CURE: Privacy-Preserving Split Learning Done Right

Halil Ibrahim Kanpak
Koç University
Istanbul, Türkiye
hkanpak21@ku.edu.tr

Aqsa Shabbir
Bilkent University
Ankara, Türkiye
aqsa.shabbir@bilkent.edu.tr

Esra Genç
Bilkent University
Ankara, Türkiye
esragenc522@gmail.com

Alptekin Küpçü
Koç University
Istanbul, Türkiye
akupcu@ku.edu.tr

Sinem Sav
Bilkent University
Ankara, Türkiye
sinem.sav@cs.bilkent.edu.tr

Abstract

Training deep neural networks often needs large datasets stored and processed in the cloud, and in sensitive fields like healthcare, these workflows must follow strict privacy rules. Split Learning (SL), a framework that divides model layers between client(s) and server(s), is widely adopted for distributed model training. While SL reduces privacy risks by limiting server access to the full parameter set, previous research has identified that intermediate outputs exchanged between server and client can compromise the client's data privacy. Homomorphic encryption (HE)-based solutions exist, but they often impose prohibitive computational burdens. To address these challenges, we propose CURE, a novel system based on HE for the single-client setting that encrypts only the server side of the model and optionally the data. CURE enables secure SL while substantially improving communication and parallelization. We propose packing schemes for efficient execution of deep learning algorithms and generalize them to MLPs and convolutional models, enabling the evaluation of large architectures using our implementations, such as ResNet blocks. We demonstrate that CURE can achieve similar accuracy to plaintext SL, while being up to 210x more efficient in terms of the runtime compared to the state-of-the-art privacy-preserving alternatives. Finally, we propose a novel estimator that enables efficient use of HE in SL settings by recommending an optimal server-client split.

Keywords

Split learning, homomorphic encryption, outsourced learning, privacy-preserving machine learning

1 Introduction

Big data has been a key driver of machine learning (ML) advancements, enabling the training of more complex models. However, massive datasets create storage and processing bottlenecks, making local computation on standard machines impractical. Additionally, handling big data raises privacy concerns due to sensitive information. Collaborative ML enables multiple parties to train a machine learning model without sharing raw data or the model itself. The most popular collaborative ML techniques include federated learning [32, 33] and split learning [21]. Federated Learning (FL) enables multiple parties to train a machine learning model without sharing their local data directly. Instead, they share local model updates with a central server, which aggregates these updates to train a global model. Split Learning (SL), on the other hand, splits the neural network (NN) architecture into client-side and server-side models. Thus, it facilitates the training of NNs without sharing the data and/or labels with the server. SL is especially useful in asymmetrical computational resource settings, where clients may lack significant computational power.

Although FL and SL reduce privacy risks by restricting the server's access to raw data or segments of the model, recent research demonstrates that the client's intermediate model updates, i.e., the gradients shared with the server, can still inadvertently leak the training data or the labels [13, 15, 16, 24, 44, 46, 55, 57, 58, 88]. Researchers focused on developing new defense strategies to mitigate this leakage in FL using differential privacy (DP) [1, 45, 73, 85, 86], homomorphic encryption (HE) [14, 28, 70, 71], or secure multiparty computation (MPC) [25, 65, 79, 81, 82, 91, 94, 95]. To mitigate various adversarial attacks in SL, several works rely on DP [3, 76, 78, 84, 87]. However, DP-based learning requires low privacy budgets, resulting in lower accuracy [64]. Some works employ outlier detection [11, 12]. Another line of research employs HE for encrypted training or inference [29–31, 59]. While Pereteanu et al. integrate HE for *inference* tasks in SL [59], most efforts to improve privacy focus only on U-shaped SL, where the neural network is divided into three segments: the client handles the initial and final layers, while the server processes the intermediate layers [29–31]. This setting assumes that the client holds its own data and labels, necessitating sufficient storage and computational capacity on the client side.

In this work, we focus on privacy-preserving training in SL framework with a single client and a single server, where the server has access to the samples and the client holds the labels. We ensure label confidentiality and, optionally, sample protection. This approach suits scenarios where clients outsource sample storage and parts of the training, such as large-scale genomic datasets. While genomic data, like that related to Autism Spectrum Disorder (ASD), may be stored unencrypted, it typically does not reveal sensitive labels. However, the labels themselves, which are critical for complex traits influenced by numerous genomic variants, remain sensitive and must be protected [10, 53, 69].

To ensure data and label confidentiality, we propose CURE, a novel system that leverages HE to encrypt model parameters on the

Proceedings on Privacy Enhancing Technologies 2026(2), 259–276

https://doi.org/10.56553/popets-2026-0047

server side only. This allows the server to work with an encrypted model, while the client –the data and label owner– operates on a plaintext model. Encrypting the server-side model allows CURE to mitigate privacy attacks and ensures label privacy by default. Additionally, CURE can optionally encrypt data samples for enhanced data privacy. Thus, our setup protects data and label confidentiality while reducing communication and computation through plaintext training on the client side.

Our contributions can be summarized as follows: (i) We introduce a novel system, CURE, for privacy-preserving SL that ensures the confidentiality of labels and (optionally) the data using HE in single-server, single-client setting. (ii) We propose two packing schemes that ensure efficient computation under various settings, for one-level server operations. (iii) We generalize our packing to support encrypted multi-layer server operations. (iv) We implement a novel estimator that, for the first time, optimally determines the best network split for SL, ensuring efficient use of CURE based on the server's and client's resources. (v) For the first time, we evaluate the performance of ResNet building blocks in an encrypted SL setting. (vi) We develop an open-source API for HE operations in deep learning networks, facilitating the extension of these techniques. Our implementation is available at https://github.com/CRYPTO-KU/CURE-Privacy-Preserving-Split-Learning.

2 Related Work

Split Learning (SL) [21] is a machine learning technique that allows model training on distributed datasets without the need to exchange raw data between participants. SL accomplishes this by dividing the ML model into sections, each handled by a different party. It gained recognition with SplitNN [80], as a more resource-efficient model than approaches like FL [33, 73, 77]. It enables various configurations [27, 38, 60, 61, 68], including the *vanilla configuration* [43, 80, 93], where the network is divided at a specific split layer. Additionally, *vertical SL* [4, 6] involves different parties holding different features of the dataset [50, 51]. In contrast, *horizontal SL* involves different dataset samples held by various parties to be processed independently [8, 63] to enhance query performance. Soon after SL gained recognition, several attacks were developed, including inference attacks [13, 58], hijacking attacks [15], backdoor attacks [88, 90], feature distribution attacks [16], data reconstruction attacks [89], and property inference attack [57].

Privacy-Preserving SL: To enhance privacy, several works integrate a mechanism called differential privacy (DP) to SL [3, 76, 78, 84, 87]. DP adds noise to the data or intermediate values shared between client and server, thereby reducing the accuracy of the results. Our work differs from DP-based methods by combining HE –encryption that supports computation on ciphertext– with SL, removing this privacy–accuracy tradeoff.

Other works [29–31, 59] integrate HE into SL. By encrypting data or model parameters, any information obtained by attackers is rendered useless without the decryption key. Pereteanu et al. [59] propose a solution leveraging HE and U-shaped split Convolutional Neural Networks (CNN) to ensure data privacy, specifically designed for fast and secure *inference*. Their model enhances secure *inference* by distributing the model weights between the client and server, with the client computation done in plaintext. In contrast, our approach focuses on efficient and secure *training* of the model using advanced packing techniques to optimize communication and computation. By encrypting server-side model parameters and utilizing an *inverted split learning* setup, where the server processes the initial layers and sends intermediate results to the client, we enable collaborative training while preserving confidentiality.

Khan et al. address the privacy challenge in SL by integrating HE into training to encrypt activation maps before transferring them from the client to the server [29–31]. For this, the authors developed a U-shaped split 1D CNN model, with the initial and final layers are computed by the client and the intermediate layers computed by the server. This design enables clients to protect the privacy of their ground truth labels. In [30], the authors enhanced the model further by ensuring that clients do not need to share either their input training samples or ground truth labels with the server. Similarly, in [29], the authors extended their experiments and introduced batch encryption to optimize memory usage and computational performance when handling encrypted data. Nguyen et al. [52] refined the approach of [30] by reducing their privacy leakage and improving communication efficiency utilizing CKKS HE scheme (see Section 3.3). Finally, while other approaches integrate HE into SL to enhance privacy, our method stands out by (i) focusing on optimizing training efficiency through HE applied *exclusively to server-side model parameters*, and (ii) implementing an estimator function to optimize the server-client split in constrained settings, improving HE utilization and overall efficiency.

3 Building Blocks

3.1 Neural Networks (NN)

NN is a model composed of interconnected layers of nodes [19]. During training, the network adjusts connection weights between neurons to minimize the loss between predictions and actual outcomes, using optimization algorithms. Input data (X) is passed through the network to produce a predicted output ($\hat{Y}$). The forward pass predicts $\hat{Y}$ by applying activation to a linear combination of the layer's weights and the previous layer's activations: $\hat{Y} = \psi(Z_l) = \psi(W_l O_{l-1} + B_l)$. Here, l is the current layer, O_{l-1} denotes the output of the previous layer $l-1$, ψ is the activation function (e.g., Sigmoid, ReLU), W_l and B_l denote the weight matrix and the bias vector at layer l, respectively. We denote the linear combination of the weights and the activations as Z_l. After the forward pass, backpropagation [66] updates the network's weights by calculating a loss function (J) using the predicted output ($\hat{Y}$) and the labels (Y). The gradient of the loss function is calculated by $g = \frac{\partial J}{\partial Z_l} = \frac{\partial J}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z_l}$ where g is the gradient of the loss function with respect to the input Z_l at layer l. The parameters (weights and biases) are then updated using: $W_l \leftarrow W_l - \alpha \frac{\partial J}{\partial W_l}$ and $B_l \leftarrow B_l - \alpha \frac{\partial J}{\partial B_l}$, where α is the learning rate.

3.2 Split Learning (SL)

SL [6, 21, 60, 80, 93] is designed to enhance privacy while enabling collaborative model training across multiple entities. In SL, the NN model with a total of $n + k$ layers is divided into two segments: the client-side segment with k layers and the server-side segment with n layers. Each client processes its local data (X) through the

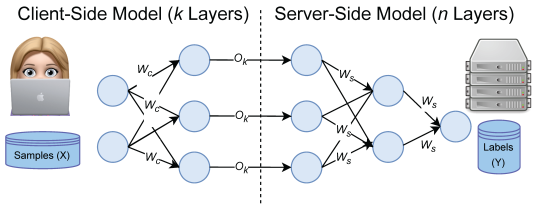
first k layers of the model. The output from the k-th layer, denoted as O_k, is then transmitted to the server. Instead of transmitting raw data, this intermediate representation is used for subsequent computations. The server then continues the forward pass through the remaining n layers to compute the predicted output, denoted as $\hat{Y}$, which is used to evaluate the loss function (J). The server computes the gradient and sends it back to the client(s). Each client uses this gradient to perform the backpropagation through its k layers and update its model parameters accordingly. This iterative process continues until the model converges or reaches a predefined number of epochs. For a detailed explanation of our SL architecture and its implementation, see Section 4.1.

SL offers several advantages. First, it enhances privacy by keeping raw data on the client side and sharing only abstract intermediate representations, reducing the privacy risk. Second, SL reduces the client's computational load, as clients only process k layers, making it ideal for devices with limited resources. Thus, SL shows promise for achieving secure and efficient collaborative learning across diverse domains. However, adversarial attacks [13, 15, 16, 24, 40, 44, 46, 57, 58, 88–90] on SL continue to pose a threat. To address this, we enhance SL by integrating HE, which we detail below.

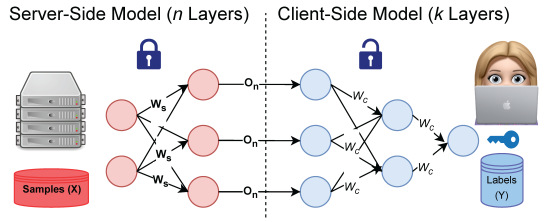
3.3 Homomorphic Encryption (HE)

HE enables computation on ciphertexts, producing encrypted results that, when decrypted, yield the same outcome as if the operations were performed on the plaintext. This is essential for privacy-preserving computations, allowing data to be processed without compromising confidentiality. In this work, we use the Cheon-Kim-Kim-Song (CKKS) scheme [9], which is a leveled HE scheme based on the ring learning with errors (RLWE) problem [42]. The scheme is well-suited for supporting approximate (floating-point) precision. CKKS significantly enhances computational efficiency with its *packing capability*, enabling simultaneous processing of multiple data points through Single Instruction, Multiple Data (SIMD) operations on encrypted data. The scheme also has effective noise management strategies, where the noise refers to the small error added to ciphertexts to ensure security. The ring in CKKS is defined as $\mathbb{Z}[X]/(X^N + 1)$, where N is a power of two. Key parameters include the cyclotomic ring size (N), the ciphertext modulus (Q), the logarithm of the moduli of the ring ($LogQP$), the noise parameter (σ), and the level of the ciphertext (L) that help manage the depth of the circuit to be evaluated before refreshing the ciphertext through the Bootstrap operation. The scheme allows packing $N/2$ values to plaintext/ciphertext slots for SIMD operations. The slots of the vector can be rearranged through an operation known as "rotations", which can be computationally expensive. We introduce the key functionalities of the CKKS scheme here:

- $\mathsf{KeyGen}(1^\lambda) \rightarrow \mathsf{PK}, \mathsf{SK}$: Generates a public key (PK) for encryption and a secret key (SK) for decryption, given a security parameter (λ) in unary.
- $\mathsf{Enc}_{\mathsf{PK}}(m) \rightarrow c$: Encrypts a plaintext message (m) into ciphertext (c) using PK.
- $\mathsf{Dec}_{\mathsf{SK}}(c) \rightarrow m$: Decrypts a ciphertext message (c) back into the plaintext message (m) using SK.
- $\mathsf{Eval}_{\mathsf{PK}}(c_v, c_w) \rightarrow c$: Performs arithmetic operations such as addition and multiplication directly on ciphertexts (c_v, c_w), producing a new ciphertext c that represents the result of the operation on the original plaintexts. Each multiplication consumes one level of the ciphertext.
- $\mathsf{Bootstrap}(c) \rightarrow c'$: Refreshes ciphertexts (c) to produce a fresh ciphertext (c') at the initial level.



(a) Traditional vanilla SL: The client processes data samples $\mathbf{X}$ through k layers, while the server holds the labels Y and computes the remaining n layers. The server then returns the gradients required for updating W_c.



(b) CURE: The inverted SL setup where the server processes data samples $\mathbf{X}$ through n layers, while the client holds the labels Y and processes k layers of the NN. Server-side weights $\mathbf{W}_s$ are encrypted, while client-side weights W_c remain plaintext.

Figure 1: *(a) Vanilla SL and (b) CURE's inverted SL model.*

We denote encrypted ciphertext vectors in bold case, e.g., $\mathbf{X}$, and encoded plaintext vectors in regular case, e.g., X, throughout the paper. We use the notation $\mathsf{Eval}^f_{\mathsf{PK}}(\{c_i\}) \rightarrow c$ to denote that a function f is evaluated over a set of ciphertexts $\{c_i\}$ (potentially through multiple basic Eval operations consuming multiple levels of ciphertexts), resulting in the ciphertext c.

4 METHOD

Problem Statement. We consider an *inverted SL scenario*, where model training is divided between a server and a client. The server potentially has access to the data samples X, but not the labels Y, which are only known to the client. This setting is motivated by a client who wishes to outsource storage and part of the computation to the server side. Our goal is to enable training within this SL framework while preserving the confidentiality of the labels and, optionally, the data samples. Note that (reconstruction, inference, etc.) attacks on the client side are beyond the scope of this work, as we assume the client owns both the samples and labels but outsources storage and some processing.

Threat Model. We consider a semi-honest threat model, which remains the most practical and widely accepted assumption in HE-based SL frameworks, including [30, 31, 52]. Most SL systems also operate in a single-client, single-server setting [72], where both

parties follow the protocol honestly without collusion. However, the server may passively attempt to infer sensitive information from exchanged messages or model parameters. This honest-but-curious model is common in collaborative ML and enables the design of more efficient cryptographic protocols [67]. Our threat model also reflects realistic deployment scenarios. In particular, the semi-honest model achieves an effective trade-off between security and efficiency. It provides strong confidentiality guarantees via semantically secure HE schemes such as CKKS, while avoiding the substantial computational and communication overhead incurred by malicious-security mechanisms (e.g., zero-knowledge proofs or verifiable computation). We further discuss potential extensions of CURE to the malicious setting in Section 4.5.

Motivation. In CURE, the client owns the data but outsources storage and computation to the server, which holds only encrypted model parameters and operates solely on ciphertexts. This design ensures that the server learns nothing about sensitive label information, crucial in clinical and proprietary settings, while maintaining practical performance. Because all server-side computations are performed on encrypted data, CURE removes the main attack surfaces targeted by input-reconstruction and membership-inference attacks [46, 58, 74, 96], as well as any other attack that rely on leaked gradients, weights, input, or output, preventing the server from extracting any meaningful information. We present a formal security definition and proof in Section 4.5.

In the inverted SL setting, the server may store or process the feature data X, either in *encrypted* or *plaintext* form, while the labels Y represent the most sensitive information kept by the client. To protect this information, CURE encrypts **all server parameters** throughout learning so that the server gains no knowledge about the labels or their influence on model updates. This design also reduces the client's computational load, as it needs to store only the labels and the final few layers of the model.

Thus, CURE is particularly suitable for settings where clients outsource data storage and portions of training, such as large-scale genomic datasets where the size of raw data (storage costs) can be dominating. Clients may offload data to the server while keeping labels private. In some cases, transferring such labels may even be restricted by regulation. An example is Autism Spectrum Disorder (ASD) [10, 53, 69]: Although genomic features (e.g., ASD-related data) may be stored unencrypted, they generally do not disclose whether an individual is affected. The labels, however, directly encode this sensitive information. Because these labels are crucial for studying complex traits shaped by many genomic variants, they require strong protection and CURE enables this by design. We note that CURE also provides optional sample privacy by encrypting client data before sending it to the server. This offers a tunable trade-off between security and computational cost based on application needs. Finally, our motivation for the resource-constrained client setting is supported in Section 5.2.2, where we compare CURE against fully client-side plaintext training.

4.1 Overview of CURE

We propose a novel framework, CURE, designed to enable collaborative machine learning across client and server with asymmetric computational resources. We employ HE, in particular the CKKS scheme (see Section 3.3), to allow computations to be performed on encrypted data, ensuring that sensitive information remains secret and eliminating attacks via communicated values throughout the training process. We present the overall workflow of CURE in comparison to vanilla SL in Figure 1. Figure 1(a) illustrates the traditional vanilla SL setup, while Figure 1(b) depicts our inverted SL design. Throughout the paper, we denote server-side and client-side parameters with a subscript of 's' and 'c', respectively.

Algorithm 1 CURE Training Phase

Server has (encrypted) data $\mathbf{X}_{[1,2,\ldots,m]}$, encrypted initial weight matrix $\mathbf{W}_s$, and the client's public key PK_c.
Client has labels Y.

for epoch = $1 \rightarrow e$ **do**
  for $\mathbf{X} \in \mathbf{X}_{[1,2,\ldots,m]}$ **do**
    Server performs encrypted forward pass:
      $\mathbf{O}_n \leftarrow \mathsf{Eval}^{f_s}_{\mathsf{PK}_c}(\mathbf{W}_s, \mathbf{X})$
      Send $\mathbf{O}_n$ to client
    Client works on plaintext:
      $O_n \leftarrow \mathrm{Dec}_{\mathsf{SK}_c}(\mathbf{O}_n)$
      $\hat{Y} \leftarrow f_c(W_c, O_n)$
      $J \leftarrow \mathrm{Loss}(\hat{Y}, Y)$
      Compute gradients g_{W_s}, g_{W_c}
      $W_c \leftarrow \mathrm{Update}_c(W_c, g_{W_c})$
      $\mathbf{g}_{\mathbf{W}_s} \leftarrow \mathrm{Enc}_{\mathsf{PK}_c}(g_{W_s})$
      Send $\mathbf{g}_{\mathbf{W}_s}$ to server
    Server performs encrypted backpropagation:
      $\mathbf{W}_s \leftarrow \mathsf{Eval}^{\mathrm{Update}_s}_{\mathsf{PK}_c}(\mathbf{W}_s, \mathbf{g}_{\mathbf{W}_s})$
  end for
end for

The server stores the data samples ($\mathbf{X}$) and performs the forward pass ($f_s(\cdot)$) up to n layers with server-side model parameters $\mathbf{W}_s$. The encrypted output ($\mathbf{O_n}$) is sent to the client, who then decrypts it and completes the forward pass ($f_c(\cdot)$) of the remaining k layers, denoted as W_c. Note that $\mathbf{W}_s$ denotes the server-side weight matrix while $\mathbf{w_i}$ are the column/row entries (vectors) of the matrix. The client, which holds the labels (Y), computes the loss (J) and its gradients (g_{W_s} and g_{W_c}), updating client weights W_c locally and sending the encrypted ($\mathbf{g}_{\mathbf{W}_s}$) to the server. The server updates its parameters in an encrypted fashion. This process ensures that (optionally the data X and) the labels Y remain confidential, adhering to the objectives of our inverted SL framework.

Our protocols' security relies on the premise that the server, despite observing encrypted gradients communicated during the training, cannot deduce the underlying labels better than random guessing, provided that the HE scheme effectively makes the encrypted values indistinguishable from random. CURE is designed to ensure that all interactions and computations are conducted securely to maintain data and/or label privacy throughout the process. This approach protects sensitive information while enabling scalable, efficient distributed learning when participants differ in computational power or data sensitivity.

4.2 CURE's Design:

4.2.1 Initialization. This phase of CURE, as detailed in Algorithm 2 in Appendix A, is crucial for setting up the necessary cryptographic keys and model parameters for secure and efficient training. Initialization begins by defining the split model architecture, where L_s represents the server-side layers $[l_1, l_2, \ldots, l_n]$ and L_c represents the client-side layers $[l_{n+1}, \ldots, l_{n+k}]$. Thus, l_n denotes the split layer index; for example, l_2 indicates that the network is split after the second layer between client and server. The client and the server randomly initialize their weights through the *GenRandomWeights* function that randomly initializes the weight matrices for a set of layers (Lines 4 and 9). The client also generates a pair of public and secret keys $(\mathsf{PK}_c, \mathsf{SK}_c)$ using KeyGen operation of HE scheme (Line 5), and then sends the public key (PK_c) to the server (Line 6). The server encrypts its weights $(\mathbf{W}_s)$ using PK_c (Line 10). Thus, initialization ensures that the server-side weights are encrypted before any data exchange, maintaining privacy from the outset. If the server does not know the input data X, which is only known by the client, then the client encrypts it locally to prepare for training, and sends the ciphertext $\mathbf{X}$ to the server.

4.2.2 Training. CURE's training algorithm is detailed in Algorithm 1. First, the server performs the forward pass of n layers under encryption, either on encrypted data $\mathbf{X}$, or on plaintext data X, depending on the application. We note that in the latter case, CURE only protects label confidentiality. At this step, the server computes a forward pass $f_s(\cdot)$ on its model portion using the encrypted weights $\mathbf{W}_s$ and the batch $\mathbf{X}$ (Line 4), producing an encrypted output $\mathbf{O}_n$ of n layers, which is then sent to the client (Line 5). The client then decrypts $\mathbf{O}_n$ using its secret key SK_c (Line 7), and performs a forward pass $f_c(\cdot)$ on its k-layer model portion using O_n and local weights W_c (Line 8), resulting in the predicted output $\hat{Y}$. The loss J is computed using $\hat{Y}$ and the true labels Y (Line 9). Gradients for the client side (g_{W_c}) and server side (g_{W_s}) are computed (Line 10). Note that Here, g_{W_s} denotes the gradient $\frac{\partial J}{\partial O_n}$ at the split index, which the server uses alongside its stored encrypted activations to compute its actual weight updates under HE. The client updates its weights W_c using its gradient g_{W_c} (Line 11), encrypts the server gradient g_{W_s} with PK_c (Line 12) as $\mathbf{gw}_s$, and sends the result to the server (Line 13). Finally, upon receiving $\mathbf{gw}_s$, the server updates its $\mathbf{W}_s$ accordingly (Line 15). This process repeats for each batch and continues for the predefined number of epochs (e), ensuring efficient and secure training of the model through collaborative computation between the client and server. The client computation performed on plaintext (Lines 7 to 11) is referred to as *UpdPlain* for convenience.

4.3 Homomorphic Operations

This section outlines our approach to homomorphic operations and their corresponding implementation details.

4.3.1 Homomorphic vs. Plain Evaluation Modes. In our framework, each neural network layer supports two modes of operation: a homomorphic mode for encrypted computations and a plaintext mode for standard computations. This dual-mode design guarantees functional equivalence of the underlying mathematical operations—up to the noise inherent in the CKKS scheme—across both settings. Such equivalence is essential for verifying the correctness of the cryptographic protocols and enables flexible placement of the server–client split point at any layer of the network.

4.3.2 Packing. In CKKS, each ciphertext carries up to $N/2$ real slots, which we exploit to co-locate data and amortize HE costs. In CURE, we support SIMD packing in various parts of the implementation to achieve operability using homomorphic operations:

1. Data Packing for Linear Layers To perform matrix-vector multiplication $y = Wx + b$ homomorphically, we employ a row-wise packing strategy. During a setup phase, each row of the weight matrix W is encoded into a dedicated ciphertext. The elements of the j-th row, $W_{j,*}$, occupy the initial slots of this ciphertext. This layout is designed for maximizing computational throughput (see Appendix B). This layout is designed to compute the dot product $W_{j,*} \cdot x$ homomorphically using a single ciphertext-ciphertext multiplication followed by a tree-like summation of the resulting slots.

For the backward pass, which requires multiplication by the transposed weight matrix W^T, weights are also pre-encoded in a column-wise fashion into separate plaintexts. Additionally, a set of one-hot plaintext vectors is prepared to facilitate the extraction and placement of specific slots during the computation.

2. Channel-Block Packing for Convolutional Layers For convolutional layers, we introduce a channel-packing technique to maximize the SIMD throughput. The goal is to compute multiple output channels in parallel within a single ciphertext. Given an input with C_{in} channels and a kernel of size $\kappa \times \kappa$, the number of slots required to process a single receptive field for one output channel is $D_{patch} = C_{in} \cdot \kappa^2$. We can therefore pack G_{size} output channels into a single ciphertext, where: $G_{size} = \left\lfloor \frac{N/2}{D_{patch}} \right\rfloor$. During a pre-computation phase, the kernel weights are encoded into a series of plaintext masks. For each spatial position (dy, dx) in the kernel, a specific mask is constructed. This mask contains the weights for that position across all input channels, structured in a block-diagonal format where each block corresponds to a group of G_{size} output channels. This layout lets one homomorphic multiplication apply the kernel weights for a given spatial offset to all corresponding inputs across a whole group of output channels at once. This significantly reduces the number of required multiplications and rotations during the forward pass. A similar mask set is used to efficiently compute gradients in the backward pass.

By tailoring the number of slots to the layer's fan-in (columns) or group of filters (channels), we minimize the number of ciphertexts, reduce rotations, and maximize SIMD throughput. See Appendix B for examples of packing implementations. Next, we explain the homomorphic evaluation of these linear and convolutional layers.

4.3.3 Homomorphic Evaluation of Linear Layers. Our linear layer provides a unified API for forward, backward, and update operations in the encrypted setting. Below, we detail the forward pass, backward pass, and weight update operations.

Forward Pass. We compute the operation $y = Wx + b$ and produce single ciphertext where the j-th slot holds the j-th element of the output vector y. A concrete breakdown of our one-level and n-level linear layer implementations, which realize this process, is provided in Appendix C and Appendix D, respectively. We summarize the sequence of operations for each output neuron j:

(1) **Homomorphic Dot Product:** The input ciphertext, containing the vector x, is multiplied element-wise with the pre-encoded ciphertext for the j-th weight row. The resulting ciphertext is then summed internally using a tree-based pattern of optimized rotations and additions. This collapses the element-wise products into a single slot (slot 0), yielding the encrypted dot product $W_{j,*} \cdot x$.
(2) **Result Placement:** The encrypted dot product, located in slot 0, is moved to the j-th slot via a rotation. A pre-computed one-hot plaintext mask is then applied to zero out other slots.
(3) **Output Assembly:** The resulting ciphertexts—each containing a single dot product at the correct output position—are added together. This assembles a final output ciphertext where the first $|l_{i+1}|$ slots contain the parallel dot-product results.
(4) **Bias Addition:** The plaintext-encoded bias vector is added to the assembled ciphertext to produce the final result.

We support two packing strategies for the dot-product:

- **Batch multiplication ($\odot$):** Pack multiple weight columns into each mask, perform one `MulNew` and tree-sum per column-group.
- **Scalar multiplication ($\otimes$):** Mask and sum one column at a time.

The ratio $\frac{N/2}{|l_2|}$ decides the preferred method (see Section 4.4). Both appear in one-level batch multiplication and one-level scalar multiplication methods (Appendix C).

Backward Pass. The backward pass homomorphically computes the weight (g_{W_s}) and the input gradients (g_X). Note that all gradients, including g_X, are denoted as g_{W_s} in Algorithm 1 for clarity.

(1) **Weight Gradients (g_{W_s}):** The gradient for the j-th weight row is the outer product of the j-th element of the output gradient and the cached input vector from the forward pass. This is computed by first isolating the j-th gradient element from the incoming gradient ciphertext. This element is then homomorphically broadcast across a new ciphertext, which is then multiplied by the cached input ciphertext. This process is repeated for each row of the weight matrix.
(2) **Input Gradients (g_X):** The input gradient is computed by multiplying the transposed weight matrix W^T by the output gradient ciphertext. This is performed by multiplication with pre-encoded plaintext columns of the weight matrix. The results are then aggregated using a tree-sum to produce the final input gradient ciphertext.

Update. The server updates its encrypted weights using the computed gradients. Each encrypted weight gradient ciphertext is multiplied by a plaintext encoding of the learning rate α. The resulting ciphertext is then subtracted from the corresponding encrypted weight ciphertext, performing a secure stochastic gradient descent.

4.3.4 Homomorphic Convolution. Our homomorphic 2D convolution maps the sliding-window operation to ciphertext manipulations by packing each input channel into a row-major ciphertext.

Forward Pass. The convolution with a $\kappa \times \kappa$ kernel is performed as follows:

(1) **Input Alignment:** For each position (dy, dx) in the kernel, the entire input ciphertext is rotated. The rotation amount is determined by the input feature map's width (W_{in}). The total rotation is calculated as $-(dy \cdot W_{in} + dx)$ slots. This rotation aligns the input data corresponding to that kernel position across all receptive fields simultaneously.
(2) **Masked Multiplication:** The rotated input is multiplied by a pre-encoded plaintext mask. This mask contains all the kernel weights for the given position (dy, dx), packed in a block-diagonal structure that aligns with groups of output channels. This single multiplication applies the kernel weights for one spatial position to the corresponding input values for all output channels in parallel.
(3) **Accumulation:** The results from each masked multiplication are accumulated via homomorphic addition. This step effectively sums the products over the entire receptive field, completing the dot-product computation.
(4) **Bias Addition:** After iterating through all kernel positions and input channels, the plaintext-encoded bias is added to the accumulated result.

This process yields a set of output ciphertexts, where each contains the feature maps for a group of output channels.

Backward Pass. The backward pass computes the gradients for the weights (g_W) and the input (g_X) gradients homomorphically.

(1) **Weight Gradients (g_W):** The gradient g_W is computed by performing a cross-correlation between the input feature maps (cached from the forward pass) and the loss gradient with respect to the convolution output. This involves multiplying the output gradient ciphertext with appropriately rotated versions of the input ciphertexts. Pre-computed gradient masks are used to efficiently collect these products according to the kernel structure and channel grouping. The results are aggregated and bias gradients are computed separately.
(2) **Input Gradients (g_X):** g_X are computed using a process similar to transposed convolution. This involves multiplying the output gradient ciphertext with rotated versions of the kernel weights. These operations are aligned and accumulated to reconstruct the gradients for the input feature maps.

Update. We follow the same procedure as the homomorphic update of the linear layer.

4.3.5 Homomorphic Average Pooling. We implement encrypted average pooling over a $p \times p$ window using a two-phase gather-and-place approach that consumes a single multiplicative level. For each non-overlapping window in the feature map:

(1) **Gather Phase:** To compute the sum of all elements in a window, each element is first isolated using a one-hot plaintext mask and then rotated to a common accumulation slot (slot 0). The results of these are homomorphically added, producing a ciphertext with the window's sum in slot 0.
(2) **Place Phase:** The ciphertext containing the sum is multiplied by a plaintext encoding the scaling factor $1/p^2$ to compute the average. The result, located in 0th slot, is rotated to its final position in the output feature map.

This procedure is repeated in parallel for all windows, producing a single output ciphertext for each input feature map.

Pooling has no trainable parameters; during backpropagation, gradients are simply scattered to contributing inputs, which in HE occurs over ciphertexts. First, each output gradient is isolated from its ciphertext using a one-hot plaintext mask. This isolated gradient

is then scaled by a plaintext encoding the factor $1/p^2$, where p is the pool size. Finally, this scaled value is distributed to the $p \times p$ original input slots through rotations, with each rotation moving the value to its correct input position. The final input gradient is the homomorphic sum of all these distributed values.

Non-linear activations cannot be applied directly under encryption (only polynomial functions are allowed). To address this, we employ polynomial approximations for both activation functions and their derivatives (see Appendix E for details). This approach allows us to evaluate widely adopted non-linear activation functions (e.g., ReLU) while keeping the multiplicative depth to a minimum. Since these element-wise operations do not require expensive ciphertext rotations, their computational cost is negligible compared to the linear layers.

4.3.6 Bootstrapping. For an initial level of L, CKKS allows for at most L multiplications to be carried out. As encrypted data undergoes multiple operations, noise accumulates, potentially making ciphertexts undecipherable. Thus, after L multiplications, Bootstrap(c) function (see Section 3.3) must be executed to refresh the ciphertext level, enabling continued operations. In CURE, we rely on bootstrapping operations when the combined number of encrypted layers n and the degree of the activation function d consumes all available levels, i.e., when $(log_2(d+1))n + n > L$. In practice, different server layers may use different approximation levels (and even different activation functions). The total number of bootstraps required during training, denoted as γ, can be expressed as:

$$\gamma = \frac{\sum_{i=1}^{l} \left(\frac{l_i \times l_{i+1}}{\frac{N}{2}} \right) \times (\mu) \times (1+d)}{L} \tag{1}$$

Here, $N/2$ denotes the number of slots in the RLWE vector, and μ is the multiplicative depth. Bootstrapping is performed when the accumulated operation depth across server-side layers exceeds the multiplicative depth limit. This allows encrypted server-side computations to proceed correctly, preserving efficiency and security.

4.4 Estimator for Server-Client Optimization

In this section, we develop a novel estimator to enhance CURE's utilization. Building on its theoretical foundation, we implement an advisor function to optimize the server-client split in CURE. This function aims to determine the optimal partitioning of neural network layers between server and client while optimizing HE operations, training latency, memory usage, and bootstrapping frequency. The advisor takes key properties such as the desired training time (T_d), computer specifications to calculate the latency of rotations, the depth of the multiplicative circuit (μ), the depth of the additive circuit (ρ), the total number of bootstrapping operations (γ) calculated in Formula 1, and the network bandwidth (O_c) available between the client and server. We focus on these parameters as our preliminary experiments suggest they are the most dominant factors. Rotations significantly affect training latency, making them crucial for time efficiency, while μ and ρ influence accuracy and the need for bootstrapping. Consequently, we provide CURE-advisor that can provide recommendations for scenarios where the client's computational power is limited, communication bandwidth is constrained, or the precision impact from the server side is critical. We present the runtime analysis, comparing the estimates from the advisor function with the actual runtime observed in Section 5.2.4. Finally, we introduce CURE's complexity analysis in Table 1 to facilitate the estimator function and its implementation.

4.4.1 Server-Side Computational Complexity. The server-side forward pass involves homomorphic matrix-matrix multiplication and one-level operations, introducing computational overhead due to HE. The computational complexity is largely dominated by rotations and HE multiplications. The total number of encrypted matrix-matrix multiplications for one forward pass is given by:

$$\sum_{i=0}^{n} \left(\frac{|\bar{l}_i| \times |\bar{l}_{i+1}|}{N/2} \right) \times \log(|\bar{l}_{i+1}|) \tag{2}$$

where $N/2$ is the number of slots, and $|\bar{l}_i|$ is the smallest power of two greater than the size of the i^{th} layer. This equation defines the time complexity of the server's forward pass, taking into account the logarithmic cost of rotations.

For other layers, such as convolutional and pooling layers, the complexity analysis similarly considers the specific HE operations required, as detailed in Table 1. This includes the cost of rotations for input alignment, masked multiplications for weight application, and accumulations. The server-side complexity estimation takes into account the number of samples processed, batch size, and the packing strategies defined for each layer type, similar to the principle illustrated in the linear layer example.

Additionally, the server also tracks the cumulative homomorphic depth up to the current layer. This depth is crucial because when δ_{current} exceeds the maximum permissible depth $\delta_{\max}$, bootstrapping is triggered, incurring additional cost. The cumulative depth $\delta^{(i)}_{\text{current}}$ for each server layer i is calculated as:

$$\delta^{(i)}_{\text{current}} = \delta^{(i-1)}_{\text{current}} + \left(\frac{l_i \times l_{i+1}}{\frac{N}{2}} \right) \times (\mu + \rho) \times (1+d) \tag{3}$$

Here, d is the degree of the activation function and μ and ρ are the depths of the multiplicative and additive circuits, respectively. The server performs δ_{current} calculation to determine when bootstrapping is necessary. For simplicity in notation, we assume all activation functions have a degree of d. Selecting the maximum degree as d among the activation functions, our analysis establishes an upper bound on the overall complexity. The server estimator ultimately provides its estimation as T_{server}, accounting for both the forward and backward pass, as well as bootstrapping if needed.

$$T_{\text{server}} \approx 2|X| \cdot \left[\sum_{i=0}^{n} (\text{Cost}_{\text{layer}_i}) + \text{Cost}_{\text{bootstrapping}} \right] \tag{4}$$

where $\text{Cost}_{\text{layer}_i}$ encompasses the cost of HE rotations for layer i, (e.g., $\frac{|\bar{l}_i| \times |\bar{l}_{i+1}|}{N/2} \times \log(|\bar{l}_{i+1}|)$ for linear layers) and $\text{Cost}_{\text{bootstrapping}}$ accounts for the cost of refreshing ciphertexts when the multiplicative depth is exceeded.

4.4.2 Client-Side Complexity and Memory Constraints. On the client side, we aim to minimize computational overhead while meeting memory and time constraints. The complexity of these

Operation	Time Complexity	Used memory float32	# Levels Used
One-Level Scalar Multiplication	$\sigma_{\otimes} \cdot \lvert l_0 \rvert$	$\lvert c \rvert \cdot \lceil \frac{\lvert l_1 \rvert}{N/2} \rceil$	1
One-Level Batch Multiplication	$\sigma_{\odot} \cdot \frac{\lvert l_0 \rvert}{N/2}$	$\lvert c \rvert \cdot \lceil \frac{2 \cdot \lvert l_1 \rvert}{N/2} \rceil$	1
Approximated Activation Polynomial	$\sigma_{\odot} \cdot \lceil \log(d) \rceil$	-	$\lceil \log(d) \rceil$
Server-Client Communication	-	$\lceil \frac{\lvert l_n \rvert}{N/2} \rceil \cdot \lvert c \rvert$	0
Bootstrapping	$T_{\text{bootstrap}} \cdot \left[\sum_{l=1}^{L} \frac{\lvert l_i \rvert \times \lvert l_{i+1} \rvert}{N/2} \times \mu \times (1+d) \right]$	$\frac{\lvert l_i \rvert \times \lvert l_{i+1} \rvert}{N/2}$	0
HE Forward Pass	$T_r \cdot \left[\sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right) \times \log(\lvert \bar{l}_{i+1} \rvert) \right]$	$\lvert c \rvert \cdot \sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right)$	$n - 1 + \sum_{i=1}^{n-1} \log(\bar{d})$
HE Back-propagation	$T_r \cdot \left[\sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right) \times \log(\lvert \bar{l}_{i+1} \rvert) \right]$	$\lvert c \rvert \cdot \sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right)$	$n - 1 + \sum_{i=1}^{n-1} \left(\log(\bar{d}) - 1 \right)$
HE Convolution	$C \left[\kappa^2 (T_r + \sigma_{\odot}) + \sigma_{\otimes} \right]$	$\lceil C \rceil \cdot \lvert c \rvert$	$n - 1 + \sum_{i=1}^{n-1} \log(\bar{d})$
HE Average Pooling	$(p^2+1)T_r + p^2 \sigma_{\otimes}$	$\lvert c \rvert$	1

Table 1: *Complexity analysis of CURE's fundamental operations for SL training with a cyclotomic ring size N. $\sigma_{\otimes}$ and $\sigma_{\odot}$ denote the execution time of scalar HE multiplications and HE batch multiplications. T_r and $T_{bootstrap}$ indicate the execution time of a rotation and bootstrapping, respectively. $\lvert c \rvert$ denotes the length of an RLWE vector ciphertext, $\lvert \bar{l}_i \rvert$ the smallest power of two exceeding the size of the i^{th} layer, d the polynomial degree of the approximated activation function, and dot$(\cdot, \cdot)$ the time for a dot product of given matrix dimensions.*

operations is primarily driven by the dot products required for matrix multiplications, which can be calculated as:

$$\sum_{i=n}^{m} \text{dot}\left(|l_i| \times |l_{i+1}|, |l_{i+1}| \times |l_{i+2}|\right) \tag{5}$$

where $\text{dot}(\cdot, \cdot)$ represents the dot product of matrices on the client side with the specified sizes, and $|l_i|$ refers to the size of the i^{th} layer. It is important to note that operations are performed in plaintext on the client side. Additionally, the client must ensure that the memory required for storing encrypted weights does not exceed available capacity. The memory required for storing ciphertexts is proportional to $|c| \cdot \left\lceil \frac{|l_n|}{N/2} \right\rceil$ where $|c|$ denotes the size of one RLWE vector under the given parameter set, and $|l_n|$ represents the size of the last layer processed by the client. This expression represents the theoretical upper bound for memory consumption on the client side. Together, the above calculations enable us to quantify the primary client-side constraints: computational load and memory usage. Client estimator provides its estimation as T_{client}.

$$T_{client} = 2|X| \times \sum_{i=n}^{m} \text{dot}\left(|l_i| \times |l_{i+1}|, |l_{i+1}| \times |l_{i+2}|\right) \tag{6}$$

4.4.3 Data Transfer and Communication Overhead. Another key factor in the advisor function is the communication overhead, determined by the number of ciphertexts transferred. Per epoch, the server sends:

$$T_{\text{comm}} = \frac{|X| \times |l_n|}{N/2} \times |c| \tag{7}$$

where $|X|$ is the number of data samples, $|l_n|$ is the size of the last layer processed by the server, and $|c|$ is the memory size of one ciphertext. By minimizing $|l_n|$, our advisor function can effectively reduce the number of ciphertexts transferred, thereby optimizing communication and enhancing overall efficiency.

4.4.4 Optimal Partitioning Strategy. The advisor function combines computational, memory, and cryptographic constraints from both server and client to determine the optimal split. It computes the server's maximum feasible split index based on rotations and memory limits, then compares it with the client's minimum feasible index to balance efficiency and memory use.

For each feasible split index where the server and client estimators agree, the advisor calculates the total computational time (including rotations and bootstrapping) and communication overhead. It then selects the partition that minimizes training time while satisfying the user's memory and time constraints. The advisor's final recommendation is guided by the following objective: Minimize $(T_{\text{server}} + T_{\text{client}} + T_{\text{comm}})$ where T_{server}, T_{client}, and T_{comm} represent the respective training times for the server, client, and communication phases. By accounting for the complexity of each operation, as outlined in Table 1, the advisor function offers a theoretically sound method for optimizing the partitioning of NN layers in CURE. It also reports which constraints –server time, client time, or communication– exceed their thresholds.

4.5 Security Definition and Proof

We start by observing that previous solutions combining SL with HE [29–31, 59] do not provide any formal security definition and proof. In contrast, we state our guarantee formally and sketch a proof. Our definition of the ideal functionality $\mathcal{F}_{InvSL}$ in Functionality 1 is inspired by [47], but adapted to the inverted SL setup. Observe that the functionality does not return any output to the server, which means the server learns no useful information. The client, on the other hand, obtains intermediate outputs at each epoch, but is considered as honest in our case, since (s)he owns the data and labels. Note that since we consider the semi-honest case, we did not complicate the functionality with aborts.

Functionality 1 ensures that the server learns nothing useful during training. Our security theorem and proof are below.

Theorem 1. *Assuming that the underlying HE scheme provides CPA security, then CURE realizes $\mathcal{F}_{InvSL}$ against a semi-honest server.*

Proof. Note that to prove ideal–real indistinguishability against a semi-honest adversary, it suffices to construct a simulator that reproduces indistinguishable outputs and views given the adversary's inputs. In our case, the server's inputs include the model description

Functionality 1 Inverted SL Ideal Functionality $\mathcal{F}_{InvSL}$

Parameters: Number of epochs e.

(1) Receive the model description from the participants. In particular, receive the forward pass f_s and backpropagation Update_s from the server, and the plaintext forward and backward pass combination $UpdPlain$ from the client.
(2) Depending on the configuration, receive the input X either from the client or the server, and receive the labels Y from the client.
(3) Initialize weights W_s and W_c using $GenRandomWeights$.
(4) For each epoch, up to e epochs, compute

$$O_n^e = f_s(W_s, X)$$
$$W_s = \text{Update}_s(UpdPlain(O_n^e, Y))$$

and send O_n^e to the client.

in the form of the forward pass f_s and backpropagation Update_s, as well as optionally the samples X. The simulator $\mathcal{S}$ sends these inputs to the ideal functionality $\mathcal{F}_{InvSL}$ and receives back nothing, since the server has no output. In our scenario of SL, the only values visible to the server are plaintext or encrypted samples (X or $\mathbf{X}$), encrypted weights ($\mathbf{W}_s$), and encrypted gradients ($\mathbf{g}_{\mathbf{W}_s}$). Initially, assume that the data X is known by the server. In that case, the simulator $\mathcal{S}$ simulates the view of the adversary simply by outputting randomly encrypted values for $\mathbf{W}_s$ and $\mathbf{g}_{\mathbf{W}_s}$. In the case where the client knows the data X and only sends its encryption $\mathbf{X}$ to the server during initialization (and therefore the server does not have plaintext access to the data samples), the simulator encrypts junk instead of samples and outputs $\mathbf{X}$, in addition to the randomly encrypted junk $\mathbf{W}_s$ and $\mathbf{g}_{\mathbf{W}_s}$, to simulate the adversary's view.

This simulator produces a view and output that is indistinguishable from those of the adversary, given that the HE scheme is CPA-secure. A full indistinguishability proof could use a hybrid argument, where ciphertexts would partly be based on real data and based on junk. Given the public key for the encryption scheme, all these ciphertexts can be generated as necessary. The challenge ciphertext received (either real or junk) would be put to a randomly picked location j within the encrypted weights/gradients, and the hybrid encrypted weights/gradients would be shared with the distinguisher. Until $j-1$ all ciphertexts would be based on real values. Up from $j+1$ all ciphertexts would be based on junk (random) data. At location j we will have the challenge ciphertext that we received (either real or junk). If the distinguisher acting as the server were able to distinguish these from real ciphertexts, it would break the CPA security of the underlying HE scheme. On one end, the hybrid would be all real ciphertexts, and on the other end, the hybrid would be all junk ciphertexts. Since there are only polynomially many values (gradients, weights, data samples), distinguishing each neighboring hybrid with negligible probability (due to the CPA security of the underlying HE solution) would mean distinguishing the two end-hybrids overall with negligible probability. Consequently, CURE ensures that no information is leaked to the server as the simulator simply encrypts random junk and realizes $\mathcal{F}_{InvSL}$.

□

Observe that even when the data X is not encrypted on the server side, after the first forward-backward pass, an honest-but-curious server still loses any correlation between the encrypted model weights $\mathbf{W}_s$, encrypted gradients $\mathbf{g}_{\mathbf{W}_s}$, and the original input X. Functionality $\mathcal{F}_{InvSL}$ guarantees that the server gains no additional information during training. For a malicious server, however, CURE needs to be combined with verifiable computation techniques [17], which we leave as future work. Recent advances in zero-knowledge proofs of training [2] demonstrate that it is now feasible to verify, in zero knowledge, that gradient updates are correctly computed without revealing the underlying model parameters or data. Such proofs represent a promising extension for future versions of CURE, integrating zero-knowledge verification mechanisms with HE to provide security against malicious servers.

Resistance to Membership Inference, Reconstruction, and Gradient Leakage Attacks. Membership inference, gradient leakage, or reconstruction attacks exploit access to plaintext outputs or gradients [72]. At a high level, since the server's entire view consists only of ciphertexts, and the CPA-secure CKKS scheme ensures that all observed values are computationally indistinguishable from random, such attacks are effectively precluded. In CURE, backward propagation and updates on the server are also executed over encrypted gradients, meaning that gradient inversion is not feasible.

In summary, CURE guarantees that, during training, the server is not exposed to any plaintext information that could facilitate such attacks. All server-side parameters and intermediate values are computed under CKKS encryption. If data features are also sensitive, the client also encrypts X before sending it. Note that attacks targeting the inference phase (e.g., by end users or third parties) are outside the scope of this work and would require additional secure inference mechanisms.

5 Experimental Evaluation

5.1 Experimental Setup

5.1.1 Implementation Details. All encrypted operators are implemented in Go with the Lattigo library v6 on the CKKS scheme. We provide further implementation details in Appendix F.

5.1.2 Setup. All timing and scalability experiments were conducted on a server with a dual Intel Xeon E5-2650 v3 CPU (providing 40 hardware cores at 2.30GHz) and 251 GB of RAM, running Ubuntu 18.04.6 LTS. All client-side experiments were conducted on a 2023 MacBook Air with the M2 chip. The communication between the client and server for our distributed setup is implemented using the Message Passing Interface (MPI). The accuracy experiments, which utilized our noise-injection simulation, were performed on a Google Colab instance with an NVIDIA T4 GPU.

CKKS Parameters. We tested a range of CKKS ring sizes (N) by varying $\log N$, from 13 to 16. Our parameter sets were chosen to align with the recommendations from the HE Security Standard [5], with our primary set at $\log N = 13$ for most of the experiments.

5.1.3 Datasets and Model Architectures. We use the following canonical architectures (all with degree-3 ReLU approximation). For fully-connected (FC) networks, (a–b–c–...) denotes the number of neurons in each layer from input to output, respectively, while for

Model	Baseline (%)	CURE l_n=2 (%)	ε_{avg}	ε_{max}
Simple MLP	94.74	93.86	4.0e-07	5.8e-07
MLP	97.25	97.19	4.0e-07	5.8e-07
LeNet	98.95	98.76	8.8e-08	1.3e-07
PTB-XL CNN	62.83	62.78	3.5e-08	5.0e-08

Table 2: *Accuracy with split at $l_n = 2$. ε_{avg}: per-sample average difference among plaintext and HE vectors in training. ε_{max}: max error among plaintext and HE vectors. These ε noise values are obtained by running the training concurrently in HE and plaintext settings for* 3 *epochs. Accuracy results are reported after model convergence.*

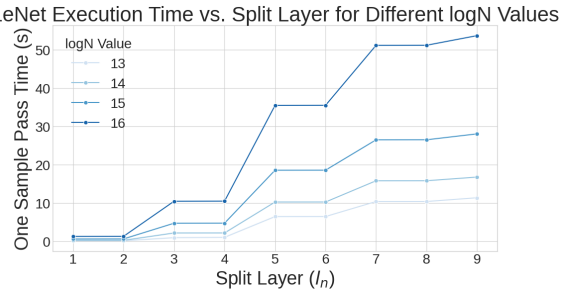


Figure 2: *One sample pass time (seconds) vs.* $\log N$ *and split location. All benchmarks run on 40 cores.*

convolutional layers, the notation '$a \longrightarrow b$' indicates a mapping from 'a' input channels to 'b' output channels:

- **Simple MLP [49]**: An FC network with layers of (64–32–16–10).
- **MLP [49]**: A FC network with layers of (784–128–32–10).
- **LeNet [35]** A classic convolutional stack consisting of two 2D convolutional layers (first $1 \longrightarrow 6$ channels with a 5×5 kernel, then $6 \longrightarrow 16$ channels with a 5×5 kernel), followed by three fully-connected layers with architecture (256-120-84-10).
- **PTB-XL CNN** Khan et al.'s model [31] is a 1D convolutional network designed for ECG classification. It comprises two 1D convolutional layers (first $12 \longrightarrow 16$ channels, then $16 \longrightarrow 8$ channels), each followed by a pooling layer, and culminates in a final fully-connected layer mapping to 5 output classes.
- **Residual blocks (ResNet) [22]** A representative block from a ResNet model, used to evaluate performance on modern, deep architectures. It consists of an initial convolutional layer ($3 \longrightarrow 64$ channels with a 7×7 kernel) followed by two residual-style convolutional layers ($64 \longrightarrow 64$ channels with a 3×3 kernel).

Unless stated otherwise, we use MNIST for the MLP, Simple MLP, and LeNet experiments, and PTB-XL for the PTB-XL CNN [31] and for residual blocks. See Appendix G for preprocessing details.

Finally, we detail our experimental methodolody in Appendix H.

5.2 Results

5.2.1 Model Accuracy. We first evaluate the impact of CURE's privacy-preserving mechanisms—namely, the degree-3 polynomial approximation of ReLU and the inherent noise from CKKS operations—on final model accuracy. We use calibrated noise simulations in our accuracy experiments. This does not bias the accuracy in favor of CURE. Crucially, rather than adding arbitrary noise, we injected the exact noise variance introduced by each HE primitive (e.g., rotation, rescaling) under our fixed parameters. Applying this empirically measured noise at each corresponding plaintext operation allowed us to accurately simulate the cumulative training error. This simulation strategy is standard in HE-based ML research [70, 71]. To validate correctness, we compared intermediate decrypted ciphertext values against their plaintext counterparts for every layer, observing only the expected CKKS approximation error. The outcomes of simulated and fully encrypted executions were statistically indistinguishable. Therefore, the reported accuracy reliably reflects the true performance of CURE, without overstating it. Table 2 summarizes the results from our calibrated noise-injection protocol. In addition to the accuracy metrics, we report ε_{avg} and ε_{max}, which represent the *actual measured divergence* between our HE implementation and the equivalent plaintext computation, not artificially injected noise. Specifically, ε_{avg} denotes the average absolute error observed in 3 epochs, while ε_{max} shows maximum error across all samples in 3 consecutive training epochs. These measurements, obtained by comparing CKKS ciphertext outputs directly against plaintext results, confirming that our noise-injection calibration faithfully reflects the true numerical behavior of HE under RLWE-based arithmetic. Here, baseline refers to the plaintext implementation of our models. Our findings demonstrate that accuracy degradation remains minimal across all evaluated models. For complex architectures such as the MNIST MLP and LeNet, the accuracy drop (Δ) is negligible, at -0.06% and -0.19%, respectively. This demonstrates that CURE can effectively preserve model utility, ensuring that the introduction of strong privacy guarantees does not significantly compromise the primary goal of the ML task.

Model	l_n=1	l_n=3	l_n=5	l_n=7	l_n=9
Simple MLP [49]	0.439	0.529	0.578	NA	NA
MLP [49]	1.476	1.781	1.929	NA	NA
LeNet [35]	0.198	0.745	2.067	2.987	3.214
PTB-XL CNN [31]	0.109	0.221	0.394	NA	NA

Table 3: *Execution times (seconds) for one sample processing with* $\log N = 13$ *on 40 cores, across varying splits (l_n denotes the last layer processed by the server). NA denotes 'Not Applicable'.*

5.2.2 Time Latency and Scalability. Here, we analyze the CURE's overall performance and its scalability with respect to *the split layer index (l_n), the number of cores*, and *CKKS ring size.*

Figure 2 illustrates the execution times on LeNet model across varying **split layer indices.** We use LeNet to illustrate the relationship between latency, split layers, and cryptographic parameters, as it incorporates all implemented layer types and thus captures all these interactions. We remind that l_n denotes the split layer, i.e., the last layer processed by the server, from which the server sends encrypted activations to the client and receives encrypted gradients back. The results indicate that as the 'split layer' position increases, latency grows substantially, reflecting the computational cost of executing more layers under HE. Table 3 quantifies this trade-off across all evaluated architectures. The table confirms that for every model, deepening the split point consistently increases the computational latency. This effect is particularly pronounced for models with large initial layers, such as the MNIST MLP, where

Model	$c = 1$	$c = 2$	$c = 4$	$c = 8$	$c = 16$	$c = 32$	$c = 40$
Simple MLP [49]	16.547	8.379	4.271	2.187	1.052	0.551	0.439
MLP [49]	55.800	28.187	14.126	7.117	3.507	1.852	1.476
LeNet [35]	7.509	3.792	1.901	0.957	0.471	0.249	0.198
PTB-XL CNN [31]	4.130	2.085	1.045	0.528	0.271	0.137	0.109

Table 4: *One sample processing (including forward pass, backpropagation, and update) times (in seconds) obtained from benchmarks for models at* $\log N = 13$, $l_n = 1$, *for all cores. Each core gets its own layer and key instances to reduce serial work. Each row reports runtimes using different numbers of cores (c), shown in the top row.*

encrypting just the first layer ($l_n = 1$) results in a latency of 1.476 seconds, significantly higher than the initial cost for the convolutional models. This trend peaks when the entire model is encrypted on the server (the final entry for each model) which incurs the full computational cost and highest latency.

We further examine the impact of **CKKS ring size** in Figure 2, varying $\log N$ from 13 to 16. Note that the ring size directly influences the security, precision, and performance of the system. The results consistently show that for any given split layer, increasing $\log N$ leads to a notable increase in execution time. For example, for the model at $l_n = 5$, the latency increases from 6.5 s at $\log N = 13$ to 35,496 s at $\log N = 16$. This illustrates the computational overhead introduced by higher ring sizes and precision, underscoring this trade-off as a critical optimization parameter for practitioners.

Finally, to assess the framework's efficiency on modern hardware, we also evaluate **scalability with the number of cores**. Table 4 reports the one-sample processing times for various models as the number of CPU cores increases from 1 to 40. For all models, the runtime decreases substantially with increasing number of cores, demonstrating near-linear speedup. For example, the LeNet model's latency drops from 7.509 seconds on a single core to just 0.198 seconds on 40 cores—a 38x speedup. This confirms that CURE efficiently exploits multi-core processors, a crucial capability for practical deployment of computationally intensive HE workloads.

Scalability for Client. Finally, we examined whether using CURE offers meaningful benefits compared to the client performing full plaintext training. We note that CURE is particularly advantageous when the client lacks sufficient resources to store the dataset X. However, even in relatively simple settings, CURE is beneficial:

We evaluated the end-to-end latency of three of our architectures, under two encryption parameter settings ($LogN = 13$ and $LogN = 14$). The results (Table 5) illustrate the trade-off between local computation execution and the cryptographic overhead introduced at the split index. By varying the cut layer (where the server takes over), we observe that even for a simple MLP, CURE becomes advantageous once the split is applied at layer $l_n = 3$. Similarly, for other architectures, we find that client-side computation drops favorably when the split occurs at $l_n = 2$ or $l_n = 3$. In summary, the client's total execution time is influenced by the encryption and decryption overhead at the split index. By strategically selecting the split index based on the layer's input dimension (fan-in), CURE maximizes packing density, thereby reducing the number of required ciphertexts and the associated cryptographic costs. Furthermore, even in scenarios where the computational advantage is

Model	Split Index	Time (ms) ($LogN$=13)	Time (ms) ($LogN$=14)
MLP	Baseline	5,901.84	5,901.84
	$l_n = 3$	5,623.58	5,563.81
	$l_n = 5$	2,350.17	2,335.23
LeNet	Baseline	23,021.99	23,021.99
	$l_n = 2$	37,052.64	35,439.52
	$l_n = 3$	15,415.45	14,937.20
	$l_n = 5$	7,978.32	7,858.75
	$l_n = 7$	4,749.25	4,693.20
	$l_n = 9$	2,301.07	2,261.84
Khan et al.	Baseline	22,020.71	22,020.71
	$l_n = 2$	46,260.19	43,531.27
	$l_n = 3$	27,244.62	25,879.74
	$l_n = 4$	12,966.00	12,286.28
	$l_n = 5$	5,571.66	5,231.80

Table 5: *End-to-end latency is compared across model architectures for varying split indices and HE parameters (*$\log N$*). The "Baseline" corresponds to fully local, plaintext training on the client.*

marginal, offloading remains architecturally significant. Deferring layers to the server alleviates client-side memory constraints by removing the need to store the full model parameters and allows for secure data storage on the server side.

5.2.3 Large Models. We evaluated our framework on a ResNet block to assess the scalability and performance trade-offs inherent in applying HE to large, modern architectures. The ResNet [22] block evaluated in our experiments represents a fundamental component of modern deep neural networks. Implementing this block in CURE provides a consistent baseline for fair performance comparisons and for analyzing noise propagation in residual architectures.

Accuracy. To quantify the impact of noise in large models, our noise injection simulation measured a final standard deviation of 1.414 after a single pass on the ResNet block. This result was derived from a simulation that models an input with a normalized noise standard deviation of 1.0, adds a fixed variance for each of the four main HE operations within the block (two convolutions and two batch normalizations), and calculates the final deviation after the skip connection which is a network design element that bypasses layers to directly add an earlier layer's output to a later layer's output. Quantified noise growth per block is a key metric for practitioners in predicting training behavior.

Time Latency Analysis. We measured the end-to-end time latency for one-sample processing, which includes the forward pass, backpropagation, and weight update. Figure 3 presents the results for a ResNet block at $\log N = 13$ on 40 cores, varying the split index. We observe that the scalability trends hold for large models, though ResNet incurs a higher execution time.

The execution time of the ResNet block stems directly from its architectural complexity, particularly the high channel depth (e.g., 64 input and 64 output channels), which demands orders of magnitude more HE operations than models like LeNet. This latency is dominated by the homomorphic `Conv2D` layers, which dramatically increase runtime. This highlights an important deployment consideration: balancing the security gains of encrypting computationally intensive layers with the associated impact on latency.

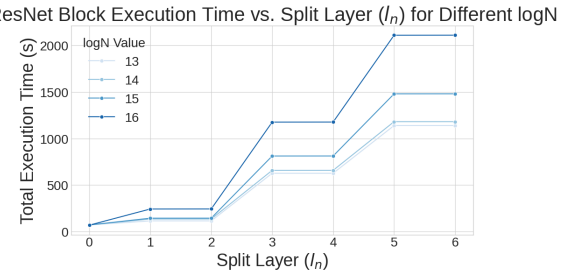


Figure 3: *Total execution time of the ResNet block as a function of the split layer (l_n) for various CKKS ring sizes.*

Model	l_n=1	l_n=3	l_n=5	l_n=7	l_n=9
MLP	2.6%	3.2%	3.4%	NA	NA
LeNet	2.7%	3.9%	4.7%	4.7%	4.3%
PTB-XL CNN	2.7%	2.7%	4.1%	NA	NA
ResNet	1.7%	NA	NA	NA	NA

Table 6: *Estimator validation across split indices (l_n) for 40-core configuration. Estimation error is defined as $\frac{|\hat{t}-t_{\text{real}}|}{t_{\text{real}}}$, representing the relative deviation between estimated and measured runtimes. For models without benchmark data at certain l_n values, the same deviation trend was propagated from available points. NA stands for "not applicable" where there is no split index existing for that network.*

5.2.4 Evaluation of Our Estimator. We conducted experiments to validate the efficacy of our estimator function (see Section 4.4). The primary success criterion is its ability to accurately predict real training times for both server and client components. Such accurate estimations are crucial for the estimator to effectively optimize training configurations under various user-defined constraints.

Table 6 summarize the estimator's quantitative fidelity across architectures, showing uniformly low deviation between predicted and actual runtimes. Overall, all models achieve less than 4.7% error, confirming that the estimator accurately captures the scaling behavior across split configurations. We note that for ResNet, we implement a single block, nevertheless, the estimator can still be applied to this block. In Figure 4 in Appendix I, we also illustrate how the estimator's predictions follow the real MLP execution profile with near-linear correspondence across split indices. Finally, Figure 5in Appendix I confirms that the estimator generalizes well to convolutional structures, preserving estimator accuracy even under deeper layer partitions.

5.3 Comparison with Prior Work

A quantitative comparison of CURE with existing privacy-preserving SL solutions is challenging for several reasons: (i) CURE employs a novel approach, and, to the best of our knowledge, no prior work has explored the inverted traditional SL setting of CURE; (ii) we were unable to identify any fully-encrypted SL work that provides publicly available implementations for reproducibility. Consequently, we compare our framework with HE-based training methods by configuring all layers as encrypted in CURE.

We contrast CURE with fully encrypted training [41, 49] and SL with HE on PTB-XL [31], and for large models, we base our comparison on the work of Lee et al. [37], which explores deep learning with HE. We note that Khan et al. [31] perform only a single convolutional layer under HE on the server side. Table 7 presents our comparison results. We observe that CURE sets a new, higher standard for performance in private ML. For fully encrypted training on MNIST under the same core budget, CURE at 30 cores completes in 5.46 hours per epoch—approximately 122× faster than Nandakumar et al.'s 30-core result (~667 hours/epoch) and about 7× faster than the Glyph framework's 38.4 hours/epoch result on a comparable 24-core setup.To attribute these gains correctly, we benchmarked the underlying schemes (CKKS vs. BGV) on Lattigo. We found that CKKS yields at most a 1.11× speedup for specific primitives (e.g., ciphertext-plaintext multiplication) and otherwise performs comparably to or slower than BGV and they are close on average in terms of corresponding operations latency. This confirms that CURE's substantial advantage derives from architectural optimization rather than the cryptosystem, holding firm even under worst-case primitive assumptions. In the SL setting with matched cores, CURE at $l_n = 1$ on 30 cores completes in 4.56 hours/epoch, giving a ~146× speedup over Nandakumar et al.'s 30-core baseline (~667 hours/epoch). On PTB-XL with 6-core server settings, CURE at $l_n = 1$ completes in 5.28 hours/epoch versus 20.15 hours/epoch for Khan et al., a ~3.8× speedup; with all layers encrypted on 6 cores, CURE is slightly faster than Khan et al. (18.93 vs. 20.15 hours/epoch). This speedup is especially crucial as it underscores the practical advantage of the SL with HE paradigm; it enables clients to securely offload the bulk of their data storage and computational workload to a powerful server, all while maintaining confidentiality and achieving feasible training times. Finally, compared to Lee et al.'s state-of-the-art ResNet-20 inference [37], our extrapolated results achieve 22.35 minutes per image—an ~8× speedup.

5.4 Discussion

Several key insights emerge from our results: (i) the impact on model accuracy is minimal. Our findings in Section 5.2.1 confirm that the combination of degree-3 polynomial activation approximations and the inherent noise from the CKKS scheme results in a negligible accuracy degradation of less than 1% across all tested models. This is a critical outcome, as it validates that the strong privacy guarantees offered by CURE do not come at the cost of the model's utility, making it a viable solution for real-world applications, (ii) The performance of CURE is dictated by a clear set of trade-offs (see Section 5.2.2). Latency is primarily determined by the number of encrypted layers, hardware capacity, and cryptographic parameters. Core-count scalability analysis shows near-linear speedup, making the framework well-suited to modern multi-core servers. However, this efficiency must be contextualized by the overhead of HE itself. As shown in our *split layer index* and log N experiments, both deepening the encrypted portion of the model and increasing the security parameters lead to substantial increases in runtime. This underscores the necessity of our estimator tool, which is designed to navigate these trade-offs and identify an optimal configuration for a given set of constraints. (iii) Our analysis of large models shows that convolutional layers are the main performance bottleneck. In ResNet, high channel depth makes Conv2D layers vastly

	Dataset	Method	Model	Time (hours)	Notes
Training	MNIST	Nandakumar et al. [49]	**MLP**	~667 hours/epoch	BGV on 30 cores
	MNIST	Glyph [41]	**MLP**	38.4 hours/epoch	BGV on 24 cores
	MNIST	CURE ($l_n = 1$)	**MLP**	**5.72 hours/epoch**	CKKS on 24 cores (SL)
	MNIST	CURE ($l_n = 1$)	**MLP**	**4.56 hours/epoch**	CKKS on 30 cores (SL)
	MNIST	CURE ($l_n = 1$)	**MLP**	**3.17 hours/epoch**	CKKS on 40 cores (SL)
	MNIST	CURE ($l_n = 5$, fully encrypted)	**MLP**	**6.83 hours/epoch**	CKKS on 24 cores (SL)
	MNIST	CURE ($l_n = 5$, fully encrypted)	**MLP**	**5.46 hours/epoch**	CKKS on 30 cores (SL)
	MNIST	CURE ($l_n = 5$, fully encrypted)	**MLP**	**3.79 hours/epoch**	CKKS on 40 cores
Training	PTB-XL	Khan et al. [31]	**PTB-XL CNN**	20.15 hours/epoch	CKKS on 6 cores (SL)
	PTB-XL	CURE ($l_n = 1$)	**PTB-XL CNN**	**5.28 hours/epoch**	CKKS on 6 cores (SL)
	PTB-XL	CURE ($l_n = 5$, fully encrypted)	**PTB-XL CNN**	**18.93 hours/epoch**	CKKS on 6 cores
	PTB-XL	CURE ($l_n = 1$)	**PTB-XL CNN**	**0.733 hours/epoch**	CKKS on 40 cores (SL)
	PTB-XL	CURE ($l_n = 5$, fully encrypted)	**PTB-XL CNN**	**2.63 hours/epoch**	CKKS on 40 cores
Inference	CIFAR-10	Lee et al. [37]	**ResNet**	2.94 hours/image	CKKS on 112 cores
	CIFAR-10	CURE	**ResNet**	**22.35 minutes/image**	CKKS on 40 cores (SL)

Table 7: *Comparison of Training Times: Prior Work vs. CURE. All listed times are for fully encrypted training unless specified as Split Learning (SL). For fair comparison, fully encrypted CURE runtimes are obtained by configuring all layers as encrypted server layers.*

more expensive under HE than layers in simpler models like LeNet, underscoring the importance of carefully choosing the split point. (iv) Benchmark results show that CURE delivers a substantial performance leap. As Table 7 illustrates, it is up to 210× faster than foundational HE training frameworks and consistently outperforms recent specialized competitors. The advantage is especially clear on the PTB-XL dataset, where CURE 's SL implementation is orders of magnitude faster than Khan et al. Overall, these results establish CURE as competitive and practical.

Finally, we note that while recent works such as BOLT [56], BumbleBee [26], and NEXUS [92] have paved the way for secure inference on transformers—where a service provider holds *static* model weights and only the user inputs are encrypted—our contribution is orthogonal: we enable server-side training with *dynamic encrypted* model parameters with *backpropagation* under SL. Rather than hard-coding specific neural networks, we implement layers as composable modules that interoperate under any split index and dimension configuration. Our packing strategies are designed to remain efficient even as layer sets or split points vary during training. In contrast, [26, 56, 92] propose optimizations tailored primarily to secure inference of transformer architectures, relying on techniques such as attention-specific packing layouts, low-rotation matrix multiplication schemes, and efficient approximations of softmax, assuming fixed weights and forward-only execution. In particular, the matrix-multiplication methods in [26, 56, 92] are tailored for fixed-weight, inference-only settings and cannot be directly applied to CURE's encrypted-training scenario: BOLT's optimizations rely on the server being able to freely access and pre-arrange its model parameters in plaintext form to minimize rotation and packing costs during computation; BumbleBee's Oblivious Linear Transformation (OLT) operates over secret-shared inputs in a two-party computation framework which contradicts with CURE's system/threat model; and NEXUS's optimization is built on having fixed parameters that allow pre-computation and reuse of encrypted operations across multiple inputs. Since CURE maintains and updates (during backpropagation) all server-side parameters as ciphertexts under the client's key during training, none of these methods are compatible with its homomorphic, single-key, SL design.

Future Work. While this work focuses on the single-client setting, an interesting future direction is to extend CURE to multi-party SL, enabling multiple clients to jointly train a model under semi-honest or malicious settings. However, we note that this necessitates distinct cryptographic solutions, such as distributed key management and the aggregation of encrypted updates. Multi-party environments introduce non-trivial challenges—such as data integrity and peer-wise privacy—that are not present in the single-client architecture of CURE. One solution would be to integrate multiparty HE [48] for secure key generation and aggregation, eliminating reliance on a trusted party as in prior private federated learning systems [71]. Moreover, unlike the single-client setting—where all updates originate from a single honest party—the multi-client scenario introduces risks of malicious or inconsistent client behavior, such as submitting malformed or adversarial gradients. Therefore, robust and verifiable mechanisms, including encrypted gradient validation and zero-knowledge proofs, are critical to ensure correctness and integrity. The estimator may also be generalized to manage dynamic cut-layer placement, heterogeneous computational capacities, and communication scheduling across clients. Finally, formalizing the multi-party security model and assessing CURE in cross-silo settings (e.g., multi-hospital collaborations) would further validate its scalability, robustness, and privacy guarantees.

6 Conclusion

We introduced CURE, an efficient privacy-preserving split-learning framework for a single-client, single-server setting that encrypts only server-side parameters. This design securely offloads storage and computation while preserving label confidentiality and, optionally, data privacy. By relying on our proposed packing strategies, CURE further enhances performance and outperforms fully encrypted training methods while achieving accuracy levels on par with both baseline and fully encrypted approaches.

Acknowledgments

This work is supported by TÜBİTAK (the Scientific and Technological Research Council of Türkiye) projects 123E462 and 124N941. The authors used ChatGPT-4o [54] to enhance the manuscript by shortening sentences, correcting typos, and improving grammar.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 308–318. https://doi.org/10.1145/2976749.2978318

[2] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. 2024. Zero-Knowledge Proofs of Training for Deep Neural Networks. Cryptology ePrint Archive, Paper 2024/162. https://eprint.iacr.org/2024/162

[3] Sharif Abuadbba, Kyuyeon Kim, Minki Kim, Chandra Thapa, Seyit A. Camtepe, Yansong Gao, Hyoungshick Kim, and Surya Nepal. 2020. Can We Use Split Learning on 1D CNN Models for Privacy Preserving Training?. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS '20)*. Association for Computing Machinery, 305–318. https://doi.org/10.1145/3320269.3384740

[4] Omar S. Ads, Mayar M. Alfares, and Mohammed A.-M. Salem. 2021. Multi-limb Split Learning for Tumor Classification on Vertically Distributed Data. In *Proceedings of the 2021 Tenth International Conference on Intelligent Computing and Information Systems (ICICIS '21)*. IEEE, 88–92. https://doi.org/10.1109/ICICIS52592.2021.9694163

[5] Martin Albrecht et al. 2018. *Homomorphic Encryption Security Standard*. Technical Report. HomomorphicEncryption.org.

[6] Corinne G. Allaart, Björn Keyser, Henri Bal, and Aart Van Halteren. 2022. Vertical Split Learning – an exploration of predictive performance in medical and other use cases. In *Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN '22)*. IEEE, 1–8. https://doi.org/10.1109/IJCNN55064.2022.9891964

[7] Maya Bakshi and Mark Last. 2020. CryptoRNN – Privacy-Preserving Recurrent Neural Networks Using Homomorphic Encryption. In *Proceedings of the 4th International Symposium on Cyber Security Cryptography and Machine Learning (CSCML '20) (Lecture Notes in Computer Science, Vol. 12161)*. Springer, 245–253. https://doi.org/10.1007/978-3-030-49785-9_16

[8] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. 1982. Horizontal data partitioning in database design. In *ACM SIGMOD*. 128–136.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT*. 409–437.

[10] Silvia De Rubeis, Xin He, Arthur P Goldberg, Christopher S Poultney, Kaitlin Samocha, A Ercument Cicek, Yan Kou, Li Liu, Menachem Fromer, Susan Walker, et al. 2014. Synaptic, transcriptional and chromatin genes disrupted in autism. *Nature* 515, 7526 (2014), 209–215.

[11] Ege Erdoğan, Alptekin Küpçü, and A Ercüment Çiçek. 2022. Splitguard: Detecting and mitigating training-hijacking attacks in split learning. In *ACM WPES*. 125–137.

[12] Ege Erdoğan, Unat Tekşen, M Salih Çeliktenyıldız, Alptekin Küpçü, and A Ercüment Çiçek. 2024. SplitOut: Out-of-the-Box Training-Hijacking Detection in Split Learning via Outlier Detection. In *CANS*. Springer, 118–142.

[13] Ege Erdoğan, Alptekin Küpçü, and A. Ercüment Çiçek. 2022. UnSplit: Data-Oblivious Model Inversion, Model Stealing, and Label Inference Attacks against Split Learning. In *WPES*. ACM, 115–124.

[14] David Froelicher, Juan R Troncoso-Pastoriza, Apostolos Pyrgelis, Sinem Sav, Joao Sa Sousa, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2021. Scalable Privacy-Preserving Distributed Learning. *PoPETs* 2 (2021), 323–347.

[15] Jiayun Fu, Xiaojing Ma, Bin B. Zhu, Pingyi Hu, Ruixin Zhao, Yaru Jia, Peng Xu, Hai Jin, and Dongmei Zhang. 2023. Focusing on Pinocchio's Nose: A Gradients Scrutinizer to Thwart Split-Learning Hijacking Attacks Using Intrinsic Attributes. In *NDSS*.

[16] Grzegorz Gawron and Philip Stubbings. 2022. Feature Space Hijacking Attacks against Differentially Private Split Learning. In *PPAI*.

[17] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*. 465–482.

[18] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

[20] Google. 2024. Google Colaboratory. https://colab.research.google.com/. Accessed April 18, 2024.

[21] Otkrist Gupta and Ramesh Raskar. 2018. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications* 116 (08 2018), 1–8. https://doi.org/10.1016/j.jnca.2018.05.003

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[23] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca Wright. 2018. Privacy-preserving Machine Learning as a Service. *PoPETS* (2018).

[24] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. 2017. Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning. In *ACM CCS*.

[25] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. 2018. Distributed learning without distress: Privacy-preserving empirical risk minimization. In *NIPS*.

[26] Wen jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and Wenguang Chen. 2025. BumbleBee: Secure Two-party Inference Framework for Large Transformers. In *Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss-paper/bumblebee-secure-two-party-inference-framework-for-large-transformers/

[27] Praveen Joshi, Chandra Thapa, Seyit Camtepe, Mohammed Hasanuzzaman, Ted Scully, and Haithem Afli. 2022. Performance and Information Leakage in Splitfed Learning and Multi-Head Split Learning in Healthcare Data and Beyond. *Methods and Protocols* 5, 4 (2022), 60.

[28] Ferhat Karakoç, Alptekin Küpçü, and Melek Önen. 2024. Fault Tolerant and Malicious Secure Federated Learning. In *CANS*. Springer, 73–95.

[29] Tanveer Khan, Khoa Nguyen, and Antonis Michalas. 2023. A More Secure Split: Enhancing the Security of Privacy-Preserving Split Learning. In *Secure IT Systems*. Springer, 307–329.

[30] Tanveer Khan, Khoa Nguyen, and Antonis Michalas. 2023. Split Ways: Privacy-Preserving Training of Encrypted Data Using Split Learning. arXiv:2301.08778 [cs.CR] https://arxiv.org/abs/2301.08778

[31] Tanveer Khan, Khoa Nguyen, Antonis Michalas, and Alexandros Bakas. 2023. Love or Hate? Share or Split? Privacy-Preserving Training Using Split Learning and Homomorphic Encryption . In *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–7. https://doi.org/10.1109/PST58708.2023.10320153

[32] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated Optimization: Distributed Machine Learning for On-Device Intelligence. *arXiv:1610.02527* (2016).

[33] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2017. Federated Learning: Strategies for Improving Communication Efficiency. *arXiv:1610.05492* (2017).

[34] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

[35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[36] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

[37] Joon-Woo Lee, Hyungchul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. *IEEE Access* 10 (2022).

[38] Zhuohang Li, Chao Yan, Xinmeng Zhang, Gharib Gharibi, Zhijun Yin, Xiaoqian Jiang, and Bradley A Malin. 2023. Split Learning for Distributed Collaborative Training of Deep Learning Models in Health Informatics. In *Annual Symposium Proceedings*. AMIA, 1047–1056.

[39] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM CCS*.

[40] Junlin Liu, Xinchen Lyu, Qimei Cui, and Xiaofeng Tao. 2024. Similarity-based Label Inference Attack against Training and Inference of Split Learning. *TIFS* (2024).

[41] Qian Lou, Bo Feng, Geoffrey C. Fox, and Lei Jiang. 2020. Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data. In *NeurIPS*, Vol. 33. 9193–9202.

[42] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *EUROCRYPT*. 1–23.

[43] Umer Majeed, Sheikh Salman Hassan, and Choong Seon Hong. 2021. Vanilla Split Learning for Transportation Mode Detection using Diverse Smartphone Sensors. In *KCC*. KIISE, 23–25.

[44] Yunlong Mao, Zexi Xin, Zhenyu Li, Jue Hong, Qingyou Yang, and Sheng Zhong. 2023. Secure Split Learning Against Property Inference, Data Reconstruction, and Feature Space Hijacking Attacks. In *ESORICS*. Springer, 23–43.

[45] H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. 2018. Learning Differentially Private Recurrent Language Models. *CoRR* abs/1710.06963 (2018). arXiv:1710.06963 [cs.LG]

[46] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. 2019. Exploiting Unintended Feature Leakage in Collaborative Learning. In *SP*. IEEE, 691–706.

[47] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *SP*. IEEE, 19–38.

[48] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2021. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies* 2021, 4 (2021), 291–311.
[49] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. 2019. Towards Deep Neural Network Training on Encrypted Data. *CVPR* (2019).
[50] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. 1984. Vertical partitioning algorithms for database design. *ACM TODS* 9, 4 (1984).
[51] Shamkant B Navathe and Mingyoung Ra. 1989. Vertical partitioning for database design: a graphical algorithm. In *ACM SIGMOD*. 440–450.
[52] Khoa Nguyen, Tanveer Khan, and Antonis Michalas. 2023. Split Without a Leak: Reducing Privacy Leakage in Split Learning. *arXiv:2308.15783* (2023).
[53] Utku Norman and A Ercument Cicek. 2019. ST-Steiner: a spatio-temporal gene discovery algorithm. *Bioinformatics* 35, 18 (2019), 3433–3440.
[54] OpenAI. 2024. GPT-4o: Multimodal AI Model. https://openai.com/research/gpt-4o Accessed: February 28, 2025.
[55] Kerem Özfatura, Emre Özfatura, Alptekin Küpçü, and Deniz Gündüz. 2024. Byzantines can also learn from history: Fall of centered clipping in federated learning. *IEEE TIFS* 19 (2024), 2010–2022.
[56] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2024. BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*. https://encrypto.de/papers/PZMZS24.pdf
[57] Mathias PM Parisot, Balazs Pejo, and Dayana Spagnuelo. 2021. Property Inference Attacks on Convolutional Neural Networks: Influence and Implications of Target Model's Complexity. In *SECRYPT*. INSTICC, 715–721.
[58] Dario Pasquini, Giuseppe Ateniese, and Massimo Bernaschi. 2021. Unleashing the Tiger: Inference Attacks on Split Learning. In *SIGSAC CCS*. ACM, 17 pages.
[59] George-Liviu Pereteanu, Amir Alansary, and Jonathan Passerat-Palmbach. 2022. Split HE: Fast Secure Inference Combining Split Learning and Homomorphic Encryption. In *PPAI*.
[60] Maarten G Poirot. 2020. *Split Learning in Health Care: Multi-center Deep Learning without sharing patient data.* Master's thesis. University of Twente.
[61] Maarten G Poirot, Praneeth Vepakomma, Ken Chang, Jayashree Kalpathy-Cramer, Rajiv Gupta, and Ramesh Raskar. 2019. Split Learning for collaborative deep learning in healthcare. *arXiv:1912.12115* (2019).
[62] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press.
[63] Minyoung Ra. 1993. Horizontal Partitioning for Distributed Database Design: A Graph-Based Approach.. In *Australian Database Conference*. 101–120.
[64] Md Atiqur Rahman, Tanzila Rahman, Robert Laganière, Noman Mohammed, and Yang Wang. 2018. Membership inference attack against differentially private deep learning model. *Trans. Data Priv.* 11, 1 (2018), 61–79.
[65] Devin Reich, Ariel Todoki, Rafael Dowsley, Martine De Cock, and Anderson C. A. Nascimento. 2021. Privacy-Preserving Classification of Personal Text Messages with Secure Multi-Party Computation: An Application to Hate-Speech Detection. *CoRR* abs:1906.02325 (2021).
[66] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[67] Sharma Sagar and Keke Chen. 2021. Confidential machine learning on untrusted platforms: a survey. *Cybersecurity* 4, 1 (2021), 1–25. https://doi.org/10.1186/s42400-021-00092-8
[68] Sambit Satpathy, Osamah Khalaf, Dhirendra Kumar Shukla, Mohit Chowdhary, and Sameer Algburi. 2024. A collective review of Terahertz technology integrated with a newly proposed split learningbased algorithm for healthcare system. *IJCDS* 15, 1 (2024), 1–9.
[69] F Kyle Satterstrom, Jack A Kosmicki, Jiebiao Wang, Michael S Breen, Silvia De Rubeis, Joon-Yong An, Minshi Peng, Ryan Collins, Jakob Grove, Lambertus Klei, et al. 2020. Large-scale exome sequencing study implicates both developmental and functional changes in the neurobiology of autism. *Cell* 180, 3 (2020), 568–584.
[70] Sinem Sav, Abdulrahman Diaa, Apostolos Pyrgelis, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2021. Privacy-Preserving Federated Recurrent Neural Networks. *PoPETs* 4 (2021), 500–521.
[71] Sinem Sav, Apostolos Pyrgelis, Juan R. Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. 2021. POSEIDON: Privacy-Preserving Federated Neural Network Learning. In *NDSS*.
[72] Aqsa Shabbir, Halil İbrahim Kanpak, Alptekin Küpçü, and Sinem Sav. 2025. A Taxonomy of Attacks and Defenses in Split Learning. *arXiv:2505.05872* (2025).
[73] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving deep learning. In *ACM CCS*.
[74] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE SP*. 3–18.
[75] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. 2015. Big Data: Astronomical or Genomical? *PLoS Biology* 13, 7 (2015).
[76] Chandra Thapa, Pathum Chamikara Mahawaga Arachchige, Seyit Camtepe, and Lichao Sun. 2022. Splitfed: When federated learning meets split learning. In *AAAI*.
[77] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, and Seyit A Camtepe. 2021. Advancements of Federated Learning Towards Privacy Preservation: From Federated Learning to Split Learning. *Federated Learning Systems: Towards Next-Generation AI* (2021), 79–109.
[78] Tom Titcombe, Adam J Hall, Pavlos Papadopoulos, and Daniele Romanini. 2021. Practical Defences Against Model Inversion Attacks for Split Neural Networks. In *ICLR Workshop on DPML*.
[79] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A hybrid approach to privacy-preserving federated learning. In *ACM AISec*.
[80] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2019. Split learning for health: Distributed deep learning without sharing raw patient data. In *ICLR AI for social good workshop*.
[81] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *PoPETS* (2019).
[82] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2020. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETS* (2020).
[83] Patrick Wagner, Nils Strodthoff, Eduardo Bietti, Tobias Schaeffter, Xu Zhu, and Radoslaw Durichen. 2020. PTB-XL, a large publicly available electrocardiography dataset. *Scientific Data* (2020). https://doi.org/10.1038/s41597-020-0495-6
[84] Wenxiao Wang, Tianhao Wang, Lun Wang, Nanqing Luo, Pan Zhou, Dawn Song, and Ruoxi Jia. 2021. DPlis: Boosting Utility of Differentially Private Deep Learning via Randomized Smoothing. *PoPETS* (2021).
[85] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H. Yang, Farhad Farokhi, Shi Jin, Tony Q. S. Quek, and H. Vincent Poor. 2020. Federated Learning With Differential Privacy: Algorithms and Performance Analysis. *IEEE TIFS* 15 (2020), 3454–3469. https://doi.org/10.1109/TIFS.2020.2988575
[86] Nan Wu, Farhad Farokhi, David Smith, and Mohamed Ali Kaafar. 2019. The Value of Collaboration in Convex Machine Learning with Differential Privacy. *CoRR* abs/1906.09679 (2019). arXiv:1906.09679 [cs.CR]
[87] Xin Yang, Jiankai Sun, Yuanshun Yao, Junyuan Xie, and Chong Wang. 2022. Differentially private label protection in split learning. *arXiv:2203.02073* (2022).
[88] Fangchao Yu, Lina Wang, Bo Zeng, Kai Zhao, Zhi Pang, and Tian Wu. 2023. How to backdoor split learning. *Neural Networks* 168 (2023), 326–336.
[89] Fangchao Yu, Lina Wang, Bo Zeng, Kai Zhao, Tian Wu, and Zhi Pang. 2024. SIA: A sustainable inference attack framework in split learning. *Neural Networks* 171 (2024), 396–409.
[90] Fangchao Yu, Bo Zeng, Kai Zhao, Zhi Pang, and Lina Wang. 2024. Chronic Poisoning: Backdoor Attack against Split Learning. In *AAAI*.
[91] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. 2020. BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning. In *USENIX Security*. Article 33, 14 pages.
[92] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. 2025. Secure Transformer Inference Made Non-interactive. In *Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2025.230868
[93] Qiulin Zhang, Zhuqing Jiang, Qishuo Lu, Jia'nan Han, Zhengxin Zeng, Shang-Hua Gao, and Aidong Men. 2020. Split to Be Slim: An Overlooked Redundancy in Vanilla Convolution. In *IJCAI*.
[94] Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2019. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *IEEE S&P*.
[95] Huafei Zhu, Rick Siow Mong Goh, and Wee-Keong Ng. 2020. Privacy-Preserving Weighted Federated Learning Within the Secret Sharing Framework. *IEEE Access* 8 (2020), 198275–198284. https://doi.org/10.1109/ACCESS.2020.3034602
[96] Ligeng Zhu, Zhijian Liu, and Song Han. 2019. Deep leakage from gradients. In *NIPS*. 14774–14784.

A CURE's Initialization Algorithm

We provide the initialization of CURE that involves the generation and encryption of weight matrices described in Section 4.1 in Algorithm 2.

B Illustration of Packing Principles

The following example outlines how we apply packing principle through a toy example. Here d_{ij} represents the entries of an arbitrary matrix, β is a scalar multiplicative, and underscores represent garbage values. Consider D a 3×3 data matrix:

Algorithm 2 Initialization

function INITIALIZATION
  if client **then**
    $L_c \leftarrow [l_{n+1}, \ldots, l_{n+k}]$ ▷ Client-side layers
    $W_c \leftarrow \text{GenRandomWeights}(L_c)$
    $(\text{PK}_c, \text{SK}_c) \leftarrow \text{KeyGen}(1^\lambda)$
    Send PK_c to server
  else if server **then**
    $L_s \leftarrow [l_1, l_2, \ldots, l_n]$ ▷ Server-side layers
    $W_s \leftarrow \text{GenRandomWeights}(L_s)$
    $\mathbf{W}_s \leftarrow \text{Enc}_{\text{PK}_c}(W_s)$
  end if
end function

$$D = \begin{bmatrix} \mathbf{d_{00}} & \mathbf{d_{01}} & \mathbf{d_{02}} \\ \mathbf{d_{10}} & \mathbf{d_{11}} & \mathbf{d_{12}} \\ \mathbf{d_{20}} & \mathbf{d_{21}} & \mathbf{d_{22}} \end{bmatrix}$$

Instead of performing a scalar multiplication in HE separately for each entry of the matrix as:

$$\begin{matrix} \beta \cdot [\mathbf{d_{00}} & \_ & \_ & \_ & \_ & \cdots & \_] \\ \beta \cdot [\mathbf{d_{01}} & \_ & \_ & \_ & \_ & \cdots & \_] \\ & & \vdots & & & & \\ \beta \cdot [\mathbf{d_{22}} & \_ & \_ & \_ & \_ & \cdots & \_] \end{matrix}$$

We pack and augment the data properly with respect to the operation to utilize the computational resources more efficiently. The packing is done as follows:

$$\beta \cdot [\mathbf{d_{00}} \quad \mathbf{d_{01}} \quad \ldots \quad \mathbf{d_{22}} \quad \_ \quad \ldots \quad \_]$$

By restructuring the data in this manner, we fill the RLWE vector with meaningful data and pad it with zeros when necessary, as described in Section 4.3.3.

C One-Level Operations in Matrix Computation

In one-level batch multiplication, we pack weight matrices as batches of columns, enabling matrix-vector multiplication. Optimizing the second layer's matrix initialization is crucial since packing efficiency depends on the dimensions of the first two layers and the number of slots in the RLWE vector. This process is complex and influenced by the dataset, HE scheme parameters, and security and computational limits. While packing the encrypted weight matrix for batch multiplication can be costly (as discussed in Section 5.3), it maximizes efficiency. The latency from batch multiplication is compensated by the performance boost from better weight matrix packing.

In contrast, for one-level scalar multiplication, there is no need for such preprocessing on the components of the network except for the encoding and encryption of the elements. However, since one-level scalar multiplication cannot utilize packing as efficiently as one-level batch multiplication, there is a possibility of higher demand for memory and computation in some cases. Therefore, it is important to carefully decide which one-level operation to use, considering the trade-offs.

Here, we provide a toy example to demonstrate the necessary operations to calculate the multiplication of a 4×4 matrix with a 4-dimensional vector:

$$\begin{bmatrix} \mathbf{v_{11}} & \mathbf{v_{12}} & \mathbf{v_{13}} & \mathbf{v_{14}} \\ \mathbf{v_{21}} & \mathbf{v_{22}} & \mathbf{v_{23}} & \mathbf{v_{24}} \\ \mathbf{v_{31}} & \mathbf{v_{32}} & \mathbf{v_{33}} & \mathbf{v_{34}} \\ \mathbf{v_{41}} & \mathbf{v_{42}} & \mathbf{v_{43}} & \mathbf{v_{44}} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} \mathbf{v_{11}}w_1 + \mathbf{v_{12}}w_2 + \mathbf{v_{13}}w_3 + \mathbf{v_{14}}w_4 \\ \mathbf{v_{21}}w_1 + \mathbf{v_{22}}w_2 + \mathbf{v_{23}}w_3 + \mathbf{v_{24}}w_4 \\ \mathbf{v_{31}}w_1 + \mathbf{v_{32}}w_2 + \mathbf{v_{33}}w_3 + \mathbf{v_{34}}w_4 \\ \mathbf{v_{41}}w_1 + \mathbf{v_{42}}w_2 + \mathbf{v_{43}}w_3 + \mathbf{v_{44}}w_4 \end{bmatrix}$$

Our one-level batch multiplication is represented as:

$$= \begin{bmatrix} w_1 \\ w_1 \\ w_1 \\ w_1 \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{11}} \\ \mathbf{v_{21}} \\ \mathbf{v_{31}} \\ \mathbf{v_{41}} \end{bmatrix} + \begin{bmatrix} w_2 \\ w_2 \\ w_2 \\ w_2 \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{12}} \\ \mathbf{v_{22}} \\ \mathbf{v_{32}} \\ \mathbf{v_{42}} \end{bmatrix} + \begin{bmatrix} w_3 \\ w_3 \\ w_3 \\ w_3 \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{13}} \\ \mathbf{v_{23}} \\ \mathbf{v_{33}} \\ \mathbf{v_{43}} \end{bmatrix} + \begin{bmatrix} w_4 \\ w_4 \\ w_4 \\ w_4 \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{14}} \\ \mathbf{v_{24}} \\ \mathbf{v_{34}} \\ \mathbf{v_{44}} \end{bmatrix}$$

And our one-level scalar multiplication is represented as:

$$= w_1 \otimes \begin{bmatrix} \mathbf{v_{11}} \\ \mathbf{v_{21}} \\ \mathbf{v_{31}} \\ \mathbf{v_{41}} \end{bmatrix} + w_2 \otimes \begin{bmatrix} \mathbf{v_{12}} \\ \mathbf{v_{22}} \\ \mathbf{v_{32}} \\ \mathbf{v_{42}} \end{bmatrix} + w_3 \otimes \begin{bmatrix} \mathbf{v_{13}} \\ \mathbf{v_{23}} \\ \mathbf{v_{33}} \\ \mathbf{v_{43}} \end{bmatrix} + w_4 \otimes \begin{bmatrix} \mathbf{v_{14}} \\ \mathbf{v_{24}} \\ \mathbf{v_{34}} \\ \mathbf{v_{44}} \end{bmatrix}$$

Notice that the column-wise multiplication can be done in a batch or scalar fashion. In other words, we can write the multiplication of a column as scalar multiplication of w_i or element-wise vector multiplication with repeated elements of w_i as an RLWE-packed vector. When the ciphertext batch size is larger than the column size (e.g., two columns fit in one RLWE vector), we can utilize packing more efficiently for the one-level batch multiplication operation, as follows:

$$\begin{bmatrix} w_1 \\ w_1 \\ w_1 \\ w_1 \\ w_2 \\ w_2 \\ w_2 \\ w_2 \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{11}} \\ \mathbf{v_{21}} \\ \mathbf{v_{31}} \\ \mathbf{v_{41}} \\ \mathbf{v_{12}} \\ \mathbf{v_{22}} \\ \mathbf{v_{32}} \\ \mathbf{v_{42}} \end{bmatrix} + \begin{bmatrix} w_3 \\ w_3 \\ w_3 \\ w_3 \\ w_4 \\ w_4 \\ w_4 \\ w_4 \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{13}} \\ \mathbf{v_{23}} \\ \mathbf{v_{33}} \\ \mathbf{v_{43}} \\ \mathbf{v_{14}} \\ \mathbf{v_{24}} \\ \mathbf{v_{34}} \\ \mathbf{v_{44}} \end{bmatrix} = \begin{bmatrix} \mathbf{v_{11}} * \mathbf{w_1} + \mathbf{v_{13}} * \mathbf{w_3} \\ \mathbf{v_{21}} * \mathbf{w_1} + \mathbf{v_{23}} * \mathbf{w_3} \\ \mathbf{v_{31}} * \mathbf{w_1} + \mathbf{v_{33}} * \mathbf{w_3} \\ \mathbf{v_{41}} * \mathbf{w_1} + \mathbf{v_{43}} * \mathbf{w_3} \\ \mathbf{v_{12}} * \mathbf{w_2} + \mathbf{v_{14}} * \mathbf{w_4} \\ \mathbf{v_{22}} * \mathbf{w_2} + \mathbf{v_{24}} * \mathbf{w_4} \\ \mathbf{v_{32}} * \mathbf{w_2} + \mathbf{v_{34}} * \mathbf{w_4} \\ \mathbf{v_{42}} * \mathbf{w_2} + \mathbf{v_{44}} * \mathbf{w_4} \end{bmatrix}$$

Upon receiving the ciphertext, the client decrypts it and calculates:

$$\begin{bmatrix} v_{11}w_1 + v_{13}w_3 \\ v_{21}w_1 + v_{23}w_3 \\ v_{31}w_1 + v_{33}w_3 \\ v_{41}w_1 + v_{43}w_3 \end{bmatrix} + \begin{bmatrix} v_{12}w_2 + v_{14}w_4 \\ v_{22}w_2 + v_{24}w_4 \\ v_{32}w_2 + v_{34}w_4 \\ v_{42}w_2 + v_{44}w_4 \end{bmatrix} = \begin{bmatrix} v_{11}w_1 + v_{12}w_2 + v_{13}w_3 + v_{14}w_4 \\ v_{21}w_1 + v_{22}w_2 + v_{23}w_3 + v_{24}w_4 \\ v_{31}w_1 + v_{32}w_2 + v_{33}w_3 + v_{34}w_4 \\ v_{41}w_1 + v_{42}w_2 + v_{43}w_3 + v_{44}w_4 \end{bmatrix}$$

D Illustration n-encrypted Layer Operations in Matrix Multiplication

Here, we provide a toy example that illustrates the packing and matrix multiplication scheme for n-encrypted layer networks described in Section 4.3.3. Consider a matrix 3×4 matrix B with the following entries:

$$\mathbf{B} = \begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} & w_{04} \\ w_{10} & w_{11} & w_{12} & w_{13} & w_{14} \\ w_{20} & w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}$$

First, we pad our matrix to achieve an efficient homomorphic dot product with optimized rotations.

$$\begin{bmatrix} w_{00} & w_{01} & & w_{04} \\ w_{10} & w_{11} & \ldots & w_{14} \\ w_{20} & w_{21} & & w_{24} \\ 0 & 0 & & 0 \end{bmatrix}$$

By concatenating and marking the entries, we achieve the placement of columns to the RLWE vectors for the dot product.

$$\begin{bmatrix} \underline{w_{00}} \\ w_{10} \\ w_{20} \\ 0 \\ \underline{w_{01}} \\ w_{11} \\ w_{21} \\ 0 \end{bmatrix} \begin{bmatrix} \underline{w_{02}} \\ w_{12} \\ w_{22} \\ 0 \\ \underline{w_{03}} \\ w_{13} \\ w_{23} \\ 0 \end{bmatrix} \begin{bmatrix} \underline{w_{04}} \\ w_{14} \\ w_{24} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

After preparing the second vector, we process the rows of the first matrix by padding them to the nearest power of two and repeating the rows as necessary. We then calculate the homomorphic dot product for each column and extract the previously marked data. It is important to note that this marking operation serves as an abstraction for explanatory purposes.

$$\mathbf{A} = \begin{bmatrix} v_{00} & v_{11} & v_{02} \\ v_{10} & v_{11} & v_{12} \\ v_{20} & v_{21} & v_{22} \end{bmatrix}$$

To prepare the first row for the homomorphic dot product calculation, we arrange the elements as $[v_{00}, v_{10}, v_{20}, 0, v_{00}, v_{00}, v_{00}, 0, ...]$ as a column vector. Next, we perform element-wise multiplication of this vector with the columns obtained from matrix $\mathbf{B}$. Subsequently, we rotate the resulting vector by powers of two until we cover all slots. After each rotation, we perform an element-wise addition with the previously accumulated vector. This process ultimately yields the dot product of the initial matrices' row-column pairs homomorphically, enabling efficient computation of the matrix-matrix product. Importantly, all operations from this stage onward are executed homomorphically.

$$\begin{bmatrix} \mathbf{w_{00}} \\ \mathbf{w_{10}} \\ \mathbf{w_{20}} \\ \mathbf{0} \\ \mathbf{w_{01}} \\ \mathbf{w_{11}} \\ \mathbf{w_{21}} \\ \mathbf{0} \end{bmatrix} \odot \begin{bmatrix} \mathbf{v_{00}} \\ \mathbf{v_{10}} \\ \mathbf{v_{20}} \\ \mathbf{0} \\ \mathbf{v_{01}} \\ \mathbf{v_{10}} \\ \mathbf{v_{21}} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{v_{00} * w_{00}} \\ \mathbf{v_{10} * w_{10}} \\ \mathbf{v_{20} * w_{20}} \\ \mathbf{0 * 0} \\ \mathbf{v_{01} * w_{02}} \\ \mathbf{v_{10} * w_{11}} \\ \mathbf{v_{21} * w_{21}} \\ \mathbf{0 * 0} \end{bmatrix}$$

Next, we rotate and add the result to itself $\lceil \log n \rceil - 1$ times, ultimately achieving the desired outcome:

$$\begin{bmatrix} \mathbf{v_{00} * w_{00}} \\ \mathbf{v_{10} * w_{10}} \\ \mathbf{v_{20} * w_{20}} \\ \mathbf{0 * 0} \\ \mathbf{v_{01} * w_{02}} \\ \mathbf{v_{10} * w_{11}} \\ \mathbf{v_{21} * w_{21}} \\ \mathbf{0 * 0} \end{bmatrix} + \begin{bmatrix} \mathbf{v_{10} * w_{10}} \\ \mathbf{v_{20} * w_{20}} \\ \mathbf{0 * 0} \\ \mathbf{v_{01} * w_{02}} \\ \mathbf{v_{10} * w_{11}} \\ \mathbf{v_{21} * w_{21}} \\ \mathbf{0 * 0} \\ \mathbf{v_{00} * w_{00}} \end{bmatrix} = \begin{bmatrix} \mathbf{v_{00} * w_{00} + v_{10} * w_{10}} \\ \mathbf{v_{10} * w_{10} + v_{20} * w_{20}} \\ \mathbf{v_{20} * w_{20} + 0 * 0} \\ \mathbf{0 * 0 + v_{01} * w_{02}} \\ \mathbf{v_{01} * w_{02} + v_{10} * w_{11}} \\ \mathbf{v_{10} * w_{11} + v_{21} * w_{21}} \\ \mathbf{v_{21} * w_{21} + 0 * 0} \\ \mathbf{0 * 0 + v_{00} * w_{00}} \end{bmatrix}$$

Rotation of this vector twice and addition will result in:

$$\begin{bmatrix} \mathbf{v_{00} * w_{00} + v_{10} * w_{10}} \\ \mathbf{v_{10} * w_{10} + v_{20} * w_{20}} \\ \mathbf{v_{20} * w_{20} + 0 * 0} \\ \mathbf{0 * 0 + v_{01} * w_{02}} \\ \mathbf{v_{01} * w_{02} + v_{10} * w_{11}} \\ \mathbf{v_{10} * w_{11} + v_{21} * w_{21}} \\ \mathbf{v_{21} * w_{21} + 0 * 0} \\ \mathbf{0 * 0 + v_{00} * w_{00}} \end{bmatrix} + \begin{bmatrix} \mathbf{v_{20} * w_{20} + 0 * 0} \\ \mathbf{0 * 0 + v_{01} * w_{02}} \\ \mathbf{v_{01} * w_{02} + v_{10} * w_{11}} \\ \mathbf{v_{10} * w_{11} + v_{21} * w_{21}} \\ \mathbf{v_{21} * w_{21} + 0 * 0} \\ \mathbf{0 * 0 + v_{00} * w_{00}} \\ \mathbf{v_{00} * w_{00} + v_{10} * w_{10}} \\ \mathbf{v_{10} * w_{10} + v_{20} * w_{20}} \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{v_{00} * w_{00} + v_{10} * w_{10} + v_{20} * w_{20} + 0 * 0} \\ \mathbf{v_{10} * w_{10} + v_{20} * w_{20} + 0 * 0 + v_{01} * w_{02}} \\ \mathbf{v_{20} * w_{20} + 0 * 0 + v_{01} * w_{02} + v_{10} * w_{11}} \\ \mathbf{0 * 0 + v_{01} * w_{02} + v_{10} * w_{11} + v_{21} * w_{21}} \\ \mathbf{v_{01} * w_{02} + v_{10} * w_{11} + v_{21} * w_{21} + 0 * 0} \\ \mathbf{v_{10} * w_{11} + v_{21} * w_{21} + 0 * 0 + v_{00} * w_{00}} \\ \mathbf{v_{21} * w_{21} + 0 * 0 + v_{00} * w_{00} + v_{10} * w_{10}} \\ \mathbf{0 * 0 + v_{00} * w_{00} + v_{10} * w_{10} + v_{20} * w_{20}} \end{bmatrix}$$

Note that the first and fifth entries of the final vector represent the desired homomorphic and efficient results of the first row's first column and the first row's second column of the resulting matrix. By proceeding with this process for each column and row, we can obtain the matrix-matrix product. For matrices with columns that do not fit into a single RLWE vector, we perform additional summation operations on the final result, based on the initial slot-to-column length ratio calculation.

E Approximated Activation Functions

Due to the fully encrypted nature of the server layers, for n encrypted server layers where $n > 1$, the activation functions of $n - 1$ layers should also be executed under encryption. However, non-linear activation functions cannot be directly applied under encryption; only polynomial functions can be used. To address this limitation, we rely on well-known approximation techniques such as the Chebyshev interpolation method [62] or the minimax approximation [75] to approximate the non-linear activation functions as polynomials. This technique is also employed by numerous privacy-preserving machine learning works [7, 14, 18, 23, 39, 70, 71]. It is important to note that using higher-degree polynomials may result in better approximations and thus better accuracy. However, higher-degree polynomials also lead to more HE multiplications, resulting in noise accumulation and potentially necessitating bootstrapping as each multiplication consumes one ciphertext level. For a degree d polynomial, the scheme consumes $log_2(d + 1)$ levels. This results in increased computational complexity and can lead to higher training latency. Therefore, careful consideration is required when selecting the function and the degree of the polynomial used for the approximation.

Backward Pass for Activation Functions. Activation functions, being non-parametric, do not require weight updates. During the backward pass, the gradient with respect to the activation function's input is computed. This is achieved by multiplying the incoming gradient (with respect to the activation's output) by the derivative of the activation function itself. For our approximated polynomial activations, we directly use the derivative of the polynomial approximation to compute this gradient.

F Detailed Implementation Details

Every NN layer used in experiments has *both* a plaintext reference implementation and an HE counterpart. This lets us (i) execute CURE with end-to-end HE encryption, meaning all computations—from forward pass to backpropagation—are performed on encrypted data without any intermediate decryptions except client operations and (ii) quantify per-layer divergence by comparing HE outputs to their plaintext mirrors at each split layer index (l_n). For accuracy testing, we also provide a Python/PyTorch simulation harness that mirrors our Go operators by using the same fixed-point quantization and degree-3 polynomial approximation of ReLU, and injecting layer-wise noise calibrated from our measurements and Lattigo's CKKS noise growth reports. Unless stated otherwise, each experiment is repeated at least three times, and we report the mean.

G Detailed Information on Datasets

We detail the datasets used in the evaluation of CURE here. We employ: (i) the hand-written digits (MNIST) dataset [36] with 60,000 images of 28×28 pixels and 10 labels, (ii) the PTB XL dataset [83] (PTB-XL) with 21,837 clinical 12-lead ECG records, 10-second recordings with 100 Hz sampling rate, annotated with up to 71 different diagnostic classes, and (iii) the CIFAR-10 dataset [34] with 60,000 color images of 32×32 pixels, categorized into 10 classes.

H Experimental Methodology

Time latency. All time-latency experiments run true end-to-end HE pipelines. For every layer, we retain both plaintext and HE implementations and log the per-layer HE↔plaintext deviation to monitor drift. Each layer processes the actual output of its predecessor, thereby capturing real training dynamics such as ciphertext scaling, rescaling, and relinearization.

Accuracy via calibrated noise-injection. To estimate accuracy under HE, we finetune training for additional epochs in PyTorch while injecting per-layer noise that emulates our measured HE deviation and Lattigo-reported CKKS noise growth. This is performed on Colab T4 [20] for fast iteration. We ensure the activation approximation and quantization match the Go/HE pipeline.

Scalability experiments. We measured scalability with respect to two primary factors: the number of CPU cores and the cryptographic context parameters, specifically by varying the ring dimension, $\log N$. For these experiments, we report the total wall-clock time for end-to-end model passes. Additionally, to provide a more granular analysis, we also benchmarked representative cryptographic primitives (e.g., $ct \times pt$, $ct \times ct$, and rotations).

I Supplementary Figures and Tables

We present here the supplementary figures and tables here that could not be included in the main manuscript due to space constraints. Discussions on these figures and tables can be found in the relevant subsections of Section 5.

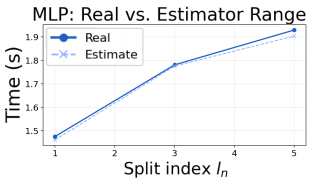


Figure 4: *MLP real and estimator times*

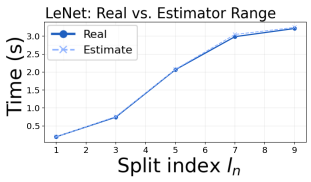


Figure 5: *LeNet real and estimator times*