CURE: Privacy-Preserving Split Learning Done Right

Halil Ibrahim Kanpak*, Aqsa Shabbir†, Esra Genç†, Alptekin Küpçü*‡, Sinem Sav†‡
*Koç University, Istanbul, Turkey
†Bilkent University, Ankara, Turkey
‡**Correspondence:** akupcu@ku.edu.tr and sinem.sav@cs.bilkent.edu.tr

***Abstract*—Training deep neural networks often requires large-scale datasets, necessitating storage and processing on cloud servers due to computational constraints. The procedures must follow strict privacy regulations in domains like healthcare. Split Learning (SL), a framework that divides model layers between client(s) and server(s), is widely adopted for distributed model training. While Split Learning 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 that encrypts *only* the server side of the model and optionally the data. CURE enables secure SL while substantially improving communication and parallelization through advanced packing techniques. We propose two packing schemes that consume one HE level for one-layer networks and then generalize our solutions to n-layer neural networks. We demonstrate that CURE can achieve similar accuracy to plaintext SL, while being $16\times$ 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. Our open-source implementation is available at https://github.com/CRYPTO-KU/CURE-Privacy-Preserving-Split-Learning.

***Keywords*—Split learning, homomorphic encryption, outsourced learning, privacy-preserving machine learning**

I. 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, and data is often distributed across multiple parties, necessitating collaborative ML approaches.

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 [1], [2] and split learning [3]. 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 sensitive information about the training data or the labels [4]–[13]. Researchers focused on developing new defense strategies to mitigate this leakage in FL using differential privacy (DP) [14]–[18], homomorphic encryption (HE) [19]–[22], or secure multiparty computation (MPC) [23]–[30].

To mitigate various adversarial attacks in SL, several works rely on DP [31]–[35]. However, DP-based learning requires low privacy budgets, resulting in lower accuracy [36]. Some works employ outlier detection [37], [38]. Another line of research employs HE for encrypted training or inference [39]–[42]. While Pereteanu et al. integrate HE for *inference* tasks in SL [39], most efforts to improve privacy focus only on U-shaped split learning, where the neural network is divided into three segments: the client handles the initial and final layers, while the server processes the intermediate layers [40]–[42]. This setting assumes that the client holds its own data and labels, necessitating sufficient storage and computational capacity on the client side. To the best of our knowledge, there is no prior work that focuses on privacy-preserving training in the SL setting where the network is divided into two parts.

In this work, we focus on privacy-preserving training within a split learning framework, 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 [43]–[45].

To ensure data and label confidentiality, we propose CURE, a novel system that leverages homomorphic encryption (HE) to encrypt model parameters on the *server side only*. This allows the server to work with an encrypted model, while the client –the data and/or 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. This setup not only protects data and label confidentiality but also reduces communication and computational overhead through plaintext training on the client side, making it especially valuable in fields like healthcare and genomics where data privacy and efficiency are paramount.

Our contributions can be summarized as follows: (i) We

introduce a novel system, CURE, for privacy-preserving split learning that ensures the confidentiality of labels and (optionally) the data using homomorphic encryption. (ii) We propose two packing schemes that ensure efficient computation under different settings, for one-level server operations. (iii) We generalize our packing to support encrypted multi-layer server models. (iv) We develop and implement a novel estimator that, for the first time, optimally determines the best neural network split for SL, ensuring efficient use of CURE based on the server's and client's available resources. (v) For the first time, we evaluate the performance of ResNet in an encrypted SL setting. (vi) We evaluate our approach through extensive experiments and analysis, demonstrating superior performance compared to state-of-the-art methods with training times improved by up to $16\times$.

II. Related Work

Split Learning (SL) [3] 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 machine learning model into sections, each handled by a different party. It gained recognition with SplitNN [46], which enables health organizations to collaboratively train models without sharing sensitive data. This method is more resource-efficient than approaches like federated learning [1], [14], [47]. It enables various configurations tailored to practical health settings [48]–[52], including the *vanilla configuration* [46], [53], [54], where the network is divided at a specific cut layer. Additionally, *vertical split learning* [55], [56] involves different parties holding different features of the dataset [57], [58]. In contrast, *horizontal split learning* involves different dataset samples held by various parties to be processed independently [59], [60] to enhance query performance, which refers to how quickly and effectively the system can distribute data across multiple parties by localizing data access.

Soon after SL gained recognition in the ML field, several attacks were developed to attain the raw data that is being processed through the SL pipeline, unveiling the critical privacy leakage issue to a spectrum of adversarial attacks, including inference attacks [4], [13], hijacking attacks [6], backdoor attacks [7], [61], feature distribution attacks [9], data reconstruction attacks [62], and property inference attack [12]. Thus, SL requires integration of further privacy mechanisms to mitigate the aforementioned attacks.

Privacy-Preserving Split Learning: To enhance privacy and eliminate various adversarial attacks, several works integrate a mechanism called differential privacy (DP) to SL [31]–[35]. DP adds noise to the data or intermediate values shared between client and server, thereby reducing the accuracy of the results. Our work distinguishes itself from DP-based approaches by integrating HE – a form of encryption that allows mathematical operations to be performed on encrypted data without the need to decrypt it – with split learning to eliminate the privacy vs. accuracy tradeoff.

Other works [39]–[42] integrate HE mechanism into the deep learning pipeline within SL. By encrypting data or model parameters, any information obtained by attackers is rendered useless without the decryption key, enhancing security and privacy. For example, Pereteanu et al. [39] 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 and the server computation done in encrypted form. In contrast, our approach focuses on the 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 data and label 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 [40]–[42]. For this, the authors developed a U-shaped split 1D CNN model, with the initial and final layers are done by the client and the intermediate layers are done by the server. The model starts with a public segment during training, then splits into two branches—one for public data and one for private data—resembling a "U". These branches reconverge during the inference phase, completing the U-shaped structure [41]. This design enables clients to protect the privacy of their ground truth labels by not sharing them with the server. In [40], 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 [42], the authors extended their experiments and introduced batch encryption to optimize memory usage and computational performance when handling encrypted data. These techniques further reduce privacy leakage by encrypting activation maps in a setup where the client controls both data and labels. Nguyen et al. [63] refined the approach of [40] by reducing their privacy leakage and improving communication efficiency utilizing CKKS HE scheme (see Section III-C). In contrast, our framework optimizes the training process by *applying HE exclusively to the server-side model parameters* within an inverted SL setup, where the server processes the initial layers under encryption and the subsequent layers are performed by the client in plaintext. Our method reduces privacy risks associated with intermediate outputs and gradients exchanged between the client and server, resulting in more efficient training than traditional techniques. This ensures minimal storage and computation on the client side, effectively balancing privacy and efficiency.

In summary, 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.

III. Building Blocks

A. Neural Networks

A neural network (NN) is a model composed of interconnected layers of nodes [64]. During training, the network adjusts connection weights between neurons to minimize the loss between predictions and actual outcomes, using optimization algorithms like gradient descent. 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 to facilitate the discussion on backpropagation below.

After the forward pass, backpropagation [65] 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. After computing the gradients, the parameters (weights and biases) are updated using the formula below to minimize the loss function: $W_l \leftarrow W_l - \alpha \frac{\partial J}{\partial W_l}$ and $B_l \leftarrow B_l - \alpha \frac{\partial J}{\partial B_l}$, where α is the learning rate.

B. *Split Learning*

Split Learning (SL) [3], [46], [51], [54], [55] is a technique designed to enhance data 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 then computes the gradient of the loss with respect to $\hat{Y}$ 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 IV-A.

SL offers several advantages. First, it enhances privacy by keeping raw data on the client side and sharing only abstract intermediate representations, minimizing the risk of sensitive information exposure. 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 [4]–[10], [12], [13], [61], [62], [66] on SL continue to pose a threat. To address this, we enhance split learning by integrating HE, which we detail below.

C. *Homomorphic Encryption*

Homomorphic encryption (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) HE scheme [67] for efficient floating-point arithmetic.

CKKS Scheme is a leveled HE scheme based on the ring learning with errors (RLWE) problem [68]. 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 (see Section IV-C1 for details). The scheme also has effective noise management strategies, where the noise refers to the small error added to ciphertexts to ensure security, making it practical for tasks such as SL. 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 $\mathsf{Bootstrap}$ operation that is detailed below. 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.

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.

IV. 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.

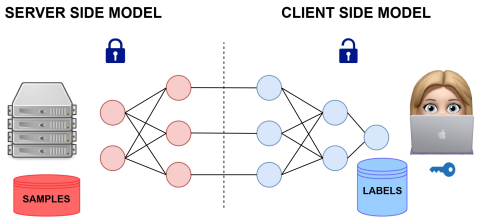

Figure 1: CURE *System's Model: the Inverted Split Learning setup. The server (left) processes data samples* $\mathbf{X}$ *through* n *layers, while the client (right) holds the labels* Y *and processes* k *layers of a neural network. Server-side weights* $\mathbf{W}_s$ *are encrypted, while client-side weights* W_c *remain unencrypted.*

Threat Model: We consider a semi-honest model without collusion between the server and the client. This is a plausible assumption regarding our motivation that the client is the initial owner of the data samples and labels, yet outsources the storage and parts of the computing. Our threat model suggests that the server might passively try to infer sensitive information, i.e. samples and/or labels, from the exchanged messages and the model parameters, but will adhere to the protocol rules and not actively inject malicious inputs. We aim to eliminate various types of input extraction attacks or membership inference attacks [4], [5], [69], [70]. These attacks typically exploit the intermediate computations and gradients shared during the training process to reconstruct sensitive data. By encrypting the server-side computations using HE, we ensure that the server cannot access any meaningful information from the encrypted data, thereby mitigating these attack vectors. We present a formal security definition and proof in Section IV-E.

A. *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 III-C), 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 illustrate the overview of CURE in Figure 1. Throughout the paper, we denote server-side and client-side parameters with a subscript of 's' and 'c', respectively.

The server is responsible for storing the data samples ($\mathbf{X}$) and performing forward pass computations ($f_s(\cdot)$) 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 using its parameters, 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 server gradients ($\mathbf{gw}_s$) back to the server. The server updates its parameters in an encrypted fashion provided by the client. This process ensures that (optionally the data X and) the labels Y remain confidential, adhering to the objectives of our inverted SL framework.

Algorithm 1 CURE Training Phase

Server has (encrypted) data $\mathbf{X}_{[1,2,...,m]}$, encrypted initial weight matrix $\mathbf{W}_s$, and the client's public key PK_c.
Client has labels Y.
1: **for** epoch $=1 \rightarrow e$ **do**
2: **for** $\mathbf{X} \in \mathbf{X}_{[1,2,...,m]}$ **do**
3: **Server performs encrypted forward pass:**
4: $\mathbf{O}_n \leftarrow \mathsf{Eval}^{f_s}_{\mathsf{PK}_c}(\mathbf{W}_s, \mathbf{X})$
5: Send $\mathbf{O}_n$ to client
6: **Client works on plaintext:**
7: $O_n \leftarrow \mathrm{Dec}_{\mathsf{SK}_c}(\mathbf{O}_n)$
8: $\hat{Y} \leftarrow f_c(W_c, O_n)$
9: $J \leftarrow \mathrm{Loss}(\hat{Y}, Y)$
10: Compute gradients g_{W_s}, g_{W_c}
11: $W_c \leftarrow \mathrm{Update}_c(W_c, g_{W_c})$
12: $\mathbf{gw}_s \leftarrow \mathrm{Enc}_{\mathsf{PK}_c}(g_{W_s})$
13: Send $\mathbf{gw}_s$ to server
14: **Server performs encrypted backpropagation:**
15: $\mathbf{W}_s \leftarrow \mathsf{Eval}^{\mathrm{Update}_s}_{\mathsf{PK}_c}(\mathbf{W}_s, \mathbf{gw}_s)$
16: **end for**
17: **end for**

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 used effectively makes the encrypted values indistinguishable from random. CURE is designed to ensure that all interactions and computations are conducted securely, leveraging HE to maintain data and/or label privacy throughout the ML process. This approach not only protects sensitive information but also allows for scalable and efficient distributed/outsourced learning, accommodating scenarios where participants have different levels of computational power and data sensitivity.

B. CURE*'s Design:*

1) Initialization: This phase of CURE, as detailed in Algorithm 2 in Supplementary Material VII-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 $[l_1, l_2, ..., l_n]$ and L_c represents the client-side layers $[l_{n+1}, ..., l_{n+k}]$. 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.

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). The gradients for both client-side (g_{W_c}) and server-side (g_{W_s}) are calculated (Line 10). 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 $\mathbf{g}_{\mathbf{W}_s}$, and sends the result to the server (Line 13). Finally, upon receiving $\mathbf{g}_{\mathbf{W}_s}$, 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 done on plaintext (Lines 7 to 11) is called $UpdPlain$ as a shorthand.

C. Homomorphic Operations

In this section, we outline how CURE leverages the CKKS HE scheme and introduce our cryptographic optimizations. We apply various optimization techniques, balancing security and practicality, such as packing, enhanced one-level operations $(n = 1)$, and minimizing resource-intensive tasks. First, we summarize the CKKS scheme's packing capability, then explain our one-level operations, and generalize our approach to encrypted execution across n server layers. Finally, we briefly discuss the bootstrapping operation to refresh ciphertexts.

1) Packing: Packing is a widely used optimization in HE-based applications, enabling multiple plaintext values to be combined into a single ciphertext. This facilitates parallel computations using SIMD operations, greatly enhancing efficiency by reducing the number of ciphertexts to manage. Consequently, choosing the right packing scheme enhances both the time and memory efficiency of HE operations. For a ring size of N, CKKS allows for packing $N/2$ values to plaintext/ciphertext slots (see Section III-C). This enables simultaneous operations on $N/2$ values through SIMD operations. For this, we identify similarities among the operations performed on data and encode (pack) similarly processed data within the same vector. We present a toy example in the Supplementary Material VII-B to demonstrate the fundamental advantage of packing.

2) One-Level Operations: Here, we explain CURE's one-level operations, i.e., operations that consume one level before decryption on the client side (before Line 7, Algorithm 1). These operations offer significant advantages by consuming only one level. They generate less noise, enhancing accuracy both empirically and theoretically. With data processed homomorphically only once, noise accumulation is avoided, eliminating the need for bootstrapping and further reducing computation time.

In CURE, we employ one-level operations when splitting the network from the second layer, allowing the server to handle only the first layer with encrypted weights. This approach offers several key advantages: (i) It reduces data transfer, requiring only the client's error calculations and the server's product results to be exchanged, and (ii) it maintains one-level operations throughout training, avoiding the need for bootstrapping. Therefore, we recommend splitting from the first layer. However, CURE is a versatile solution that supports splitting at any layer, as described in the next subsection. Below, we outline our one-level plaintext-ciphertext multiplications. Although ciphertext-ciphertext operations are also one-level, we focus on plaintext-ciphertext multiplication for simplicity.

Batch multiplication primarily involves element-wise multiplication of RLWE vectors, denoted as $\odot$, between plaintext and ciphertext elements. In contrast, scalar multiplication, denoted as $\otimes$, computes the product of each plaintext element with each component of the ciphertext vector individually with scalar multiplication, obtaining the result by summing the vectors obtained from these scalar products. Let v and w be the arbitrary vectors with w being an encrypted ciphertext and v being a plaintext. Pairwise element batch multiplication can be represented as:

$$[v_0, v_1 ...] \odot [\mathbf{w_0}, \mathbf{w_1} ...] = [\mathbf{v_0 w_0}, \mathbf{v_1 w_1} ...]$$

Scalar multiplication can be represented as:

$$v \otimes [\mathbf{w_0}, \mathbf{w_1} ...] = [\mathbf{v w_0}, \mathbf{v w_1} ...]$$

While scalar multiplication is approximately 2.7 times faster in our experiments, batch multiplication leverages packing more effectively, resulting in better throughput. We propose the one-level batch and one-level scalar methods, incorporating packing in both. We provide the details of one-level batch and scalar multiplication in Supplementary Material VII-C. Note that both scalar and batch methods utilize packing. In both cases, each column of the weight matrices is encoded using batch encoding. In the one-level scalar method, a single column is stored per ciphertext, while in the one-level batch method, multiple columns are packed into the same ciphertext, improving packing efficiency as previously discussed.

Our proposed methods differ from each other and require modified implementations for different ratios of $\frac{N/2}{|l_2|}$ where $N/2$ is the number of slots and $|l_2|$ is the size of the second layer considering the weight matrix obtained by the first two layers. If the second layer is large enough to benefit from improved packing utilization and the efficiency of scalar multiplication of RLWE vectors compared to batch multiplication, the one-level scalar approach becomes preferable. Conversely, when the size of the second layer is small, the one-level batch approach is more advantageous. More explicit decision thresholds are given in Section IV-D.

3) Execution of n-encrypted layer networks and matrix-matrix operations: Here, we briefly explain CURE with multiple encrypted server layers. In an edge case, CURE allows for the execution of all layers of a network under encryption on the server side, except for the last layer (to hide the labels from the server), when the client has extremely low computational power. It is important to note that in such an SL setup, while the overall computational demand increases, it accommodates the client's computational limitations, ensuring training proceeds despite the client's capability. Overall, CURE empowers users to choose where to split, i.e., the number of encrypted layers, optimizing the balance between security, latency, and computational resources.

In settings with multiple encrypted layers, unlike one-level operations, several computational challenges emerge due to the need for encrypted matrix-matrix multiplications on the server side. This leads to noise accumulation and the need for bootstrapping (see Section IV-C4), encrypted execution of activation functions (Supplementary Material VII-E), increased computational demand, and potentially lower expected accuracy. Therefore, we introduce additional optimizations to efficiently implement CURE in settings with n-encrypted layers.

First, we employ log-scaling operations to compute the inner products of vectors during matrix-matrix multiplications. This method involves summing vector elements by shifting them as powers of 2 to reduce the multiplicative depth in HE operations. This approach effectively reduces computational overhead and noise induced by HE operations. Additionally, for vector addition, multiplication, and scalar multiplication, we use packing strategies to enhance the efficiency of vector inner product computations. Packing optimizes memory usage by consolidating data and reduces the number of separate computational steps, thereby accelerating overall processing speed.

Our packing method for matrix-matrix multiplication involves two main steps. First, we determine how the columns of the second matrix will be placed on the RLWE interface for computation. We start by padding the columns of the second matrix with zeros to the nearest larger power of two. After padding, we concatenate the columns until they fill one RLWE interface vector. If a column is longer than the slot sizes, we repeat the padding and division process until each segment fits into one RLWE interface vector. Once the columns are packed, we define the number of "division steps" as $\frac{N/2}{|\mathbf{B}|}$ where $|\mathbf{B}|$ is the size of the column matrix $\mathbf{B}$. We mark the positions on the RLWE vectors in increments of this step size. For long columns that do not fit into a single RLWE vector, we calculate this quotient to ensure the correct summation of dot products for those column-row pairs. We provide the details of this process in Supplementary Material VII-D with a toy example. With this optimized implementation, we can efficiently calculate the product of two matrices. This allows us to delegate more layers to the server, enhancing ciphertext-ciphertext operations up to the last layer. When the training phase reaches the server's last layer, the network is treated as a single-layer encrypted network, and a one-level operation is performed in the final layer.

We note here that for n encrypted server layers where $n > 1$, the activation functions of $n-1$ layers should be executed under encryption. Since HE does not support non-polynomial functions, we approximate the activation functions to polynomials. The details of these approximations can be found in Supplementary Material VII-E.

4) 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, $\mathsf{Bootstrap}(c)$ function (see Section III-C) must be executed to refresh the ciphertext level, enabling continued operations on the ciphertext. 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 $(log_2(d+1))n+n>L$. Note that in practice, it is possible to use different degrees of approximation for different server layers regarding activation functions (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 depth of the multiplicative circuit. Bootstrapping is triggered when the operation depth, combined with the number of server-side layers, exceeds the multiplicative depth. This ensures that the server-side encrypted computations can proceed without corruption, maintaining the overall efficiency and security of the training process.

D. 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 ($\mathcal{O}_c$) 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 the experiments in Figures 6 and 7, in Supplementary Material VII-I. Finally, we introduce CURE's complexity analysis in Table I to facilitate the estimator function and its implementation.

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.

Additionally, the server calculates the cumulative depth of homomorphic operations up to and including the current layer. This depth is crucial because when δ_{current} exceeds the maximum permissible depth δ_{max}, bootstrapping is triggered,

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
Server Forward Pass	$T_r \cdot \left[\sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right) \times \log(\lvert \bar{l}_{i+1} \rvert) \right]$	$\lvert c \rvert \cdot \sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right)$	$n-1+\sum_{i=1}^{n-1} \log(\bar{d})$
Client Forward Pass	$\sum_{i=n}^{m} \text{dot}(\lvert l_i \rvert \times \lvert l_{i+1} \rvert, \lvert l_{i+1} \rvert \times \lvert l_{i+2} \rvert)$	$\sum_{i=n}^{m} \lvert l_i \rvert$	0
Server Back-propagation	$T_r \cdot \left[\sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right) \times \log(\lvert \bar{l}_{i+1} \rvert) \right]$	$\lvert c \rvert \cdot \sum_{i=0}^{n} \left(\frac{\lvert \bar{l}_i \rvert \times \lvert \bar{l}_{i+1} \rvert}{N/2} \right)$	$n-1+\sum_{i=1}^{n-1} \left(\log(\bar{d})-1 \right)$
Client Back-propagation	$\sum_{i=n}^{m} \text{dot}(\lvert l_i \rvert \times \lvert l_{i+1} \rvert, \lvert l_{i+1} \rvert \times \lvert l_{i+2} \rvert)$	$\sum_{i=n}^{m} \lvert l_i \rvert$	0

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

incurring additional computational cost. The cumulative depth $\delta_{\text{current}}^{(i)}$ for each server layer i is calculated as:

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

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 side 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 pass and back-propagation during training, as well as bootstrapping if needed.

$$T_{\text{server}} = 2|X| \times T_r \cdot \left[\sum_{i=0}^{n} \left(\frac{|\bar{l}_i| \times |\bar{l}_{i+1}|}{N/2} \right) \times \log(|\bar{l}_{i+1}|) \right] + \max_i \{ \frac{|l_i| \times |l_{i+1}|}{N/2} \times |c| \} \tag{4}$$

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

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

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, above calculations enable us to quantify the primary client-side constraints: computational load and memory usage. Client estimator provides its estimation as T_{client}.

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

3) Data Transfer and Communication Overhead: Another crucial factor considered by the advisor function is the communication overhead between the server and the client, which is determined by the number of ciphertexts that must be transferred. The amount of data transferred from the server to the client per epoch can be calculated as:

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

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 under given parameter set. By minimizing $|l_n|$, our advisor function can effectively reduce the number of ciphertexts transferred, thereby optimizing communication time and improving overall efficiency.

4) Optimal Partitioning Strategy: The advisor function integrates computational and memory constraints with cryptographic parameters on both the server and client sides to determine the optimal network split index. It first calculates the maximum feasible split index on the server, accounting for rotations and memory limits, then compares it to the client's minimum feasible index, ensuring a balance between computational efficiency and memory usage.

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 identifies the partition that minimizes overall training time while meeting user-defined 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 I, the advisor function offers a theoretically sound method for optimizing the partitioning of NN layers in CURE's split learning setting. Additionally, our advisor function implementation provides feedback indicating which constraints—server time, client time, or communication time—exceed the predetermined thresholds based on these calculations.

E. Security Definition and Proof

We start by observing that previous solutions combining SL with HE [39]–[42] 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 [71], 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 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.

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 is enough to provide a simulator that simulates indistinguishable outputs and view for the adversary, given the adversary's inputs. In our case, the server's inputs include the model description 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 as depicted in Figure 1, and our solution in Algorithm 1, 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 that 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 $\mathcal{F}_{InvSL}$ ensures that the server learns nothing new out of the training process. It is important to note that, against a malicious server, CURE needs to be combined with verifiable computation techniques [72], which we leave as future work. Additionally, exploring multi-client scenarios and other SL use cases is also deferred to future work.

V. Experimental Evaluation

Here, we experimentally evaluate CURE across various settings. We first describe the experimental setup, then present results on model accuracy and runtime, including scalability analysis. Lastly, we compare our findings with previous work.

A. Experimental Setup

1) Implementation Details: We use the Go programming language for its compiler-based execution and its support for parallel programming, essential for our encrypted experiments. We employ Lattigo[1] cryptographic operations.

We also simulate CURE using the Python Pytorch library, using approximated activation functions, CKKS noise [67] addition and fixed-precision for encrypted layers to expedite

[1]Lattigo v5, Online: https://github.com/tuneinsight/lattigo

Model	Server Layers (**Encrypted**)	Client Layers	Data Amount	Batch Size (b)	Execution Time LAN (m)	Execution Time WAN (m)
Model 1	784×128	32×10	10,000	60	29.216	29.256
Model 1	784×128	32×10	10,000	128	23.821	23.861
Model 2	784×128	$128\times128\times128\times128\times128\times32\times10$	10,000	60	35.703	35.743
Model 2	$784\times128\times128$	$128\times128\times128\times128\times32\times10$	10,000	60	42.995	43.035
Model 2	$784\times128\times128\times128$	$128\times128\times128\times32\times10$	10,000	60	49.923	49.963
Model 2	$784\times128\times128\times128\times128$	$128\times128\times32\times10$	10,000	60	53.727	53.767
Model 2	$784\times128\times128\times128\times128\times128$	$128\times32\times10$	10,000	60	58.202	58.242
Model 2	$784\times128\times128\times128\times128\times128\times128$	32×10	10,000	60	62.608	62.648
Model 3	8192×8192	$1024\times512\times128\times32\times10$	100	60	331.774	334.378
Model 3	$8192\times8192\times1024$	$512\times128\times32\times10$	100	60	342.733	343.058
Model 3	$8192\times8192\times1024\times512$	$128\times32\times10$	100	60	351.760	351.928
Model 3	$8192\times8192\times1024\times512\times128$	32×10	100	60	362.473	362.520
Model 4	16384×16384	$8192\times32\times10$	100	60	1183.345	1188.553
Model 4	16384×16384	$8192\times32\times10$	100	128	925.156	930.364
Model 4	16384×16384	$8192\times32\times10$	100	256	784.802	790.010
Model 4	16384×16384	$8192\times32\times10$	100	1024	615.381	620.589
Model 5	ResNet[:3]	ResNet[3:]	1000	256	4229.85	4231.53
Model 5	ResNet[:8]	ResNet[8:]	1000	256	6605.73	6607.41

TABLE II: CURE*'s runtime for one training epoch (to cover all data) with various NN architectures. Parallelization on a 40-core CPU machine is used. We use the cryptographic parameter **Set 2** and extrapolate WAN results with 1 Mbps bandwidth (seeSection VII-F). [:] indicates a slice of the layers, e.g., ResNet[3:] refers to the layers starting from the fourth layer.*

accuracy experiments (see Section V-B1). We note here that while we evaluate the correctness of our encrypted implementation, we rely on simulations to expedite accuracy tests in Section V-B1, as our main focus is optimizing the HE-based split learning pipeline. All our experiments were repeated twice and the average numbers are provided.

2) Setup: We experiment on an Ubuntu 18.04.6 LTS server with a 40-core Intel Xeon Processor E5-2650 v3 2.3GHz CPU 251GB of RAM for the evaluation of CURE with parallelization. Additionally, we experimented with varying the number of cores utilized during execution to obtain a broader range of results for analysis. We use two set of cryptographic parameters. **Set 1:** We use $N=2^{14}$ as CKKS ring size and logQP=438, which represents the logarithm of the number of moduli in the ring. The scale is 2^{30}. This setting allows us to encrypt vectors of size $N/2=2^{13}$ into one ciphertext.
Set 2: We use $N=2^{13}$ and logQP$=218$. The scale is 2^{30}. This setting allows us to encrypt vectors of size $N/2=2^{12}$. We chose our default parameters to achieve 128-bit security according to the HE standard whitepaper [73]. We also provide information on our network setup in Supplementary Material VII-F

3) Datasets and Model Architectures: We employ 4 datasets in our experiments: (i) Breast Cancer Wisconsin dataset (BCW) [74], (ii) MNIST [75], (iii) the default of credit card clients (CREDIT) dataset [76], (iv) the PTB XL dataset [77] (PTB-XL). We provide detailed information on these datasets in Supplementary Material VII-G. For the scalability evaluations, we use both the MNIST dataset and synthetic data with varying numbers of features and samples. Finally, we employ eight different neural network architectures, referred to as **Model #X**, with details provided in Supplementary Material VII-H.

B. Experimental Results

1) Model Accuracy: Table IV in Supplementary Material VII-I displays the accuracy results on various datasets. For all baselines, we use NN structures of **Models 6, 7,** and **8** and the same learning parameters as CURE. We use the sigmoid for all layers in the network. For the baseline, we use the original sigmoid function, while for the encrypted version, we use the approximated version of the sigmoid. We observe that the accuracy loss between plaintext and CURE is between 0.04% and 0.6% when encryption is simulated. For example, CURE achieves 97.37% training accuracy on the BCW dataset, which is only $\%0.08$ lower than the plaintext learning in terms of accuracy loss. Moreover, it should be noted that these training results are obtained using the same number of epoch iterations for a fair comparison. Therefore, the accuracy loss could be further reduced if the model is trained with a higher number of epochs using CURE.

2) Model Time Latency: We experiment with various NN architectures and various distributions of server and client layers to observe their effect on the training runtime, as detailed in Table II. The table presents the runtime of one epoch of various combinations of server and client layers. The server layers are encrypted. We vary the number of neurons in each layer, e.g., server layers 784×128 indicate an input layer of size 784 followed by a layer of 128 neurons. We use 100 or $10{,}000$ samples with batch sizes of $60{,}128{,}256{,}1024$ to cover a comprehensive range of networks.

Effect of the number of server layers (n). We observe in Table II that with increasing n, the runtime increases. This is due to the computationally demanding nature of HE operations, which are primarily executed on the server. Note that some rows, e.g., rows 3-8, describe the same network with different n ranging from 2 to 7, illustrating how CURE behaves under different network splits and emphasizing that HE operations are the primary determinant of time latency. Here, we aim to demonstrate the effect of different sequences of HE operations discussed in Section IV-C. Considering the first 6 rows for **Model 2** and the subsequent 4 rows for **Model 3** we observe the impact of n on training time latency. Variations in n alter the number of matrix-matrix products, directly impacting training time.

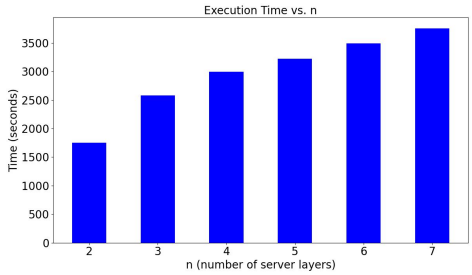

Figure 2: CURE*'s runtime for one epoch on* ***Model 2****, with* $b = 60$ *and* 40 *CPU cores with varying* n*, excluding the bootstrapping time.*

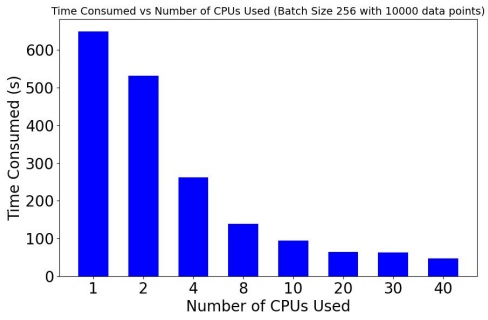

Figure 3: CURE*'s runtime on* ***Model 1*** *for one epoch with 10,000 samples and* $b=256$ *with varying numbers of CPUs.*

We observe that the addition of a third encrypted layer to the server significantly increases time latency for our experimental setting. This is due to the involvement of ciphertext-ciphertext operations, approximated activation functions, and subsequent bootstrapping required for the given cryptographic parameters (with $N=2^{13}$). Adding further layers also increases the training time as expected while reducing the computational and memory load on the client. The additional time induced by adding a layer ranges from $1.0758\times$ to $1.204\times$ of the original, with the highest increase occurring on the third layer of the server, as mentioned. Note that when a different set of cryptographic parameters is used, this observation of the third layer may vary to some other layer. Similarly, for **Model 3**, we observe a similar pattern, with runtime increasing in the range of $1.0263\times$ to $1.330\times$, again with the highest increment occurring on the third layer addition to the server. Similar results can be observed in Figure 2 which displays the runtime of one epoch with **Model 1** when the network is split from the second layer leaving the third and fourth layers to the client, i.e., only the first layer is being encrypted. We use a $b=60$ with parallelization on the server side.

Effect of the batch size (b). The influence of b on training duration is also evident, where larger b generally reduce training times by optimizing the use of computational resources and reducing the overhead associated with managing many smaller batches. This finding highlights the CURE's capability to handle larger batches more efficiently. We can observe in the first 2 rows (**Model 1**) and last 4 rows (**Model 4**) of Table II that increasing layer sizes improves the time latency of training when incrementing b. Doubling the b leads to a reduction in time latency ranging from $0.848\times$ to $0.718\times$. Although we use a $b=60$ for **Model 1** to ensure a fair comparison with [78] (see Section V-C), we found that varying b yield better results in terms of training time latency. b is also highly related to training time latency performance. With higher concurrency utilization, we observe that increasing b improves performance results in terms of training time latency. Conversely, the impact of performance degradation due to the increased number of CPU cores used diminishes and eventually vanishes as the b increases.

Effect of the neural network (NN) architecture. Moreover, Table II includes results from setups with different complexities of network structures - from simpler models with fewer layers to more complex ones with many layers. The results demonstrate a manageable increase in training times for more complex models, indicating that the CURE framework efficiently handles increased computational demands. We can see that even with large network sizes, CURE achieves practically applicable training times in most cases.

Effect of the number of cores. The model's scalability was further evaluated by varying the number of CPU cores used during training. As shown in Table V in Supplementary Material VII-I the runtime significantly decreases with an increase in the number of CPU cores, demonstrating the model's ability to efficiently leverage parallel computing. The results indicate a substantial reduction in training time, with the time latency advantage ranging from $2.204\times$ to $0.215\times$ as more cores are utilized.

It is important to note that, particularly in smaller networks (i.e., networks with fewer layers that exhibit varying behavior under HE operations in a parallelized environment), increasing the number of cores does not always lead to improved runtime performance. In some instances, performance may even degrade due to race conditions.

Our observations show that CURE is influenced by both the number of cores used and the batch size (b). As illustrated in Table V in Supplementary Material VII-I, batch sizes of $b=60$ and $b=128$ display different trends based on the number of CPUs. This is a crucial factor when determining the appropriate batch size for training, taking into account computational constraints and the nature of the training process. Notably, larger b tend to benefit from a large number of CPUs, while smaller b perform more efficiently with fewer CPUs. This behavior with an increasing number of CPUs used for training a network is not present with larger b. Figure 3 shows the training of **Model 1** with $n=2$, $k=2$, using $10{,}000$ samples and a batch size b of 256 on a LAN setting. We observe that while the improvement in performance is not as pronounced when changing the number of CPUs from 2 to 4 compared to changes from 20 to 30 or 30 to 40, there is no penalty in performance. This suggests that the number of CPUs is less significant when using larger b during training.

Table II also shows that configurations where the server handles a greater computational load benefit from parallelization. By leveraging the server's superior processing power for encrypted data operations, we achieve a more favorable balance of the computational burden compared to client-side processing. Combined with the analysis in Section IV-D, these insights emphasize the importance of architectural decisions in optimizing training and highlight CURE's ability to handle computational complexities in a privacy-focused learning environment.

Effect of the number of samples. We measure CURE's scalability with the increasing number of samples by considering only the homomorphic ciphertext-ciphertext dot product mentioned in Section IV-C. Figure 4 in Supplementary Material VII-I illustrates the impact of increasing the number of samples, demonstrating a linear scalability in runtime.

C. Comparison with Prior Work

Our qualitative analysis focuses on the efficiency and overhead of different frameworks (see Section II). CURE optimizes performance by transferring only encrypted intermediate gradients, utilizing the server's computational power for encrypted tasks. This reduces client-side computation, minimizes homomorphic operations, and lowers communication overhead, leading to improved time efficiency and accuracy.

A quantitative comparison of CURE with existing privacy-preserving split learning (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 that CURE introduces; (ii) we were unable to identify any fully-encrypted SL work that provides publicly available implementations for reproducing experiments under the same system configuration. Consequently, we compare CURE with HE-based training methods and rely on the results reported in the respective papers.

CURE employs a unique training paradigm that restricts encrypted training to predefined server-side layers in the network. Thus, CURE also supports fully encrypted training when the number of server-side encrypted layers n equals the number of layers in the network, leading to a fair comparison. We choose two works of fully encrypted training [78], [79] and one privacy-preserving SL framework [41] for the comparison and provide the results in Table III. The choice of these methods is based on their goal of improving HE-based SL setups through various approaches. It is worth noting that, although prior works conducted their experiments on different hardware, our setup closely mirrors the conditions used in those studies [78], [79], but is less favorable compared to the GPU-enhanced setup utilized in [41]. Therefore, comparing our results to those numbers places us at a disadvantage.

In Glyph [78], a fully encrypted setting requires 134 hours per epoch, while in CURE, the same setting (fully encrypted) takes approximately 8 hours per epoch, making CURE $16\times$ faster in the worst-case scenario. Notably, in the best case, CURE can achieve $64\times$ faster execution for the same task compared to [78] due to enabling the same task by encrypting only parts of the network on the server side with the same privacy guarantees. CURE also has the advantage on a WAN setting compared to [41] achieving $10\times$ faster execution.

CURE demonstrates substantial improvement for the scalability as the size of the second-layer multiplicand in HE dot product increases (see Figure 5 in Supplementary Material VII-I). This improvement is primarily due to the efficiency of our one-level operations, which eliminate the need for rotations, as well as optimizations tailored to different second-layer scales. Additionally, CURE exhibits superior scalability in ciphertext-ciphertext products, largely owing to the versatility of our packing schemes. This scalability is critical, as it enables users to apply CURE to a wider range of networks in split learning setups.

Dataset / Setting	Method	Time Latency (h)
MNIST / LAN	[79]	111.11
	Glyph [78]	134.26
	CURE	**8.32**
	CURE- One Level	**4.87**
PTB XL / LAN	[41]	20.14
	CURE	**34.58**
PTB XL / WAN	[41]	341.37
	CURE	**34.58**

TABLE III: *The comparison with [41], [78], [79] with a fully encrypted network for 1 epoch on MNIST and 10 epochs on PTB-XL dataset. We extrapolate the runtime for 10 epochs on the PTB-XL dataset with a 100 MBps network bandwidth. Due to unavailable code, we extrapolated training latencies based on related papers.*

VI. Discussion

Based on our experimental evaluation, the scalar multiplication method performs consistently well in one-level operations, regardless of the size of the second layer. In contrast, the one-level batch method's performance deteriorates with increasing second layer size. A practical heuristic for estimating packing method performance is given by $\frac{N/2}{|l_2|}$ where $N/2$ is the number of slots and $|l_2|$ is the size of the second layer. If this ratio is less than ~ 2.7, the scalar multiplication method is preferable. It is also important to note that the size of the first layer affects both methods linearly since the matrices are stored column-wise.

Our comparison with the prior work [78] shows that CURE is faster than the state of the art. The proposed one-level batch method is $16\times$ and the one-level scalar method is $2.4\times$ faster than Glyph [78]. Additionally, we have shown that by adjusting batch size during training and the number of CPUs used in execution, users may achieve even better results. CURE also achieves accuracy levels on par with baseline (plaintext) or fully-encrypted approaches.

CURE excels in various tasks due to its innovative approach to privacy-preserving SL. It effectively mitigates reconstruction attacks through encrypted gradients. Our novel one-level operations reduce noise, enhancing the overall accuracy. Moreover, CURE incorporates bootstrapping for various network configurations with higher circuit depths, and our empirical results demonstrate successful implementation and superior performance. CURE also transfers a minimal amount of data per epoch. This efficiency is achieved by only exchanging the server's last layer and the client's first layer gradients per iteration. Consequently, CURE minimizes data transfer by leveraging the NN architecture's tendency to reduce data dimensionality. Additionally, CURE scales well with an increasing number of samples used in training. Quantitative descriptions of these results are discussed in Section V-B2.

We have also demonstrated that CURE is more broadly applicable to generic, n-layer networks. These innovations enable users to allocate more encrypted layers to the server, thereby reducing computational and memory loads on the client side and expanding the feasibility of real-world applications. The applicability of CURE's operations extends beyond resource allocation to different training scenarios. CURE achieves practical results with complex networks.

VII. Conclusion

We presented CURE, a novel and efficient privacy-preserving split learning framework that offers substantial improvements over previous methods. CURE is the first framework that encrypts only the server-side parameters, enabling secure outsourcing of storage and computational tasks from the client while ensuring the confidentiality of labels and optionally data samples. This approach effectively mitigates reconstruction attacks. 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. In conclusion, CURE not only enhances data security through encrypted server-side computations but also demonstrates practicality and broad applicability across diverse training scenarios and complex network architectures.

References

[1] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated Learning: Strategies for Improving Communication Efficiency," *arXiv:1610.05492*, 2017.

[2] J. Konečnỳ, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated Optimization: Distributed Machine Learning for On-Device Intelligence," *arXiv:1610.02527*, 2016.

[3] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 08 2018.

[4] D. Pasquini, G. Ateniese, and M. Bernaschi, "Unleashing the Tiger: Inference Attacks on Split Learning," in *SIGSAC CCS*. ACM, 2021.

[5] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting Unintended Feature Leakage in Collaborative Learning," in *SP*. IEEE, 2019, pp. 691–706.

[6] J. Fu, X. Ma, B. B. Zhu, P. Hu, R. Zhao, Y. Jia, P. Xu, H. Jin, and D. Zhang, "Focusing on Pinocchio's Nose: A Gradients Scrutinizer to Thwart Split-Learning Hijacking Attacks Using Intrinsic Attributes," in *NDSS*, 2023.

[7] F. Yu, L. Wang, B. Zeng, K. Zhao, Z. Pang, and T. Wu, "How to backdoor split learning," *Neural Networks*, vol. 168, p. 326–336, 2023.

[8] Y. Mao, Z. Xin, Z. Li, J. Hong, Q. Yang, and S. Zhong, "Secure Split Learning Against Property Inference, Data Reconstruction, and Feature Space Hijacking Attacks," in *ESORICS*. Springer, 2023, pp. 23–43.

[9] G. Gawron and P. Stubbings, "Feature Space Hijacking Attacks against Differentially Private Split Learning," in *PPAI*, 2022.

[10] B. Hitaj, G. Ateniese, and F. Perez-Cruz, "Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning," in *ACM CCS*, 2017.

[11] K. Özfatura, E. Özfatura, A. Küpçü, and D. Gündüz, "Byzantines can also learn from history: Fall of centered clipping in federated learning," *IEEE TIFS*, vol. 19, pp. 2010–2022, 2024.

[12] M. P. Parisot, B. Pejo, and D. Spagnuelo, "Property Inference Attacks on Convolutional Neural Networks: Influence and Implications of Target Model's Complexity," in *SECRYPT*. INSTICC, 2021, pp. 715–721.

[13] E. Erdoğan, A. Küpçü, and A. E. Çiçek, "Unsplit: Data-Oblivious Model Inversion, Model Stealing, and Label Inference Attacks against Split Learning," in *WPES*. ACM, 2022, p. 115–124.

[14] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *ACM CCS*, 2015.

[15] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *ACM CCS*, 2016.

[16] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," *CoRR*, vol. abs/1710.06963, 2018.

[17] K. Wei, J. Li, M. Ding, C. Ma, H. H. Yang, F. Farokhi, S. Jin, T. Q. S. Quek, and H. V. Poor, "Federated learning with differential privacy: Algorithms and performance analysis," *IEEE TIFS*, vol. 15, pp. 3454–3469, 2020.

[18] N. Wu, F. Farokhi, D. Smith, and M. A. Kaafar, "The value of collaboration in convex machine learning with differential privacy," *CoRR*, vol. abs/1906.09679, 2019.

[19] D. Froelicher, J. R. Troncoso-Pastoriza, A. Pyrgelis, S. Sav, J. S. Sousa, J.-P. Bossuat, and J.-P. Hubaux, "Scalable Privacy-Preserving Distributed Learning," *PoPETs*, no. 2, pp. 323–347, 2021.

[20] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J.-P. Bossuat, J. S. Sousa, and J.-P. Hubaux, "Poseidon: Privacy-preserving federated neural network learning," in *NDSS*, 2021.

[21] F. Karakoç, A. Küpçü, and M. Önen, "Fault tolerant and malicious secure federated learning," in *CANS*. Springer, 2024, pp. 73–95.

[22] S. Sav, A. Diaa, A. Pyrgelis, J.-P. Bossuat, and J.-P. Hubaux, "Privacy-preserving federated recurrent neural networks," *PoPETs*, no. 4, pp. 500–521, 2021.

[23] B. Jayaraman, L. Wang, D. Evans, and Q. Gu, "Distributed learning without distress: Privacy-preserving empirical risk minimization," in *NIPS*, 2018.

[24] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou, "A hybrid approach to privacy-preserving federated learning," in *ACM AISec*, 2019.

[25] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," in *IEEE S&P*, 2019.

[26] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning," in *USENIX Security*, 2020.

[27] D. Reich, A. Todoki, R. Dowsley, M. D. Cock, and A. C. A. Nascimento, "Privacy-preserving classification of personal text messages with secure multi-party computation: An application to hate-speech detection," *CoRR*, vol. abs:1906.02325, 2021.

[28] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-Party Secure Computation for Neural Network Training," *PoPETS*, 2019.

[29] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: Honest-majority maliciously secure framework for private deep learning," *PoPETS*, 2020.

[30] H. Zhu, R. S. Mong Goh, and W.-K. Ng, "Privacy-preserving weighted federated learning within the secret sharing framework," *IEEE Access*, vol. 8, pp. 198 275–198 284, 2020.

[31] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "Splitfed: When federated learning meets split learning," in *AAAI*, 2022.

[32] T. Titcombe, A. J. Hall, P. Papadopoulos, and D. Romanini, "Practical Defences Against Model Inversion Attacks for Split Neural Networks," in *ICLR Workshop on DPML*, 2021.

[33] S. Abuadbba, K. Kim, M. Kim, C. Thapa, S. A. Camtepe, Y. Gao, H. Kim, and S. Nepal, "Can we use split learning on 1D CNN models for privacy preserving training?" in *ACM ASIACCS*, 2020, pp. 305–318.

[34] W. Wang, T. Wang, L. Wang, N. Luo, P. Zhou, D. Song, and R. Jia, "DPlis: Boosting Utility of Differentially Private Deep Learning via Randomized Smoothing," *PoPETS*, 2021.

[35] X. Yang, J. Sun, Y. Yao, J. Xie, and C. Wang, "Differentially private label protection in split learning," *arXiv:2203.02073*, 2022.

[36] M. A. Rahman, T. Rahman, R. Laganière, N. Mohammed, and Y. Wang, "Membership inference attack against differentially private deep learning model." *Trans. Data Priv.*, vol. 11, no. 1, pp. 61–79, 2018.

[37] E. Erdoğan, A. Küpçü, and A. E. Çiçek, "Splitguard: Detecting and mitigating training-hijacking attacks in split learning," in *ACM WPES*, 2022, pp. 125–137.

[38] E. Erdoğan, U. Tekşen, M. S. Çeliktenyıldız, A. Küpçü, and A. E. Çiçek, "Splitout: Out-of-the-box training-hijacking detection in split learning via outlier detection," in *CANS*. Springer, 2024, pp. 118–142.

[39] G.-L. Pereteanu, A. Alansary, and J. Passerat-Palmbach, "Split HE:

Fast Secure Inference Combining Split Learning and Homomorphic Encryption," in *PPAI*, 2022.

[40] T. Khan, K. Nguyen, and A. Michalas, "Split Ways: Privacy-Preserving Training of Encrypted Data Using Split Learning," in *EDBT/ICDT*, 2023.

[41] T. Khan, K. Nguyen, A. Michalas, and A. Bakas, "Love or Hate? Share or Split? Privacy-Preserving Training Using Split Learning and Homomorphic Encryption," in *PST*. IEEE, 2023, pp. 1–7.

[42] T. Khan, K. Nguyen, and A. Michalas, "A More Secure Split: Enhancing the Security of Privacy-Preserving Split Learning," in *Secure IT Systems*. Springer, 2023, pp. 307–329.

[43] S. De Rubeis, X. He, A. P. Goldberg, C. S. Poultney, K. Samocha, A. Ercument Cicek, Y. Kou, L. Liu, M. Fromer, S. Walker *et al.*, "Synaptic, transcriptional and chromatin genes disrupted in autism," *Nature*, vol. 515, no. 7526, pp. 209–215, 2014.

[44] F. K. Satterstrom, J. A. Kosmicki, J. Wang, M. S. Breen, S. De Rubeis, J.-Y. An, M. Peng, R. Collins, J. Grove, L. Klei *et al.*, "Large-scale exome sequencing study implicates both developmental and functional changes in the neurobiology of autism," *Cell*, vol. 180, no. 3, pp. 568–584, 2020.

[45] U. Norman and A. E. Cicek, "St-steiner: a spatio-temporal gene discovery algorithm," *Bioinformatics*, vol. 35, no. 18, pp. 3433–3440, 2019.

[46] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," in *ICLR AI for social good workshop*, 2019.

[47] C. Thapa, M. A. P. Chamikara, and S. A. Camtepe, "Advancements of Federated Learning Towards Privacy Preservation: From Federated Learning to Split Learning," *Federated Learning Systems: Towards Next-Generation AI*, pp. 79–109, 2021.

[48] Z. Li, C. Yan, X. Zhang, G. Gharibi, Z. Yin, X. Jiang, and B. A. Malin, "Split Learning for Distributed Collaborative Training of Deep Learning Models in Health Informatics," in *Annual Symposium Proceedings*. AMIA, 2023, p. 1047–1056.

[49] S. Satpathy, O. Khalaf, D. Kumar Shukla, M. Chowdhary, and S. Algburi, "A collective review of Terahertz technology integrated with a newly proposed split learningbased algorithm for healthcare system," *IJCDS*, vol. 15, no. 1, pp. 1–9, 2024.

[50] M. G. Poirot, P. Vepakomma, K. Chang, J. Kalpathy-Cramer, R. Gupta, and R. Raskar, "Split Learning for collaborative deep learning in healthcare," *arXiv:1912.12115*, 2019.

[51] M. G. Poirot, "Split Learning in Health Care: Multi-center Deep Learning without sharing patient data," Master's thesis, University of Twente, 2020.

[52] P. Joshi, C. Thapa, S. Camtepe, M. Hasanuzzaman, T. Scully, and H. Afli, "Performance and Information Leakage in Splitfed Learning and Multi-Head Split Learning in Healthcare Data and Beyond," *Methods and Protocols*, vol. 5, no. 4, p. 60, 2022.

[53] U. Majeed, S. S. Hassan, and C. S. Hong, "Vanilla Split Learning for Transportation Mode Detection using Diverse Smartphone Sensors," in *KCC*. KIISE, 2021, pp. 23–25.

[54] Q. Zhang, Z. Jiang, Q. Lu, J. Han, Z. Zeng, S.-H. Gao, and A. Men, "Split to Be Slim: An Overlooked Redundancy in Vanilla Convolution," in *IJCAI*, 2020.

[55] C. G. Allaart, B. Keyser, H. Bal, and A. Van Halteren, "Vertical Split Learning - an exploration of predictive performance in medical and other use cases," in *IJCNN*. IEEE, 2022, pp. 1–8.

[56] O. S. Ads, M. M. Alfares, and M. A.-M. Salem, "Multi-limb Split Learning for Tumor Classification on Vertically Distributed Data," in *ICICIS*. IEEE, 2021, pp. 88–92.

[57] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical partitioning algorithms for database design," *ACM TODS*, vol. 9, no. 4, 1984.

[58] S. B. Navathe and M. Ra, "Vertical partitioning for database design: a graphical algorithm," in *ACM SIGMOD*, 1989, pp. 440–450.

[59] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal data partitioning in database design," in *ACM SIGMOD*, 1982, pp. 128–136.

[60] M. Ra, "Horizontal partitioning for distributed database design: A graph-based approach." in *Australian Database Conference*, 1993, pp. 101–120.

[61] F. Yu, B. Zeng, K. Zhao, Z. Pang, and L. Wang, "Chronic Poisoning: Backdoor Attack against Split Learning," in *AAAI*, 2024.

[62] F. Yu, L. Wang, B. Zeng, K. Zhao, T. Wu, and Z. Pang, "SIA: A sustainable inference attack framework in split learning," *Neural Networks*, vol. 171, pp. 396–409, 2024.

[63] K. Nguyen, T. Khan, and A. Michalas, "Split without a leak: Reducing privacy leakage in split learning," *arXiv:2308.15783*, 2023.

[64] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[65] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[66] J. Liu, X. Lyu, Q. Cui, and X. Tao, "Similarity-based Label Inference Attack against Training and Inference of Split Learning," *TIFS*, 2024.

[67] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *ASIACRYPT*, 2017, pp. 409–437.

[68] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *EUROCRYPT*, 2010, pp. 1–23.

[69] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *IEEE SP*, 2017, pp. 3–18.

[70] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *NIPS*, 2019, pp. 14 774–14 784.

[71] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *SP*. IEEE, 2017, pp. 19–38.

[72] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *CRYPTO*, 2010, pp. 465–482.

[73] M. Albrecht *et al.*, "Homomorphic Encryption Security Standard," HomomorphicEncryption.org, Tech. Rep., 2018.

[74] "Breast cancer wisconsin (original)," https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original), (Accessed: 2024-05-05).

[75] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[76] I.-C. Yeh and C. hui Lien, "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients," *Expert Systems with Applications*, vol. 36, no. 2, pp. 2473 – 2480, 2009.

[77] P. Wagner, N. Strodthoff, E. Bietti, T. Schaeffter, X. Zhu, and R. Durichen, "PTB-XL, a large publicly available electrocardiography dataset," *Scientific Data*, 2020.

[78] Q. Lou, B. Feng, G. C. Fox, and L. Jiang, "Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data," in *NeurIPS*, vol. 33, 2020, pp. 9193–9202.

[79] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi, "Towards Deep Neural Network Training on Encrypted Data," *CVPR*, 2019.

[80] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[81] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big Data: Astronomical or Genomical?" *PLoS Biology*, vol. 13, no. 7, 2015.

[82] M. Bakshi and M. Last, "Cryptornn - privacy-preserving recurrent neural networks using homomorphic encryption," in *CSCML*, 2020.

[83] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *ICML*, 2016.

[84] E. Hesamifard, H. Takabi, M. Ghasemi, and R. Wright, "Privacy-preserving machine learning as a service," *PoPETS*, 2018.

[85] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via MiniONN transformations," in *ACM CCS*, 2017.

SUPPLEMENTARY MATERIAL

A. CURE's *Initialization Algorithm*

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

Algorithm 2 Initialization

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

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:

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

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:

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

By restructuring the data in this manner, we fill the RLWE vector with meaningful data and pad it with zeros when necessary, as detailed in Section IV-C3.

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 V-C), 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 IV-C3. 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.

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

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 Chebyshev interpolation method [80] or minimax approximation [81] to approximate the non-linear activation functions as polynomials. This technique is also employed by numerous privacy-preserving machine learning works [19], [20], [22], [82]–[85].

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.

F. Network Setup

We use MPI (Message Passing Interface) which is a standard for passing messages between different processes in a distributed memory system [2], enabling parallel computing architectures to communicate efficiently, to implement the communication between client and server. The experiments are conducted on LAN and WAN environments. Our primary focus is on LAN because, for CURE, the determining factor is the computational expense of the HE operations. Finally, we extrapolate WAN results, by calculating the amount of data to be transferred for a given network through our estimator in Section IV-D (see formula 7).

G. Detailed Information on Datasets

We detail the datasets used in the evaluation of CURE here. We employ: (i) Breast Cancer Wisconsin dataset (BCW) [74] with 699 samples, 9 features and 2 labels, (ii) the hand-written digits (MNIST) dataset [75] with $70{,}000$ images of 28×28 pixels and 10 labels, (iii) the default of credit card clients (CREDIT) dataset [76] with $n = 30{,}000$ samples, 23 features and 2 labels, (iv) the PTB XL dataset [77] (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.

H. Model Architectures and Split Learning Setup Used For Experimental Evaluations

We employ varying network models for our different types of experiments. The models are structured as follows: For time latency experiments we use models; **Model 1**: $784 \times 128 \times 32 \times 10$ (adapted from [78]), **Model 2**: $784 \times 128 \times 128 \times 128 \times 128 \times 128 \times 128 \times 32 \times 10$, **Model 3**: $8192 \times 8192 \times 1024 \times 512 \times 128 \times 32 \times 10$, **Model 4**: $16384 \times 16384 \times 8192 \times 32 \times 10$ and **Model 5** serves as a computational counterpart to ResNet, designed to test scalability with respect to an increasing number of layers. and for simulation tests to obtain accuracy results we use models; **Model 6**: $[input] \times 128 \times 32 \times [output]$, **Model 7**: $[input] \times 1024 \times 32 \times [output]$, **Model 8:** $[input] \times 2048 \times 32 \times [output]$ where [input] and [output] are the number of input and outputs in the respective datasets. The structure of these networks is designed to clearly demonstrate how the proposed methods perform across various settings. Unless otherwise stated, we use a batch size (b) of 60 for a fair comparison with [78]. We also experiment with various batch sizes to empirically show the effect of b on the runtime. Using our first four models for time latency experiments, we created several split learning setups for a chosen NN architecture to demonstrate how CURE provides advantages in different scenarios. We achieved this by varying n, which refers to the number of encrypted layers or, in other words, the position of the last layer of the server in the entire NN. We observe that n is a crucial determinant in the performance as it represents the network layers where the server homomorphically executes operations, as discussed in Section IV-C.

For unencrypted layers, we use the sigmoid activation function, and for encrypted layers, we approximate it with a polynomial using Chebyshev interpolation [80] or minimax approximation [81]. After testing various degrees and intervals, we selected a degree of 7 and an interval of [-15, 15] for an optimal balance of accuracy and efficiency. These baselines allow us to assess CURE's accuracy loss due to activation function approximation, fixed precision, encryption, and the effects of privacy-preserving split learning.

[2]Message Passing Interface, Online: https://www.mpi-forum.org/docs/

Network	Dataset	CURE Accuracy	Plaintext Accuracy
[input]x128x32x[output]	MNIST	95.97%	95.83%
	BCW	97.37%	98.25%
	CREDIT	81.61%	81.70%
[input]x1024x32x[output]	MNIST	96.11%	96.16%
	BCW	98.25%	99.12%
	CREDIT	81.73%	81.77%
[input]x2048x32x[output]	MNIST	95.74%	96.32%
	BCW	99.12%	99.12%
	CREDIT	81.16%	81.25%

TABLE IV: CURE*'s accuracy results for plaintext and encrypted learning with 10 epochs.*

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 V.

Figure 4 displays the effect of increasing number of samples when matrices of dimensions 1×2^{13} and $2^{13} \times$ number of samples taken into dot product we have implemented. By doubling the number of samples to be processed and recording the timing results, we observe a linear growth in the complexity of our ciphertext-ciphertext dot product.

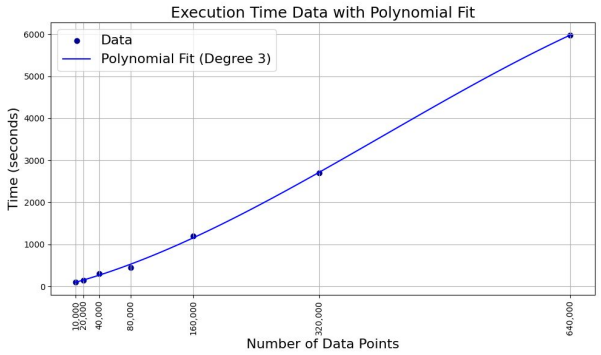

Figure 4: *Homomorphic dot product scalability with respect to increasing number of data samples.*

Table V displays the impact of increasing the number of CPU cores on runtime for varying batch sizes. Figure IV displays CURE's accuracy results for plaintext and encrypted learning with 10 epochs and we discuss the results in Section V-B1. Figure 5 illustrates an asymptotic comparison between one-level operations and ciphertext-ciphertext multiplications of CURE and prior works [78], [79].

Figure 6 and 7 illustrate the comparison between the runtime estimated by the advisor function and the actual runtime

Batch Size (b)	#CPU cores	Execution Time (m)
60	1	62.5
60	2	51.0
60	10	23.3
60	20	24.2
60	30	27.6
60	40	29.2
128	1	131.3
128	2	92.2
128	10	29.1
128	20	23.5
128	30	24.9
128	40	23.5

TABLE V: CURE*'s runtime for one training epoch with* $10{,}000$ *samples,* ***Model 1*** *network with the varying number of* b *and CPU cores. The first two layers are encrypted* ($n=2$) *on a LAN setting.*

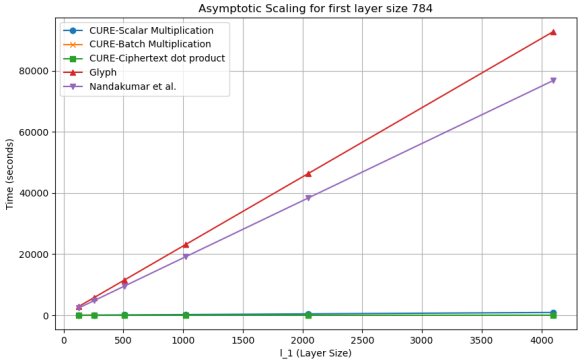

Figure 5: *Asymptotic comparisons between one-level operations and ciphertext-ciphertext multiplications. The figure illustrates the impact of increasing the size of the second multiplicand in the product. It is important to note that results from prior work have been extrapolated due to the lack of implementation and available data across various architectural configurations.*

observed in the experiments. We observe that the advisor function estimates the actual runtime with minimal error.

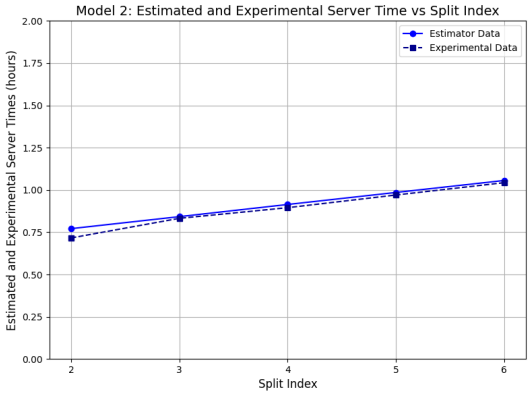

Figure 6: *Runtime comparison of advisor function estimation and experimental results for server-side operations and for* ***Model 2*** *training evaluation.*

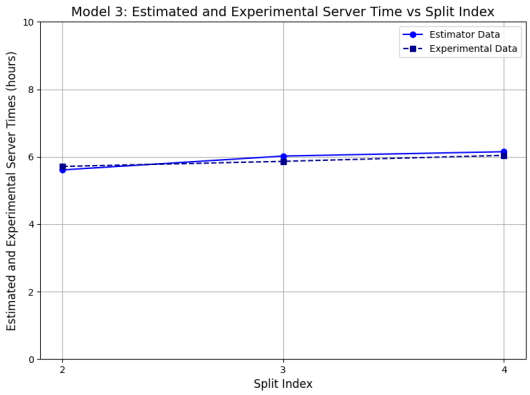

Figure 7: *Runtime comparison of advisor function estimation and experimental results for server-side operations and for* ***Model 3*** *training evaluation.*