans in two different settings: a centralized location and a decentralized environment. We will utilize the same set of parameters in both scenarios to ensure a fair comparison. The objective is to measure the variations and performance differences between training the model locally and employing FL techniques, with and without post-quantum secured model updates. By conducting these experiments, we can assess the impact of decentralization and post-quantum security measures on the training process and model performance.

2.2. Privacy-Preserving FL

FL is a machine learning approach that facilitates the training of models across distributed devices or servers without transferring data to a centralized location. This method was first introduced in 2016 by Google researchers as a solution to the challenges associated with training machine learning models on large and sensitive datasets while ensuring data privacy and security. FL allows the model to be trained locally on individual devices or servers, with only the model updates being shared and aggregated instead of raw data. This decentralized approach preserves privacy and reduces communication and bandwidth requirements, making it suitable for scenarios with limited network connectivity or stringent privacy regulations [11].

The traditional approach of centralizing data in FL is replaced with a decentralized training process. Instead of transferring data to a central server, the training is conducted locally on each participating device using the data available on that specific device. The local models generated on each device are then aggregated to create a global model. This global model is then sent back to each device, which is used to refine and improve the local models. This iterative training, aggregation, and refinement process continues until the global model converges to a desired level of accuracy. By training locally and aggregating models, FL enables privacy-preserving machine learning while benefiting from multiple devices' collective knowledge [12].

FL offers numerous advantages compared to conventional centralized machine learning methods. One of its key strengths is preserving privacy, as data remain localized on individual devices without being shared with external parties. FL also facilitates training models on expansive and heterogeneous datasets, enhancing the accuracy and generalization capabilities of the models. By leveraging diverse data sources, FL enables the capturing of a broader spectrum of patterns and variations, leading to more robust and reliable models [13].

However, several adversarial threats must be considered despite the significant advantages of employing FL over traditional learning methods. These threats include model poisoning attacks [14], inference attacks [15], sybil attacks [16], model inversion attacks [17], deep leakage, and others [18]. These various attacks exploit vulnerabilities to either leak or reconstruct sensitive data or introduce backdoor functionality.

FL can be categorized into three distinct types [19]:

1. Centralized FL is a variant of FL characterized by a centralized server that orchestrates the training process across multiple devices. While the data remain on the devices, updates to the model parameters are transmitted to the server, which aggregates them and updates the global model [20].
2. Decentralized FL is a variant of FL where no centralized server exists. Instead, the training process is coordinated in a decentralized manner among 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 from each node [20].
3. Heterogeneous FL is a variant of FL wherein devices or nodes possess varying capabilities, such as processing power, storage capacity, or network bandwidth. The training process is designed to accommodate these disparities, with computation and communication tasks allocated to each device or node based on its capabilities [21].

While FL inherently provides a certain level of data privacy, our study focuses on implementing privacy-preserving FL, as depicted in Figure 1. This implementation incorporates additional measures, including differential privacy to introduce noise to the model updates and multi-key HE to enable encrypted aggregation of model updates. We have chosen to employ centralized learning in this project to showcase model aggregation through a single centralized server, which allows for better management and control over the encryption keys typically handled by the central server. Additionally, we will integrate a proposed solution to mitigate model inversion attacks.

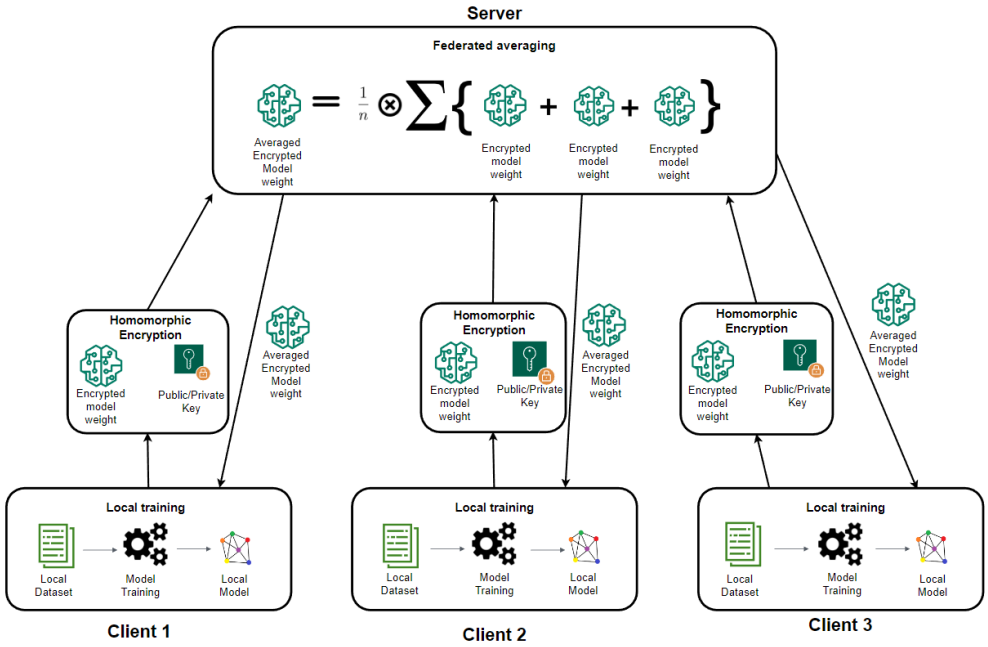

Figure 1. Centralized FL with privacy preserving encryption.

2.3. Flower: A FL Framework

Flower (Flwr) is an open-source Python library for FL. Its primary purpose is to provide a convenient and scalable infrastructure that facilitates the seamless movement of machine learning models between training and evaluation on client devices and the aggregation of model updates on servers. Flower offers a unified approach for model training, analytics,

and evaluation across different workloads, machine learning frameworks (e.g., TensorFlow, PyTorch), and programming languages. This flexibility allows developers to work with Flower using their preferred tools and frameworks [22]. Another noteworthy framework in the field is PySyft, developed by OpenMined. Although still under development, PySyft shows promise as a potential candidate for the future industry standard in privacy-preserving FL. Once fully developed and documented, PySyft may offer advanced features and functionality that further enhance the privacy and security aspects of FL. It is worth keeping an eye on its progress [23].

A key reason we chose this framework is its very modular and open architecture that allows us to easily subclass core components such as the clients, server, and the aggregation method to more easily tailor things to our specific needs. Unfortunately, the feature for creating 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 [24]. Due to Flower's third party documentation on creating custom messages including errors and lacks many of the necessary changes needed, we will in the solution approach section show, through an example, how to successfully modify the code.

2.4. Homomorphic Encryption

Unlike today's traditional encryption techniques, HE allows for the ability to perform computation with encrypted data. This type of encryption can be split into three categories: partial, somewhat, and fully HE [25].

1. Partially HE (PHE) schemes only allow for a single operation, such as addition or multiplication, to be performed over encrypted data an arbitrary number of times.
2. Somewhat HE (SHE) schemes allow for bounded computation of a set of operations. An example is the BGN encryption schemes that allow for an arbitrary number of additions and a single multiplication operation.
3. Fully HE (FHE) schemes allow for an arbitrary number of both addition and multiplication [26]

HE and FHE have significantly improved since their inception by Rivest, Adleman, Dertouzos (RAD) in 1978 and Craig Gentry in 2009. However, despite these advancements, there remain notable drawbacks to consider. One significant challenge is the increased computational overhead required for encryption, decryption, and mathematical operations compared to traditional methods, which can reduce overall system performance and efficiency. Additionally, data expansion is a concern associated with HE, given that the encrypted ciphertext is typically larger in size than the corresponding plaintext, leading to increased storage and communication requirements. This expansion poses challenges in situations with limited bandwidth or storage capabilities. Noise management also presents a crucial issue in HE as each homomorphic evaluation introduces additional noise to the ciphertext. Over time, the accumulation of noise can reach a point where the ciphertext becomes undecryptable. To address this issue, techniques such as bootstrapping, modulus reduction, or dimension reduction are employed to manage and mitigate noise accumulation. Considering these challenges when implementing HE in practical systems is essential. Ongoing research aims to address these limitations and further enhance the efficiency and effectiveness of HE techniques.

Implementing FL is a significant step towards improved security, as it allows clients to conduct training on their sensitive data locally without sharing it with other parties. However, adding HE can further enhance the security of this process. Clients can encrypt their model updates by employing HE, enabling the server to aggregate them without accessing the raw training data. This technique, known as private summation, provides an additional layer of security and privacy to the FL framework [27].

Our study aims to practically implement the theory outlined in Section 2.9, focusing on private summation using the xMK-CKKS scheme. This scheme utilizes HE to securely aggregate the encrypted model updates within the FL framework. Incorporating this tech-

nique will provide heightened security and privacy for sensitive data, ensuring protection against potential breaches or unauthorized access.

The xMK-CKKS encryption system is a significant improvement in keeping the process of federated learning private. It enhances efficiency by utilizing a collaborative approach to generating public keys, which reduces the computational load on individual clients. This system encrypts model updates with a combined public key derived from individual client public keys, enhancing privacy protection. This collaborative encryption ensures that model updates remain unknown to the server and other clients, ensuring data privacy. Additionally, the partial decryption process, which involves multiple clients, adds another layer of security.

2.5. Traditional vs. Lattice-Based Encryption

Encryption methods like RSA and ECC rely on the complex computation of large integers and discrete logarithms for security measures. However, quantum computers can utilize algorithms like Shor's Algorithm to solve these problems at a much faster pace than classical computers, resulting in traditional encryption schemes becoming vulnerable. Post-quantum cryptographic algorithms are being developed and standardized to ensure secure encrypted communications and data in the face of advancing quantum computers. These algorithms are designed to withstand attacks from quantum computers and provide robust and secure encryption methods. Research and development efforts are continuously ongoing to guarantee the long-term security of encrypted communications and data.

As quantum computer threats loom, researchers have diligently explored new methods to safeguard against quantum attacks. One up-and-coming area of focus is lattice-based cryptography, which employs intricate mathematical problems derived from lattice theory [28]. The critical establishment algorithms based on lattice theory are advantageous due to their simplicity, efficiency, and parallelizability. Moreover, the security of these algorithms is provably guaranteed under worst-case assumptions [28,29]. In this project, we aim to augment a Ring Learning With Errors (RLWE) scheme and integrate it into our federated learning architecture to enable multi-key HE. RLWE is a mathematical problem that leverages lattices over polynomial rings to create an encryption scheme. We will delve into a detailed discussion on RLWE in Section 2.8.

2.6. Multi-Key Homomorphic Encryption

In a typical plain HE system, operations can only be performed on data that has been encrypted using the same key [30]. However, this approach could be better suited for an FL environment, particularly when preserving privacy. Requiring all clients to use the same private key poses significant security concerns and directly undermines the objective of ensuring client data confidentiality. If all clients share a common private key, a malicious client could decrypt the learning updates from other clients and carry out model inversion attacks to access sensitive information about their local data . This scenario is especially problematic when dealing with sensitive healthcare or financial information. Therefore, an alternative approach that provides stronger privacy guarantees and mitigates these risks, such as multi-key HE, is necessary for FL settings.

Multi-Key HE, also known as MKHE, is a highly secure variant of HE. This encrypted computation method enables servers to perform computations on encrypted data from various clients, even using different private keys. In this process, each client generates a public key based on their private key, which the server then combines to create a single public key shared with all clients. With this public key, clients can encrypt their data, but the ciphertext can only be decrypted once the server has received partial decryption shares from all participating clients. This collaborative decryption process ensures that the server cannot access complete decryption capability, thereby preserving privacy and security in the FL setting.

2.7. Protobuf and gRPC

Our Flower system incorporates gRPC and Protocol Buffers (protobuf) to enable efficient communication between the server and clients. gRPC, an open-source and high-performance framework developed by Google, streamlines the Remote Procedure Call (RPC) communication process. RPC communication allows a program on one machine to call a function on another networked machine as if it were a local call without the added complexities of network communication. Similarly, protobufs, also developed by Google, provide a language-agnostic and platform-neutral mechanism for serializing and deserializing structured data. Within our project, the server and clients must convert Python objects into bytes for data transfer and then back into Python objects upon receipt of data. As a critical project component, we have modified the original Flower source code to allow for customized protobuf messages. This customization feature enables the implementation of our MKHE scheme within Flower's FL architecture, ensuring secure and privacy-preserving computations during training.

2.8. Ring Learning with Errors (RLWE)

Ring Learning With Errors (RLWE) is a variant of the Learning With Errors (LWE) problem deeply rooted in lattice theory. These mathematical problems are widely known to be challenging for quantum computers to solve efficiently. The complexity of the challenge lies in solving an equation with slight random noise or errors added to the results. Despite the small size of the errors, they are enough to obscure the solution and make the problem computationally demanding. Consequently, cryptographic schemes based on RLWE and LWE are considered secure, providing a solid foundation for privacy-preserving techniques in various applications, including FL ([29], pp. 30–33, [31,32]).

RLWE and LWE lies in the algebraic structures they employ. While LWE operates in vector spaces over finite fields, RLWE operates in polynomial rings. This structural variance presents certain advantages and complexities.

Regarding efficiency, RLWE surpasses LWE due to ring operations' speed and memory benefits. Ring operations are known to be comparatively faster and require less memory than their vector space counterparts. This efficiency advantage can be beneficial in practical implementations of RLWE-based cryptographic schemes. However, working with polynomial rings adds complexity, necessitating knowledge of number theory and abstract algebra. A firm grasp of these mathematical concepts is essential to utilize polynomial rings in RLWE-based encryption schemes effectively.

A ring can be defined as a group of polynomials incorporating a single unknown variable. The process of securing plaintext involves its transformation into a sequence of integers. This conversion facilitates the expression of plaintext as a polynomial, where each value in the plaintext is assigned a coefficient in the polynomial. These coefficients represent integers within a modular arithmetic system, usually selected with a prime number q as the modulus. This modulus enables a convenient and symmetrical depiction of the data. When adding polynomials modulo q, the coefficients are added together, and the result is reduced by applying the modulus q. However, multiplying polynomials results in a higher degree polynomial, which necessitates reduction back to the original degree through modulus reduction. The symmetric value range provided by the modulus q is beneficial during modulus reduction and other operations. Any errors or noise introduced, frequently modelled as zero-mean Gaussian distributions, are generated within this value range, usually with a slight standard deviation.

2.9. xMK-CKKS

2.9.1. Conceptual Theory

The CKKS encryption scheme is a handy tool for securely conducting approximate arithmetic operations on encrypted data, thereby maintaining the privacy of sensitive information without the need for decryption. This is a particularly advantageous feature in scenarios where accuracy is prioritized over exact precision. Building on the foundation of

CKKS, the MK-CKKS encryption scheme enables multiple parties to perform homomorphic operations on encrypted data using their respective private keys, thus ensuring privacy while maintaining control over encryption keys. The xMK-CKKS variant, introduced in a recent preprint study authored by Jing Ma, Si-Ahmed Naas, Stephan Sigg, and Xixiang Lyu, further advances the capabilities of multi-key HE based on the CKKS scheme, with notable enhancements to efficiency and applicability in privacy-preserving contexts [27].

A recent study by Jing Ma, Si-Ahmed Naas, Stephan Sigg, and Xixiang Lyu has presented a novel privacy-preserving FL scheme, namely xMK-CKKS, that addresses privacy concerns associated with mobile services and networks. The scheme aims to mitigate privacy leakage from publicly shared information and protect against client and server collaboration. The proposed scheme employs multi-key HE based on the CKKS scheme, facilitating collaborative training and computation on encrypted data, thus preserving individual clients' data privacy. The study includes experimental evaluations to assess the performance of the xMK-CKKS scheme. The results demonstrate that the scheme provides privacy protection and preserves model accuracy, thereby ensuring the effective extraction of valuable insights from the encrypted data while maintaining data privacy and security during the FL process.

In their proposed scheme, a crucial factor in enhancing the privacy of FL is using an aggregated public key to encrypt model updates. The aggregated public key is obtained by summing the individual public keys generated by each participating client. This collaborative approach of combining public keys introduces an extra level of security, ensuring that the model update from any individual client remains obfuscated and protected from decryption or analysis by both the server and other clients.

However, the decryption of the aggregated ciphertexts requires collaboration among all participating clients. Each client holds a unique piece of the puzzle necessary to decrypt the aggregated ciphertext. Once the server has collected all the puzzle pieces from the clients, it can retrieve the plaintext corresponding to the combined model updates. The server can then calculate the average of the integrated model updates, considering the number of participating clients, and send back the finalized model updates to all clients before proceeding to the next training round.

2.9.2. Mathematical Approach

In xMK-CKKS, generating a public key for encryption is a collaborative process among the clients. On the other hand, decryption only occurs on the server side once it receives partial decryption shares from all the participating clients. The setup of this encryption scheme follows the methodology detailed in the referenced study [27].

Setup: Begin by defining the RLWE parameters, including the parameter n, the ciphertext modulus q, and three Gaussian distributions: χ for key distribution, ψ for encryption noise, and ϕ for partial decryption noise. Generate a shared polynomial a that is drawn from a uniform distribution.

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

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

Encryption (m_i, $\widetilde{b}$, a): The plaintext m_i of client i is encrypted using the following equation: (See Figure 2)

$$\begin{aligned} ct_i &= (c_0^{d_i}, c_1^{d_i}) \\ &= (v^{d_i} \times b + m_i + e_0^{d_i}, v^{d_i} \times a + e_1^{p_i}) \mod q. \end{aligned} \tag{2}$$

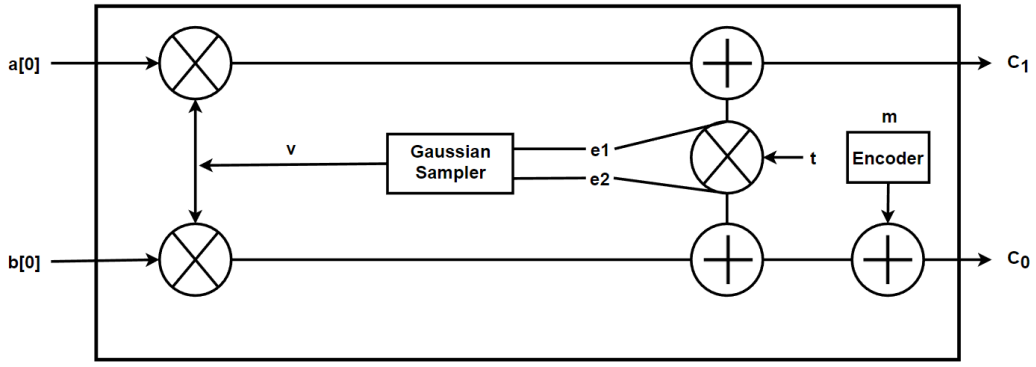

Figure 2. Utilizing aggregated public key b[0], shared polynomial a[0] and noise polynomials to encrypt plaintext m into ciphertext tuple (c_0, c_1). Homomorphic addition and multiplication are represented by "+" and "x" respectively.

Homomorphic Addition ($ct_1, \ldots, ct_n$): The summation of ciphertexts is as follows:

$$\begin{aligned} C_{\text{sum}} &= \sum_{i=1}^{N} ct_i = (C_{\text{sum}_0}, C_{\text{sum}_1}) \\ &= \left(\sum_{i=1}^{N} c_0^{d_i}, \sum_{i=1}^{N} c_1^{d_i} \right) \\ &= \left(\sum_{i=1}^{N} (v^{d_i} \times b + m_i + e_0^{d_i}), \right. \\ &\left. \sum_{i=1}^{N} (v^{d_i} \times a + e_1^{d_i}) \right) \pmod{q}. \end{aligned} \tag{3}$$

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 3)

$$\begin{aligned} D_i &= s_i \times C_{\text{sum}_1} + e_i^* \\ &= s_i \times \sum_{i=1}^{N} (v_i \times a + e_1^{d_i}) + e_i^* \pmod{q}. \end{aligned} \tag{4}$$

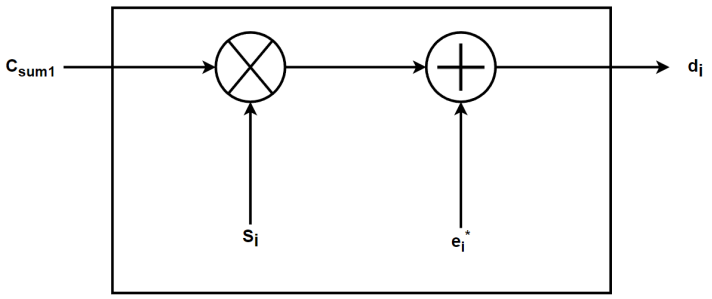

Figure 3. How the aggregated c_{sum_1} is used to compute a partial decryption share d_i using a client's private key and some added noise. Performed by all clients.

Sum of all plaintexts can be recovered as follows: (See Figure 4)

$$
\begin{aligned}
C_{sum_0} + \sum_{i=1}^{N} D_i \mod q &= C_{sum_0} + \sum_{i=1}^{N} s_i \times C_{sum_1} + \sum_{i=1}^{N} e_i^* \mod q \\
&= \sum_{i=1}^{N} (v^{d_i} \times b + m_i + e_0^{d_i}) + \sum_{i=1}^{N} s_i \times \sum_{i=1}^{N} (v^{d_i} a + e_1^{d_i}) + \sum_{i=1}^{N} e_i^* \mod q \\
&= -\sum_{i=1}^{N} v^{d_i} s_i a + \sum_{i=1}^{N} v^{d_i} \sum_{i=1}^{N} e_i + \sum_{i=1}^{N} e_0^{d_i} + \sum_{i=1}^{N} v^{d_i} s_i a + \sum_{i=1}^{N} (s_i e_1^{d_i} + e_i^*) \mod q \\
&= \sum_{i=1}^{N} m_i + \sum_{i=1}^{N} v^{d_i} \times \sum_{i=1}^{N} e_i + \sum_{i=1}^{N} e_0^{d_i} + \sum_{i=1}^{N} (s_i e_1^{d_i} + e_i^*) \mod q \\
&\approx \sum_{i=1}^{N} m_i.
\end{aligned}
\tag{5}
$$

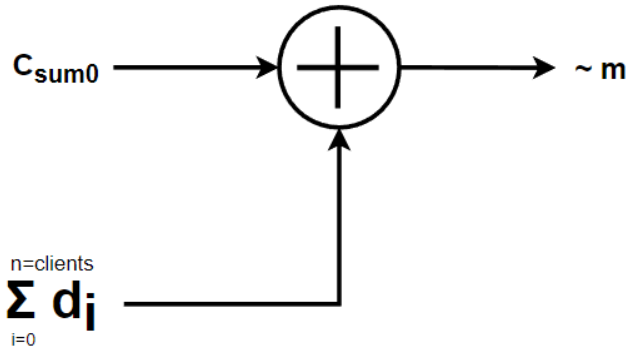

Figure 4. An approximate recovery of all plaintexts, performed by the server.

3. System Model

The privacy of individuals in healthcare institutes is jeopardized by the sharing of sensitive data across different institutions, posing a substantial challenge for machine learning applications in the medical field. To tackle this issue, our study presents a proposed solution that adopts a centralized FL framework, integrating an RLWE-based MKHE scheme for secure communication between the server and clients. Using this approach, we aim to enhance privacy-preserving machine learning in healthcare institutes and alleviate the inherent risks associated with data sharing.

3.1. Existing Approaches/Baselines

Currently, Wibawa et al. [33] have proposed a model that utilizes SHE, where each client trains their copy of the global model with their local dataset before updating the global model by sending the weights from their trained model. Like our implementation, Wibawa's model must precisely convert the ciphertext back to its original plaintext value. Instead, it provides an approximation with a small amount of noise due to the encryption process. However, a significant distinction between Wibawa's proposed solution and our proposed solution lies in the encryption approach. While Wibawa's model employs a shared key across all clients, which may not be suitable for real-life scenarios where clients may need more trust in one another, our proposed model incorporates private summation using a multi-key encryption scheme. This ensures that neither the clients nor the server can decrypt the sensitive training data of individual clients, significantly enhancing the security of our proposed model for real-life scenarios.

Ma et al. [27] presented an enhanced privacy preserving FL scheme known as xMK-CKKS. This scheme utilizes multi-key HE to ensure secure model updates. The proposed approach involves encrypting the model using an aggregated public key before sharing it

with a server for aggregation. Collaboration among all participating devices is required for the decryption process. In our study, we adopted the mathematical encryption and decryption techniques proposed by Ma et al.. We implemented them with our custom communication framework within the Flower FL library.

3.2. RLWE: Original and Improved

3.2.1. Overview of Original RLWE Implementation

The initial implementation of RLWE was obtained from a GitHub repository [34] and will be the basis for our modifications. The existing RWLE implementation consists of three primary Python files. The utils.py file contains utility functions, the Rq.py file is responsible for creating ring polynomial objects, and the RLWE.py file is used to instantiate RLWE instances for tasks such as key generation, encryption, decryption, and modular operations required for HE.

Key Generation, Encryption, and Decryption in Original RLWE

The original RLWE.py file includes a class that creates an RLWE instance, which requires four variables: n, t, q, and std. These variables represent the degrees (length) of the polynomial plaintext to be encrypted (which must be a power of 2), two large prime numbers defining the value ranges for the coefficients of the plaintext and ciphertext, and the standard deviation of a zero-mean Gaussian distribution utilized for generating the private keys and errors.

In the original RLWE implementation and our modified version, we incorporate three Gaussian distributions: χ, ψ, and ϕ, vital in key and error generation. All distributions are zero-mean Gaussian but with distinct standard deviations. χ is employed to generate private keys, ψ is utilized for generating error polynomials during the encryption process, and ϕ is introduced in our modified RLWE scheme specifically for partial decryption. In the partial decryption phase, ϕ generates error polynomials with a slightly higher standard deviation than the error polynomials used during encryption. In our project, we use a standard deviation of 3 for χ and ψ (making them equivalent), while ϕ has a standard deviation of 5.

When setting the variable t, it is crucial to ensure that the minimum and maximum coefficients of the polynomial plaintext fall within the range of $-\frac{t}{2}$ and $\frac{t}{2}$. This is of utmost importance in RLWE, as the coefficients undergo a wrapping operation within the ring. Consequently, a coefficient exceeding $\frac{t}{2}$ will be interpreted as a negative value, and vice versa.

Likewise, the ratio between the modulus for the ciphertext q and the modulus for the plaintext t holds great significance. As discussed by Peikert in the context of the hardness of LWE and its variants, as well as Ring-LWE [29], the selection of these moduli must carefully consider the desired security level and the specific parameters of the scheme. In our proof-of-concept work, we arbitrarily chose the modulus q to be a prime number greater than twenty times the modulus t, aiming to balance computational efficiency and security.

Regarding the selection of standard deviations, our choices of 3 and 5 are somewhat arbitrary. As highlighted in Peikert's papers [29,31], the standard deviation is a crucial factor in determining the security level in conjunction with other parameters. A larger ratio between the modulus q and t enables a higher standard deviation, resulting in increased noise that enhances decryption difficulty and thus augments security. However, these choices must be carefully balanced in practice, considering both security and efficiency requirements. For more precise guidance on parameter selection, 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. Listing 1 shows the generate keys method.

Listing 1. Key generation.

```
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. Listing 2 shows the encryption method.

Listing 2. encrypting a message 'm' using lattice-based cryptography with error terms and public key 'a'.

```
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: Listing 3 shows the decryption method.

Listing 3. decrypting a ciphertext 'c' using a lattice-based cryptography scheme with the secret key 's'.

```
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
```

3.2.2. Modifications to RLWE.py for xMK-CKKS Integration

Our aim of implementing secure FL using the lattice-based xMK-CKKS scheme necessitated certain modifications to the original RLWE implementation. These modifications primarily involved the functions for key generation, encryption, decryption, and the inclusion of additional utility functions. In this subsection, we will follow a similar structure as

before but with the inclusion of formulas to justify the changes made to the original RLWE implementation. All the procedures presented in this section are derived from [27], which outlines the setup of the xMK-CKKS scheme for the key-sharing process and provides a detailed explanation of the encryption and decryption steps (steps 1 to 5).

Changes in Key Generation for xMK-CKKS Integration

In the FL context, each client must have a unique private key while utilizing a shared public key. To accomplish this, we employ the formulas proposed in the xMK-CKKS paper. Each client generates a distinct public key, denoted as b, by combining a shared polynomial a with their private key s. The server subsequently aggregates these individual public keys through modular addition, resulting in a shared public key distributed back to all the clients.

For each client i, we assume that the secret key $\boldsymbol{s_i}$ is drawn from the distribution χ, and the error polynomial e_i is drawn from the distribution ψ. In contrast to the original RLWE implementation, where both the secret key and error polynomials have the same degree as the plaintext, in the xMK-CKKS scheme, the secret key s_i is required to be a polynomial of a single degree. The polynomial a is shared among all clients by the server before they generate their respective keys. Each client's public key b_i can then be computed using the formula: $b_i = -s_i \times a + e_i$.

Listing 4 shows the modified key generation function.

Listing 4. Key Generation Method for Cryptographic Operations.

```
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 has also undergone slight modifications from the original RLWE implementation. While the overall outcome remains the same, which is to encrypt a polynomial plaintext using a public key and three error polynomials, the modified scheme differs in that it only requires the first value of the aggregated public key for encryption instead of the entire public key. As a result, the encryption produces a tuple of two ciphertexts: ct = (c_0, c_1).

For each client i, let m_i be the client's polynomial plaintext to be encrypted into the ciphertext ct_i. Let $\tilde{b} =$ **b[0]** and $a =$ **a[0]**, which are the same for all clients. For each client, generate three error polynomials: v^i, $e[1]^i$, and $e[2]^i$. The polynomial v^i is drawn from the key distribution χ and is used in the computation of both c_0^i and c_1^i. On the other hand, **e[1]** and **e[2]** are unique to either c_0^i or c_1^i and are drawn from the error distribution ψ. The computed ciphertext can then be obtained using the equation:

$$ct_i = (c_0^i, c_1^i) = (v^i \times b + m_i + e_0^i, v^i \times a + e_1^i)(mod q). \quad (6)$$

Listing 5 shows the updated encryption function.

Listing 5. Encryption Method for Cryptographic Operation.

```
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 FL with xMK-CKKS, clients cannot fully decrypt ciphertexts that have undergone homomorphic operations. Instead, clients compute a partial decryption share, and once the server has successfully retrieved all the partial results, it can decrypt the aggregated ciphertexts from all the clients.

For each client i, let s_i be the client's secret key, and let e_i^* be an error polynomial sampled from distribution ϕ, assuming that ϕ has a more considerable variance than the distribution χ used in the basic scheme. The server collects all the ciphertexts c_0^i and c_1^i from each client, and let C_{sum_1} be the aggregated result of the modular addition of all the c_1^i ciphertexts. With these variables, a client can compute their partial decryption share d_i using the following equation:

$$D_i = s_i \times C_{sum_1} + e_i^* = s_i \sum_{i=1}^{N} (v_i \times a + e_1^i) + e_i^* (mod q). \tag{7}$$

Listing 6 shows the updated (now only partial) decryption function.

Listing 6. Decryption Method for Cryptographic Operations.

```
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 were made to enhance functionality. These modifications include adding methods to set and retrieve a shared polynomial a, which enables the server to generate a polynomial and distribute it to the clients for storage in their respective RLWE instances. Another modification involves including a function to convert a Python list into a polynomial object of type Rq, defined over a specified modulus. These modifications contribute to the improved versatility and convenience of the RLWE implementation.

3.2.3. Alterations to Utility.py

While our work did not require any modifications to the Rq file, we introduced several additional functions to the utils file. Specifically, we implemented functions to:

1. Get model weights and flatten them: This function retrieves the weights of a CNN model, which are organized in nested tensors, and flattens them into a single Python list. The function also returns the original shape of the weights for future reference. Clients use this function during the encryption stage, as we require the weights to be in a long Python list format to represent them as a polynomial. Listing 7 shows the extracting flattened weights and original shapes.

Listing 7. Extracting flattened weights and original shapes from a CNN model's parameters.

```
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 serves as the inverse operation of the previous function. It takes a long list of flattened weights and the original shape of the weights, allowing the function to reconstruct the weights back into their original nested tensor format. This function is used by clients at the end of each training round after the decryption process. Once all clients have sent their model updates to the server, the server computes an average and sends back the updated model weights to all clients before the next training round begins. At this point, clients need to update their local CNN models by retrieving the updated model weights and converting them back into the nested tensor format required by the CNN model. Listing 8 shows the unflattening weights using the original shapes.

Listing 8. Unflattening weights using the original shapes to reconstruct the parameters.

```
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 the plaintext has a length of 2^n. We provide clients with a function to pad their plaintext to the required length to satisfy this requirement. The target length is the smallest power of 2, more significant than the length of the plaintext.

Furthermore, CNN model weights are initially floating-point numbers ranging from -1 to 1 with eight decimal places. Since the RLWE scheme cannot directly handle floating-point numbers, we scale the weights to large integers in our CNN class. By default, the weights are scaled up by a factor of 10^8, ensuring nearly complete precision of the model as all decimal places are preserved during the encryption process. However, we have made this scaling factor a static variable that can be modified to evaluate the trade-off between the accuracy and speed of the model by using fewer decimal places in the model weights. Listing 9 shows the padding a list of flattened parameters.

Listing 9. Padding a list of flattened parameters to a specified length.

```
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. Flower Implementation of xMK-CKKS

The Flower FL library offers a versatile and user-friendly interface for initializing and running FL prototypes. It is designed to be compatible with various machine learning frameworks and programming languages, with a particular emphasis on Python, as discussed in Section 2.3.

In a typical Flower library setup, the user will implement the server and client functionalities in separate scripts, commonly named [server.py] and [client.py]. The [server.py] script primarily handles the initiation of the FL process. This typically entails creating a server instance and specifying various optional parameters, such as a strategy object. A strategy object instructs the server on aggregating and computing the model updates the clients received. Users can choose from various pre-existing strategies or customize their own by subclassing the predefined strategy class.

On the contrary, the [client.py] script is responsible for defining the logic of a client. Its main objective is to perform computations, such as training a local model on its dataset and sending the computed results (i.e., model updates) back to the server. To implement a client, the user must create a subclass of the Client or NumpyClient class provided by the Flower library and define the required methods: get_parameters, set_parameters, and fit. The get_parameters method is responsible for retrieving the parameters of the local model, while the set_parameters method updates the model with new parameters received from the server. The fit method, on the other hand, oversees the training process of the current model.

This architecture provides a flexible, scalable, and highly adaptable environment for implementing FL across various specifications. Users can utilize their preferred machine learning framework while still enjoying the inherent privacy and security benefits of FL.

While the standard Flower setup offers excellent flexibility, it does have a limitation regarding the support for custom communication messages between the server and client. This limitation becomes apparent when implementing the xMK-CKKS scheme, as it requires data exchange beyond the pre-defined message types in the original Flower library. We had to fork the library's repository and modify the source code to overcome this limitation. The modifications involved changing and compiling several related files to incorporate our custom messages. These changes were crucial in successfully implementing the xMK-CKKS scheme. The following section will provide a comprehensive guide on the necessary steps for creating custom message types. However, before delving into that, we will discuss integrating the xMK-CKKS scheme into the Flower source code, assuming that all the required message types have already been created.

3.3.1. 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. Listing 10 shows the dynamic settings for initializing an RLWE.

Listing 10. Dynamic settings for initializing an RLWE encryption scheme.

```
# 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).
```

3.3.2. Key Sharing Process

Until now, when referring to the server.py file, we have been discussing the user-created server.py file, typically used to start a Flower server. However, starting from this section onwards, when we mention server.py, we are explicitly referring to the server.py file located within the source code of the Flower library itself. This server.py file serves as the foundation for the library's operations, and our modified version allows us to make significant changes to the communication and encryption procedures between the server and clients. It is important to note that our Python code directly deals with polynomial objects for encryption and decryption computations. In contrast, the transmission of polynomials between the server and clients requires them to be converted into Python lists of integers.

Step 1: Polynomial Generation and Key Exchange

The key sharing process is initiated in the modified server.py script. Just before the training loop in Flower, our custom communication message types are employed to ensure that all participating clients possess the same aggregated public key. In the modified server.py script, the local RLWE instance is utilized to generate a uniformly generated polynomial a. This polynomial "a" is then sent by the server to all the clients, who use it to generate their private keys and corresponding public keys "b". The clients respond to the server's request by returning their public key "b".

Listings 11and 12 show the server and client side code for the first step.

(Forked) Server.py:

Listing 11. Server-side code for the first step in an RLWE-based secure computation process.

```
# 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:

Listing 12. Client-side code for the first step in an RLWE-based secure computation process.

```
# 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

After collecting all the public keys from the participating clients, the server performs modular additions to aggregate these keys. The resulting aggregated public key is then sent to all clients through a message request from the server. After training, clients store this shared public key in their RLWE instance to encrypt their local model weights. Upon receiving the shared public key, the clients respond to the server with a confirmation. Listings 13 and 14 show the server and client side code for the second step.

Listing 13. Server-side code for the second step in an RLWE-based secure computation process. The server aggregates public keys received from clients and sends the aggregated 'allpub' to clients then awaits confirmation from each client.

```
# 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).
```

Listing 14. Client-side code for the second step in an RLWE-based secure computation process. Clients receive the aggregated public key 'allpub' from the server and confirm the reception.

```
# 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.
```

3.3.3. Training Loop and Weight Encryption Process

After completing the key sharing process, the server initiates the federated training loop. During this loop, each client independently trains its local model using its local dataset. Once the training is complete, the server collects and aggregates the model updates from all clients. The server then calculates the average of the aggregated model updates and redistributes the updated model to all clients for the next training round. This process ensures collaboration and synchronization among all clients in the FL process. Listing 15 shows the loop of a distributed training process.

Listing 15. The loop that iterates through multiple rounds of a distributed training process where a local model is trained simultaneously on all participating clients in each 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 setup of the Flower library, clients transmit their unencrypted model weights to the server, which raises significant security concerns. If any client trains its model on sensitive data, an untrusted server or malicious client could perform an inversion attack to infer the client's training data based on the updated model weights. In contrast, our implementation of the xMK-CKKS scheme significantly enhances the security and privacy of this process. In our implementation, clients encrypt their model weights before transmitting them to the server. The server then homomorphically aggregates all the encrypted updates, ensuring that neither the server nor any client can read the individual model updates of other clients. This encryption scheme provides a strong layer of security and protects the confidentiality of each client's training data throughout the FL process.

Moreover, in the standard setup of Flower, clients transmit their complete updated model weights. In contrast, our approach deviates from this by sending only the gradients of the updated weights. Each client locally compares the new weights with the old ones and communicates only the weight changes to the server. Similarly, the server homomorphically aggregates all the gradients and returns the weighted average to the clients. This approach ensures that the server never has access to the complete weights of any model, thereby reducing the level of trust required from the participating clients. By transmitting only the weight differentials, the privacy and confidentiality of the client's model are further protected, as the server only receives information about the changes made to the weights rather than the total weight values.

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 following encryption will result in two ciphertexts c0, c1. The 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 (8), while the server aggregates the ciphertexts accoAfter each round, the server requests the updated weights when the clients have completed training their local models. The clients convert the nested tensor structure of the weights into a long Python list, which serves as the plaintext for encryption. Subsequently, the encryption process produces two ciphertexts, c0 and c1. The server then performs homomorphic aggregation on all the received c0 and c1 ciphertexts, resulting in the variables c0sum and c1sum.

The encryption of plaintexts is carried out by the client's RLWE instance, following the equation shown in Equation (8). On the other hand, the server aggregates the ciphertexts using Equation (9):

$$\begin{aligned} ct^{d_i} &= (c_0^{d_i}, c_1^{d_i}) \\ &= (v^{d_i} \times b + m_i + e_0^{d_i}, v^{d_i} \times a + e_1^{d_i})(\mod q) \end{aligned} \tag{8}$$

$$\begin{aligned} C_{\text{sum}} &= \sum_{i=1}^{N} ct^{d_i} \\ &= (\sum_{i=1}^{N} c_0^{d_i} \sum_{i=1}^{N} c_1^{d_i}) \\ &= (C_{\text{sum}_0}, C_{\text{sum}_1})(\mod q). \end{aligned} \tag{9}$$

Listings 16 and 17 show the server and client side code for Step 3 in the process. (Forked) Server.py:

Listing 16. Step 3 in the process where all clients encrypt plaintext 'p' into (c0 c1) polynomials and the server aggregates and sums up these polynomials 'c0sum' and 'c1sum' from all clients.

```
# 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").
```

Step 4: Aggregation and Partial Decryption

Upon receiving the encrypted ciphertexts from all the clients, the server requests each client to send a partial decryption share based on the previously computed c1sum. The clients compute their partial decryption share, denoted as "d", through modular operations involving c1sum, their private key, and an error polynomial. The server performs homomorphic aggregation on all the partial decryption shares, resulting in the variable dsum. The client's RLWE instance carries out the partial decryption process, following the equation described in Equation (10).

$$\begin{aligned} D_i &= s_i \times C_{sum_1} + e_i^* \\ &= s_i \sum_{i=1}^{N} (v_i \times a + e_1^{d_i}) + e_i^*(\mod q). \end{aligned} \tag{10}$$

Client.py:

Listing 17. Step 3 of the process involves encrypting a flat list of parameters into two lists 'c0' and 'c1' using the RLWE encryption scheme.

```
# 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.
```

Listings 18 and 19 show the server and client side code for Step 4 in the process.

Listing 18. Step 4 in the process where the server sends 'c1sum' to clients and retrieves all decryption shares 'd_i'. The shares are aggregated and summed up to obtain 'dsum'.

```
# 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")
```

Listing 19. Step 4 of the process involves computing the decryption share 'di' using 'csum1'. The function processes 'csum1' then decrypts it and returns 'di' as a list of integers.

```
# 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

In the final step of the xMK-CKKS scheme, the server performs modular addition between c0sum and dsum. This operation yields a very close approximation of the sum of all the plaintexts that would have resulted from unencrypted standard addition. The server then calculates the average of the aggregated updates, taking into account the number of

participating clients. The resulting average becomes the final model update for the next training round, which the server sends to all the clients.

Upon receiving the updated model weights, the clients overwrite their current weights with the new ones. This step also involves evaluating the performance of the new model weights. The modular addition is performed according to the following equation:

$$\sum_{i=1}^{N} m_1 \approx C_{sum_0} + \sum_{i=1}^{N} D_i(\mod q). \tag{11}$$

Listings 20 and 21 show the server and client side code for Step 5 of the process.

Listing 20. Step 5 of the process involves using 'c0sum' and decryption shares 'dsum' to retrieve plaintext then calculate the average and send back the final weights to clients. The average weights are calculated and sent to clients and memory is freed up by deleting unnecessary variables.

```
# 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.
```

Listing 21. Step 5 of the process involves receiving approximated model weights from the server and setting the new weights on the client side. The function converts the received weights then updates them and then restores the weights into the original tensor structure of the neural network model.

```
# 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. Flower Guide: Create Custom Messages

This section guides implementing custom messages, a crucial aspect of integrating the xMK-CKKS scheme into the Flower architecture. Including custom messages enables data transmission between the server and clients beyond the predefined message types provided by the library.

Regrettably, the Flower library does not provide a straightforward method for creating custom messages. Consequently, we are compelled to fork the library and modify the source code to manually introduce the required changes for implementing this functionality. It is important to note that this task is nontrivial, as making incorrect changes can disrupt the existing gRPC communication and other components, failing the FL process.

The Flower library's documentation site features a contributor section where a third party has attempted to provide a guide on modifying the source code. However, despite some typographical errors, this guide needs more files and changes necessary for the task [35]. To address this gap, our section aims to serve as a comprehensive guide to implementing custom messages in the Flower library. We provide extensive code snippets and explanations to facilitate this process, which is currently largely undocumented.

With our contribution, we aim to support and assist future projects seeking to modify and enhance FL frameworks.

Now, we will proceed with a step-by-step example demonstrating creating a custom communication system, similar to how we developed our protobuf messages for transferring list-converted polynomials. The initial step involves forking the library repository, and upon obtaining access to all the source code files, the following files must be modified: 'client.py', 'transport.proto', 'serde.py', 'message_handler.py', 'numpy_client.py', 'app.py', 'grpc_client_proxy.py', and finally 'server.py'.

Let us consider a scenario where the server needs to send a string request to the clients, asking them to respond with a list of integers representing the weights of their local models. Here is an example function from client.py. Listing 22 shows the retrieving model weights.

Listing 22. Function for retrieving model weights based on the request type.

```
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.
```

3.4.1. transport.proto (Defining Message Types)

The utilization of Google's Protocol Buffers (protobuf) by Flower has provided us with an advantageous ability to make modifications with ease. Protobuf is a language-agnostic and platform-neutral tool that enables the serialization of structured data, granting versatility by creating multiple message types, including integers, floats, booleans, strings, and even other message types. The first step in this process involves defining message types for the RPC communication system [36].

A protobuf file consists of various message types, primarily divided into two main blocks: ServerMessage and ClientMessage. The ServerMessage block includes the kinds of messages the server can send, while the ClientMessage block has a similar structure for client-side messages. Conventionally, when designing protobuf messages, it is common practice to pair the server request and client response messages with similar names to indicate their association as part of the same operation. The official proto3 documentation often uses pairing words like ExampleRequest and ExampleResponse, while Flower's protobuf file uses ExampleIns (Ins for Instruction) and ExampleRes (Res for Response). Ultimately, the choice of naming convention can be arbitrary. Listings 23 and 24 show the server and client side code for Protocol Buffer message definition.

Within the "ServerMessage" block:

Listing 23. Protocol Buffer message definition in which the 'GetWeightsIns' message includes a 'request' field of type string.

```
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:

Listing 24. Protocol Buffer message definition for the 'GetWeightsRes' message which includes a 'response' field of type string and a repeated field 'l' of type int64.

```
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 generating 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
```

3.4.2. Serde.py (Serialization and Deserialization)

Serialization and deserialization, commonly known as SerDe, play a crucial role in distributed systems as they convert complex data types into a format suitable for transmission and storage and vice versa. To facilitate this conversion for our defined RPC message types, our next step is to include functions in the serde.py file. This file will handle the serialization process of Python objects into byte streams and the deserialization process of byte streams back into Python objects.

To ensure the proper serialization and deserialization of data in each server and client request-response pair, we need to add four functions to the serde.py file. Firstly, we require a function on the server side to serialize the Python data into bytes for transmission. Secondly, on the client side, we need a function to deserialize the received bytes back into

Python data. Thirdly, on the client side, we need a function to serialize the Python data into bytes for transmission back to the server. Lastly, we need a function on the server side to deserialize the received bytes back into Python data. These four functions collectively enable seamless data conversion between Python objects and byte streams for effective communication. Listing 25 defines functions for serializing and deserializing messages.

Listing 25. Define functions for serializing and deserializing messages between the server and client using Protocol Buffers.

```
# 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
```

3.4.3. grpc_client_proxy.py (Sending the Message from the Server)

This file is crucial for smooth data exchange between the client and server. Its primary function entails the serialization of messages originating from the server and the deserialization of responses transmitted by the clients. Listing 26 shows the client-side implementation for requesting vector.

Listing 26. The client-side implementation for requesting vector 'b' in a specific weight format from a server.

```
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
```

3.4.4. message_handler.py (Receiving Message by Client)

This particular file is responsible for managing the processing of incoming messages. Its various functions are designed to efficiently extract the message payload from the server, convert it into an understandable format, and then direct it towards the appropriate function within your clients. After completing this process, the results are converted into a legible form and transmitted to the server. Listings 27 and 28 show the conditional statements.

Within the handle function:

Listing 27. Conditional statement that checks if the received 'server_msg' contains a 'get_weights_ins' field.

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

Add a new function:

Listing 28. Client Request for Local Weights with Serialization and Deserialization

```
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)
```

3.4.5. numpy_client.py

This file defines how the pre-defined numpyclient interacts with the server. If your client.py subclasses NumpyClient, one must add new methods for sending and receiving custom messages. Listing 29 shows below.

Listing 29. Function for requesting and retrieving local model weights from the client based on the provided 'weightformat' received from the server.

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

3.4.6. app.py

Here, you implement the functionality for handling your custom messages. If your message were a request to perform some computation, one would write the function that performs this computation here. Listing 30 shows the import for client.py file.

Listing 30. Imports.

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

Listing 31 defines this function above the _wrap_numpy_client function.

Listing 31. Wrapper function that retrieves model weights from a client using the specified 'weightformat'.

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

Listing 32 adds wrapper type method inside the _wrap_numpy_client function

Listing 32. If the 'numpyclient' has a 'get_model_weights' method assign the '_get_model_weights' function to the 'member_dict' dictionary with the key 'get_model_weights'.

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

4. Experimental Evaluation

4.1. Experimental Setup and Data Set

4.1.1. Setup

After receiving the encrypted ciphertexts from all the clients, the server requests each client to send a partial decryption share based on the previously computed c1sum. The clients compute their partial decryption shares, denoted as "d", through modular operations involving c1sum, their private key, and an error polynomial. Subsequently, the server performs homomorphic aggregation on all the received partial decryption shares, resulting in the variable dsum. The client's RLWE instance executes the partial decryption process, following the equation described in Equation (10). This procedure allows the server to collectively decrypt the aggregated ciphertexts without having access to individual clients' specific model updates.

The t3.large instance served as a cost-effective starting point for our experiments, while the c5a.8xlarge and g4dn.4xlarge instances offered substantially higher computational capacity, enabling faster model training for our experimental evaluations. Among these two, the g4dn.4xlarge instance, with GPU-accelerated processing, significantly expedited the model training process. However, it is worth noting that the encryption process, involving the transmission of large polynomials and CPU-based homomorphic operations, did not benefit significantly from GPU acceleration.

During development, we leveraged Visual Studio Code's remote development features, which allowed us to SSH into the EC2 instances from any device. This approach facilitated real-time collaborative work, eliminating the need for constant pushes and pulls to a GitHub repository. Since our team consisted of only two members, this setup proved to be highly efficient, as it did not require extensive version control or merging processes. The flexibility and collaborative nature of this setup significantly contributed to the overall efficiency of our work. The finalized code for our project is available on GitHub (https://github.com/MetisPrometheus/MSc-thesis-xmkckks) and Zenodo (https://zenodo.org/record/8135902 accessed on 30 September 2023).

4.1.2. Dataset

This study collected 7593 COVID-19 images from 466 patients, 6893 normal images from 604 patients, and 2618 CAP (Community-acquired pneumonia) images from 60 patients. For our experiments, we utilized a COVID-19 X-ray lung scan dataset from a study [37] available on Kaggle, which served as our training, validation, and test datasets. The original image size for all images was 512×512 pixels, which we resized to 128×128 pixels for consistency.

The dataset was categorized into two classifications: COVID and non-COVID. Specifically, we gathered 7593 images from 466 COVID-19-infected patients and 6893 images from 604 normal patients. We had 14,486 images, with a nearly balanced distribution between the two classifications, resulting in a close to 50/50 split. Figure 5 show samples of the dataset.

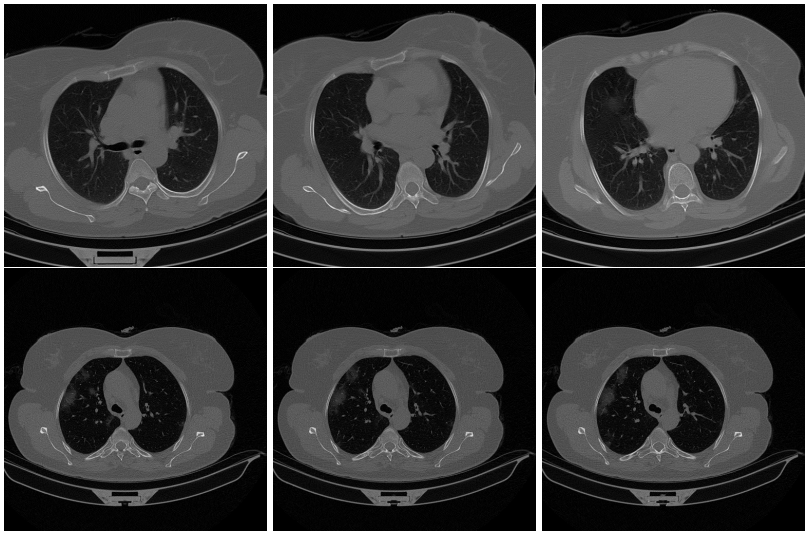

Figure 5. An example of an X-ray lung scan image taken from the dataset. The first row is COVID-19 negative, the second row is COVID-19 positive.

The X-ray lung scan images were placed into a folder directly which was divided into "COVID" and "non-COVID" folders. Figure 6 below shows the dataset folder structure.

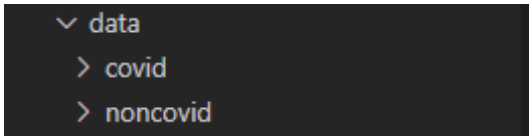

Figure 6. Dataset folder structure.

The folder structure was designed to maintain a balanced classification while limiting the data used during model training. We randomly selected 2000 images from each classification and converted the dataset to the ".npy" format to achieve this. Subsequently, we split the dataset into train and test datasets.

Specifically, 75% of the samples were allocated for training, where 10% of this training data was further set aside as the validation dataset. The remaining 25% of the data were used as the test dataset. In other words, 300 records were used for validation, 2700 for training, and the remaining 1000 for the test dataset. This partitioning ensured a balanced distribution of samples across the different sets and allowed us to evaluate the model's performance effectively.

Table 1 shows the data segregation.

Table 1. Dataset description.

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

4.1.3. Preprocessing

We conducted two main data preprocessing steps in this implementation: grayscaling and labeling. Grayscaling was employed to simplify algorithms and reduce computational complexities. Additionally, we assigned corresponding labels to the images, and to visualize the conversion process of both the COVID and non-COVID datasets, we utilized the "progressbar" library. Once the data was preprocessed, we split it into train and test sets, later utilized in

the machine learning model. This preprocessing phase ensured that the data was in a suitable format for training and evaluation, facilitating the subsequent stages of the experiment.

4.1.4. Experimental Setup

In this research, we conducted experiments using varying numbers of clients, ranging from two to ten clients. The FL process was executed over a total of five rounds per client, during which we closely monitored the runtime and evaluation metrics to analyze the model prediction performance of the encrypted FL model. All models were trained and evaluated on different AWS EC2 instances. For the final tests, we utilized a g4dn.x4large instance, simulating the server and clients by running multiple scripts in separate terminals. This approach allowed us to assess the model's performance under various client configurations and AWS instances, contributing to a comprehensive evaluation of the proposed encrypted FL scheme.

4.1.5. Python Libraries

The implementation of our research was developed using Python 3.10.6 and various essential libraries, including Keras, TensorFlow, NumPy, Flower 1.3.0, time, and Scikit.

We relied on the Keras and TensorFlow libraries to build and preprocess the Convolutional Neural Network (CNN) model used in prediction. These open-source software libraries are powerful tools for machine learning tasks, particularly emphasising the training and inference of deep neural networks.

Numpy was employed to handle array operations, such as concatenating the COVID and NON-COVID arrays and calculating the precision and accuracy of predicted model values. In addition, we utilized Scikit's sklearn metrics module to generate evaluation metrics for the prediction results.

Flower Library, originating from a research project at the University of Oxford, was our chosen FL (FL) library to create the FL environment. Notably, Flower's components are known for being relatively simple to customize, extend, or override, making it an ideal platform for implementing our multi-key HE scheme for secure communication between the server and clients. Its modular architecture played a crucial role in facilitating the integration of our encryption scheme into the FL framework.

4.1.6. Model Infrastructure

Table 2 below shows the model summary of the CNN model used in the study:

Table 2. CNN classification model summary.

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

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 crucial performance metric used in classification tasks to assess the model's ability to predict the classes of instances in the dataset accurately. It quantifies the proportion of correctly classified instances, encompassing both true positives and true negatives, among all instances in the dataset. Given the binary classification nature of our case (Positive or Negative), the accuracy can be calculated as follows using the formula:

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

where:

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

4.2.2. Precision

Precision is an essential performance metric used in classification tasks to assess the accuracy of positive predictions made by a model. It quantifies the proportion of true positive (TP) instances among all the instances the model predicted as positive (TP + FP). In other words, it gauges the model's capability to avoid false positives. The precision is calculated using the following formula:

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

.

4.2.3. Recall

Recall, also known as sensitivity or true positive rate, is a vital performance metric used in classification tasks to assess a model's ability to identify positive instances correctly. It quantifies the proportion of true positive (TP) instances among all the actual positive instances (TP + FN). In other words, it measures the model's capability to avoid false negatives. The recall is calculated using the following formula:

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

4.2.4. F1-Score

The F1-score is a fundamental performance metric utilized in classification tasks, effectively combining precision and recall into a unified measure. It represents the harmonic mean of precision and recall, evaluating the model's performance on both positive and negative instances. The F1-score is calculated using the following formula:

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

4.2.5. Time

Time refers to the duration a program or algorithm takes to accomplish a specific task or operation. In our study, we measured time in seconds. The time was calculated using the following formula:

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

where T_0 is the start time and 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 refers to the amount of memory consumed by the code during the execution of commands. To measure memory usage, we utilized the "memory_profiler" package, a module that queries the operating system kernel to determine the amount of memory allocated by the current process.

4.3. Experimental Results

4.3.1. Experimentation on Locally Trained Model

At the beginning of the experiment, we conducted training on the COVID-19 labeling model in a plain centralized environment, involving only one client in the training process. This setup is typical for machine learning models, where a single client is usually responsible for the training. Table 3 presents the precision evaluation results of the locally trained model.

Table 3. Initial results of a locally trained CNN model.

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

4.3.2. Experimentation on Plain and Encrypted Model with FL

Next, we utilized the FL framework with two, three, five, and ten clients, conducting the same tests as the locally trained model. Table 4 illustrates the precision evaluation results of a model trained using FL.

Table 4. Results from plain FL and homomorphic encrypted FL.

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	184 s	237 s	1314 MB	1488 MB	1040 MB	1182 MB
3	0.92	0.912	0.92	0.92	0.92	0.91	0.91	0.92	185 s	248 s	1592 MB	1638 MB	936 MB	1128 MB
5	0.93	0.904	0.93	0.96	0.93	0.86	0.92	0.90	191 s	317 s	1797 MB	1887 MB	796 MB	1008 MB
10	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	225 s	417 s	2322 MB	2546 MB	662 MB	814 MB

4.3.3. Experimentation with a Bigger Dataset

Subsequently, we also assessed the framework's performance with a larger dataset. In this case, we utilized 10,000 images, divided into an 80/20 split between training and testing sets. Table 5 presents the description of the larger dataset.

Table 5. Description of the bigger dataset.

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

The training/validation data split remains the same, with a 90/10 distribution. Specifically, 7200 out of the 8000 records are used for training, while the remaining 800 are allocated for validation. Table 6 displays the results of the precision evaluation for a model trained with FL. Additionally, Table 7 illustrates the time required for a single polynomial transmission between server and client compared to a single homomorphic operation, providing insight into the additional time required by our encryption scheme.

Table 6. Results with bigger dataset.

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	542 s	612 s	1398 MB	1601 MB	1504 MB	1906 MB
3	0.968	0.964	0.98	0.98	0.97	0.97	0.97	0.96	564 s	633 s	1596 MB	1710 MB	1322 MB	1730 MB
5	0.962	0.933	0.96	0.98	0.97	0.89	0.97	0.93	667 s	841 s	1965 MB	1977 MB	1001 MB	1175 MB
10	0.938	0.927	0.94	0.98	0.94	0.89	0.94	0.92	765 s	940 s	2528 MB	2659 MB	901 MB	1088 MB

Table 7. xMK-CKKS: Cause of additional time required.

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

4.3.4. Result Comparison

The figures below illustrate a comparison of the results obtained from both the small and large datasets.

5. Discussion

The comparison of performance between encrypted and unencrypted models showed similar results, as depicted in Figures 7–11. Tests with two, three, and five clients yielded comparable scores ranging from 89% to 97%, but the performance varied significantly with ten clients. Larger datasets provided satisfactory results across all clients, while smaller datasets lacked diverse data for each client, leading to ineffective learning of the models.

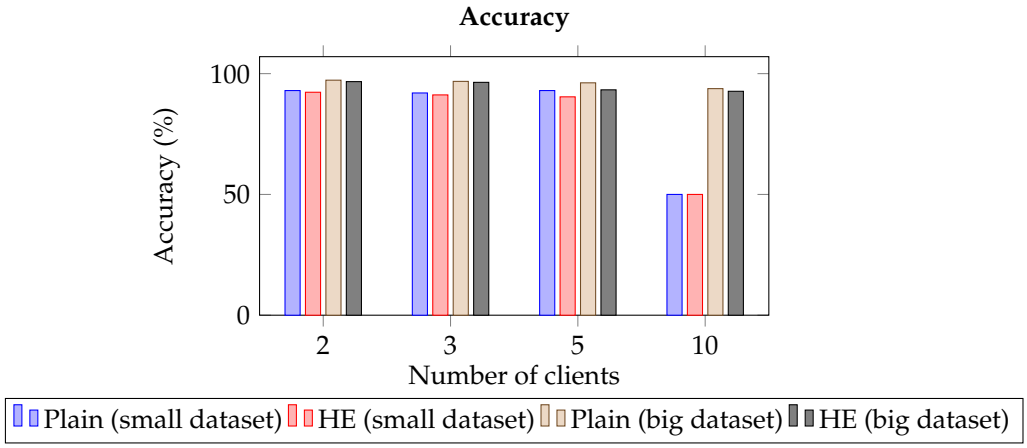

Figure 7. Accuracy for different numbers of clients.

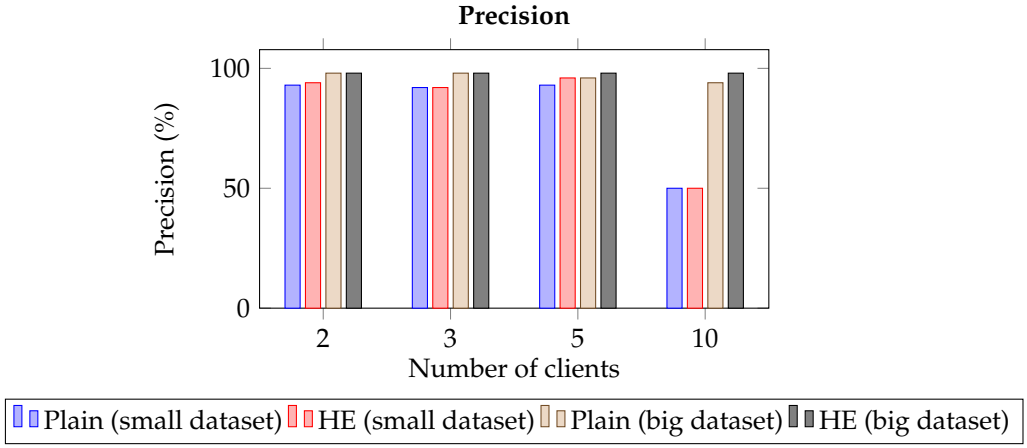

Figure 8. Precision for different numbers of clients.

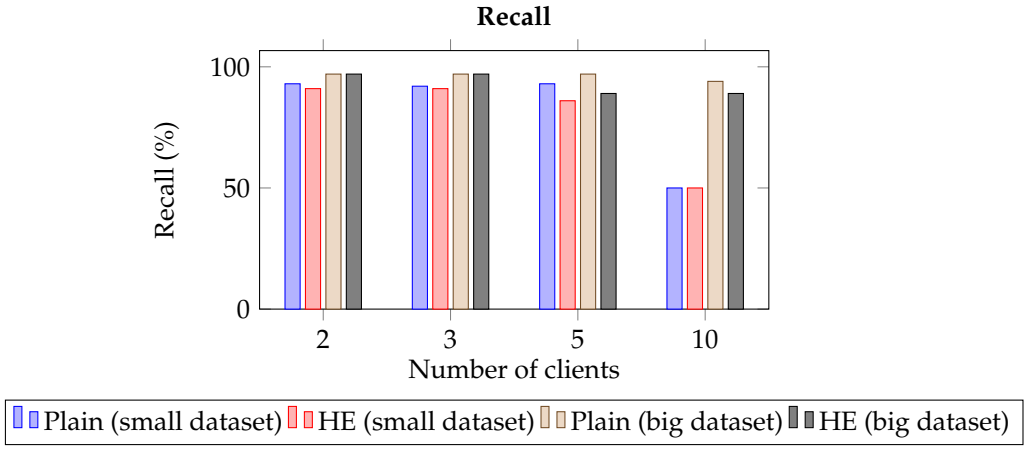

Figure 9. Recall for different numbers of clients.

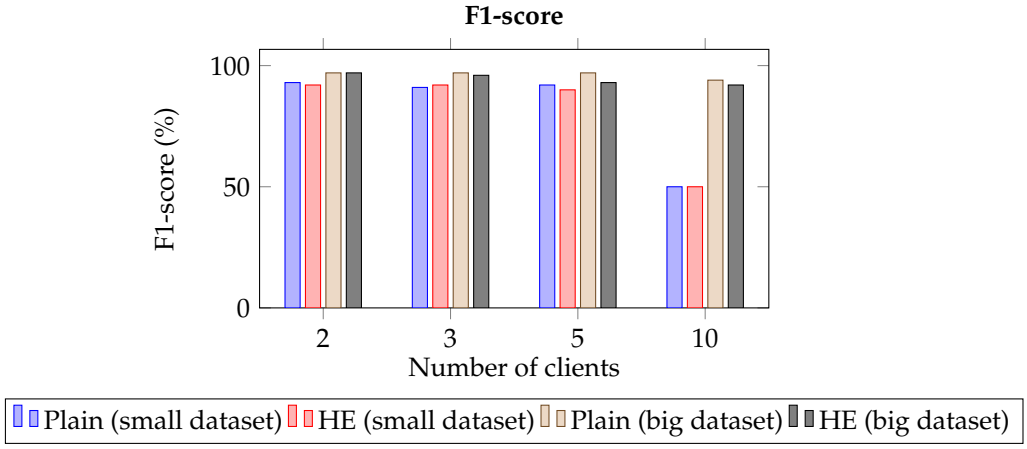

Figure 10. F1-score for different numbers of clients.

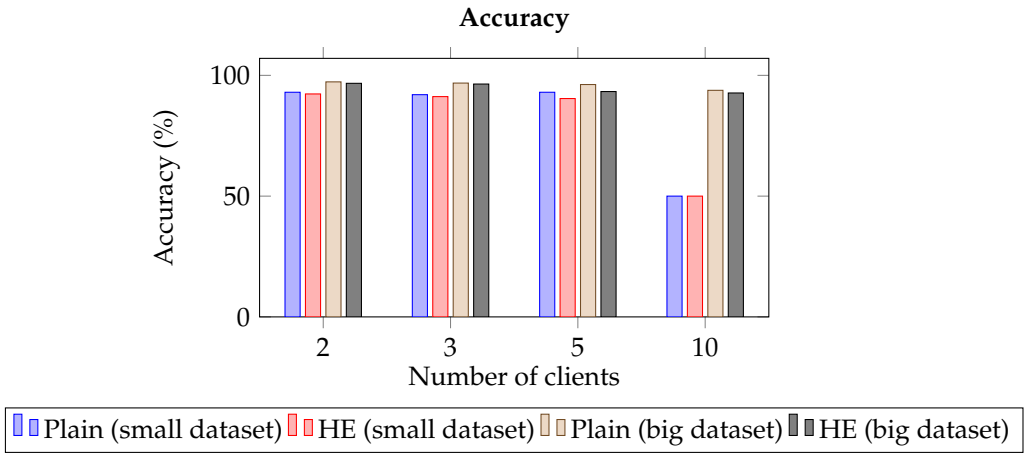

Figure 11. Accuracy for different numbers of clients.

Regarding execution time, as shown in Figure 12, encrypted FL (FL) naturally took longer than its plain FL counterpart, which aligns with our expectations. The number of clients significantly impacted execution time, mainly because of our sequential design, where data aggregation from each client occurs individually, resulting in an additional 4 s per client per training round.

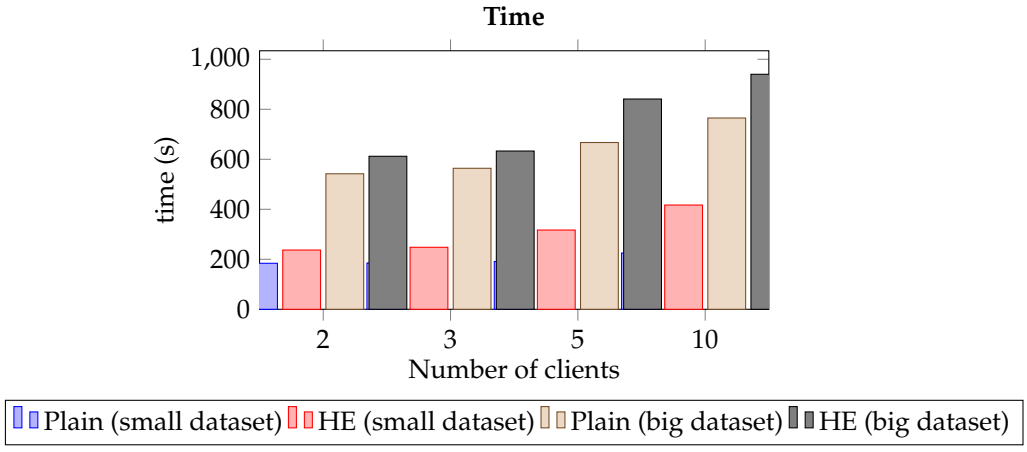

Figure 12. Time taken for different numbers of clients.

Interestingly, as demonstrated with a t3 and c5 instance in Table 7, more than 80% of this additional time is due to the transmission of large polynomials. In comparison, less

than 20% is allocated to the actual homomorphic operations. Notably, these operations maintained similar efficiency levels regardless of whether we used three- or nine-digit coefficients, indicating that the operation time is primarily influenced by the length of the polynomials rather than the size of their coefficients. This finding shows that the homomorphic operations are very efficient, even with over one million nine-digit coefficients.

Examining Table 4 reveals a notable observation. The precision measurements for both plain and encrypted data scored 50% when training with 10 clients. This can be attributed to the limited and divided dataset, causing clients to be unable 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.

6. Conclusions and Future Directions

We propose a privacy-preserving FL scheme based on homomorphic multi-key encryption with the assistance of RLWE and Flower to safeguard private data. Specifically, we enhance the solution presented by Wibawa et al. [33] by implementing a more secure HE approach in an FL framework. The previous paper employed a single public key for all clients participating in training, which is unsuitable in real-world scenarios where not all clients can be trusted. To address this issue, we adopt the xMK-CKKS scheme proposed by Ma et al. [27] and introduce custom communications between clients and the server within the Flower framework. This approach ensures that model update confidentiality is maintained through multi-key HE.

Our evaluation comprehensively examines an HE-based FL scheme, considering metrics such as accuracy, precision, recall, F1 score, time, and memory usage. We compare this scheme against plain FL and locally trained models, conducting tests on an Amazon Web Services EC2 instance with scenarios involving up to 10 clients.

Upon thorough evaluation and comparison of the different schemes, we conclude that the xMK-CKKS HE scheme effectively preserves privacy while achieving comparable accuracy, precision, recall, and F1-score levels compared to the other schemes. However, it is essential to note that the HE 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, mainly when user privacy is of utmost importance.

Looking ahead, there are potential enhancements that could further improve our implementation's privacy, efficiency, and effectiveness. Firstly, ensuring all clients start with identical initial weights could enhance privacy by eliminating the need to share full model weights during the first round. Secondly, leveraging Flower's built-in code for parallel processing during encrypted weight sharing could significantly reduce overall execution time depending on various factors. Thirdly, a more nuanced model updating process could be achieved by adopting a weighted averaging approach that considers the size of each client's training data. Finally, to mitigate memory constraints, we could reduce the precision of model weights, allowing us to use smaller integer types. However, it is crucial to strike a balance to ensure that memory optimization does not compromise the model's learning capacity. Further research could determine the optimal level of precision reduction that maximizes memory savings without significantly impacting model performance.

Author Contributions: Conceptualization, I.W., M.C.T. and F.O.C.; software, I.W. and M.C.T.; validation, I.W., M.C.T. and F.O.C.; formal analysis,I.W., M.C.T. and F.O.C.; writing—original draft preparation, I.W., M.C.T. and F.O.C.; writing—review and editing, I.W., M.C.T. and F.O.C.; visualization, I.W. and M.C.T.; supervision, F.O.C. Authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Dataset used in the manuscript can be found at: https://www.kaggle.com/datasets/tawsifurrahman/covid19-radiography-database accessed on 30 September 2023.

Conflicts of Interest: The authors have no conflict of interest to declare. All co-authors have seen and agreed with the contents of the manuscript. We certify that the submission is original work and is not under review at any other publication.

References

1. Abouelmehdi, K.; beni hssane, A.; Khaloufi, H.; Saadi, M. Big data security and privacy in healthcare: A Review. *Procedia Comput. Sci.* **2017**, *113*, 73–80. [CrossRef]
2. Kaissis, G.; Makowski, M.; Rückert, D.; Braren, R. Secure, privacy-preserving and federated machine learning in medical imaging. *Nat. Mach. Intell.* **2020**, *2*, 305–311. [CrossRef]
3. Lyubashevsky, V.; Peikert, C.; Regev, O. On Ideal Lattices and Learning with Errors over Rings. In Proceedings of the Advances in Cryptology—EUROCRYPT 2010, French Riviera, France, 30 May–3 June 2010; Gilbert, H., Ed., Springer: Berlin/Heidelberg, Germany, 2010; pp. 1–23.
4. Truong, N.; Sun, K.; Wang, S.; Guitton, F.; Guo, Y. Privacy Preservation in Federated Learning: An insightful survey from the GDPR Perspective. *arXiv* **2021**, arXiv:2011.05411.
5. Li, S.; Xue, K.; Yang, Q.; Hong, P. PPMA: Privacy-Preserving Multisubset Data Aggregation in Smart Grid. *IEEE Trans. Ind. Inform.* **2018**, *14*, 462–471. [CrossRef]
6. Pu, C.; Wall, A.; Choo, K.K.R.; Ahmed, I.; Lim, S. A Lightweight and Privacy-Preserving Mutual Authentication and Key Agreement Protocol for Internet of Drones Environment. *IEEE Internet Things J.* **2022**, *9*, 9918–9933. [CrossRef]
7. Sala, R.D.; Scotti, G. Exploiting the DD-Cell as an Ultra-Compact Entropy Source for an FPGA-Based Re-Configurable PUF-TRNG Architecture. *IEEE Access* **2023**, *11*, 86178–86195. [CrossRef]
8. Sun, C.; Liu, J.; Xu, X.; Ma, J. A Privacy-Preserving Mutual Authentication Resisting DoS Attacks in VANETs. *IEEE Access* **2017**, *5*, 24012–24022. [CrossRef]
9. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
10. What are Convolutional Neural Networks? | IBM. Available online: https://www.ibm.com/topics/convolutional-neural-networks (accessed on 30 September 2023)
11. McMahan, H.B.; Moore, E.; Ramage, D.; Hampson, S.; Arcas, B.A. Communication-Efficient Learning of Deep Networks from Decentralized Data. *arXiv* **2023**, arXiv:1602.05629.
12. Li, T.; Sahu, A.K.; Talwalkar, A.; Smith, V. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Process. Mag.* **2020**, *37*, 50–60. [CrossRef]
13. Smith, V.; Chiang, C.K.; Sanjabi, M.; Talwalkar, A. Federated Multi-Task Learning. *arXiv* **2018**, arXiv:1705.10467.
14. Bagdasaryan, E.; Veit, A.; Hua, Y.; Estrin, D.; Shmatikov, V. How To Backdoor Federated Learning. *arXiv* **2019**, arXiv:1807.00459.
15. Melis, L.; Song, C.; De Cristofaro, E.; Shmatikov, V. Exploiting Unintended Feature Leakage in Collaborative Learning. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 691–706. [CrossRef]
16. Fung, C.; Yoon, C.J.M.; Beschastnikh, I. Mitigating Sybils in Federated Learning Poisoning. *arXiv* **2020**, arXiv:1808.04866.
17. Fredrikson, M.; Jha, S.; Ristenpart, T. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 1322–1333. [CrossRef]
18. Zhu, L.; Liu, Z.; Han, S. Deep Leakage from Gradients. *arXiv* **2019**, arXiv:1906.08935.
19. Rieke, N.; Hancox, J.; Li, W.; Milletarì, F.; Roth, H.R.; Albarqouni, S.; Bakas, S.; Galtier, M.N.; Landman, B.A.; Maier-Hein, K.; et al. The future of digital health with federated learning. *NPJ Digit. Med.* **2020**, *3*, 119. [CrossRef]
20. Kairouz, P.; McMahan, H.B.; Avent, B.; Bellet, A.; Bennis, M.; Bhagoji, A.N.; Bonawitz, K.; Charles, Z.; Cormode, G.; Cummings, R.; et al. Advances and Open Problems in Federated Learning. *arXiv* **2021**, arXiv:1912.04977.
21. Diao, E.; Ding, J.; Tarokh, V. HeteroFL: Computation and Communication Efficient Federated Learning for Heterogeneous Clients. *arXiv* **2021**, arXiv:2010.01264.
22. Authors, T.F. Flower: A Friendly Federated Learning Framework. *arXiv* **2020**, arXiv:2007.14390.
23. Quickstart. 2023. Original-Date: 2017-07-18T20:41:16Z. Available online: https://openmined.github.io/PySyft/getting_started/index.html (accessed on 30 September 2023)
24. Flower: A Friendly Federated Learning Framework, 2023. Original-Date: 2020-02-17T11:51:29Z. Available online: https://flower.dev (accessed on 30 September 2023)
25. Wood, A.; Najarian, K.; Kahrobaei, D. Homomorphic Encryption for Machine Learning in Medicine and Bioinformatics. *ACM Comput. Surv.* **2020**, *53*, 70. [CrossRef]
26. Gentry, C. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM* **2010**, *53*, 97–105. [CrossRef]
27. Ma, J.; Naas, S.A.; Sigg, S.; Lyu, X. Privacy-preserving Federated Learning based on Multi-key Homomorphic Encryption. *arXiv* **2021**, arXiv:2104.06824.
28. Chen, L.; Jordan, S.; Liu, Y.K.; Moody, D.; Peralta, R.; Perlner, R.; Smith-Tone, D. *Report on Post-Quantum Cryptography*; Technical Report NIST IR 8105; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2016. [CrossRef]
29. Peikert, C. A Decade of Lattice Cryptography. *Found. Trends Theor. Comput. Sci.* **2016**, *10*, 283–424. [CrossRef]

30. Dowlin, N.; Gilad-Bachrach, R.; Laine, K.; Lauter, K.; Naehrig, M.; Wernsing, J. Manual for Using Homomorphic Encryption for Bioinformatics. *Proc. IEEE* **2017**, *105*, 552–567. [CrossRef]
31. Lyubashevsky, V.; Peikert, C.; Regev, O. On Ideal Lattices and Learning with Errors Over Rings. *J. ACM* **2013**, *60*, 43. [CrossRef]
32. Körtge, N. The Idea behind Lattice-Based Cryptography. 2021. Available online: https://medium.com/nerd-for-tech/the-idea-behind-lattice-based-cryptography-5e623fa2532b (accessed on 30 September 2023).
33. Wibawa, F.; Catak, F.O.; Sarp, S.; Kuzlu, M. BFV-Based Homomorphic Encryption for Privacy-Preserving CNN Models. *MDPI Cryptogr.* **2022**, *6*, 34. [CrossRef]
34. GitHub—Yusugomori/Rlwe-Simple: Simple RLWE (Ring Learning with Errors) Implementation with Python. Available online: https://github.com/yusugomori/rlwe-simple (accessed on 30 September 2023).
35. Creating New Messages—Flower 1.5.0. Available online: https://flower.dev/docs/framework/tutorial-series-get-started-with-flower-pytorch.html (accessed on 30 September 2023).
36. Protocol Buffer Basics: Python. Available online: https://protobuf.dev/getting-started/pythontutorial/ (accessed on 30 September 2023).
37. Maftouni, M.; Law, A.C.C.; Shen, B.; Grado, Z.J.K.; Zhou, Y.; Yazdi, N.A. A robust ensemble-deep learning model for COVID-19 diagnosis based on an integrated CT scan images database. In Proceedings of the IIE Annual Conference, Online, 22–25 May 2021; Institute of Industrial and Systems Engineers (IISE): Peachtree Corners, GA, USA, pp. 632–637.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.