PP-Stream: Toward High-Performance Privacy-Preserving Neural Network Inference via Distributed Stream Processing

Qingxiu Liu[1,3], Qun Huang[1], Xiang Chen[2,1], Sa Wang[4], Wenhao Wang[5], Shujie Han[1], Patrick P. C. Lee[3]
[1]*Peking University*, [2]*Zhejiang University*, [3]*The Chinese University of Hong Kong*,
[4]*Institute of Computing Technology, Chinese Academy of Sciences*,
[5]*Institute of Information Engineering, Chinese Academy of Sciences*

***Abstract*—Privacy preservation is critical for neural network inference, which often involves collaborative execution of different parties to make predictions on sensitive data based on sensitive neural network models. However, the expensive cryptographic operations of privacy preservation also pose performance challenges to neural network inference. We address this performance-security tension by designing PP-Stream, a distributed stream processing system for high-performance privacy-preserving neural network inference. PP-Stream adopts hybrid privacy-preserving mechanisms for linear and non-linear operations of neural network inference. It treats inference data as real-time data streams, and parallelizes the inference operations across multiple pipelined stages that are executed by multiple servers and threads. It also solves the load-balanced resource allocation across servers and threads as an optimization problem. We prototype PP-Stream and show via testbed experiments that it achieves low inference latencies on various neural network models.**

***Index Terms*—homomorphic encryption, obfuscation, distributed stream processing**

I. Introduction

A. Motivation and Challenges

Neural network inference has been widely applied in various domains, such as face recognition [43], [47], medical diagnosis [30], [48], and home monitoring [8], [13]. Its model training and inference procedures are often collaboratively executed by two parties, namely the *model provider* and the *data provider*. In the training phase, the model provider designs the neural network model and employs algorithms to train the model parameters with sufficient training data as input, while in the inference phase, the data provider feeds data of interest into the model provider to yield inference results from the trained model. To make neural network inference practical and scalable, we need to address two primary deployment challenges: *privacy preservation* and *high performance*.

Privacy preservation. The collaborative nature of neural network inference makes privacy preservation critical [40]. From the model provider's perspective, the model parameters are often their proprietary assets, since the designs of models and training algorithms require dedicated domain knowledge from human experts. Also, the massive datasets for training need laborious collection and pre-processing and the training procedure consumes extensive hardware resources, making the model provider unwilling to disclose such critical information [17], [19], [29]. From the data provider's perspective, the input data for inference is often related to personal privacy [19], [53], [59], and the model is expected to produce inference results without knowing the sensitive input data. Thus, both the model parameters for the model provider and the input data for the data provider should be well protected. Furthermore, it is critical to support simple deployment without relying on third parties for privacy preservation, as third-party services need to be trusted and hence require strong security assumptions. Even though prior studies address the privacy issue in neural network inference (see Section VII for details), they suffer from various limitations, such as accuracy drops in inference (e.g., [31], [33], [34], [38]), high performance overhead in using multiple protocols (e.g., [24], [45], [51]), loss of generality (e.g., [49], [51]), and reliance on trusted third parties or environments (e.g., [50], [57], [64]).

High performance. It is also critical to maintain high performance in neural network inference, yet privacy preservation and high performance are conflicting goals. In particular, privacy preservation builds on expensive cryptographic operations; for example, homomorphic encryption, a widely used cryptographic primitive in prior privacy-preserving approaches (e.g., CryptoNets [31] and CryptoDL [33], [34]), can increase the computation and communication times by two orders of magnitude over plaintexts in model training [66]. Some privacy-preserving approaches (e.g., [24], [45], [51]) combine multiple cryptographic protocols, but require expensive secure transformations across different protocols [56].

To validate the performance overhead, we implement a simple privacy-preserving residual network [32] based on Paillier's homomorphic encryption [55] using the GMP library [5]. Specifically, we encrypt a *tensor* (multi-dimensional array) of size 28×28, perform scalar multiplication (with the scaling constant 10^6) on the encrypted tensor, perform homomorphic addition between the original encrypted tensor and scalar multiplication results, and finally obtain the homomorphic addition results by decryption. We repeat the above steps 1,000 times with different input tensors from the images of the MNIST dataset [10], and benchmark the average computational latency of each step (see the testbed details in Section VI-A). Figure 1 shows the latency results for processing an input tensor versus the key size in Paillier's homomorphic encryption. The latencies for encryption/decryption and arithmetic operations for privacy preservation are on the order of seconds and milliseconds, respectively. For comparisons, we also perform the same scalar multiplication and addition on plaintexts (not shown in the

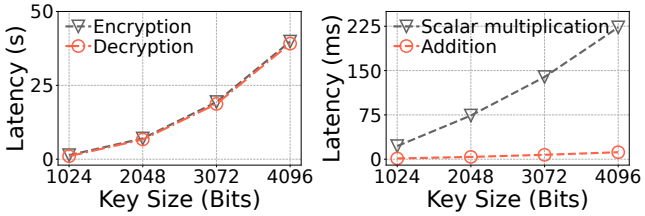


Fig. 1: Homomorphic encryption benchmark.

figure), whose average computational times are only 2.1 μs and 1.7 μs, respectively. Therefore, we see that homomorphic encryption incurs heavy computation overhead in privacy-preserving neural networks.

B. Contributions

Our core idea is to design lightweight yet secure privacy-preserving mechanisms for neural network inference, and adapt them into *distributed stream processing* to parallelize neural network inference by treating input data for inference as real-time data streams. To this end, we design PP-Stream, a distributed stream processing framework for high-performance privacy-preserving neural network inference. PP-Stream aims for: (i) privacy preservation for both model and data providers in neural network inference, (ii) high inference accuracy, (iii) low inference latency, (iv) supporting general neural networks, and (v) being deployable in untrusted environments. To the best of our knowledge, PP-Stream is the first system that simultaneously satisfies all these goals.

To achieve privacy preservation, PP-Stream decomposes the inference procedure into *linear* and *non-linear* operations and protects them with *hybrid* privacy-preserving techniques. Specifically, for linear operations, PP-Stream applies homomorphic encryption to directly process encrypted tensors without decryption and obtain the encrypted linear operation results. Since practical homomorphic encryption approaches only work for linear operations, for non-linear operations, PP-Stream adopts a lightweight obfuscation protocol, in which the model provider performs permutation on the element positions of each tensor and sends the obfuscated tensors to the data provider to perform non-linear operations. The model provider later receives the non-linear operation results from the data provider, performs inverse permutation on the element positions of the received results, and feeds the non-obfuscated outputs to the next round of linear operations.

To alleviate the high performance overhead in privacy-preserving mechanisms (e.g., homomorphic encryption), we exploit system-level optimizations that are scalable and can support general neural networks. PP-Stream is designed as a distributed stream processing system that encapsulates linear and non-linear operations into alternate pipelined stages that run on different servers between the model and data providers. In each stage, PP-Stream assigns multiple threads to parallelize tensor processing (i.e., homomorphic encryption and obfuscation operations). For efficient resource usage, PP-Stream performs *load-balanced resource allocation* by solving an integer linear programming problem to assign CPU resources across different stages. To mitigate the communication overhead, it further performs *tensor partitioning*, which divides an input tensor into sub-tensors and feeds the sub-tensors (instead of the whole tensor) into multiple threads, each of which produces some elements of the output tensor.

In summary, we make the following contributions:

- We design PP-Stream based on hybrid privacy-preserving mechanisms. We also discuss its correctness and security guarantees. (Section III)
- We design and implement PP-Stream as a distributed stream processing system that builds on several design elements, such as scaling floating-point numbers to integers for cryptographic operations, encapsulating operations into pipelined stages, performing load-balanced resource allocation, and partitioning tensors to multiple threads. To the best of our knowledge, PP-Stream is the *first* system that maps privacy-preserving neural network inference into a distributed stream processing system. (Section IV)
- We evaluate PP-Stream on nine neural network models trained from public datasets [1]–[3], [7], [10]. PP-Stream achieves low inference latency, while preserving high inference accuracy. Its load-balanced resource allocation reduces the inference latency by up to 64.94% compared to without load-balanced resource allocation, and its tensor partitioning reduces the inference latency by up to 61.64% compared to without tensor partitioning. We further measure the information leakage of PP-Stream, and show that PP-Stream reduces the inference latencies of state-of-the-art systems, including SecureML [51], CryptoNets [31], CryptoDL [33], [34], and EzPC [24]. (Section VI)

The PP-Stream prototype is now open-sourced at the following link: **https://github.com/calanquee/PP-Stream-Project**.

II. Preliminaries

A. Basics of Neural Networks

We provide an overview of neural network inference. We consider a neural network that feeds the data through a sequence of *layers*. The data that is fed into each layer can be expressed as tensors (i.e., multi-dimensional arrays), and each layer transforms a tensor by some *activation functions*. The first layer (called the *input layer*) receives the raw input tensors; the last layer (called the *output layer*) outputs the inference results; each of the intermediate layers (called the *hidden layers*) receives the output tensors from its previous layer as input tensors, operates on the received tensors, and forwards the output tensors to the next layer. The weights of the connections across the layers are called the *model parameters*. The training step learns the model parameters, while the inference step outputs the prediction results based on the model parameters.

In this paper, we consider a *collaborative* inference scenario that involves two parties, namely the *model provider* and the *data provider*. Specifically, the model provider starts with a well-trained neural network model, which can be trained on plaintexts through general frameworks (e.g., PyTorch [12] and TensorFlow [14]). It also manages the model parameters and

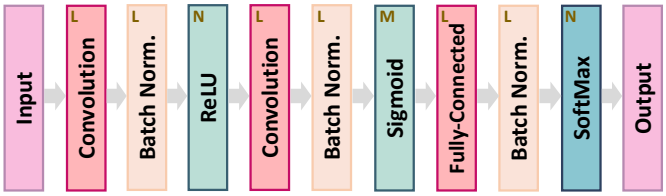


Fig. 2: A CNN model whose hidden layers can be classified into linear (L), non-linear (N), and mixed (M) layers.

outputs inference results from the model. The data provider provides the raw data to the model provider for inference.

Given a neural network, we can classify each hidden layer by its operations into three types: (i) *linear layer*, which contains only linear operations (i.e., tensor addition and multiplication) between model parameters and input tensors; (ii) *non-linear layer*, which contains only non-linear activation functions (e.g., ReLU and SoftMax); and (iii) *mixed layer*, which contains a mix of linear and non-linear operations. To illustrate, Figure 2 depicts a convolution neural network (CNN) [44] that includes nine hidden layers with different types of operations. Convolution, batch normalization, and fully-connected layers are linear layers; the ReLU and SoftMax layers are non-linear layers; the Sigmoid layer is a mixed layer with both linear (e.g., scalar multiplication between input tensors and model parameters) and non-linear (e.g., exponentiation) operations.

B. Design Goals

PP-Stream aims for the following design goals:

- **Privacy preservation:** It protects both model parameters for the model provider and input data for the data provider in neural network inference.
- **High accuracy:** It maintains nearly identical inference accuracy compared with the plain inference without privacy preservation.
- **High performance:** It achieves high throughput and low latency in neural network inference.
- **Generality:** It supports general neural networks without the need of modifying the training and inference protocols.
- **Simple deployment:** It can be deployed in untrusted environments, without relying on trusted services.

Table I compares state-of-the-art privacy-preserving neural network inference systems and PP-Stream, and shows how existing systems are limited in achieving some of the goals besides privacy preservation. In terms of accuracy, CryptoNets [31], CryptoDL [33], [34], E2DM [38], Faster CryptoNets [25], FHE-DiNN100 [22], nGraph-HE [21], LoLa [23], CHET [27], and DELPHI [49] all apply approximate fitting for non-linear operations (e.g., activation functions), leading to accuracy degradations. In terms of performance, CryptoNets [31], CryptoDL [33], [34], E2DM [38], Faster CryptoNets [25], and nGraph-HE [21] have high inference latency based on the reported results in their published papers; for example, CryptoNets [31] has two orders of magnitude higher latency than SecureML [51]. We also evaluate EzPC [24] in our testbed and show its high inference latency especially in large models

TABLE I: Comparisons with state-of-the-arts (in order of publication years) in privacy-preservation (PP), high accuracy (HA), high performance (HP), generality (G), and simple deployment (SD).

Framework	Design Goals				
	PP	HA	HP	G	SD
CryptoNets [31]	✓	✗	✗	✗	✓
CryptoDL [33], [34]	✓	✗	✗	✗	✓
SecureML [51]	✓	✓	✓	✗	✗
E2DM [38]	✓	✗	✗	✗	✓
Faster CryptoNets [25]	✓	✗	✗	✗	✓
Chameleon [57]	✓	✓	✓	✓	✗
Slalom [64]	✓	✓	✓	✓	✗
FHE-DiNN100 [22]	✓	✗	✓	✗	✓
nGraph-HE [21]	✓	✗	✗	✗	✓
SecureNN [65]	✓	✓	✓	✓	✗
LoLa [23]	✓	✗	✓	✗	✓
CHET [27]	✓	✗	✓	✗	✓
EzPC [24]	✓	✓	✗	✓	✓
QuantizedNN [15]	✓	✓	✓	✓	✗
DELPHI [49]	✓	✗	✓	✗	✓
CrypTFlow [41]	✓	✓	✓	✓	✗
PP-Stream (ours)	✓	✓	✓	✓	✓

(Section VI). In terms of generality, some systems [21]–[23], [25], [27], [31], [33], [34], [38], [49] need model retraining (i.e., the training protocol is changed) or special activation functions [51], so they sacrifice generality. In terms of simple deployment, SecureML [51], SecureNN [65], QuantizedNN [15], and CrypTFlow [41] rely on two or three non-colluding servers to act as the model provider, while Chameleon [57] and Slalom [64] rely on trusted execution environments. Thus, they cannot achieve the simple deployment goal.

C. Threat Model

We define the threat model about the adversarial capabilities. We assume that both the model and data providers are honest-but-curious (as in prior studies [49], [57]), meaning that they provide correct model parameters and raw data, respectively, and faithfully execute the protocols, so that the data provider will eventually obtain the inference results. However, they are curious about the sensitive information from the other provider and attempt to extract the sensitive information through protocol execution. Because the model and data providers exchange information through the network, we assume that the adversaries can launch passive attacks and eavesdrop on the communication between the model and data providers. Based on the threat model, we provide two guarantees for privacy-preserving neural network inference.

- **Correctness.** Given the privacy-preserving inference protocol, the data provider should obtain the same inference results as in the original inference protocol without any privacy-preserving mechanism.
- **Security.** Given the honest-but-curious assumption, we specify the security guarantees against the compromised model and data providers as well as passive adversaries that

eavesdrop on the communication between the model and data providers.

- We ensure that the compromised model provider cannot obtain any information regarding both the data provider's raw input tensors (except for the dimension) and intermediate results during protocol execution.
- We ensure that it is computationally infeasible for the compromised data provider to derive the model parameters based on the information obtained from the inference phase, even though there may be a negligible amount of information exposure (e.g., tensor dimensions) to the compromised data provider.
- We ensure that the adversaries cannot extract the model parameters from the model provider and the raw data from the data provider by eavesdropping on the communication between the model and data providers.

Note that a compromised data provider can execute model stealing attacks by training a new model based on the queried samples and corresponding inference results. A possible countermeasure is to rate-limit the number of requests issued by the data provider [39].

III. Hybrid Privacy Preservation

For privacy preservation, the model parameters should always reside in the model provider, while the raw input data and inference results should always reside in the data provider. This requirement motivates our design to decompose the inference phase. Specifically, PP-Stream protects neural network inference via *hybrid privacy preservation* by applying different privacy preservation mechanisms to linear and non-linear operations. We first provide an overview of the collaborative workflow of neural network inference in PP-Stream (Section III-A). We then describe the designs for privacy-preserving linear and non-linear operations based on homomorphic encryption (Section III-B) and obfuscation (Section III-C), respectively. Finally, we discuss the security implications of our hybrid design (Section III-D). Note that each of the privacy-preserving techniques has been studied in prior work (including homomorphic encryption [55] and obfuscation [52]), so we do not claim that we propose novel privacy-preserving methods. Instead, our contribution here is to seamlessly integrate these techniques into distributed stream processing (Section IV).

A. Overview of Collaboration Workflow

Figure 3 shows the collaborative inference workflow between the model and data providers in PP-Stream. Suppose that the model provider maintains a neural network and the data provider sends a tensor to the model provider for inference. Here, we assume that a neural network starts with a linear layer and ends with a non-linear layer (which is true in typical neural networks, say, CNNs (Figure 2)). We can divide the workflow into three parts, each of which has a different mix of steps (i.e., encryption, decryption, obfuscation, inverse obfuscation, linear operations, and non-linear operations).

In the first round, the data provider encrypts the raw input tensor (Step 1.1) and sends the encrypted tensor to the model provider for inference (Step 1.2). The model provider performs linear operations on the encrypted tensor based on homomorphic encryption (Step 1.3). It then obfuscates the resulting tensor (Step 1.4) and sends the obfuscated tensor back to the data provider (Step 1.5).

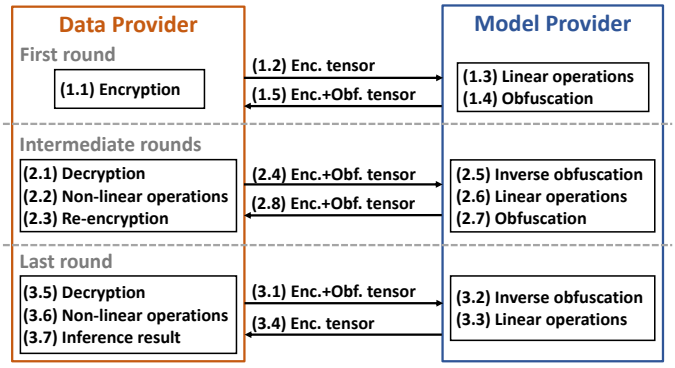


Fig. 3: Workflow of PP-Stream, in which the model and data providers exchange encrypted tensors (with or without obfuscation).

In each of the intermediate rounds, the data provider decrypts the obfuscated tensor (Step 2.1), performs non-linear operations (Step 2.2), re-encrypts the resulting tensor (Step 2.3), and sends the encrypted tensor to the model provider (Step 2.4). The model provider performs inverse obfuscation to recover the true encrypted tensor (Step 2.5), executes linear operations on the encrypted tensor (Step 2.6), obfuscates the resulting tensor (Step 2.7), and sends the obfuscated tensor to the data provider (Step 2.8). The model and data providers repeat Steps 2.1-2.8 in each subsequent intermediate round.

In the last round, the data provider sends the encrypted tensor to the model provider (Step 3.1), which performs inverse obfuscation (Step 3.2) and linear operations (Step 3.3) on the tensor from the data provider. It then sends directly the encrypted tensor to the data provider without obfuscation (Step 3.4) (we discuss the security implications in Section III-D). The data provider then decrypts the received tensor (Step 3.5), performs the last non-linear operation on the tensor (Step 3.6), and finally obtains the inference result (Step 3.7).

B. Privacy-Preserving Linear Operations

To protect linear operations, we leverage homomorphic encryption, which allows linear operations (including addition and multiplication) over ciphertexts without decryption. Homomorphic encryption can be classified as fully homomorphic encryption (FHE) and partially homomorphic encryption (PHE). In this work, we choose PHE, which has better computational performance than FHE. In particular, we use Paillier's public-key cryptosystem [55] (referred to as Paillier's PHE in short). Let $E(.)$ and $D(.)$ denote the public-key encryption and decryption functions in Paillier's PHE, respectively. For the linear operations of a privacy-preserving neural network, we leverage the homomorphic property of Paillier's PHE to support two types of arithmetic operations on ciphertexts, namely addition and scalar multiplication, as elaborated below (note that we omit the modulo operations for brevity):

- **Addition.** Let m_1 and m_2 be two plaintext integers. The addition of m_1 and m_2 can be expressed as:

$$m_1 + m_2 = D(E(m_1) \cdot E(m_2)). \quad (1)$$

- **Scalar multiplication.** Let w be a scalar and m be a plaintext integer. The scalar multiplication of w and m can be expressed as:

$$w \times m \ = \ D(E(m)^w). \quad (2)$$

Linear operations in neural networks are often in the form of $\sum_i w_i m_i + b$, where w_i is the i-th weight, b is the bias (both w_i's and b are the model parameters), and m_i is the i-th element of the input tensor from the data provider. Under Paillier's PHE, a linear operation can be expressed as:

$$\begin{aligned} \textstyle\sum_i w_i m_i + b \ &= \ D(E(\textstyle\sum_i w_i m_i + b) \\ &= \ D(\textstyle\prod_i E(m_i)^{w_i} \cdot E(b)). \end{aligned} \quad (3)$$

Thus, the data provider can send the encrypted tensors (i.e., $E(m_i)$'s) to the model provider, which performs an encrypted linear operation as $\prod_i E(m_i)^{w_i} \cdot E(b)$ and returns the result to the data provider (assuming no obfuscation on the returned result). Then, the data provider decrypts the result to obtain $\sum_i w_i m_i + b$, without revealing the plaintext tensors m_i's to the model provider. Note that Paillier's PHE (and public-key cryptosystems in general) needs to work on integers, while neural network parameters can be floating-point numbers. Thus, we need to first convert floating-point numbers into integers before we apply Paillier's PHE in PP-Stream (Section IV-A).

C. Privacy-Preserving Non-Linear Operations

To protect non-linear operations, we leverage a lightweight obfuscation mechanism, in which the model provider obfuscates a tensor (which is encrypted by the data provider) by *permuting* its element positions before sending the tensor to the data provider for performing non-linear operations. Specifically, to obfuscate a multi-dimensional tensor, denoted by $\mathbf{T}$, the model provider first *reshapes* $\mathbf{T}$ into a one-dimensional vector $\mathbf{v}$, where the dimension size of $\mathbf{v}$ is the product of all dimension sizes of $\mathbf{T}$, by representing the elements of $\mathbf{T}$ in lexicographic order in $\mathbf{v}$; for example, for a two-dimensional array, the elements of the first row of $\mathbf{T}$ are first assigned to $\mathbf{v}$, followed by the elements of the second row of $\mathbf{T}$, and so on. Given $\mathbf{v}$, the model provider randomly permutes the elements of $\mathbf{v}$ to form $\mathbf{v}'$, and sends $\mathbf{v}'$ to the data provider for performing non-linear operations. Note that in the first round and each of the intermediate rounds (i.e., Steps 1.4 and 2.7 in Figure 3, respectively), the model provider randomly permutes a tensor with different random seeds, so the permuted element positions vary across the intermediate rounds. Since the permutation is in essence a one-to-one mapping, the model provider can invert the permutation order of a tensor and restore its original element positions.

In this work, we target four non-linear functions: ReLU, Sigmoid, SoftMax, and MaxPooling; ReLU, Sigmoid, and SoftMax are activation functions, while MaxPooling is a downsampling function. Both ReLU and Sigmoid can work on permuted tensors without compromising the correctness as they perform element-wise operations, while SoftMax and MaxPooling can only work on non-obfuscated tensors (i.e., the element positions cannot be permuted). Nevertheless, since SoftMax is often used in the last hidden layer, the model provider does not perform obfuscation (Section III-A), meaning that the data provider can perform SoftMax on non-obfuscated tensors. MaxPooling can be replaced with a convolution layer where the stride is two plus a ReLU layer [62].

D. Discussion

We discuss how our hybrid privacy-preserving mechanisms maintain the correctness and security guarantees described in Section II-C. For correctness, both Paillier's PHE and obfuscation maintain the correctness of linear and non-linear operations, respectively, so the inference results remain unaffected.

For security, since Paillier's PHE is provably semantically secure under certain intractability assumptions [55], the tensors that are sent by the data provider for linear operations are secure against a compromised model provider. Also, the obfuscation mechanism builds on the random permutation of a multi-dimensional tensor, in which the total number of possible permutations is $P!$, where P is the product of all dimension sizes of the tensor. The probability that a compromised data provider can infer the original tensor from a randomly permuted tensor is $1/P!$, which is negligibly small for practical neural network applications (e.g., P is up to $32 \times 32 \times 8 = 8192$ in our evaluated neural networks (Section VI)). Note that the permuted tensor still leaks some information since the obfuscation changes permutations rather than values. We measure this information leakage via distance correlation [63] in Section VI.

Recall that the model provider sends directly the encrypted tensor to the data provider without obfuscation in the last round (Section III-A). To extract model parameters of a linear operation, a compromised data provider needs to obtain both the input and output tensors. While the compromised data provider can access the output tensor of the linear operations in the last round since there is no obfuscation, it only has access to the obfuscated input tensor from the previous round (i.e., the tensor sent to the model provider in Step 3.1 in Figure 3). Thus, the model parameters used in the linear operations in the last round remain secure.

Furthermore, both the model and data providers always exchange encrypted information over the network, as shown in Figure 3. Therefore, the communication is secure against eavesdropping attacks from passive adversaries.

IV. PP-Stream Design

PP-Stream is a distributed stream processing system that realizes the collaborative workflow of privacy-preserving neural network inference (Section III-A). It treats the layers of a neural network model as stream operations across different servers and pipelines the processing of inference requests across the layers in a streaming fashion; each layer can be further parallelized with multiple threads for further performance gains. This improves the inference performance.

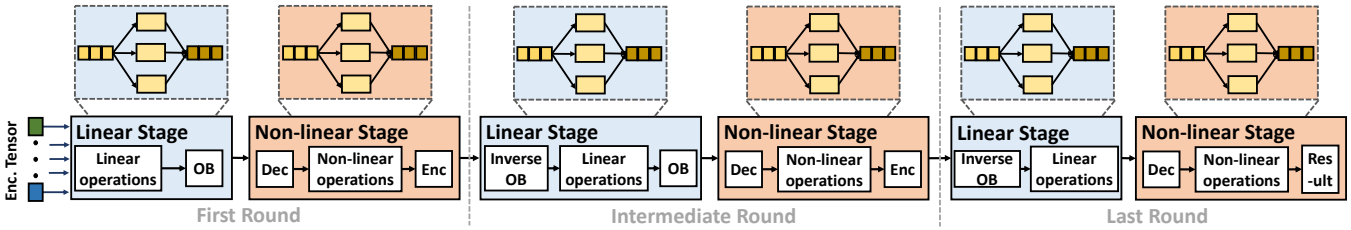


Fig. 4: PP-Stream architecture. Here, we show the stages of linear and non-linear primitive layers for three rounds of the collaborative workflow in Figure 3. Each stage performs encryption (Enc), decryption (Dec), obfuscation (OB), inverse obfuscation (inverse OB), linear operations, or non-linear operations; the last stage also outputs the inference result.

However, adapting privacy-preserving neural network inference into distributed stream processing is non-trivial. First, we should ensure that the linear and non-linear operations are only executed in the model and data providers for privacy preservation, respectively. Second, the distributed deployment includes multiple servers equipped with multiple CPU cores, and the resource demands also vary across different neural network layers. How to efficiently allocate the available computational resources across servers and CPU cores is critical. Third, we should address security requirements during resource allocation (e.g., linear and non-linear operations cannot be deployed on the same server), while ensuring load-balanced resource allocation. Finally, we should mitigate the resource usage overhead in distributed deployment, both in communication and computation.

To this end, we design PP-Stream with four key components: (i) *parameter scaling*, in which we scale floating-point model parameters to integers for cryptographic operations, while limiting the computational overhead; (ii) *operation encapsulation*, in which we encapsulate the linear and non-linear operations of neural network inference into alternate pipelined stages that run between model and data providers; (iii) *load-balanced resource allocation*, in which we perform both server and CPU core allocations for different stages in a load-balanced manner by formulating an integer programming problem; and (iv) *tensor partitioning*, in which we only send sub-tensors as input to each thread if applicable to reduce the communication overhead.

A. Parameter Scaling

Recall that we perform Paillier's PHE [55] for privacy-preserving linear operations (Section III-B). Since Paillier's PHE cannot work for floating-point numbers (e.g., model parameters), we need to convert floating-point numbers into integers before PP-Stream's deployment. Note that prior studies (e.g., [42], [60]) also address parameter scaling in machine learning and adopt a similar scaling approach as ours (see below), so as to avoid parameter overflows for specific hardware (e.g., programmable switches). In contrast, we apply parameter scaling for compatibility with cryptographic operations in a privacy-preserving setting. As shown in our experiments (Section VI-B), PP-Stream preserves the neural network inference accuracy even though it casts floating-point numbers into integers.

We require the model provider to perform parameter scaling for a neural network; that is, for each model parameter, the model provider first multiplies the model parameter with a *scaling factor* F (which is in power of 10) and rounds the result off to the nearest integer. We then perform neural network inference on the scaled setting (where all parameters are scaled by F to integers). How to choose a proper scaling factor presents a trade-off between accuracy and efficiency: a larger scaling factor achieves higher precision for float-point model parameters, yet it increases the computational burdens and hence degrades the inference performance, as the scalar multiplication operations in Paillier's PHE now need to work on larger numbers (and vice versa for a smaller scaling factor).

We adopt a parameter scaling approach to preserve the accuracy of neural network inference, while limiting the computational overhead. We perform the following three steps. In Step 1, for a given neural network model, we test the inference accuracy (denoted by A) using the training set; here, we define the inference accuracy as $(TP+TN)/(TP+TN+FP+FN)$, where TP, TN, FP, and FN are the numbers of true positives, true negatives, false positives, and false negatives, respectively. In Step 2, we round each model parameter to f decimal places, starting from $f=0$, to extract a new (approximate) model with the rounded parameters. We then test the new model's accuracy (denoted by A') using the same training set. If A and A' differ by less than a pre-specified threshold (0.01% in our default setting) or f hits a maximum limit (6 in our case), we choose the value of f and proceed to Step 3; otherwise, we increment f by one and repeat Step 2. In Step 3, we select the scaling factor F as $F=10^f$. Note that we bound f to a maximum limit in Step 2 to avoid operating on very large numbers, while keeping the inference accuracy after scaling close to that before scaling as much as possible.

B. Operation Encapsulation

Figure 4 illustrates the architecture of PP-Stream, which performs privacy-preserving neural network inference into pipelined *stages*. We first map each hidden layer (i.e., a linear, non-linear, or mixed layer, as defined in Section II-A) into a *primitive layer* that contains either only linear operations or only non-linear operations. Specifically, we map each linear (resp. non-linear) layer into a linear (resp. non-linear) primitive layer, and decompose each mixed layer into a linear primitive layer and a non-linear primitive layer. PP-Stream then encapsulates the primitive layers into pipelined stages.

There are two extreme approaches of encapsulating primitive layers into stages, both of which should be avoided. One

TABLE II: Notations used in resource allocation.

Symbol	Meaning
$\mathcal{N}$	merged neural network
ℓ	number of primitive layers in $\mathcal{N}$
L_i	i-th primitive layer in $\mathcal{N}$, $1 \le i \le \ell$
I_i	variable indicating i-th primitive layer in $\mathcal{N}$ belongs to linear or non-linear category, $1 \le i \le \ell$
s	number of servers for allocation
S_j	i-th server for allocation, $1 \le j \le s$
c_j	number of CPU cores in server S_j, $1 \le j \le s$
T_i	CPU time L_i in $\mathcal{N}$ requires, $1 \le i \le \ell$
$x_{i,j}$	0-1 variable indicating whether L_i in $\mathcal{N}$ deploys on server S_j, $1 \le i \le \ell$, $1 \le j \le s$
y_i	number of allocated threads of L_i in $\mathcal{N}$, $1 \le i \le \ell$

extreme is to encapsulate each primitive layer into a single stage. However, it can introduce substantial serialization and deserialization overhead to pass the data across stages. Another extreme is to encapsulate all primitive layers into a single stage. However, it destroys privacy preservation (Section II-B), as the linear and non-linear primitive layers need to be executed by the model and data providers, respectively.

To this end, PP-Stream merges the adjacent primitive layers of the same type (i.e., linear or non-linear) of a given input neural network into a *merged* primitive layer. It deploys each merged primitive layer into a single stage. Thus, the linear and non-linear merged primitive layers are executed in alternate pipelined stages, as shown in Figure 4. For brevity, we refer to a "merged primitive layer" simply as a "primitive layer" in the following discussion.

C. Load-balanced Resource Allocation

Given a merged neural network, PP-Stream assigns each primitive layer into a single stage that runs on a server. Note that a server may have one or multiple primitive layers of the same type (i.e., linear or non-linear primitive layer). Within each server, PP-Stream performs multi-threading by allocating a number of threads to each primitive layer, subject to the available resources. Note that PP-Stream performs resource allocation *offline* before the start of processing inference requests. Since the computational load of each primitive layer depends on the neural network model, which is static, it can be determined in advance through offline profiling (see below).

Notation. We first define the notations. Consider a merged neural network $\mathcal{N}$ with ℓ primitive layers $L_1, L_2, \cdots, L_\ell$. Let I_i ($1 \le i \le \ell$) be an indicator variable, where $I_i = 1$ if L_i is a linear primitive layer, or $I_i = -1$ if L_i is a non-linear primitive layer. Let s be the total number of servers $S_1, S_2, \cdots, S_s$, and c_j ($1 \le j \le s$) be the number of CPU cores in server S_j. Also, let T_i ($1 \le i \le s$) be the time for running L_i, obtained through offline profiling. Finally, let $x_{i,j}$ ($1 \le i \le \ell$, $1 \le j \le s$) be a 0-1 indicator variable, where $x_{i,j} = 1$ if L_i is deployed in S_j, and y_i ($1 \le i \le \ell$) be the number of threads allocated to L_i.

Objective. Our high-level goal is to evenly distribute the computational resources across primitive layers. Since each primitive layer has varying computational loads, we aim to balance the execution times of all primitive layers, so that no primitive layer becomes the bottleneck. In this work, we consider a homogeneous setting and assume that each server has identical processing capacities, meaning that the load-balanced allocation across threads can imply load balancing at a system-wide level; a heterogeneous setting across servers is posed as our future work.

Before we solve the load balancing problem, we first profile T_i (i.e., the time to run the primitive layer L_i) offline, which can be viewed as part of the training phase. Specifically, for a given merged neural network $\mathcal{N}$, we run each of its primitive layers in a single process (i.e., a stage). We feed an input tensor into $\mathcal{N}$ and measure the execution time of each primitive layer L_i. We repeat the measurement for 100 input tensors randomly selected from the training set of $\mathcal{N}$ and obtain the average execution time for L_i.

We formulate our load-balanced resource allocation problem as an integer linear programming (ILP) problem. Our optimization objective is to find $x_{i,j}$ and y_i ($1 \le i \le \ell$ and $1 \le j \le s$) that solve the following problem:

$$\min \sum_{i=1}^{\ell} \sum_{i'=1}^{\ell} \left| \frac{T_i}{y_i} - \frac{T_{i'}}{y_{i'}} \right|, \tag{4}$$

subject to the following constraints:

$$\sum_{j=1}^{s} x_{i,j} = 1 \qquad \text{for each } i \in [1,\ell], \tag{5}$$

$$\left|\sum_{i=1}^{\ell} (x_{i,j} \times I_i)\right| = \sum_{i=1}^{\ell} x_{i,j} \qquad \text{for each } j \in [1,s], \tag{6}$$

$$y_i \ge 1 \qquad \text{for each } 1 \le i \le \ell, \tag{7}$$

$$\sum_{i=1}^{\ell} (x_{i,j} \times y_i) \le c_j \times 2 \qquad \text{for each } 1 \le j \le s. \tag{8}$$

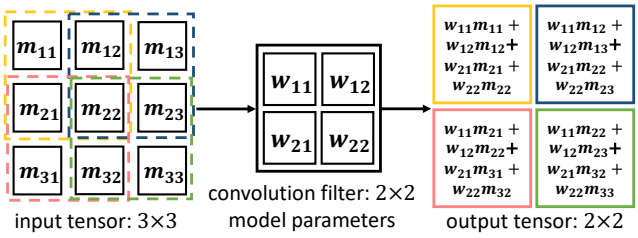
The objective function (Equation (4)) is to minimize the sum of absolute differences in execution times across all pairs of primitive layers, where T_i/y_i represents the average execution time per thread. Other objective functions (e.g., minimizing the maximum difference of execution times of a pair of primitive layers) are also applicable.

The optimization problem is subject to the following four constraints. The first constraint (Equation (5)) ensures that each primitive layer L_i is assigned to only one server. The second constraint (Equation (6)) ensures that each server S_j deploys either all linear primitive layers (i.e., $I_i = 1$ for all $x_{i,j} = 1$) or all non-linear primitive layers (i.e., $I_i = -1$ for all $x_{i,j} = 1$), so as to satisfy privacy preservation. The third constraint (Equation (7)) ensures that each primitive layer is assigned at least one thread. The final constraint (Equation (8)) is that the total number of threads assigned to the primitive layers in each server does not exceed the available CPU resources. Here, if hyper-threading is enabled, at most two threads can simultaneously run on each physical CPU core.

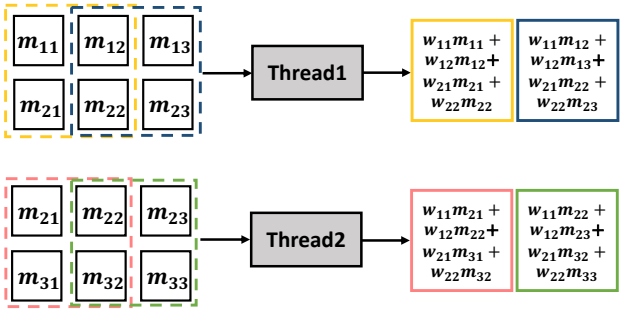
The above ILP problem can be solved with the branch-and-bound method [6]. Note that the solving process is performed offline (within a few hours on a commodity machine), so the cost of problem solving is acceptable in practice.

D. Tensor Partitioning

After solving the load-balanced resource allocation problem, we need to determine how to distribute the computation tasks



(a) Convolution on a 3×3 tensor



(b) Convolution with tensor partitioning

Fig. 5: Tensor partitioning.

across threads. A straightforward approach is to feed an input tensor directly to each thread, which produces one element of the output tensor at a time. Our observation is that the operations for some neural network layers can be decomposed into partial operations that can be parallelized across threads, where each partial operation only requires an input of part of the input tensor (i.e., sub-tensor). In this case, we can send only a sub-tensor, instead of an entire tensor, to a thread, thereby mitigating the communication overhead.

We explain the above idea via an example of a convolution layer (a common type of neural network operations), as shown in Figure 5(a). Suppose that the convolution layer processes a 3×3 input tensor (whose elements are denoted by m_{ij} for $1 \leq i \leq 3$ and $1 \leq j \leq 3$) and produces a 2×2 output tensor, with a 2×2 filter (whose elements are denoted by w_{ij} for $1 \leq i \leq 2$ and $1 \leq j \leq 2$), a stride of 1, and no padding. The input tensor is divided into four sub-tensors, each of which is multiplied with the filter to produce an element of the output tensor. Note that each element of the output tensor only requires a sub-tensor as input.

Based on the above example, we propose a tensor partitioning approach that feeds only the input sub-tensors into each thread and determines how each thread generates output sub-tensors. Specifically, suppose that a primitive layer L_i is allocated with y_i threads. PP-Stream first determines the number of elements in the output tensor and evenly partitions these elements across all y_i threads (i.e., each thread produces a fraction $1/y_i$ of all the elements in the output tensor). It then partitions an input tensor into sub-tensors and distributes only the elements of the required sub-tensors into each thread. For example, Figure 5(b) shows the tensor partitioning design, in which the primitive layer is assigned two threads and each thread only receives six (out of nine in total) elements of the input tensor to generate two (out of four in total) elements of the output tensor.

It is worth noting that output tensor partitioning can also be employed for fully-connected operations, while input tensor partitioning can only be applied for convolution operations in neural networks as they operate based on the input tensor's local information. For a neural network model that contains convolution operations, PP-Stream employs both input and output tensor partitioning; otherwise, it only performs output tensor partitioning. Furthermore, tensor partitioning can operate with CPU processing, without relying on specialized hardware (e.g., GPU [54] or TPU [4]).

V. Implementation

We build a prototype of PP-Stream in C++ with around 4.6 K LoC. Specifically, we use the GMP library [5] to implement the cryptographic operations in Paillier's PHE [55] based on modular arithmetic, where the key size is set as 2,048 bits due to security considerations [16]. In addition, we implement PP-Stream as a distributed stream processing system based on AF-Stream [36], which performs stream processing across multiple *workers* (i.e., processes) that run on multiple servers. We map each worker in AF-Stream into a stage. Inside each stage, we implement load-balanced resource allocation (Section IV-C) and tensor partitioning (Section IV-D).

VI. Evaluation

We evaluate PP-Stream via testbed experiments on real-world datasets. Our experimental findings include:

- Parameter scaling allows PP-Stream to choose a suitable scaling factor that preserves inference accuracy (defined in Section IV-A) with a low inference latency. (Exp#1)
- By encapsulating operations into a distributed stream processing pipeline, PP-Stream effectively reduces the inference latency than the centralized approach by 96.54%. (Exp#2)
- With load-balanced resourced allocation, PP-Stream reduces the inference latency by up to 64.94%, and the performance gain is higher for larger neural network models. (Exp#3)
- With tensor partitioning, PP-Stream reduces the inference latency by up to 61.64%, and the performance gain is higher when more CPU cores are available. (Exp#4)
- Our obfuscation mechanism efficiently reduces information leakage, especially for larger neural networks. (Exp#5)
- PP-Stream achieves lower inference latency than state-of-the-art systems. (Exp#6)

A. Setup

Testbed. We deploy PP-Stream on a testbed composed of nine servers, each of which is equipped with an Intel Xeon E5-2630 2.20 GHz 24-core CPU, 256 GB memory, and an Intel 82599ES 10 Gbps Ethernet NIC. We assign the stages in PP-Stream to CPU cores across these nine servers.

Datasets and models. We evaluate PP-Stream on nine neural network models from MNIST [10], CIFAR-10 [3] and healthcare datasets [1], [2], [7], as summarized in Table III; such datasets are widely used in the literature on privacy-preserving neural network inference (e.g., [20], [31], [49], [56]). Specifically, the MNIST dataset [10] consists of a collection

TABLE III: Description of datasets and models.

Dataset	Model	# Samples		# Servers
		Train	Test	Model / Data
Breast [1]	3FC	456	113	2 / 1
Heart [7]	3FC	820	205	2 / 1
Cardio [2]	3FC	60,000	10,000	2 / 1
MNIST-1 [10]	3FC	60,000	10,000	2 / 1
MNIST-2 [10]	1Conv+2FC	60,000	10,000	2 / 1
MNIST-3 [10]	2Conv+2FC	60,000	10,000	2 / 2
CIFAR-10-1 [3]	VGG13 [61]	50,000	10,000	6 / 3
CIFAR-10-2 [3]	VGG16 [61]	50,000	10,000	6 / 3
CIFAR-10-3 [3]	VGG19 [61]	50,000	10,000	6 / 3

of 70,000 grayscale images of handwritten digits from 0 to 9 with 28×28 pixels each. The CIFAR-10 dataset [3] consists of 60,000 32×32 color images (with $32 \times 32 \times 3$ pixels each) in 10 classes with 6,000 images each. The breast cancer (Breast) dataset [1] consists of 569 instances with 30 features each, where each instance includes a binary target variable indicating if a tumor is malignant or benign. The heart disease (Heart) dataset [7] contains 1,025 samples with 13 features, in which each sample includes a binary target variable that indicates if the patient has heart disease. The cardio disease (Cardio) dataset [2] includes 70,000 records with 11 features each, and its target variable indicates the presence of cardiovascular disease.

Each dataset is trained with a neural network model, including 3FC (three fully-connected layers), 1Conv+2FC (one convolution layer and two fully-connected layers), 2Conv+2FC (two convolution layers and two fully-connected layers), and VGG13/16/19 [61]; note that all models are commonly used in the computing vision domain. The models are trained by Matlab for neural networks [9] and PyTorch [12], and are consistent with those used in prior studies [31], [45], [49], [51], [56]–[58]. Furthermore, we deploy different numbers of servers for the model and data providers, as shown in Table III.

Due to the insufficient CPU resources in our testbed, we unfortunately cannot evaluate PP-Stream for larger models (e.g., ResNet50 [32] and DenseNet121 [35]), and we pose such evaluation as future work. We conjecture that PP-Stream still maintains high inference performance for larger models due to its distributed stream processing design.

Metrics. Our experiments focus on the following two metrics: the *inference accuracy* (defined in Section IV-A) and the *inference latency* (defined as the execution time of processing an inference request). Note that for the inference accuracy, our goal is not to improve the accuracy of the original model; instead, we focus on mitigating the difference of the inference accuracy before and after parameter scaling (Section IV-A).

B. Results

(Exp#1) Scaling factors. We first evaluate the efficiency of our parameter scaling approach (Section IV-A) and how different scaling factors affect the inference accuracy.

Table IV shows the results for different scaling factors on the nine models; the rightmost column also shows the inference accuracy on the training set without parameter scaling (the original case). In general, the inference accuracy increases with the scaling factor. However, for some models, the inference accuracy under a small scaling factor can match the inference accuracy without parameter scaling. For example, the scaling factor can be selected as 10^4 for Cardio, MNIST-2, MNIST-3, and CIFAR-10-2. Note that the inference accuracy for Breast (slightly) deviates from the inference accuracy without parameter scaling by more than 0.01% due to the small size of the training set, the deviation is limited and does not compromise the testing accuracy (see below).

TABLE IV: (Exp#1) Inference accuracy versus scaling factor on the training set.

Model	Accuracy tested by training set (%)							Original
	10^0	10^1	10^2	10^3	10^4	10^5	10^6	(%)
Breast	59.21	59.21	98.24	98.24	98.24	98.24	**98.46**	98.68
Heart	47.93	64.51	99.27	99.51	99.51	99.51	**99.76**	99.76
Cardio	50.09	55.93	71.12	71.48	**71.52**	-	-	71.52
MNIST-1	9.87	9.75	9.75	97.99	98.09	**98.10**	-	98.10
MNIST-2	9.87	10.22	67.80	67.80	**98.24**	-	-	98.24
MNIST-3	10.44	11.24	43.81	96.32	**96.51**	-	-	96.51
CIFAR-10-1	10.00	37.49	90.25	91.05	91.05	91.05	**91.06**	91.06
CIFAR-10-2	10.00	19.46	82.72	92.43	**92.45**	-	-	92.45
CIFAR-10-3	10.00	16.35	31.18	92.09	92.67	**92.68**	-	92.68

(Note: the bold numbers correspond to the selected scaling factors)

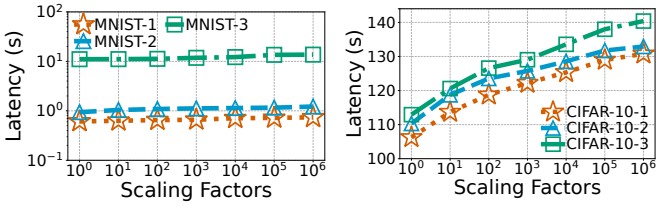
TABLE V: (Exp#1) Inference accuracy versus scaling factor on the testing set.

Model	Accuracy tested by testing set (%)							Original
	10^0	10^1	10^2	10^3	10^4	10^5	10^6	(%)
Breast	70.34	70.34	96.46	96.46	96.46	96.46	**97.34**	97.34
Heart	50.74	65.86	97.57	98.06	98.06	98.54	**98.54**	98.54
Cardio	51.12	56.28	71.07	71.46	**71.46**	71.46	71.46	71.46
MNIST-1	9.80	9.74	9.74	96.37	96.54	**96.54**	96.54	96.54
MNIST-2	9.80	10.10	10.10	67.98	**97.38**	97.38	97.38	97.38
MNIST-3	10.28	11.35	44.36	96.60	**96.76**	96.76	96.76	96.76
CIFAR-10-1	10.00	37.06	83.95	84.65	84.66	84.66	**84.66**	84.66
CIFAR-10-2	10.00	19.51	77.91	85.99	**85.99**	85.99	85.99	85.99
CIFAR-10-3	10.00	16.57	29.91	86.10	86.39	**86.39**	86.39	86.39

(Note: the bold numbers correspond to the selected scaling factors)

To prove that the selected scaling factors can preserve the inference accuracy in practice, we evaluate the inference accuracy for different scaling factors using the testing set. For each model, we first perform parameter scaling based on different scaling factors (from 10^0 to 10^6) to extract the corresponding approximate models. We then evaluate the inference accuracy using the testing set.

Table V shows the results; the rightmost column shows the inference accuracy on the testing set without parameter scaling (the original case). The inference accuracy is significantly worse than that of the original case for a small scaling factor (e.g., 10^0), yet it increases with the scaling factor. We observe that for



(a) MNIST Models (b) CIFAR-10 Models

Fig. 6: (Exp#1) Scaling factors.

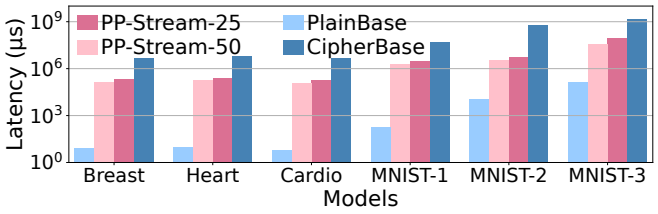
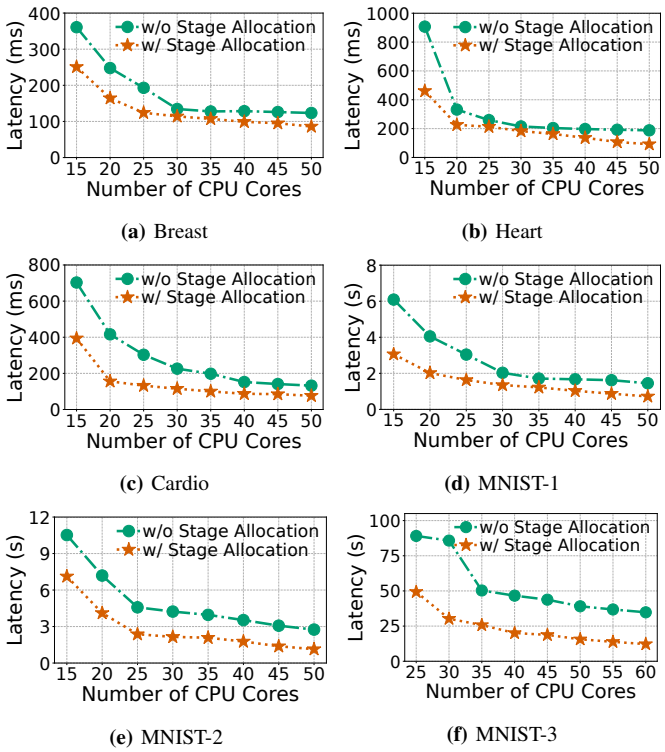
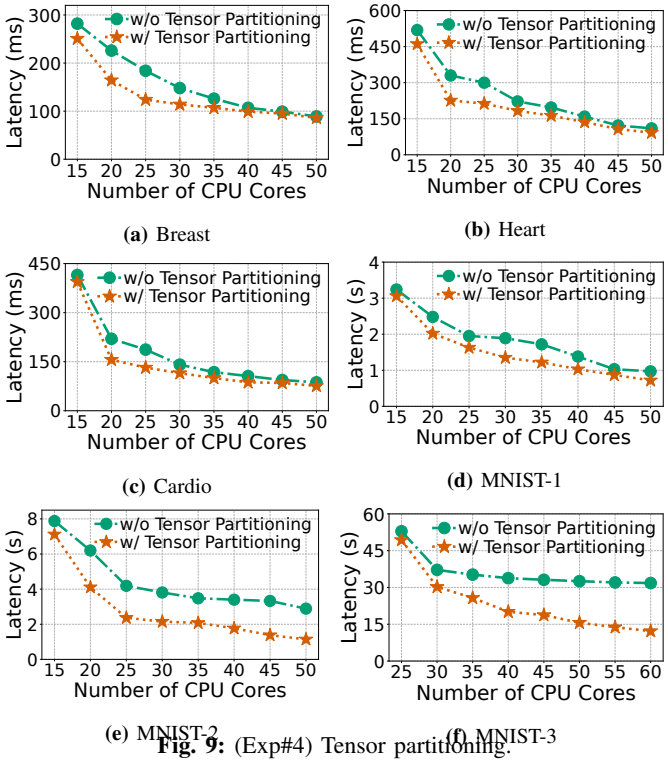


Fig. 8: (Exp#2) Effectiveness of distributed stream processing.



(a) Breast (b) Heart (c) Cardio (d) MNIST-1 (e) MNIST-2 (f) MNIST-3

Fig. 7: (Exp#3) Load-balanced resource allocation.



(a) Breast (b) Heart (c) Cardio (d) MNIST-1 (e) MNIST-2 (f) MNIST-3

Fig. 9: (Exp#4) Tensor partitioning.

all nine models, the selected scaling factors by our parameter scaling approach preserve the same inference accuracy in the testing set as in the original case without parameter scaling. It is noting that PP-Stream applies scaling factors selected by our parameter scaling scheme for all the models in the following experiments (i.e., Exp#2-4, and 6).

We also evaluate the inference latency of PP-Stream for different scaling factors. We focus on the MNIST and CIFAR-10 models, as the healthcare models have small scales and cannot show performance differences for different scaling factors. In our evaluation, we enable all features of PP-Stream (i.e., operation encapsulation, load-balanced resource allocation, and tensor partitioning).

Figure 6 shows the inference latency results versus the scaling factor. The inference latency generally increases with the scaling factor since larger scaling factors will introduce more homomorphic encryption operations (i.e., scalar multiplication). For example, when the scaling factor increases from 10^0 to 10^6, the inference latency increases by 29% and 23% in MNIST and CIFAR-10 models, respectively. The results show the performance-accuracy trade-off in neural network inference.

(Exp#2) Effectiveness of distributed stream processing. We evaluate how PP-Stream reduces the inference latency by mapping the primitive layers into distributed stream processing. We compare four variants: (i) *PP-Stream-25*, which runs distributed stream processing and performs inference on ciphertexts on 25 CPU cores in total; (ii) *PP-Stream-50*, which runs distributed stream processing and performs inference on ciphertexts on 50 CPU cores in total; (iii) *PlainBase*, which runs as a centralized system on a single server and performs inference on plaintexts; and (iv) *CipherBase*, which runs as a centralized system on a single server and performs inference on ciphertexts. For PP-Stream-25 and PP-Stream-50, we disable load-balanced resource allocation and tensor partitioning; instead, for a given total number of CPU cores and a neural network model, PP-Stream evenly distributes the CPU cores across the stages (note that some stages may have one more CPU core than the others if the number of CPU cores is not divisible by the number of servers) and feeds an input tensor to each thread to generate one element of the output tensor at a time.

Figure 8 shows the inference latency results; we only show the results for the MNIST and healthcare models in the interest of space. We make the following observations. First,

comparing PlainBase and CipherBase, CipherBase suffers from a significantly higher inference latency than PlainBase. For example, for the MNIST-3 model, the inference latency of PlainBase is 0.13 s only, while that of CipherBase is 1,461.09 s. This implies that privacy-preserving mechanisms incur severe performance overhead. For a larger scale of the model, the performance overhead of privacy-preserving mechanisms is more severe. Second, comparing PP-Stream-25/50 and CipherBase, PP-Stream's inference latency is one order of magnitude less than CipherBase's. By mapping primitive layers into distributed stream processing, PP-Stream-25 and PP-Stream-50 reduce the inference latency of CipherBase on average by 95.63% and 97.46%, respectively. This shows the significant performance gain through distributed stream processing. Third, PP-Stream-50 reduces the inference latency of PP-Stream-25 on average by 39.24%. This shows that PP-Stream can achieve a higher performance gain with more available CPU cores.

(Exp#3) Load-balanced resource allocation. We evaluate PP-Stream with and without load-balanced resource allocation (Section IV-C) by varying the total number of CPU cores in the whole system. For the case without load-balanced resource allocation, we configure PP-Stream to evenly distribute the CPU cores to each stage; for the case with load-balanced resource allocation, we evenly distribute the CPU cores to all allocated servers, and assign the CPU cores to different stages by solving the optimization problem formulated in Section IV-C. In both cases, we enable distributed stream processing and tensor partitioning during runtime.

Figure 7 shows the inference latency results; here, we focus on the healthcare and MNIST models. For each model, load-balanced resource allocation reduces the inference latency by around 42.55% on average. In particular, the maximum reduction is 64.94%, which occurs for the MNIST-3 model. The reason is that the MNIST-3 model has the largest scale among the six models and it benefits the most from load-balanced resource allocation. Thus, we expect that load-balanced resource allocation is more effective for larger models in reducing the inference latency. Furthermore, we observe that assigning more CPU cores improves inference performance, yet there exists a diminishing return as we assign more CPU cores. While it is intriguing to explore and characterize such a trade-off, it is beyond the scope of our paper and is posed as our future work.

(Exp#4) Tensor partitioning. We evaluate PP-Stream with and without tensor partitioning (Section IV-D) by varying the total number of CPU cores. For the case without tensor partitioning, we configure PP-Stream to feed the whole input tensor to each thread, which produces one element of the output tensor at a time. For the case with tensor partitioning, each linear stage employs output tensor partitioning, while some linear stages that are responsible for convolution operations also leverage input tensor partitioning. In both cases, we enable distributed stream processing and employ load-balanced resource allocation to distribute given CPU resources.

Figure 9 shows the inference latency results; here, we focus on the healthcare and MNIST models. We have the following two observations. First, when PP-Stream is allocated only a small number of CPU cores, tensor partitioning only marginally reduces the inference latency, as the bottleneck is primarily due to limited CPU resources. As the number of CPU cores increases, tensor partitioning shows higher performance gains, as it can now partition an input tensor into smaller tensors that are sent to more threads and each thread generates a smaller output sub-tensor during inference (i.e., each thread has less load now). We see that tensor partitioning reduces the inference latency by up to 61.64%. Second, tensor partitioning reduces more latency in MNIST-2 and MNIST-3 models than that in healthcare and MNIST-1 models. This discrepancy arises because the healthcare and MNIST-1 models lack convolution operations, limiting their utilization of tensor partitioning to solely focusing on output tensor partitioning.

TABLE VI: (Exp#5) Information leakage measurement.

Tensor Length	Distance	Tensor Length	Distance
2^5	0.2898	2^{10}	0.0566
2^6	0.1767	2^{11}	0.0405
2^7	0.1232	2^{12}	0.0289
2^8	0.1087	2^{13}	0.0200
2^9	0.0783	-	-

(Exp#5) Information leakage measurement. Recall that we use the obfuscation mechanism to protect model parameters (Figure 3). However, PP-Stream still has some information leakage since the obfuscation mechanism only re-orders the tensors instead of changing their values. Here, we quantify the information leakage by *distance correlation* [63], [67], which has been widely used in the theory of statistics. Specifically, for each model in Table III, we run the privacy-preserving inference phase in the testing sets and export all the tensors that need to be obfuscated. The tensor length (i.e., the number of values in a tensor) varies from 2^5 to 2^{13} for all exported tensors. Then, we perform obfuscation on these tensors and measure the distance correlation between the before-obfuscated and after-obfuscated tensors via Python Dcor [11]. A smaller distance correlation implies that the two tensors are more different; in particular, the distance correlation is one if two tensors are identical, while it is zero if they are totally different.

Table VI shows the distance correlation results between the before-obfuscated and after-obfuscated tensors during the inference for all models in Table III. Here, we only show the results versus the tensor length, as our observation indicates that when two before-obfuscated tensors have an equal number of values, the distance correlation results remain nearly identical with differences less than 0.1%.

We make the following observations. First, our obfuscation mechanism effectively reduces information leakage. The highest recorded distance correlation is 0.2898, which is significantly lower from one (i.e., without obfuscation). We believe that such a distance correlation is weak, meaning that the obfuscated tensors are weakly correlated with the non-obfuscated tensors. Second, as the tensor becomes larger, the distance correlation becomes smaller, meaning that larger tensors become more different after obfuscation. This shows that our obfuscation mechanism shows less information leakage for larger neural

TABLE VII: (Exp#6) Comparisons with state-of-the-arts.

	Inference latency (s)		
	MNIST-1	**MNIST-2**	**MNIST-3**
SecureML [51]	4.88*	-	-
CryptoNets [31]	-	297.5*	-
CryptoDL [33], [34]	-	320*	-
EzPC [24]	2.42	2.92	25.66
PP-Stream (ours)	0.72	1.14	12.20

(Note: $*$ means the numbers are reported in corresponding papers)

networks. Since neural networks typically have large tensors (e.g., our CIFAR-10 models have a tensor length of up to 2^{13}), our obfuscation mechanism maintains high privacy in practice.

(Exp#6) Comparisons with state-of-the-arts. We show that PP-Stream achieves a low latency compared with state-of-the-art systems, including SecureML [51], CryptoNets [31], CryptoDL [33], [34], and EzPC [24] on privacy-preserving neural network inference. For SecureML, CryptoNets, and CryptoDL, since their artifacts are not publicly available, we compare the latency based on the numbers reported in their respective publications using the same neural network models (i.e., the MNIST-1 and MNIST-2 models); note that such high-level comparisons are also adopted in prior studies [45], [56]. For EzPC, we deploy its open-source system in our testbed and reproduce the three MNIST models to compare the performance of PP-Stream and EzPC in the same evaluation environment.

Table VII shows the inference results. We first consider SecureML, CryptoNets, and CryptoDL. For the MNIST-1 model with a scaling factor 10^5, the inference latency of PP-Stream is 0.72 s, while the inference latency of SecureML [51] is 4.88 s on two Amazon EC2 `c4.8xlarge` instances with 60 GB of RAM each; for the MNIST-2 model with a scaling factor of 10^4, the inference latency of PP-Stream is 1.14 s, while the inference latency of CryptoNets [31] is 297.5 s on a server with a single Intel Xeon E5-1620 3.5 GHz CPU and 16 GB of RAM, and that of CryptoDL [33], [34] is 320 s on a virtual machine with 48 GB of RAM and 12 CPU cores. Thus, PP-Stream achieves high inference performance.

We further compare PP-Stream and EzPC. The inference latencies of EzPC are 2.42 s, 2.92 s, and 25.66 s for the MNIST-1, MNIST-2, and MNIST-3 models, respectively (i.e., 236%, 156%, and 110% higher than those of PP-Stream, respectively). PP-Stream outperforms EzPC for two reasons. First, EzPC suffers from its high protocol transition overhead due to the frequent switching between secret sharing and garbled circuits. Second, PP-Stream achieves high inference performance by pipelining the processing of inference requests in a streaming fashion, which is not readily done in EzPC as its secret sharing and garbled circuits require multiple rounds of interactions for each layer of neural network inference.

VII. Related Work

Prior studies address privacy preservation in neural network inference. Earlier privacy-preserving neural network inference systems [18], [52] support non-linear operations, but they do not preserve the correctness of the inference results (Section II-C). CryptoNets [31], CryptoDL [33], [34] and E2DM [38] build on homomorphic encryption as in PP-Stream, yet they suffer from accuracy loss and need to retrain models as they replace the ReLU function with approximated polynomials. In contrast, PP-Stream maintains the correctness of the inference results and preserves model accuracy.

Some studies build neural network inference frameworks through hybrid approaches. MiniONN [45] and SecureML [51] build on homomorphic encryption, garbled circuits, and secret sharing. Specifically, MiniONN [45] uses approximately fitting to replace activation functions, while SecureML [51] uses special activation functions that may not support general neural network functions (e.g., ReLU). EzPC [24] builds on garbled circuits and secret sharing. Since the formats vary across cryptographic protocols, the transitioning across the protocols in the above systems [24], [45], [51] incurs high performance overhead. DELPHI [49] proposes a planner that makes a decision for ReLU or approximated polynomials, which has the same limitations as SecureML in terms of generality. ABY^3 [50] builds on a three-server model that involves a trusted third party, while Chameleon [57] relies on a trusted execution environment. QuantizedNN [15], FLEXBNN [28], and Force [26] build on non-colluding three-party or four-party computation frameworks. In contrast, PP-Stream addresses the above limitations by supporting general neural network functions and untrusted environments, without relying on specialized settings such as trusted execution environments.

Some studies address the performance issues in neural network inference. For example, there exist hardware acceleration approaches based on GPU [54] and TPU [4], but they do not address privacy preservation. FALCON [46] uses Fast Fourier Transform to accelerate secure linear operations in neural networks. Cheetah [37] builds on lattice-based homomorphic encryption for fast performance. In contrast, PP-Stream is readily deployable in commodity servers.

VIII. Conclusion

PP-Stream is a distributed stream processing system for privacy-preserving neural network inference. It decomposes the inference workflow into linear and non-linear operations, and protects them with hybrid privacy-preserving techniques. It also maps the inference operations into a distributed stream processing architecture, and further performs load-balanced resource allocation and tensor partitioning for better resource utilization. Our evaluation shows that our PP-Stream prototype effectively supports widely used neural network models with high inference accuracy and low inference latencies.

Acknowledgements: This work was supported by National Key R&D Program of China (2019YFB1802600), National Natural Science Foundation of China (62172007, 62090022, and 62172388), Youth Innovation Promotion Association of Chinese Academy of Sciences (2020105), and Research Grants Council of Hong Kong (AoE/P-404/18). The corresponding author is Qun Huang.

REFERENCES

[1] Breast cancer. https://www.kaggle.com/uciml/breast-cancer-wisconsin-data.

[2] Cardio vascular disease. https://www.kaggle.com/sulianova/cardiovascular-disease-dataset.

[3] CIFAR-10 dataset. https://www.kaggle.com/datasets/pankrzysiu/cifar10-python.

[4] Cloud TPU. https://cloud.google.com/tpu.

[5] The GNU multiple precision arithmetic library. https://gmplib.org/.

[6] Gurobi optimizer. http://www.gurobi.com.

[7] Heart disease. https://www.kaggle.com/johnsmith88/heart-disease-dataset.

[8] Kuna AI. https://getkuna.com/blogs/news/2017-05-24-introducingkuna-ai.

[9] Matlab deep learning toolbox. https://ww2.mathworks.cn/products/deep-learning.html.

[10] The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[11] Python dcor. https://pypi.org/project/dcor/.

[12] PyTorch. https://pytorch.org/.

[13] Wyze: contact and motion sensors for your home. https://www.wyze.com.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: a system for large-scale machine learning. In *Proc. of USENIX OSDI*, pages 265–283, 2016.

[15] A. Barak, D. Escudero, A. P. Dalskov, and M. Keller. Secure evaluation of quantized neural networks. *IACR Cryptol*, page 131, 2019.

[16] E. Barker, E. Barker, W. Burr, W. Polk, M. Smid, et al. *Recommendation for key management: Part 1: General*. 2006.

[17] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In *ESORICS*, pages 424–439, 2009.

[18] M. Barni, C. Orlandi, and A. Piva. A privacy-preserving protocol for neural-network-based computation. In *Proc. of ACM MM&Sec*, pages 146–151, 2006.

[19] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*, pages 190–209, 2011.

[20] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame. MP2ML: a mixed-protocol machine learning framework for private inference. In *Proc. of ACM ARES*, pages 1–10, 2020.

[21] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proc. of ACM CF*, pages 3–13, 2019.

[22] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO*, pages 483–512, 2018.

[23] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha. Low latency privacy preserving inference. In *ICML*, pages 812–821, 2019.

[24] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. EzPC: programmable and efficient secure two-party computation for machine learning. In *Proc. of IEEE EuroS&P*, pages 496–511, 2019.

[25] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei. Faster CryptoNets: leveraging sparsity for real-world encrypted inference. *arXiv*, 2018.

[26] T. Dai, L. Duan, Y. Jiang, Y. Li, F. Mei, and Y. Sun. Force: making 4pc$>$ 4$\times$ pc in privacy preserving machine learning on GPU. *Cryptology*, 2023.

[27] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proc. of ACM PLDI*, pages 142–156, 2019.

[28] Y. Dong, X. Chen, X. Song, and K. Li. FLEXBNN: fast private binary neural network inference with flexible bit-width. *IEEE Trans. on Information Forensics and Security*, 2023.

[29] D. Evans, Y. Huang, J. Katz, and L. Malka. Efficient privacy-preserving biometric identification. In *Proc. of USENIX NDSS*, volume 68, pages 90–98, 2011.

[30] R. Fakoor, F. Ladhak, A. Nazi, and M. Huber. Using deep learning to enhance cancer diagnosis and classification. In *ICML*, volume 28, pages 3937–3949, 2013.

[31] 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*, pages 201–210, 2016.

[32] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*, pages 770–778, 2016.

[33] E. Hesamifard, H. Takabi, and M. Ghasemi. CryptoDL: deep neural networks over encrypted data. *arXiv*, 2017.

[34] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright. Privacy-preserving machine learning as a service. *Privacy Enhancing Technologies*, 2018(3):123–142.

[35] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proc. of IEEE CVPR*, pages 4700–4708, 2017.

[36] Q. Huang and P. P. Lee. Toward high-performance distributed stream processing via approximate fault tolerance. *Proc. of the VLDB Endowment*, 10(3):73–84, 2016.

[37] Z. Huang, W.-j. Lu, C. Hong, and J. Ding. Cheetah: lean and fast secure two-party deep neural network inference. In *In Proc. of USENIX Security*, 2022.

[38] X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. In *Proc. of ACM CCS*, pages 1209–1222, 2018.

[39] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: a low latency framework for secure neural network inference. In *Proc. of USENIX Security*, pages 1651–1669, 2018.

[40] B. Kacsmar, K. Tilbury, M. Mazmudar, and F. Kerschbaum. Caring about sharing: user perceptions of multiparty data sharing. In *S&P*, 2022.

[41] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: secure tensorflow inference. In *S&P*, pages 336–353, 2020.

[42] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. M. Swift. ATP: in-network aggregation for multi-tenant learning. In *Proc. of USENIX NSDI*, pages 741–761, 2021.

[43] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: a convolutional neural-network approach. *IEEE Trans. on Neural Networks*, 8(1):98–113, 1997.

[44] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[45] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proc. of ACM CCS*, pages 619–631, 2017.

[46] Q. Lou, W.-j. Lu, C. Hong, and L. Jiang. FALCON: fast spectral inference on encrypted data. *NeurIPS*, 33:2364–2374, 2020.

[47] Z. Ma, Y. Liu, X. Liu, J. Ma, and K. Ren. Lightweight privacy-preserving ensemble classification for face recognition. *IEEE Internet of Things*, 6(3):5778–5790, 2019.

[48] D. Malathi, R. Logesh, V. Subramaniyaswamy, V. Vijayakumar, and A. K. Sangaiah. Hybrid reasoning-based privacy-aware disease prediction support system. *Computers & Electrical Engineering*, 73:114–127, 2019.

[49] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. DELPHI: a cryptographic inference service for neural networks. In *Proc. of USENIX Security*, pages 2505–2522, 2020.

[50] P. Mohassel and P. Rindal. ABY^3: a mixed protocol framework for machine learning. In *Proc. of ACM CCS*, pages 35–52, 2018.

[51] P. Mohassel and Y. Zhang. SecureML: a system for scalable privacy-preserving machine learning. In *Proc. of IEEE S&P*, pages 19–38, 2017.

[52] C. Orlandi, A. Piva, and M. Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007:1–11.

[53] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. Scifi-a system for secure face identification. In *S&P*, pages 239–254, 2010.

[54] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proc. of IEEE*, 96(5):879–899, 2008.

[55] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[56] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *Proc. of USENIX Security*, pages 1501–1518, 2019.

[57] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: a hybrid secure computation framework for machine learning applications. In *Proc. of ACM AsiaCCS*, pages 707–721, 2018.

[58] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. DeepSecure: scalable provably-secure deep learning. In *Proc. of ACM DAC*, pages 1–6, 2018.

[59] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, pages 229–244, 2009.

[60] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *Proc. of USENIX NSDI*, 2021.

[61] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.

[62] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: the all convolutional net. *ICLR*, 2015.

[63] G. J. Székely, M. L. Rizzo, and N. K. Bakirov. Measuring and testing dependence by correlation of distances. 2007.

[64] F. Tramer and D. Boneh. Slalom: fast, verifiable and private execution of neural networks in trusted hardware. 2019.

[65] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-party secure computation for neural network training. *Proc. on Privacy Enhancing Technologies*, 2019(3):26–49.

[66] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu. BatchCrypt: efficient homomorphic encryption for cross-silo federated learning. In *Proc. of USENIX ATC*, pages 493–506, 2020.

[67] F. Zheng, C. Chen, X. Zheng, and M. Zhu. Towards secure and practical machine learning via secret sharing and random permutation. *Knowledge-Based Systems*, 2022.