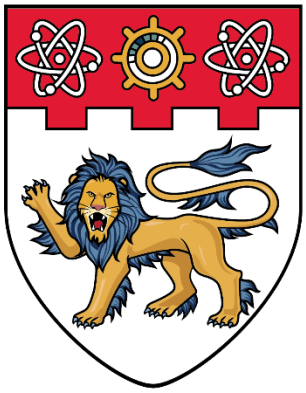

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Multi-Key Fully Homomorphic Encryption on Machine Learning Models

CCDS24-0128

Submitted in Partial Fulfilment of the Requirements

for the Degree of Bachelor of Engineering (Computer Engineering)

of the Nanyang Technological University

by

Tan Jian Wei

College of Computing and Data Science

2025

Abstract

With the rapid rise of Machine Learning (ML) and Artificial Intelligence (AI) across multiple fields, concerns over data privacy as well as model privacy have become ever more pressing. As these technologies matures and integrate itself into the core of sensitive sector such as healthcare or finance, the need for fully privacy-preserving models is increasing critical. In this paper, we will explore the use of Multi-Key Fully Homomorphic Encryption (MK-FHE) on ML models to protect both the input data, as well as the model parameters within a Multi-Party Computation (MPC) Environment. Through experimental results and complexity analyses, we address the challenges and trade-offs in using MK-FHE for ML applications, such as performance, accuracy, noise accumulation, and scalability.

Keywords: Machine Learning, Artificial Intelligence, Multi-Key Homomorphic Encryption, Multi-Party Computation

Acknowledgements

I would like to extend my sincere gratitude to Associate Professor Anupam Chattopadhyay for his invaluable guidance throughout the duration of my project. His deep expertise in cryptography and machine learning greatly contributed to the meaningful progress achieved in this work.

I am also deeply grateful to PhD candidate Ngo Anh Tu for his prompt responses and constant guidance whenever I encountered challenges in the project's implementation.

This project would not have been possible without the generous support and contributions of all the individuals mentioned above.

Table of Contents

1 Introduction

1.1 Project Background

Machine Learning (ML) models today are used in a wide range of fields including healthcare, finance, automotive and manufacturing. Organizations looking to use ML as part of their operation have 2 options – On premise ML Infrastructure or Machine Learning as a Service (MLaaS). Most organizations will adopt the latter as they are able to leverage ML without building their own infrastructure from scratch which saves them time, money and manpower compared to building their own infrastructure.

However, a key concern with using MLaaS is that these services are hosted on the cloud by third-party providers like Amazon, Google, Microsoft, and Tencent. Organizations lack direct oversight of the data they send to these providers, which limits their ability to ensure that the data isn't misused or leaked. This issue is particularly critical for organizations working with sensitive data as input for model processing. Any potential misuse or leakage of such data can seriously impact company reputation and may lead to non-compliance with regulations like GDPR and PDPA.

On the other hand, MLaaS has become a crucial revenue stream for providers as more organizations turn to MLaaS [1]. As a result, protecting the intellectual property (IP) of their ML models—such as model parameters, structure, and algorithms—is essential for them. If this proprietary information were exposed, it would be vulnerable to a practice known as model stealing, where third parties or competitors replicate a provider's model without incurring the costs associated with its development, training, and fine-tuning. This can have direct financial and competitive repercussions for MLaaS providers.

To safeguard the confidentiality of input data, privacy-preserving models have been extensively researched and have proven effective. In terms of defences against model stealing, most approaches fall into one of three categories: behaviour detection, prediction perturbation, and watermarking. However, there is a notable gap in leveraging cryptographic methods as a defence against model stealing, which this paper will set out to explore.

1.2 Project Objective

This project aims to evaluate and propose strategies to enhance data privacy and intellectual property protection within Machine Learning as a Service (MLaaS) environments by utilizing Multi-Key Fully Homomorphic Encryption (MK-FHE). Through analysing the performance and identifying the challenges associated with implementing MK-FHE in ML models, this study will demonstrate the practicality of MK-FHE in real-world applications. Additionally, it will provide a comprehensive framework for applying MK-FHE in ML models, guiding future implementations to ensure secure, privacy-preserving data processing within MLaaS platforms.

1.3 Project Scope

This project will focus on investigating the application of Multi-Key Fully Homomorphic Encryption (MK-FHE) to enhance data privacy and intellectual property protection within Machine Learning as a Service (MLaaS) platform. This is accomplished by researching current data security methods, analysing the feasibility and performance of MK-FHE in ML models, and developing a framework for its application within MLaaS. The scope will specifically address computational challenges and privacy concerns but will exclude the deployment of MK-FHE in live production environments.

2 Background and Related Work

2.1 Machine Learning

Machine Learning (ML) is a subset of Artificial Intelligence (AI) that focuses on learning from training data and improving performance on tasks without explicit programming. It uses algorithms and models to analyse and identify patterns in the training data, applying that knowledge to make predictions on previously unseen data. ML has a wide range of applications across almost all industries, thanks to the diverse algorithms developed for various purposes. For example, Logistic Regression (LR) is used for binary classification, Neural Networks (NN) are applied in image recognition, and Support Vector Machines (SVM) are effective for classification tasks.

In this paper, we will focus specifically on two types of models: Logistic Regression (LR) and Convolutional Neural Network (CNN).

2.1.1 Logistic Regression

Logistic Regression (LR) models are primarily used for binary classification tasks, where the goal is to predict one of two possible outcomes [2]. It utilizes the sigmoid function, defined as:

$$S(x) = \frac{1}{1 + e^{-x}}$$

This function maps the predicted values (linear combinations of input features) to probabilities ranging between 0 and 1. These probabilities represent the likelihood of the input belonging to the positive class. a threshold is applied to the probability to make a final classification decision. Typically, a threshold of 0.5 is used.

2.1.2 Convolutional Neural Network

Convolutional Neural Network (CNN) is a type of deep learning model commonly used for tasks like image recognition, object detection, and processing data with spatial patterns. It uses small filters, called kernels, to scan the input data and learn important features such as edges, textures, and patterns.

The figure below illustrates a high-level overview of how a CNN works, starting from raw input data, extracting features layer by layer, and finally making predictions.

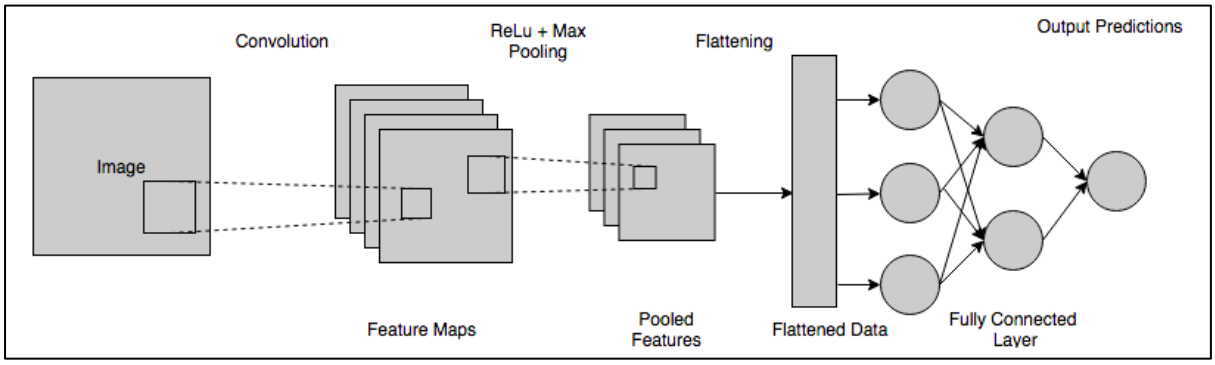

Figure 2-2 Example of a Convolutional Neural Network [3]

2.2 Attacks against ML models

There are many ways attackers target machine learning models, depending on what they want to achieve. Some try to uncover sensitive training data; others aim to manipulate predictions or exploit weaknesses in the system. For the purpose of this paper, we will focus on model extraction attacks**.** As training models for use in MLaaS takes up significant amount of computing resource, training data and time, efforts have been made towards stealing model – specifically model parameters, allowing others to use the same parameters to replicate an equally effective or identical model without dedicating resources to training it from scratch. These model stealing attacks if carried out successfully, could significantly harm service providers, as they may lead to Intellectual Property (IP) theft, loss of competitive advantage, and reduced revenue. Model stealing attacks can fall into 2 categories – Whitebox and Blackbox.

Extraction Method	**Description**
Whitebox Model Extraction	• Direct access to internal structure and parameters • Uses architecture, weights, and training details • High accuracy with lower complexity
Blackbox model Extraction	• No access to internal details • Relies on input-output queries • Complexity depends on inference

Table 2-3 Summary of whitebox and blackbox model extraction

2.2.1 Whitebox Model Stealing

In a Whitebox scenario, the adversary has complete access to the victim model's parameters and architecture, in addition to the ability to query the model. With the inner workings of the model fully exposed, the adversary can effortlessly replicate it, creating an identical clone.

2.2.2 Blackbox Model Stealing

Meanwhile in a blackbox scenario, the threat actor does not have any information of the inner working of the model. They only can infer and build their knowledge from specially crafted input and their corresponding output.

Multiple studies on blackbox model stealing have been done successfully as seen in [4], [5], [6], [7]. Most of these studies employ similar methodologies to achieve their goals, albeit with slight variations in implementation. In a black-box attack, the primary challenge lies in the lack of information about the victim model. Therefore, all such attacks must begin with gathering information about the victim model. This is typically achieved by querying the victim model with various inputs and analysing its outputs, a common strategy employed in the attacks discussed below.

2.2.2.1 Knockoff Net

In Knockoff Net [4], they focused on creating a model functionality stealer that aims to replicate a victim model. This is achieved by querying a set of input images to the Blackbox model, obtaining predictions, and using those input-prediction pairs to train a knockoff version of the model.

The first step of the proposed attack involves creating a set of input-prediction pairs to gather the model's predictions. This can be accomplished using two strategies: random and adaptive.

1. **Random Strategy:** As the name implies, this involves randomly sampling inputs. However, this approach has a drawback—there is a risk of querying irrelevant inputs (e.g., over-querying dog images to a bird classifier).
2. **Adaptive Strategy:** This approach utilizes a feedback signal that is fed into an algorithm to decide which sample to query next. The algorithm considers multiple factors to maximize the coverage of the potential dataset, ensuring more efficient and relevant sampling.

Once the set of input-prediction pair have been generated, the knockoff models will then be trained multiple times on the given dataset. The result of this model stealing methodology are promising with the knockoff models being able to obtain 81% -96% performance of the victim box model using the random strategy while adaptive strategy boost performance up to 4.5% across all experiments.

2.2.2.2 Extraction with confidence value

Another method of stealing models is to use the confidence values returned by the victim model and using them in an equation solving attack [5]. This is especially effective against prediction models with a logistic layer, as the attack closely resembles training a logistic regression model. The attacker minimizes a loss function based on the difference between the confidence values of the victim model and the predictions of the stolen model, using optimization techniques similar to those used in standard logistic regression training.

This method have proven to be both efficient and effective, with the replica model achieving 99.9% accuracy with 5410 sample on average [5].

2.2.2.3 Reverse Engineering

Unlike the earlier model stealing methods, which focus on replicating the victim model by training a copy using its input-output pairs, [6] introduces three unique approaches to uncover the victim model's attributes. Instead of recreating the model's functionality, these methods aim to reveal details about its internal structure, such as its architecture or specific hyperparameters.

1. **Kennen-O:** Predicts multiple attributes of a victim model simultaneously by using a fixed set of query inputs selected from a dataset. The goal is to train a meta-classifier m_θ to map the concatenated outputs of meta-training models on the fixed query set to their corresponding attributes. Once trained, m_θ can infer the attributes of a black-box model by analyzing its outputs on the same query set.

2. **Kennen-I:** Infers a single attribute of a black-box victim model by crafting a specialized query input $\tilde{x}$ The crafted input is optimized using meta-training models with known attributes to produce outputs strongly correlated with the target attribute. Once the query input $\tilde{x}$ is learned, it is submitted to the black-box model, and the target attribute is inferred from its prediction.

3. **Kennen-IO:** Combines the strengths of Kennen-O and Kennen-I to overcome the limitation of Kennen-I, which can only infer one attribute at a time. Kennen-IO incorporates the simultaneous prediction capabilities of Kennen-O while still leveraging crafted query inputs inspired by Kennen-I.

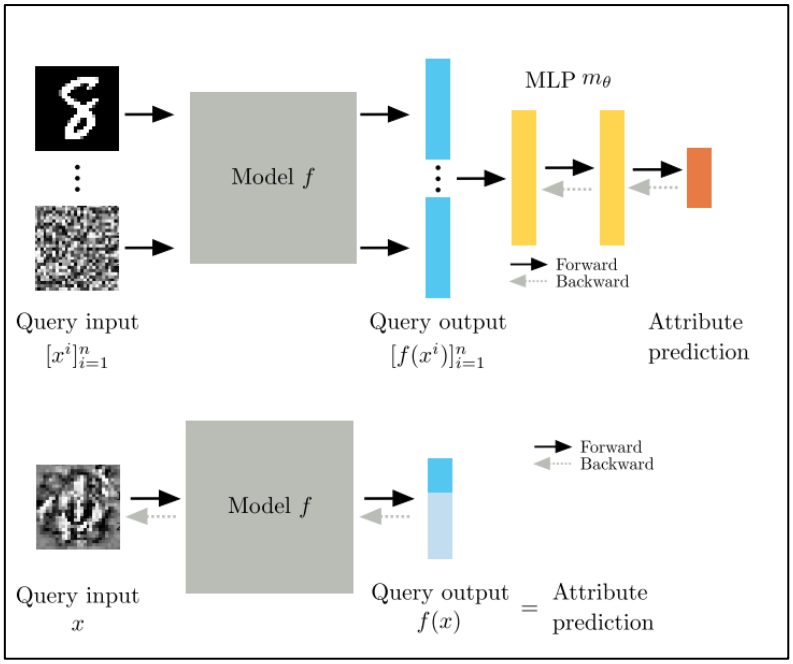

Figure 2-4 High level visualisation of Kennen-IO [6]

The performance of Kennen-O and Kennen-I can vary depending on the experiment, but Kennen-IO has shown great potential, with an average accuracy of 80.1%. This demonstrates the power of combining Kennen-O's ability to predict multiple attributes at once with Kennen-I's precision in crafting targeted queries, making it a more effective and versatile approach overall.

2.3 Defences for ML models

After understanding the various types of model extraction attacks that threaten MLaaS providers, it is essential to explore the defences that can mitigate these risks. Model extraction defences can generally be categorized into three main types: behavioural detection, prediction perturbation, and model watermarking. Each of these approaches addresses specific attack vectors, and when combined, they provide a more comprehensive and holistic defence against potential threat actors.

2.3.1 Behavioural Detection

The first layer of defence that can be implemented is monitoring the queries sent to the model and flagging any suspicious query trends that may indicate an attempt to extract the victim model. As demonstrated in attacks on machine learning models, most extraction methods require the perpetrator to either send a series of queries or craft specific inputs to gather information about the victim model. This common behaviour can therefore be analysed and used to detect potential model extraction attacks, allowing them to be intercepted and stopped before they succeed.

We see this type of defence being employed by [7], [8] which uses different algorithms to detect malicious queries. In [7], They used a classifier that returns a temperature-scaled

maximum softmax probability (TMSP) to determine whether a query is malicious. When a malicious query is detected, the model responds by flipping the label of an optimal sample in the hopes of misleading the attacker and decreasing the performance of the replica model. The paper also takes into account the scenario where the attacker might notice different predictions for the same input, potentially alerting them to the possibility of label flipping. To address this, they introduced another algorithm that considers the input's Phash, which is used to identify similar inputs and their corresponding results.

Meanwhile in [8], the assumption is that malicious queries behave differently from benign ones since they are designed to extract as much information as possible. With this in mind, benign queries are expected to follow a normal distribution in consecutive interactions, while malicious queries are more likely to deviate from this pattern. Upon detection, the suggested counter measure would be to simply block further queries or modifying the prediction to deceive the adversary.

2.3.2 Prediction Perturbation

In prediction perturbation, as the name suggests, the goal is to perturb the predictions for certain inputs to make it more difficult for an attacker to extract accurate information about the underlying model. This approach involves deliberately modifying the model's output for specific queries, either by adding noise, flipping labels, or generating randomized responses. By doing so, it becomes challenging for the attacker to accurately reconstruct the model's decision boundaries or replicate its functionality. An example of such defence employ can be seen in ModelGuard [9].

ModelGuard achieves prediction perturbation by using a constrained optimization approach which is to maximize the loss between true predictions and the attacker's recovered predictions while making the prediction accurate enough for legitimate users.

Since the attacker's method of reversing predictions is unknown, ModelGuard uses two strategies:

1. **ModelGuard-W**: Assumes attackers are less capable and approximates their recovery process as nearly identical to the altered predictions. It then solves the optimization problem based on this assumption.
2. **ModelGuard-S**: Assumes attackers use the best possible recovery method (the Bayes estimator) and maximizes an information-theoretic lower bound to counter such optimal attacks.

2.3.3 Model Watermarking

Lastly, we have watermarking, which has been used to deter and detect counterfeit documents or physical currency since its introduction in 1282 [10]. Watermarking is essentially the practice of embedding a unique identifier that is difficult to replicate but easily verifiable into a medium that can be distributed on a large scale to ensure its authenticity and traceability. This technique has evolved into the digital era, finding applications in digital media such as audio files, images, and videos.

As the popularity of machine learning (ML) has grown over the past decade, the concept of watermarking has been adapted to address the challenge of proving the legitimacy and ownership of ML models. This is achieved by embedding a unique, hidden signature during the training process which have been implemented in [11], [12], [13] . The signature is designed such that only when a specific input is provided, the model will produce a unique, predefined output, serving as a verifiable marker of ownership. This approach not only helps protect intellectual property but also enables tracing and accountability in the use of ML models.

2.4 Asymmetric Key Encryption

Asymmetric key encryption differs from symmetric key encryption in that it uses two separate keys for the encryption and decryption processes. In this system, a public key is used for encryption, which can be shared openly, while a private key is kept confidential and used for decryption. This distinction eliminates the need for both parties to share a single secret key, addressing the challenge of secure key distribution present in symmetric key systems [14].

One of the major applications of asymmetric key encryption is in secure communication over the internet, such as SSL/TLS protocols for websites. Additionally, it plays a critical role in digital signatures, ensuring data integrity and authentication by allowing recipients to verify the sender's identity using their public key. Popular algorithms that employ this cryptography system include RSA, ECC (Elliptic Curve Cryptography), and Diffie-Hellman.

This property of secure key distribution will be vital as we delve into Secure Multi-Party Computation (SMPC) in later sections. SMPC relies on the ability of parties to securely share information without revealing their private data, making public key cryptography an essential building block for achieving this goal.

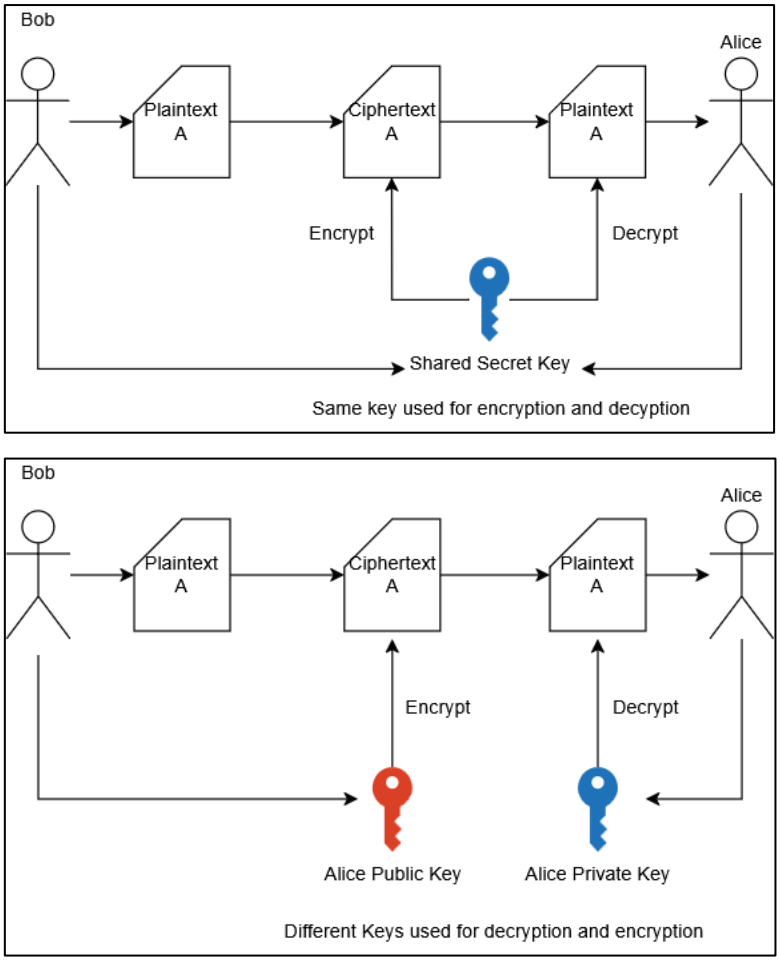

Figure 2-5 Comparison between Symmetric (top) and Asymmetric Cryptography (bottom)

2.5 Homomorphic Encryption

Homomorphic Encryption refers to a specific group of encryption scheme which allows for arithmetic operations to be carry out on the ciphertext without decryption [15]. This property of Homomorphic Encryption is especially useful when highly sensitive data needs to be handled by third parties who are not fully trusted. There are 3 classes of Homomorphic Encryption: Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SHE) and Fully Homomorphic Encryption (FHE).

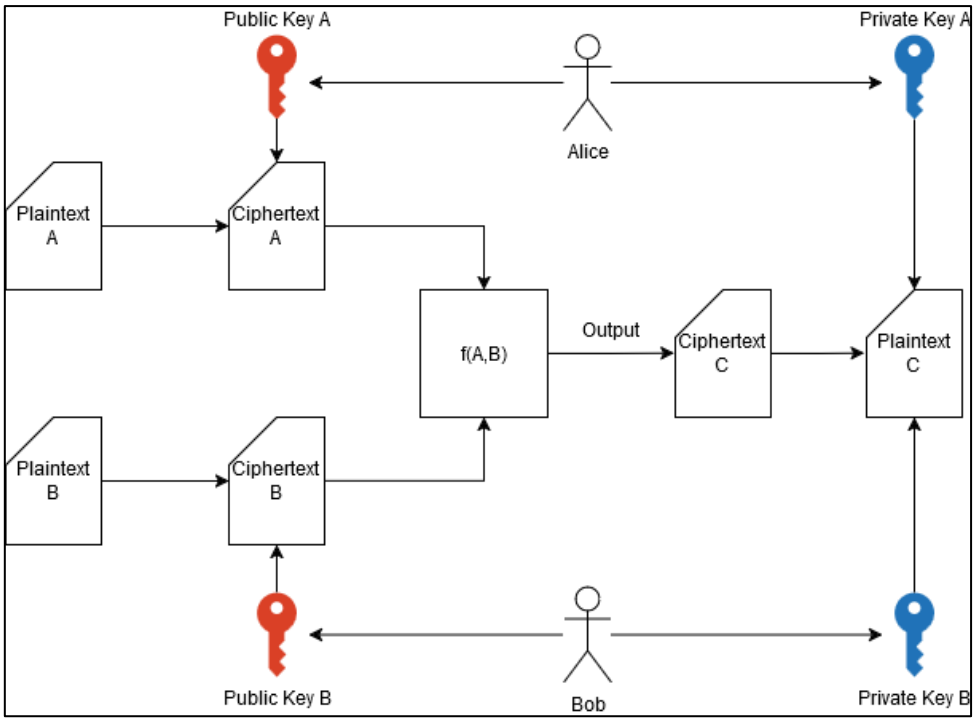

Figure 2-6 High level visualisation of homomorphic encryption operation

2.5.1 Partial Homomorphic Encryption (PHE)

Partially Homomorphic Encryption (PHE) is the simplest form of Homomorphic Encryption. It only allows one arithmetic operation to be performed on a ciphertext, either addition or multiplication for an unlimited number of times. One popular PHE would be the classic Rivest-Sharmir-Adleman (RSA) encryption scheme.

Rivest- Sharmir-Adleman (RSA)

RSA [16] achieves homomorphic multiplication through the property that multiplying two ciphertexts results in a ciphertext that corresponds to the product of the plaintexts. In RSA encryption, a message m is encrypted by raising it to an exponent e and taking it modulo n, where n is the product of two large primes and serves as the public modulus:

$$\mathrm{c} = m^{e} \bmod \mathrm{n}$$

If two messages, m_1 and m_2, are encrypted as c_1 and c_2, then multiplying the two ciphertext gives:

$$c_1 \cdot c_2 = (m_1^e \, mod \, n) \cdot (m_2^e \, mod \, n) \equiv (m_1 \cdot m_2)^e \bmod \mathrm{n}$$

This result shows that RSA preserves multiplication homomorphically, meaning that multiplying ciphertexts corresponds to multiplying the underlying plaintexts [16].

2.5.2 Somewhat Homomorphic Encryption (SHE)

For Somewhat Homomorphic Encryption (SHE) schemes, both addition and multiplication operations can be performed on encrypted data. However, these operations can only be applied a limited number of times due to the accumulation of noise in the ciphertext after each operation. This noise grows with each homomorphic operation, and once it reaches a certain threshold, the ciphertext can no longer be decrypted correctly. SHE schemes are often more efficient than fully homomorphic schemes since they do not involve techniques to manage noise over an unlimited number of operations.

Boneh-Goh-Nissim (BGN)

BGN [17] is a public key system that relies on bilinear pairings on elliptic curves to achieve its homomorphic properties. Bilinear pairings allow for one level of multiplication between ciphertexts, which is what limits BGN to a single multiplication after which further multiplications are not possible.

2.5.3 Fully Homomorphic Encryption (FHE)

FHE encryption scheme allows for the full suite of endless arithmetic operations at the expense of being the most resource intensive homomorphic encryption scheme among the 3 types. In this paper, we will be focusing on Cheon-Kim-Kim-Song (CKKS) encryption scheme due to its suitability for applications in machine learning and data analysis.

Cheon-Kim-Kim-Song (CKKS)

The CKKS [18] encryption scheme is specifically crafted for homomorphic encryption with approximate arithmetic, making it well-suited for real and complex data. CKKS achieves its homomorphic capabilities by encoding plaintexts as polynomials, managing noise, and utilizing a bootstrapping process to support extensive computations without significant precision loss.

To control noise, CKKS employs both rescaling and bootstrapping techniques. Rescaling reduces the modulus size and noise level after each multiplication, maintaining a balance in precision. Bootstrapping, in contrast, fully resets the noise, allowing CKKS to handle deeper computations that would otherwise suffer from excessive noise buildup and precision degradation.

2.6 Secure Multi-Party Computation

Secure Multi-Party Computation (SMPC) is a protocol that allows multiple parties to collaboratively compute or process data while ensuring that each party's private data remains confidential. This enables secure computations without revealing any individual party's input to the others [19].

SMPC is particularly useful in scenarios where confidentiality is critical, such as joint data analysis by competing organizations, secure voting systems, or privacy-preserving machine learning. By enabling secure collaboration without compromising privacy, SMPC facilitates data sharing in situations where trust is limited or non-existent. The key principle of SMPC is that the computation is distributed among the participating parties, and no single party gains access to the others' private data during the process and hence no trusted 3rd party have to be established.

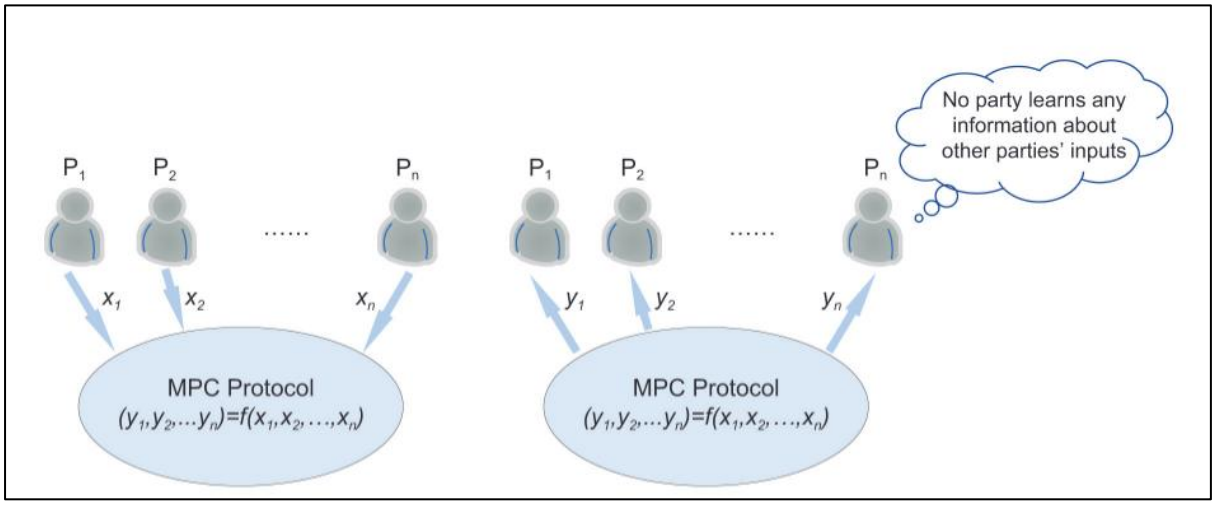

Figure 2-7 Example of Secure Multi-Party Computation [19]

This confidentiality is achieved through a combination of cryptographic techniques as mentioned above, such as asymmetric key encryption and homomorphic encryption. Asymmetric key encryption ensures secure communication and key exchange between participants, while homomorphic encryption allows computations to be performed directly on encrypted data without requiring decryption.

3 Threat Model

In this section, we define the threat model of our adversary to systematically identify potential risks and develop appropriate defences. This threat model helps us understand the adversary's capabilities, goals, and constraints, serving as a foundation for formulating our defence framework.

3.1 Adversary Types

3.1.1 Data Owner

The data owner, aiming to leverage MLaaS within their organization, will have two main risks to consider: privacy guarantees on their submitted data and the protection of the outputs derived from the machine learning model.

Firstly, the data submitted—especially sensitive information such as personally identifiable information (PII), financial records, or proprietary organizational data—must be safeguarded against any form of data leakage or mishandling. A failure to secure this data could result in severe consequences, including reputational damage, loss of stakeholder trust, or regulatory penalties due to non-compliance with data protection laws such as General Data Protection Regulation (GDPR) enacted in the European Union (EU).

Secondly, the outputs generated by the ML model must also be protected. These outputs could contain valuable insights or sensitive predictions that, if exposed, could allow adversaries to infer private information about the underlying data.

However, a data owner could also have malicious intentions. One such intent could involve stealing the model parameters from the model owner to gain unauthorized access to the proprietary technology. By extracting the model parameters, the data owner could host the model in-house, effectively replicating its functionality without needing to pay the service provider. This would allow them to bypass licensing or subscription fees, leading to significant financial losses for the model owner.

3.1.2 Model Owner

As the model owner or service provider of MLaaS, they bear the responsibility of securing both their customers' data and their own intellectual property (IP). Protecting customer data, particularly while it is being processed on their infrastructure, requires implementing strict privacy measures, safeguarding sensitive information from unauthorized access, and ensuring compliance with relevant data protection regulations. Any breaches or mishandling of customer data again could lead to severe consequences, including significant reputational damage, loss of customer trust, and substantial regulatory penalties.

However, this does not necessarily guarantee that the model owner will uphold their end of the agreement. The model owner could potentially misuse or mishandle customer data, intentionally for purposes beyond the agreed scope. For example, they might exploit the data for their own research or share it with third parties without consent.

At the same time, model owners must also safeguard their own IP. These models are valuable assets, representing years of research, development, and computational investment. Unauthorized extraction of the models' parameters could severely undermine their competitive advantage and business model.

3.1.3 External Adversary

Finally, as an external adversary, the primary goal would be to steal both the customer data and the model parameters. These adversaries typically operate without any authorized access to the MLaaS platform and employ various attack methods to achieve their objectives.

Stealing customer data could involve intercepting communications, exploiting vulnerabilities in data storage or transmission, or launching sophisticated attacks such as data exfiltration through APIs. The motivation behind such attacks could range from financial gain to industrial espionage, where sensitive customer information is sold, leaked, or used maliciously.

Similarly, extracting the model parameters poses a significant threat to the model owner’s IP. Successfully stealing the model parameters would allow the third party to duplicate the functionality of the MLaaS system, bypassing the need for payment and potentially using the stolen model for their own gain or resale.

3.2 Threat Scenario

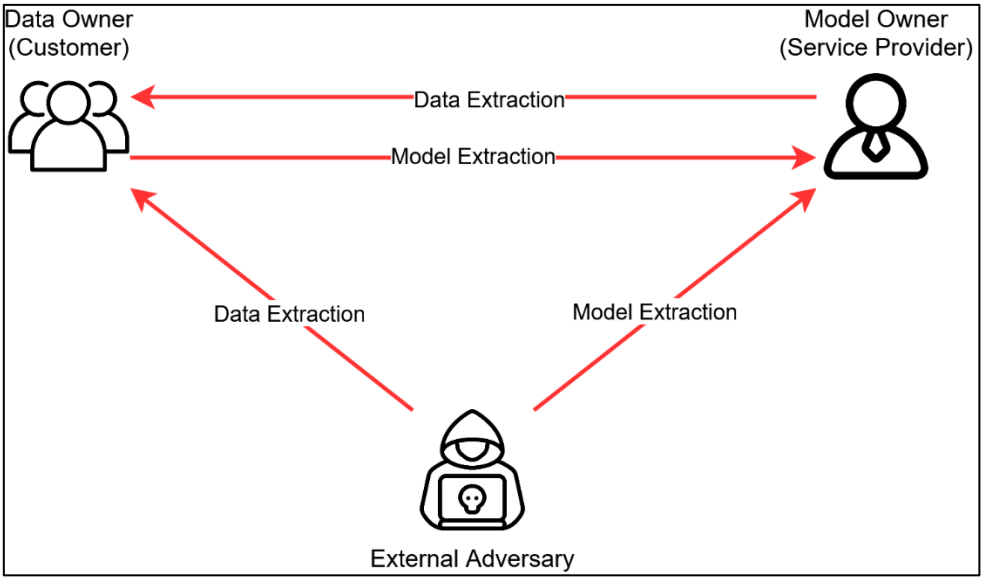

Figure 3-1 Possible threats in MLaaS ecosystem

The diagram above effectively showcases the high-level interactions and potential threats among the parties involved in the MLaaS ecosystem: the Data Owner (Customer), the Model Owner (Service Provider), and the external adversary. Each arrow illustrates the specific threats and attack goals. The following table breaks down the security goals of each party.

Party	Security Goal
Data Owner (Customer)	Ensure sensitive data is not exposed, extracted, or misused during processing.
Model Owner (Service Provider)	Protect model parameters from being stolen or reverse-engineered by any party.
External Adversary	No security goals; aims to exploit vulnerabilities to steal data or model parameters.

Table 3-2 Summary of Security Goal for each party in MLaaS ecosystem

4 Experiment Design

In this section, we will outline the key considerations for our implementation of Multi-Key Fully Homomorphic Encryption (MK-FHE) in machine learning (ML) models. This includes the selection of the FHE library, choice of ML models, datasets, and underlying assumptions that guide our approach.

4.1 Design Methodology

With the threat model established, we can categorize the threats into two main areas: Data Privacy and Model Privacy.

Data Privacy focuses on ensuring that data provided to the model is protected from mishandling, leakage, or unauthorized access. This involves safeguarding sensitive information during processing, transmission, and storage to maintain confidentiality and prevent potential breaches.

Model Privacy, on the other hand, ensures that the model's parameters cannot be extracted or reverse-engineered by any means. Protecting the model's intellectual property is critical for maintaining the competitive advantage of the service provider and preventing unauthorized replication or misuse.

As highlighted in Section 2, numerous studies have proposed frameworks to defend against black-box extraction attacks, where adversaries attempt to infer the model through query-based interactions. However, white-box defences, which protect models when adversaries have direct access to the model parameters or gradients, have been notably lacking to our knowledge.

We now propose the use of MK-FHE as an extension to the SMPC to meet our dual security goals of data privacy and model privacy. MK-FHE enables secure computations on encrypted data from multiple parties without requiring decryption, ensuring that sensitive customer data remains confidential during processing. Simultaneously, it protects the model parameters by performing computations in an encrypted domain, preventing adversaries from extracting or reverse-engineering the model even if they have access to the computation process.

4.2 Design Considerations

The first key consideration for our project was selecting an appropriate cryptographic library for our experiment. The available options for Homomorphic Encryption libraries are currently limited, with sparse support, as the adoption of Homomorphic Encryption is still in its early stages. Furthermore, our requirement for Multi-Key functionality significantly narrowed our choices, ultimately leading us to the OpenFHE library [20], the only viable option for our needs.

Within OpenFHE, we selected the CKKS FHE scheme due to its ability to perform Fully Homomorphic Encryption (FHE), supporting all arithmetic operations. Additionally, CKKS is highly efficient for approximate computations on real numbers, making it an ideal

choice for our ML use case. Since CKKS operates exclusively on real numbers, our chosen dataset must also be numeric in nature. A detailed discussion on dataset selection will be provided in a later section.

Another important consideration in our experiment was balancing security with the availability of our model. In the context of the CIA Triad [21], availability ensures that information remains accessible when needed. However, implementing security measures such as encryption inevitably impacts availability due to the additional processing overhead required for encryption and decryption. Hence in our experiment we will explore different variations of implementation which includes full and partial implementation of MK-FHE into our selected ML models.

4.3 Assumptions

Given the limited timeframe for this project, we establish several key assumptions to ensure a practical and focused implementation of MK-FHE in ML models.

1. We assume that the chosen MK-FHE encryption scheme is secure and cannot be broken cryptographically.
2. All established parties have access to the encrypted data but cannot decrypt without the appropriate keys.
3. Only legitimate users have access to their respective private keys and no private keys are leaked, lost or compromised.

4.4 Environment Setup

We will conduct our experiments in Jupyter Notebook using the following hardware configuration:

CPU: AMD Ryzen 7 3700X @ 3.60GHz

GPU: NVIDIA GeForce RTX 3070

RAM: 32GB (2×16GB) DDR4 @ 3200MHz

4.5 Machine Learning Models

For our experiment, we have chosen to run two models: Logistic Regression (LR) and a Convolutional Neural Network (CNN). As discussed in our literature review, LR is a relatively simple model commonly used for binary classification. Its simplicity allows us to better analyse the impact of integrating an MK-FHE security layer without the added complexity of deep learning models.

Meanwhile, the CNN enables us to expand on our findings from implementing MK-FHE on LR models and evaluate its feasibility in more complex, deep learning-based architectures. This will allow us to assess how MK-FHE impacts performance, computation time, and overall model accuracy in a more advanced setting.

The ML models used have minimal layers to clearly highlight the performance impact of adding MK-FHE to the models, ensuring that any observed computational overhead or latency can be directly attributed to the encryption rather than model complexity. Below shows the model shape of the 2 models used for our experiment.

Layer	Output Shape	Param #
Linear-1	[-1, 1, 1]	8 or 16

Table 5-1 LR model shape

Layer	Output Shape	Param #
Conv1d-1	[-1, 1, 260]	8
Linear-2	[-1, 10]	2,610

Table 5-2 CNN model shape

4.6 Datasets

4.6.1 Pima Indians Diabetes Database

The Pima Indians Diabetes Database (PIDD) is a well-known dataset for binary classification, used to predict diabetes based on numerical medical attributes such as glucose levels, BMI, and insulin levels. With 768 samples and 8 fully numeric features, it is well-suited for Multi-Key Fully Homomorphic Encryption (MK-FHE), particularly with the CKKS scheme, which operates on real numbers. Additionally, its small size makes it computationally feasible, serving as a benchmark for the least computationally intensive scenario in our experiment.

Column Name	**Description**
Pregnancies	Number of times a patient has been pregnant.
Glucose	Blood glucose concentration (mg/dL) from an oral glucose tolerance test.
Blood Pressure	Diastolic blood pressure (mm Hg).
Skin Thickness	Thickness of the triceps skin fold (mm), a measure of body fat.
Insulin	Serum insulin level (μU/mL) measured during an oral glucose tolerance test.
BMI (Body Mass Index)	Weight-to-height ratio (kg/m^2), an indicator of body fat.
Diabetes Pedigree Function	A measure of genetic predisposition to diabetes based on family history.
Age	Patient's age (years).
Outcome	Indicates whether the patient has diabetes (1) or not (0).

Table 5-1 Summary of PIDD dataset

4.6.2 Framingham Heart Study

The Framingham Heart Study dataset is another dataset used for binary classification, differing from PIDD in its larger size, with 4,420 entries, 16 columns, and 15 attributes. It covers key cardiovascular risk factors such as age, sex, cholesterol levels, blood pressure, smoking status, diabetes, and family history, making it a more complex dataset for evaluating the impact of MK-FHE on model performance.

Column Name	**Description**
Sex	Gender of the participant (0 = Female, 1 = Male).
Age	Age of the participant (years).
Current Smoker	Whether the participant is a current smoker (1 = Yes, 0 = No).
Cigs Per Day	Number of cigarettes smoked per day (if a smoker).
BP Meds	Whether the participant is on blood pressure medication (1 = Yes, 0 = No).
Prevalent Stroke	Whether the participant has had a stroke before (1 = Yes, 0 = No).
Prevalent Hypertension	Whether the participant has high blood pressure (1 = Yes, 0 = No).
Diabetes	Whether the participant has diabetes (1 = Yes, 0 = No).
Tot Cholesterol	Total cholesterol level (mg/dL).
Systolic BP	Systolic blood pressure (mm Hg).
Diastolic BP	Diastolic blood pressure (mm Hg).
BMI (Body Mass Index)	Body Mass Index (kg/m^2), a measure of body fat.
Heart Rate	Resting heart rate (beats per minute).
Glucose	Blood glucose level (mg/dL).
Ten-Year CHD	Whether the participant developed Coronary Heart Disease within 10 years (1 = Yes, 0 = No).

Figure 5-2 Summary of Framingham Heart Study Dataset

4.6.3 MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a collection of 70,000 grayscale images of handwritten digits, each sized 28x28 pixels [22]. This dataset serves as an excellent benchmark for image recognition tasks, as it is complex enough to evaluate various machine learning and deep learning algorithms while remaining simple and manageable for prototyping and experimentation. Its popularity as a benchmark for image recognition tasks also makes it a well-established dataset for comparing results and assessing the performance of different models.

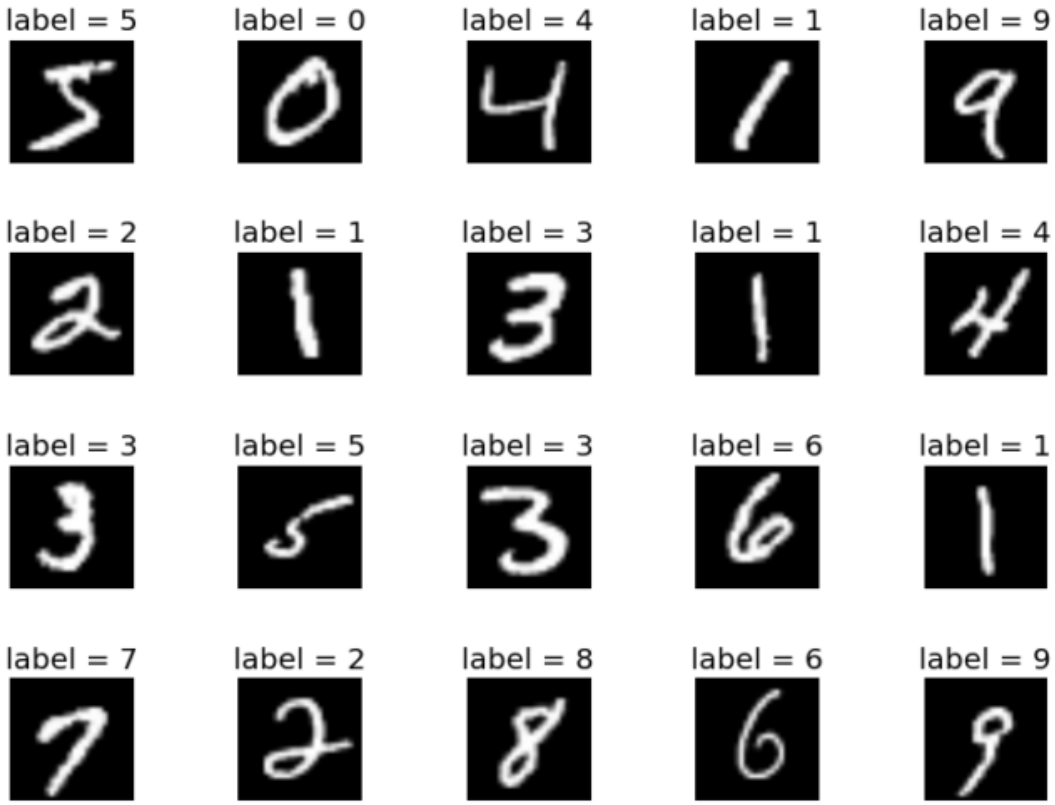

Figure 5-3 Sample of MNIST Dataset

4.6.4 Fashion-MNIST Dataset

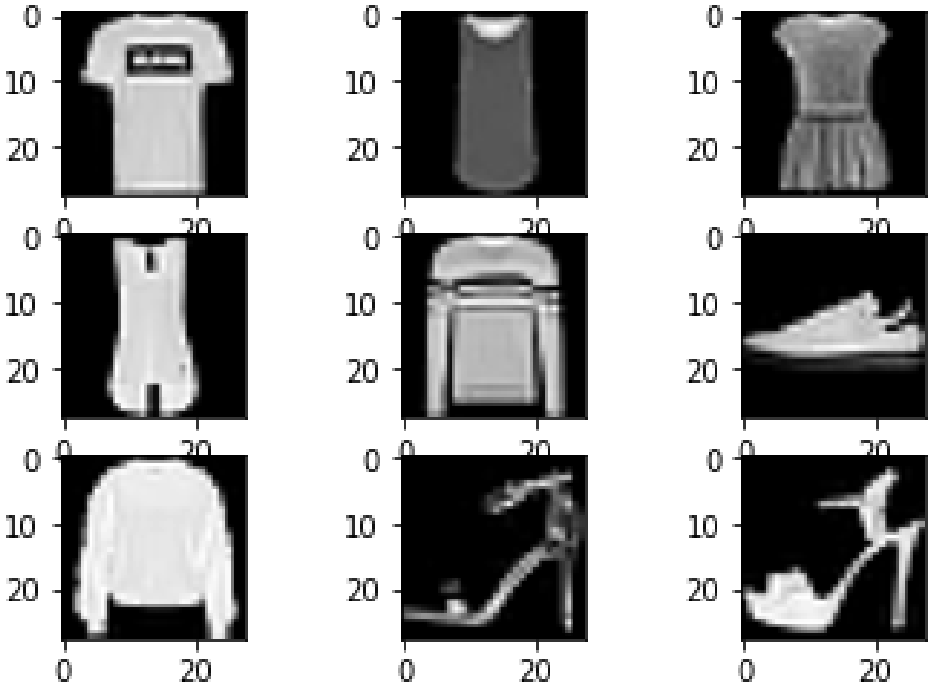

Figure 5-4 Sample of Fashion-MNIST Dataset [23]

The Fashion-MNIST dataset consists of 70,000 grayscale images of clothing items, each sized 28×28 pixels [24]. It serves as a widely used benchmark for image classification tasks, providing a more challenging alternative to the original MNIST dataset while maintaining the same structure. With 10 categories representing different types of apparel, it is well-suited for evaluating machine learning and deep learning models. In this study, we use Fashion-MNIST to examine the accuracy impact of using a 1D convolution layer instead of the conventional 2D convolution layer, which is typically recommended for image recognition tasks. Due to certain limitations, which we discuss in a later section, we will be implementing 1D convolution instead of 2D convolution for our experiments.

4.7 Training

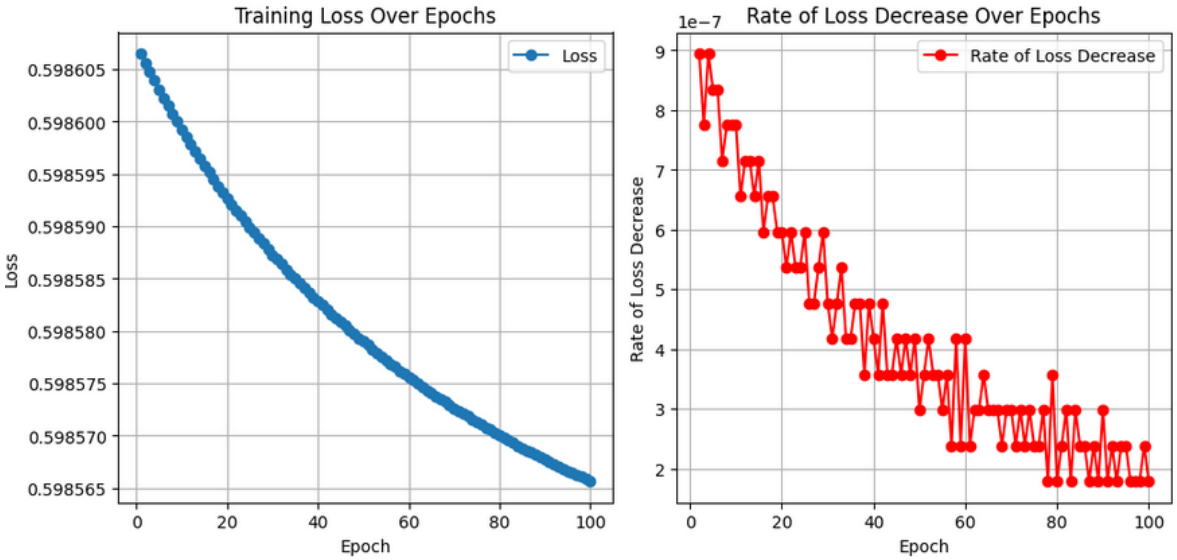

Figure 5-5 Training loss for Logistic Regression Model

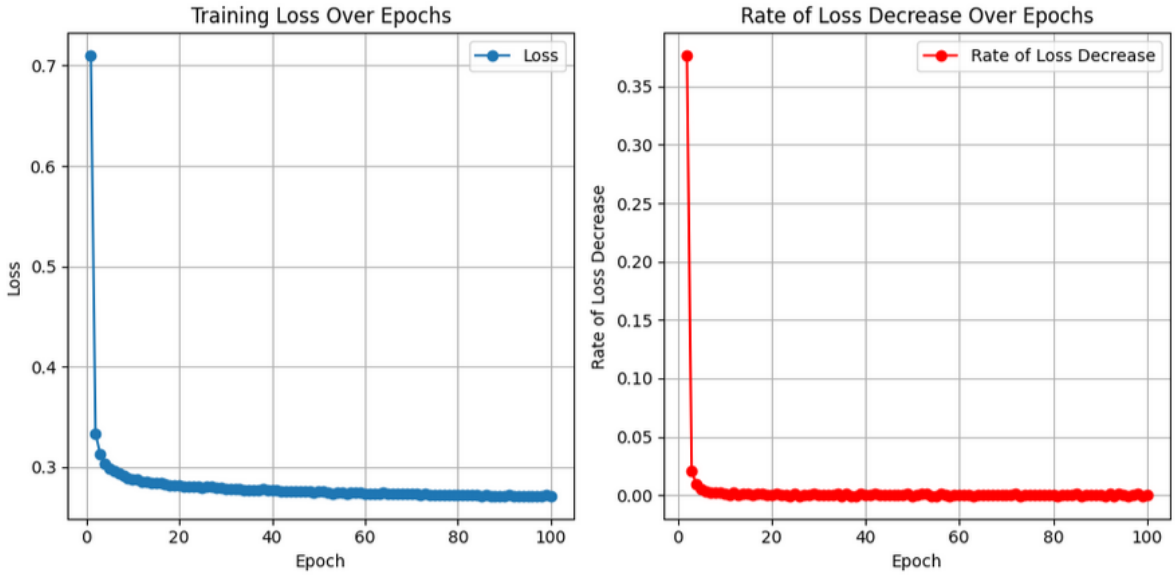

Figure 5-6 Training loss for Convolutional Neural Network

During our logistic regression training, we observed that the loss exhibited minimal reduction even after 100 epochs. Given this trend, we determined that training beyond 5 epochs provides negligible benefit. Therefore, we will standardize our logistic regression training to 5 epochs for efficiency.

Similarly, for our CNN model, we noted that the rate of loss reduction slows significantly after the 10th epoch. Based on this observation, we will standardize CNN training to 10 epochs to balance model performance and training efficiency.

5 Implementation

5.1 Crypto Context Setup

Key parameters	Value
Scaling Mod Size	40
Multiplicative Depth	9
Security Level	HEStd_128_classic
Batch Size	16

Table 6-1 Table of crypto context key parameters

The table above showcases some of the key parameter that was used to generate the crypto context one of which is the security level, which is set to 128 bits, ensuring compliance with classical 128-bit security, the current industry standard. This level of security provides a robust defence against classical attacks, aligning with best practices in cryptographic implementations.

Another crucial parameter is the multiplicative depth, which defines the number of consecutive multiplications a ciphertext can undergo before noise accumulation compromises its accuracy, necessitating bootstrapping. While a higher multiplicative depth enables more complex computations, it also increases computational overhead. To optimize performance, the multiplicative depth in our experiment will be set to the minimum necessary level, ensuring sufficient depth for accurate computations while minimizing computational costs. The pseudocode for all key implementations can be found in Appendix A.

5.2 Key Generation

After setting up the cryptographic context, we proceed with generating key pairs for two parties (Party A and Party B). In our implementation, we need to support multiple homomorphic operations, including:

- Summation (EvalSum) – Used for aggregating encrypted values.
- Multiplication (evalMult) – Required for performing homomorphic multiplication.
- Dot Product (EvalAtIndex) – Utilizes index-based evaluation keys to enable element-wise computations.

To achieve this, we generate evaluation keys for each of these operations and map them to their respective key pairs. Since we are implementing Multi-Key Fully Homomorphic Encryption (MK-FHE), we must:

- Generate the necessary evaluation keys for Party A.
- Extend these keys to Party B using multi-party key generation.
- Merge both parties' evaluation keys to create a shared key that enables homomorphic operations across multiple parties.
- Insert the final merged keys into the cryptographic context to facilitate secure computation.

This setup allows both parties to perform encrypted computations collaboratively without decrypting their individual data. The block diagram below showcases the key generation flow.

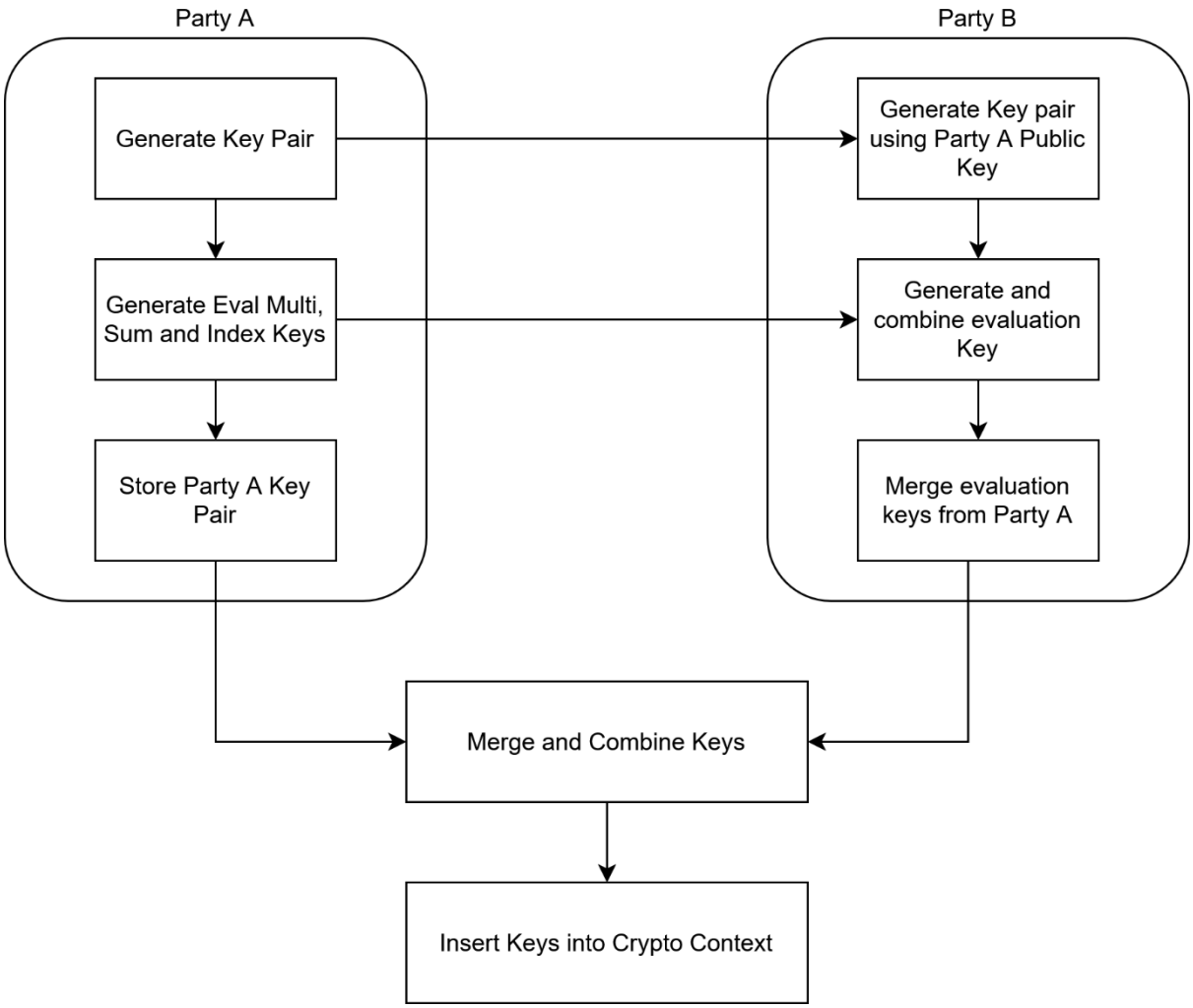

Figure 6-2 Block diagram of key generation flow

5.2 Encryption and Decryption

Encryption in OpenFHE is a straightforward process consisting of two main steps: packing the plaintext and encrypting it with the public key. However, decryption is more complex. Each party must decrypt the same ciphertext separately using their respective secret keys. These partially decrypted ciphertexts are then merged to reconstruct the original plaintext.

Since CKKS uses approximate arithmetic, the decrypted results may contain leading zeros, which we trim for reliability. Additionally, we remove any extraneous values that are not part of the original input. Finally, we adjust the output to match the original input length, ensuring consistency despite the transformations performed in FHE.

5.3 Bootstrapping

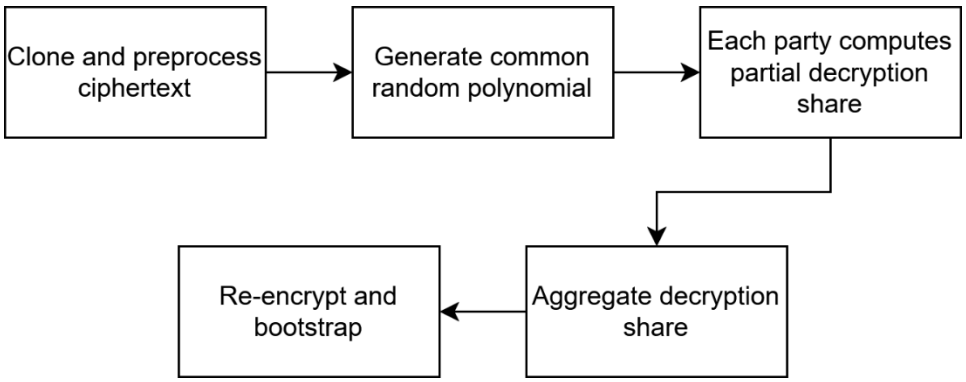

Figure 6-3 Block diagram of bootstrapping flow

With bootstrapping, we effectively reset the noise level in a ciphertext, enabling indefinite homomorphic computations without exceeding the noise threshold. This process involves both parties partially decrypting the ciphertext using their respective secret keys. These partial decryptions are then aggregated to reconstruct the original plaintext in an encrypted form. Finally, the aggregated result is re-encrypted using the public key, producing a refreshed ciphertext with reduced noise.

5.4 Model Functions

Since OpenFHE is a homomorphic encryption library designed to support FHE arithmetic rather than a machine learning framework with built-in API functionality for model creation, we must manually implement fundamental operations such as the forward pass, backward pass, and parameter updates during training. This involves leveraging OpenFHE's encrypted arithmetic functions to replicate core machine learning computations while ensuring that data remains encrypted throughout the process.

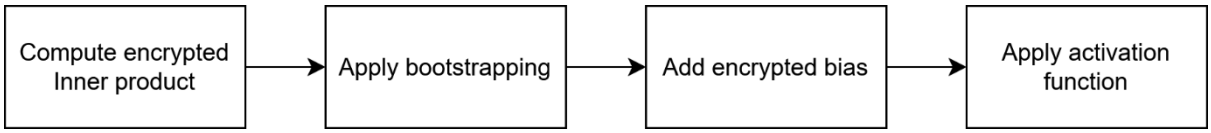

Figure 6-4 Block diagram of forward pass

In the forward pass, we compute the inner product between the encrypted input and encrypted weights, followed by rescaling to maintain numerical stability. Next, it applies bootstrapping to refresh the ciphertext and reduce noise accumulation. The encrypted bias is then added to the result, and an activation function (sigmoid) is applied to introduce non-linearity.

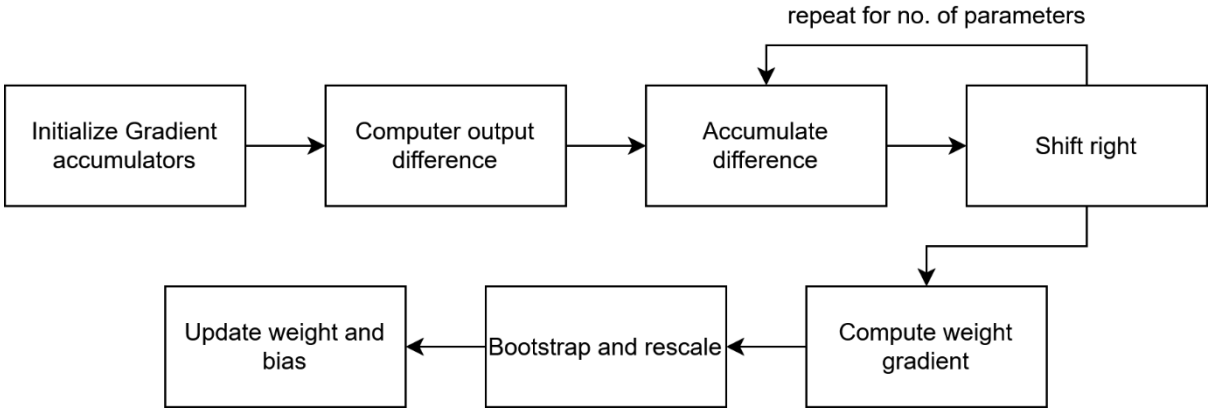

Figure 6-5 Block diagram of backward pass

The backward pass begins by initializing gradient accumulators and defining a mask to isolate the first element of the encrypted output difference. It then computes the difference between the encrypted output and encrypted target (EncOut - EncY) and applies the mask to focus on the first element. The difference is iteratively accumulated over the width of the dataset, shifting the masked values to ensure proper summation.

Next, the intermediate weight gradient is computed by multiplying the encrypted input with the accumulated difference. To manage encryption noise, bootstrapping is applied, followed by rescaling to maintain numerical stability. Finally, the encrypted gradients for weights (ΔW) and bias (ΔB) are updated, and the update count is incremented.

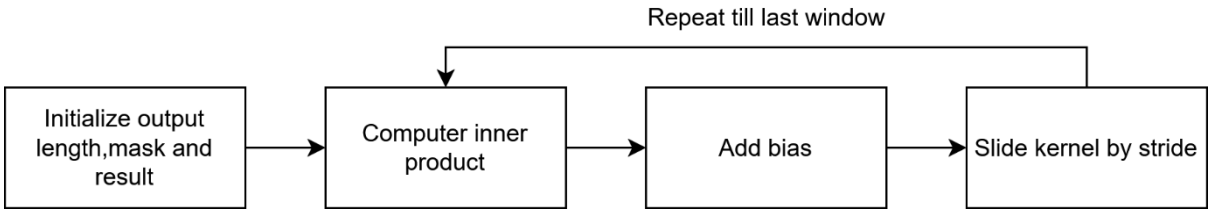

Figure 6-6 Block diagram of 1D convolution

Since OpenFHE does not provide a built-in function for convolution, we must recreate the operation using its existing primitive arithmetic functions. The above block diagram demonstrates how 1D convolution is implemented, ensuring that all computations remain encrypted throughout. This method serves as the convolution layer in our CNN experiment.

The function begins by calculating the output length and initializing an encrypted result. It then iterates over the input, computing the inner product between the encrypted input and kernel, adding an encrypted bias, and applying a masking operation. To simulate the convolution's sliding window, encrypted rotations are performed.

5.7 Experiment Variations

To understand the impact of Multi-Key Fully Homomorphic Encryption (MK-FHE) on machine learning models, we will explore various scenarios where it can be applied. These scenarios include full encryption and partial encryption of both models and datasets. Additionally, in the case of Convolutional Neural Networks (CNNs), we will investigate the effect of encrypting a single layer, providing insights into the feasibility and trade-offs of secure model inference under MK-FHE. We will name our model(M)/data(D) with the following subscript: U= Unencrypted, E= Encrypted, ET= Encrypted post training, CET= Convolution layer encrypted post training.

Model/Data	Description
$\mathrm{M}_U\mathrm{D}_U$	Normal unencrypted training and inference. Act as control for baseline comparison.
$\mathrm{M}_U\mathrm{D}_E$	Model parameters are unencrypted, but data is encrypted during training and inference. Used to ensure data privacy.
$\mathrm{M}_E\mathrm{D}_U$	Model parameters are encrypted, but data is encrypted during training and inference. Used to ensure model privacy.
$\mathrm{M}_E\mathrm{D}_E$	Model parameters and data are encrypted during training and inference. Used to ensure model and data privacy.
$\mathrm{M}_{ET}\mathrm{D}_U$	Model parameters are encrypted after training while data remains unencrypted. Used for securing the model parameters during inference.
$\mathrm{M}_{ET}\mathrm{D}_E$	Model parameters are encrypted after training, data is encrypted during training and inference. Used for securing the model parameters during inference while ensuring data privacy throughout training and inference.
$\mathrm{M}_{CET}\mathrm{D}_E$	Only the convolutional layer of a CNN is encrypted after training, while the rest of the model and data remain unencrypted. Used to protect specific layers while minimizing overhead.

Table 6-3 Description of experiment model/data permutation pair

5.8 Limitations

At the time of writing, 2D convolution is not easily achievable in OpenFHE, as matrix multiplication is not yet supported. However, we can perform 1D convolution because OpenFHE provides built-in inner product and rotation functions, which enable computing the dot product between the encrypted input and kernel while shifting the input as needed.

Another limitation we faced was the lack of computing power and RAM required to fully encrypt the CNN in our experiment. Encrypting the images alone demands significant memory and processing resources, making it infeasible to extend encryption across all layers of the network with our current environment. As a result, we had to focus on partial encryption, such as encrypting only specific layers, to balance security and computational feasibility.

6 Experiment Results

6.1 Logistic Regression

6.1.1 Pima Indians Diabetes Database

Model / Data	Accuracy	Train time	Test time
$M_U D_U$	0.6812	0.08s	0.07s
$M_U D_E$	0.6875	1165.71s	54.70s
$M_E D_U$	0.6875	1168.11s	57.01s
$M_E D_E$	0.6875	1275.64s	60.89s
$M_{ET} D_U$	0.6812	0.01s	42.83s
$M_{ET} D_E$	0.6812	0.08s	51.04s

Table 7-1 Experiment accuracy, train and test duration for PIDD dataset

Additional Preparation Steps

Time taken to encrypt model parameters	0.08s
Time taken to encrypt train set	27.00s
Time taken to encrypt test set	5.59s

Table 7-2 Additional preparation steps for PIDD dataset

6.1.2 Framingham Heart Study Dataset

Model / Data	Accuracy	Train time	Test time
$M_U D_U$	0.6946	0.09s	0.07s
$M_U D_E$	0.6976	3004.45s	122.75s
$M_E D_U$	0.6976	2973.25s	126.15s
$M_E D_E$	0.6976	3034.43s	138.23s
$M_{ET} D_U$	0.6946	0.09s	90.49s
$M_{ET} D_E$	0.6946	0.09s	106.29s

Table 7-3 Experiment accuracy, train and test duration for framingham heart study dataset

Time taken to encrypt model parameters	0.09s
Time taken to encrypt train set	65.79s
Time taken to encrypt test set	14.05s

Table 7-4 Additional preparation steps for framingham heart study dataset

6.2 Convolution Neural Network

6.2.1 MNIST Dataset

Model / Data	Accuracy	Train time	Test time
$\mathrm{M}_U\mathrm{D}_U$	0.9208	64.80s	1.05s
$\mathrm{M}_{CET}\mathrm{D}_E$	0.9208	64.80s	121313.33s

Table 7-5 Experiment accuracy, train and test duration for MNIST dataset

Additional Preparation Steps

Time taken to encrypt model parameters	0.02s
Time taken to encrypt test set	78.47s

Table 7-6 Additional preparation steps for MNIST dataset

6.2.2 Fashion MNIST Dataset

Model / Data	Accuracy	Train time	Test time
$\mathrm{M}_U\mathrm{D}_U$	0.8444	64.95s	1.05s
$\mathrm{M}_{CET}\mathrm{D}_E$	0.8444	64.95s	126715.98s

Table 7-7 Experiment accuracy, train and test duration for fashion MNIST dataset

Additional Preparation Steps

Time taken to encrypt model parameters	0.02s
Time taken to encrypt test set	87.60s

Table 7-8 Additional preparation steps for fashion MNIST dataset

7 Discussion on Experiment Result

7.1 Logistic Regression

The results of logistic regression experiments using the Pima Indians Diabetes Database (PIDD) and the Framingham Heart Study Dataset highlight significant trade-offs between encryption-enhanced privacy and computational efficiency.

Impact on Accuracy

One of the key observations was that the accuracy remained constant across all variations of encryption. This suggests that Multi-Key Fully Homomorphic Encryption (MK-FHE) does not introduce any computational errors or biases into the model. The classification performance remained unchanged, indicating that MK-FHE successfully preserves the integrity of ML computations, even when both the model parameters and the input data are encrypted.

Impact on Training Time

However, a major drawback was the drastic increase in computation time due to encryption. For the PIDD dataset, the fully encrypted model ($\mathrm{M_E D_E}$) required 1275.64s (21.26 minutes) for training, compared to just 0.08s in the unencrypted control case— a staggering increase of 14,571 times. The Framingham dataset exhibited an even greater performance hit, with training taking 3034.43s (50.57 minutes) when encrypted, compared to 0.09s in the unencrypted case.

While both datasets differ in content, they share similar structures: both are numerical tabular datasets, where the primary difference is the number of parameters and entries. This led to an additional analysis of the performance degradation factors to identify the primary contributors to the slowdown.

Investigating the Cause of Performance Degradation

To further understand the source of the computational overhead, an additional experiment was conducted on the Framingham dataset, where various columns were systematically dropped. This allowed us to simulate datasets with varying numbers of parameters and observe how encryption overhead scales with dataset complexity. The result of the experiment is shown below.

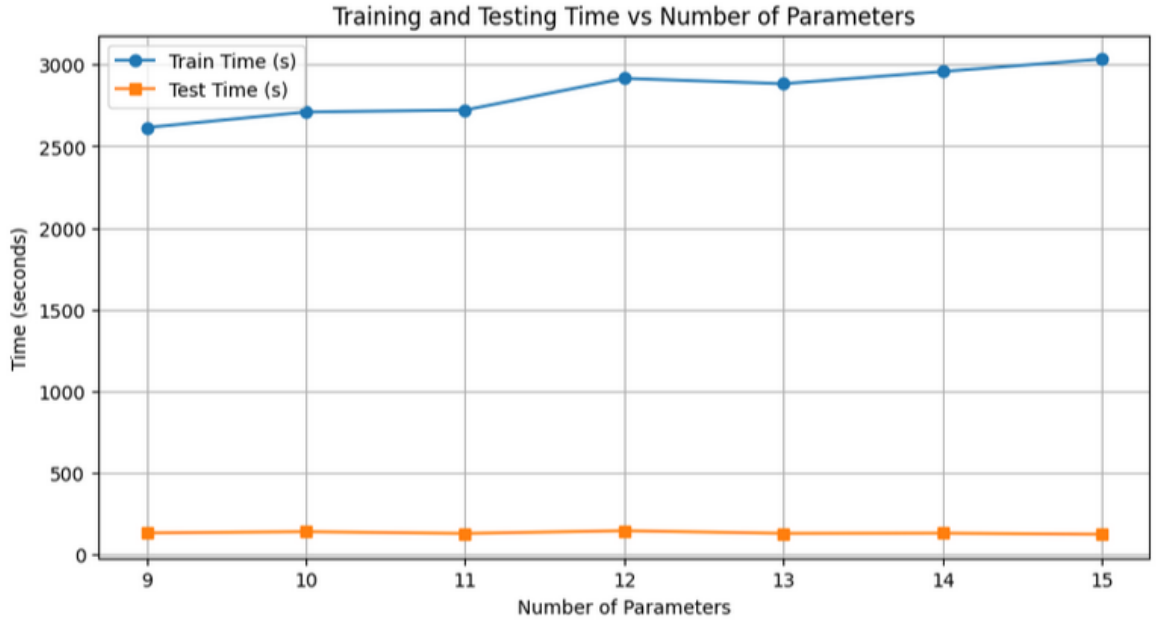

Figure 8-1 Training and test time across different number of parameters

It is observed that as the number of parameters increases from 9 to 15, the training time shows a slight upward trend, indicating that while the number of features contributes to the computational overhead, its impact is relatively minor compared to the overall encryption cost. In contrast, the testing time remains almost constant, suggesting that the number of parameters has little to no effect on inference time. Given the significant slowdown observed in the original experiment, these findings imply that the number of data entries plays a much larger role in performance degradation than the number of features.

Partial Encryption Benefits

For partially encrypted models ($\mathrm{M_U D_E}$, $\mathrm{M_E D_U}$) we noted a very slight decrease in the training and test timing compared to the fully encrypted model. Specifically, for the PIDD dataset, the training time for $\mathrm{M_E D_U}$ was 1275.64s, whereas $\mathrm{M_U D_E}$ and $\mathrm{M_E D_U}$ had slightly reduced training times of 1168.11s and 1165.71s, respectively. A similar trend was observed in the Framingham Heart Study dataset, where the fully encrypted model ($\mathrm{M_E D_E}$) took 3034.43s, while $\mathrm{M_U D_E}$ and $\mathrm{M_E D_U}$ recorded 2973.25s and 3004.45s respectively.

This suggests that when either the model parameters or the input data remain unencrypted, the computational overhead is somewhat reduced, albeit not significantly. The marginal improvement in efficiency indicates that while encryption contributes to the performance bottleneck, the primary factor driving the slowdown remains the arithmetic operations performed in the encrypted domain itself. The minor variation in computation time highlights that MK-FHE's computational cost is dominated by the complexities of encrypted arithmetic.

Post Training Encryption benefits

Our most promising approach in balancing security and computational efficiency was post-training encryption ($\mathrm{M_{ET} D_{E,} M_{ET} D_U}$) which allowed training to be performed in plaintext while ensuring model security during inference. This method eliminated the massive computational overhead of encrypted training, reducing training time to match that of the unencrypted baseline. However, inference time remained higher due to encrypted arithmetic, with test times increasing from 0.07s ($\mathrm{M_U D_U}$) to 42.83s ($\mathrm{M_{ET} D_U}$) and 51.04s ($\mathrm{M_{ET} D_E}$) for the PIDD dataset, and from 0.07s to 90.49s ($\mathrm{M_{ET} D_U}$) and 106.29s ($\mathrm{M_{ET} D_E}$) for the Framingham dataset. While this is a significant slowdown, it remains far more practical than fully encrypted models, which suffered an order-of-magnitude increase in both training and inference times.

7.2 Convolution Neural Network

Impact on Training Time

With our CNN experiment, it reinforces the trend that encrypted-domain arithmetic introduces significant computational overhead, particularly in models that require complex operations such as convolutions. Unlike logistic regression, which relies on simple linear transformations, CNNs involve convolutional layers that require multiple matrix operations, making them inherently more expensive in the encrypted domain. In our setup, only the convolutional layer was encrypted, while the rest of the model remained unencrypted, allowing us to isolate the impact of encrypted convolutions. Despite this partial encryption, inference time increased drastically, from 1.05s (unencrypted) to 121313.33s (33.7 hrs) for MNIST and 126715.98s (35.2 hrs) for Fashion-MNIST. This highlights the computational challenges of applying MK-FHE to deep learning, where even encrypting a single layer incurs substantial latency.

Impact on Accuracy

Additionally, we observed that the Fashion-MNIST model had lower accuracy compared to MNIST, despite using the same CNN architecture. This can be attributed to the forced usage of 1D convolutions instead of standard 2D convolutions due to OpenFHE's lack of native support for encrypted 2D matrix operations. Since 2D convolutions are better suited for spatial feature extraction in image classification, the use of 1D convolutions likely resulted in suboptimal feature representations, leading to reduced classification performance.

8 Conclusion and Future Works

8.1 Conclusion

In this paper, we explored various model extraction attacks and defences, identifying a critical gap in protecting models from white-box extraction attacks. To address this, we proposed the use of MK-FHE as a cryptographic defence mechanism, ensuring both data privacy and model security in MLaaS environments. Our experimental results demonstrated that while MK-FHE successfully preserves model confidentiality without compromising accuracy, its implementation comes with significant computational overhead, particularly in deep learning architectures. Through various experimental permutations, we also found that post-training encryption offers the most pragmatic balance between security and performance, allowing for efficient model training while ensuring model protection during inference.

8.2 Future Works

While this paper has demonstrated the feasibility of implementing MK-FHE on ML models, several areas remain for further exploration and improvement.

Firstly, reducing computational overhead will be crucial for MK-FHE to achieve widespread adoption in real-world applications. This can be addressed through algorithmic optimizations that enhance performance in the encrypted domain.

Secondly, enabling efficient 2D matrix computations in OpenFHE will significantly expand the applicability of MK-FHE to more complex ML architectures, such as GPTs and GNNs, further broadening its potential use cases.

Thirdly, improving the scalability of MK-FHE to support larger datasets and deeper models is essential for practical deployment. This may involve developing more efficient key-switching mechanisms, optimizing ciphertext packing strategies, and exploring hybrid approaches that balance security and performance.

9 References

[1] “Machine Learning as a Service (MLAAS) market size | Mordor Intelligence.” [Online]. Available: https://www.mordorintelligence.com/industry-reports/global-machine-learning-as-a-service-mlaas-market

[2] X. Zou, Y. Hu, Z. Tian, and K. Shen, “Logistic Regression Model Optimization and Case Analysis,” in *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*, 2019, pp. 135–139. doi: 10.1109/ICCSNT47585.2019.8962457.

[3] Admin, “What is Convolutional Neural Network? What are all the layers used in it?” Oct. 2019. [Online]. Available: https://www.i2tutorials.com/what-is-convolutional-neural-network-what-are-all-the-layers-used-in-it/

[4] T. Orekondy, B. Schiele, and M. Fritz, “Knockoff Nets: Stealing Functionality of Black-Box Models,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2019, pp. 4949–4958. doi: 10.1109/cvpr.2019.00509.

[5] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Stealing Machine Learning Models via Prediction APIs,” in *USENIX Security Symposium*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:2984526

[6] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz, “Towards Reverse-Engineering Black-Box Neural Networks.” 2017.

[7] W. Jiang, H. Li, G. Xu, T. Zhang, and R. Lu, “A Comprehensive Defense Framework Against Model Extraction Attacks,” *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 1–16, 2024, doi: 10.1109/TDSC.2023.3261327.

[8] M. Juuti, S. Szyller, S. Marchal, and N. Asokan, “PRADA: Protecting against DNN Model Stealing Attacks.” 2019. [Online]. Available: https://arxiv.org/abs/1805.02628

[9] M. Tang *et al.*, “ModelGuard: Information-Theoretic Defense Against Model Extraction Attacks,” in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5305–5322. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/tang

[10] “HOLBEIN WATERMARK.” [Online]. Available: https://museum.wales/blog/783/HOLBEIN-WATERMARK/

[11] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein, “A Watermark for Large Language Models.” 2024. [Online]. Available: https://arxiv.org/abs/2301.10226

[12] J. Xu, F. Wang, M. D. Ma, P. W. Koh, C. Xiao, and M. Chen, “Instructional Fingerprinting of Large Language Models.” 2024. [Online]. Available: https://arxiv.org/abs/2401.12255

[13] M. Christ, S. Gunn, and O. Zamir, “Undetectable Watermarks for Language Models.” 2023. [Online]. Available: https://arxiv.org/abs/2306.09194

[14] K. Brush and M. Cobb, “asymmetric cryptography.” Mar. 2024. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/asymmetric-cryptography

[15] “Homomorphic Encryption: How it works | Splunk.” [Online]. Available: https://www.splunk.com/en_us/blog/learn/homomorphic-encryption.html

[16] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978, doi: 10.1145/359340.359342.

[17] D. Boneh, E.-J. Goh, K. Nissim, and J. Kilian, “Evaluating 2-DNF Formulas on Ciphertexts,” in Lecture Notes in Computer Science. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 325–341. doi: 10.1007/978-3-540-30576-7_18.

[18] J. H. Cheon *et al.*, “Homomorphic Encryption for Arithmetic of Approximate Numbers,” vol. 10624, in Lecture Notes in Computer Science, vol. 10624. , Switzerland: Springer International Publishing AG, 2017, pp. 409–437. doi: 10.1007/978-3-319-70694-8_15.

[19] *2842-2021 : IEEE Recommended Practice for Secure Multi-Party Computation*. New York, NY, USA: IEEE, 2021. doi: 10.1109/IEEESTD.2021.9604029.

[20] A. A. Badawi *et al.*, “OpenFHE: Open-Source Fully Homomorphic Encryption Library.” 2022. [Online]. Available: https://eprint.iacr.org/2022/915
[21] K. A. Cochran, “The CIA Triad: Safeguarding Data in the Digital Realm,” United States: Apress L. P, 2024, pp. 17–32. doi: 10.1007/979-8-8688-0432-8_2.
[22] L. Deng, “The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web],” *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, 2012, doi: 10.1109/MSP.2012.2211477.
[23] GeeksforGeeks, “Fashion MNIST with Python Keras and Deep Learning.” Mar. 2023. [Online]. Available: https://www.geeksforgeeks.org/fashion-mnist-with-python-keras-and-deep-learning/
[24] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.” 2017. [Online]. Available: https://arxiv.org/abs/1708.07747

Appendix A

```
def setup_openfhe():
    """
    Function to set up the OpenFHE CKKS context and generate encryption keys.
    """
    # Create an instance of CKKS parameters
    params = CCParamsCKKSRNS()
    # Set the scaling factor size for CKKS encryption
    params.SetScalingModSize(40)
    # Set the multiplicative depth (determines the number of multiplications
that can be performed before re-encryption)
    params.SetMultiplicativeDepth(9)
    # Set the security level to 128-bit classical security
    params.SetSecurityLevel(HEStd_128_classic)
    # Set the ring dimension (determines the size of polynomials used in
encryption)
    params.SetRingDim(32768)
    # Set the scaling technique (determines how rescaling is handled in CKKS)
    params.SetScalingTechnique(FIXEDAUTO)
    # Set the batch size (number of packed values per ciphertext)
    params.SetBatchSize(16)
    # Generate the CKKS CryptoContext based on the configured parameters
    cc = GenCryptoContext(params)
    # Enable Public Key Encryption (PKE) feature, allowing encryption and
decryption
    cc.Enable(PKESchemeFeature.PKE)
    # Enable Key Switching, which allows ciphertexts to be transformed
between keys
    cc.Enable(PKESchemeFeature.KEYSWITCH)
    # Enable Leveled Homomorphic Encryption (LHE), allowing multiple
operations without bootstrapping
    cc.Enable(PKESchemeFeature.LEVELEDSHE)
    # Enable Advanced Homomorphic Encryption (SHE), supporting more complex
operations
    cc.Enable(PKESchemeFeature.ADVANCEDSHE)
    # Enable Multi-Party Computation (MPC), allowing multiple users to
interact with encrypted data
    cc.Enable(PKESchemeFeature.MULTIPARTY)
    # Return the configured CryptoContext
    return cc
```

Table A-1 Code fragment of crypto context parameters

```
# For Party A
kp1 = cc.KeyGen()
evalMultKey = cc.KeySwitchGen(kp1.secretKey, kp1.secretKey)
cc.EvalSumKeyGen(kp1.secretKey)
evalSumKeys = cc.GetEvalSumKeyMap(kp1.secretKey.GetKeyTag())
cc.EvalAtIndexKeyGen(kp1.secretKey, [-1])
evalAtIndexKeys=cc.GetEvalAutomorphismKeyMap(kp1.secretKey.GetKeyTag())

# For Party B
kp2 = cc.MultipartyKeyGen(kp1.publicKey)
evalMultKey2 = cc.MultiKeySwitchGen(kp2.secretKey, kp2.secretKey, eval-
MultKey)

# Combine evaluation multiplication keys from both parties
evalMultAB = cc.MultiAddEvalKeys(evalMultKey, evalMultKey2, kp2.pub-
licKey.GetKeyTag())
evalMultBAB = cc.MultiMultEvalKey(kp2.secretKey, evalMultAB, kp2.pub-
licKey.GetKeyTag())

# Summation keys for both parties
evalSumKeysB = cc.MultiEvalSumKeyGen(kp2.secretKey, evalSumKeys, kp2.pub-
licKey.GetKeyTag())
evalSumKeysJoin = cc.MultiAddEvalSumKeys(evalSumKeys, evalSumKeysB,
kp2.publicKey.GetKeyTag())
cc.InsertEvalSumKey(evalSumKeysJoin)

# Final evaluation multiplication keys from both parties
evalMultAAB = cc.MultiMultEvalKey(kp1.secretKey, evalMultAB, kp2.pub-
licKey.GetKeyTag())
evalMultFinal = cc.MultiAddEvalMultKeys(evalMultAAB, evalMultBAB, eval-
MultAB.GetKeyTag())

# Insert the final multiplication key
cc.InsertEvalMultKey([evalMultFinal])
evalAtIndexKeysB=cc.MultiEvalAtIndexKeyGen(kp2.secretKey, evalAtIndexKeys,
[-1], kp2.publicKey.GetKeyTag())
evalAtIndexKeysJoin = cc.MultiAddEvalAutomorphismKeys(evalAtIndexKeys,
evalAtIndexKeysB, kp2.publicKey.GetKeyTag())
cc.InsertEvalSumKey(evalAtIndexKeysJoin)
```

Figure A-2 Code fragment for key generation

Algorithm 1 *Encrypt Input using CKKS*

Require: CC (Crypto Context), PublicKey, InputData
Ensure: Ciphertext
1: ***Create CKKS Packed Plaintext:***
2: Plaintext ← MakeCKKSPackedPlaintext(CC, InputData)
3: ***Encrypt the Plaintext:***
4: Ciphertext ← Encrypt(CC, PublicKey, Plaintext)
return *Ciphertext*

Algorithm 2 *Decrypt Ciphertext using CKKS Multiparty Decryption*

Require: *CC (Crypto Context), KP 1 SecretKey, KP 2 SecretKey, Ciphertext, OriginalSize*
Ensure: *DecryptedValues*
1: ***Compute Partial Decryption Shares:***
2: CiphertextPartial1 ← MultipartyDecryptLead([Ciphertext], KP 1 SecretKey)
3: CiphertextPartial2 ← MultipartyDecryptMain([Ciphertext], KP 2 SecretKey)
4: ***Aggregate Partial Decryptions:***
5: PartialCiphertextVec ← [CiphertextPartial1[0], CiphertextPartial2[0]]
6: ***Perform Multiparty Decryption Fusion:***
7: DecryptedPlaintext ← MultipartyDecryptFusion(PartialCiphertextV ec)
8: ***Extract and Process Decrypted Values:***
9: DecryptedValues ← GetCKKSPackedV alue(DecryptedPlaintext)
10: TrimmedValues ← DecryptedValues[: OriginalSize]
11: RealDecryptedValues ← [RealIfClose(v, 1e−15)forvinTrimmedValues]
12: FloatDecryptedValues ← [float(Real(v))forvinRealDecryptedValues]
return *FloatDecryptedValues*

Figure A-3 Pseudocode for decryption and encryption

Algorithm 3 *Two-Party Bootstrapping*

Require: *Ciphertext, PartyA, PartyB, CC*
Ensure: *BootstrappedCiphertext*
1: ***Check Bootstrapping Condition:***
2: ***if*** *Ciphertext.GetLevel() < 2* ***then return*** *Ciphertext*
3: ***end if***
4: ***Clone and Preprocess Ciphertext:***
5: C1 ← Ciphertext.Clone()
6: C1.RemoveElement(0)
7: ***Generate Common Random Polynomial:***
8: A ← IntMPBootRandomElementGen(PartyB.PublicKey)
9: ***Each Party Computes Decryption Share:***
10: Share1 ← IntMPBootDecrypt(PartyA.SecretKey, C1, A)
11: Share2 ← IntMPBootDecrypt(PartyB.SecretKey, C1, A)
12: ***Aggregate Decryption Shares:***
13: AggregatedShares ← IntMPBootAdd([Share1, Share2])
14: ***Re-encrypt and Bootstrap:***
15: BootstrappedCiphertext ← IntMPBootEncrypt(PartyB.PublicKey, AggregatedShares, A, Ciphertext
return *BootstrappedCiphertext*

Figure A-4 Pseudocode for bootstrapping

Algorithm 4 *Encrypted Forward Pass*

Require: *EncX, WeightEnc, BiasEnc, CC, KP1, KP2*
Ensure: *Encrypted Output*
1: ***Compute Encrypted Inner Product:***
2: Rescale WeightEnc
3: EncOut ← InnerProduct(EncX, WeightEnc)
4: Rescale EncOut
5: ***Apply Two-Party Bootstrapping:***
6: EncOut ← Bootstrapping(EncOut, KP1, KP2, CC)
7: ***Add Encrypted Bias:***
8: EncOut ← Add(EncOut, BiasEnc)
9: ***Apply Activation Function:***
10: EncOut ← Sigmoid(EncOut)
return *EncOut*

Figure A-5 Pseudocode for forward pass

Algorithm 5 *Encrypted Backward Pass*

Require: *EncX, EncOut, EncY , CC, KP1, KP2*
Ensure: *Updated encrypted weight and bias gradients*
1: ***Initialize Gradient Accumulators:***
2: ExtOutMinusY ← Encrypt(Key, PackedPlaintext([0] × 15))
3: Mask ← PackedPlaintext([1,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
4: ***Compute Output Difference:***
5: OutMinusYW ← EncOut − EncY
6: OutMinusYW ← OutMinusYW × Mask
7: OutMinusYB ← OutMinusYW
8: ***Accumulate Output Difference:***
9: ***for*** *i = 0 to 14* ***do***
10: ExtOutMinusY ← ExtOutMinusY + OutMinusYW
11: Shift OutMinusYW left by 1 position
12: ***end for***
13: ***Compute Intermediate Weight Gradient:***
14: IntermediateV al ← EncX × ExtOutMinusY
15: Apply two-party bootstrapping on IntermediateVal
16: Rescale IntermediateVal
17: ***Update Gradients:***
18: ΔW ← ΔW + IntermediateV al
19: ΔB ← ΔB + OutMinusYB
20: Increment update count
return *Updated ΔW , ΔB*

Figure A-6 Pseudocode for backward pass

Algorithm 6 Encrypted Parameter Update

Require: CC, WeightEnc, BiasEnc, ΔW , ΔB, Count, KP1, KP2
Ensure: Updated WeightEnc and BiasEnc
1: **Check if Forward Pass Was Run:**
2: if Count = 0 then
3: Raise Error: ”You should at least run one forward iteration”
4: end if
5: **Compute Regularization Term**
6: Term1 ← EvalMult(WeightEnc, MakeCKKSPackedPlaintext([0.05] ×15))
7: RescaleInPlace(T erm1)
8: **Compute Scaled Gradient Update**
9: Term2 ← EvalMult(ΔW, MakeCKKSPackedPlaintext([1/Count] × 15))
10: RescaleInPlace(Term2)
11**: Aggregate and Bootstrap to Manage Noise**
12: IntermediateSum ← EvalAdd(Term1, Term2)
13: **Update Encrypted Weights**
14: WeightEnc ← EvalSub(WeightEnc, IntermediateSum)
15: WeightEnc ← TwoPartyBootstrapping(WeightEnc, KP1, KP2, CC)
16: **Update Encrypted Bias with Correct Scaling**
17: DeltaB Scaled ← EvalMult(ΔB, MakeCKKSPackedPlaintext([1/Count]))
18: DeltaB Scaled ← TwoPartyBootstrapping(DeltaB Scaled, KP1, KP2, CC)
19: BiasEnc ← EvalSub(BiasEnc, DeltaB Scaled)
20: **Reset Gradient Accumulators and Count**
21: ΔW ← Zero9
22: ΔB ← Zero1
23: Count ← 0
return Updated WeightEnc, BiasEnc

Figure A-7 Pseudocode for parameter update

Algorithm 7 *Encrypted 1D Convolution*

Require: *EncInput, EncKernel, EncBias, InputLen, KernelLen, Stride*
Ensure: *TensorOutput*
1: ***Compute Output Size:***
2: OutputLen ← (InputLen−KernelLen)/Stride + 1
3: ***Initialize Encrypted Result:***
4: Mask ← MakeCKKSPackedPlaintext([1])
5: EncResult ← Encrypt([0] × OutputLen)
6: ***for*** *i = 0 to OutputLen − 1 do*
7: EncDotProd ← EvalInnerProduct(EncInput, EncKernel)
8: EncDotProd ← EvalAdd(EncDotProd, EncBias)
9: EncDotProd ← EvalMult(Mask, EncDotProd)
10: EncResult ← EvalAdd(EncResult, EncDotProd)
*11: **if** i < OutputLen − 1 **then***
12: EncInput ← EvalRotate(EncInput, Stride)
13: EncResult ← EvalRotate(EncResult, −1)
*14: **end if***
15: ***end for***
16: ***Decrypt and Process Output:***
17: DecryptedResult ← Decrypt(EncResult)
18: ReversedResult ← Reverse(DecryptedResult)
19: TensorOutput ← TorchTensor(ReversedResult, dtype = torch.f loat32)
20: TensorOutput ← Reshape(T ensorOutput, (1, 1, −1))
return *TensorOutput*

Figure A-8 Pseudocode for 1D convolution

Appendix B

The Jupyter Notebook used for our experiment is available on GitHub and requires setting up the OpenFHE Python library before execution. Detailed setup instructions can also be found in the following GitHub repository:

https://github.com/lumiinous/CCDS24-0128-FYP