# Privacy-Preserving Hierarchical Model-Distributed Inference

Fatemeh Jafarian Dehkordi
*University of Illinois Chicago*
fjafar3@uic.edu

Yasaman Keshtkarjahromi
*Seagate Technology*
yasaman.keshtkarjahromi@seagate.com

Hulya Seferoglu
*University of Illinois Chicago*
hulya@uic.edu

***Abstract*—This paper focuses on designing a privacy-preserving Machine Learning (ML) inference protocol for a hierarchical setup, where clients own/generate data, model owners (cloud servers) have a pre-trained ML model, and edge servers perform ML inference on clients' data using the cloud server's ML model. Our goal is to speed up ML inference while providing privacy to both data and the ML model. Our approach (i) uses model-distributed inference (model parallelization) at the edge servers and (ii) reduces the amount of communication to/from the cloud server. Our privacy-preserving hierarchical model-distributed inference, `privateMDI` design uses additive secret sharing and linearly homomorphic encryption to handle linear calculations in the ML inference, and garbled circuit and a novel three-party oblivious transfer are used to handle non-linear functions. `privateMDI` consists of offline and online phases. We designed these phases in a way that most of the data exchange is done in the offline phase while the communication overhead of the online phase is reduced. In particular, there is no communication to/from the cloud server in the online phase, and the amount of communication between the client and edge servers is minimized. The experimental results demonstrate that `privateMDI` significantly reduces the ML inference time as compared to the baselines.**

## I. Introduction

Machine learning (ML) has become a powerful tool for supporting applications such as mobile healthcare, self-driving cars, finance, marketing, agriculture, etc. These applications generate vast amounts of data at the edge, requiring swift processing for timely responses. On the other hand, ML models are getting more complex and larger, so they require higher computation, storage, and memory, which are typically constrained in edge networks but abundant in the cloud. Thus, the typical scenario is that the data owner (at the edge) is geographically separated from the model owner (in the cloud).

The geographically separated nature of data and model owners poses challenges for ML inference as a service. We can naturally use a client/server-based approach where the data owner (client) sends its data to the model owner (cloud server) for ML inference. This approach violates data privacy and introduces communication overhead between the data and model owners, which is usually considered a bottleneck link in today's systems. Very promising privacy-preserving mechanisms have been investigated in the literature [1]–[3], which preserve the privacy of data in the client/server ML inference setup, but still suffer from high communication costs between data and model owners. Such high communication cost undermines the effectiveness of ML inference applications that require less latency and real-time response.

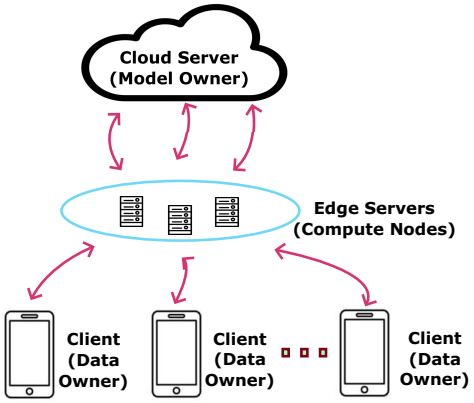



Fig. 1: Hierarchical ML inference.

A promising solution is a hierarchical setup, where clients own/generate data, model owners (cloud servers) have a pre-trained ML model, and compute nodes (edge servers) perform ML inference on clients' data using the cloud server's ML model, Fig. 1. This approach advocates that edge servers perform ML inference by preserving the privacy of data from the client's perspective and ML model from the cloud server's perspective [4]–[6]. Hierarchical ML inference is promising to reduce the communication overhead between the client and cloud server by confining the communication cost between the client and edge servers.

Despite the promise, the potential of hierarchical ML inference has not yet been fully explored in terms of utilizing available resources in edge servers. In this work, we consider (i) model-distributed inference (model parallelization) at the edge servers to speed up ML inference and (ii) reducing the amount of communication to/from the cloud server while preserving the privacy of both data and model.

Model-distributed inference is emerging as a promising solution [7], [8], where an ML model is distributed across edge servers, Fig. 2. The client transmits its data to an edge server, which processes a few layers of an ML model and transmits the feature vector of its last layer/block to the next edge server. Each edge server that receives a feature vector processes the layers that are assigned to it. The edge server that calculates the

This work was supported in parts by the Army Research Office (W911NF2410049), the National Science Foundation (CCF-1942878, CNS-2148182, CNS-2112471, CNS-1801708), and Seagate Technology (00118496.0).

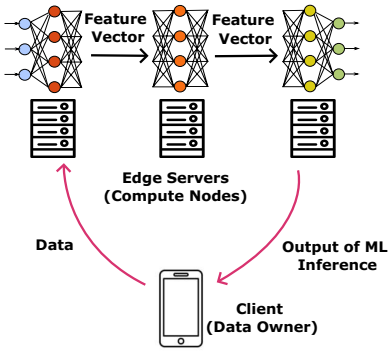


Fig. 2: Model-distributed inference.

last layers of the ML model obtains and sends the output back to the client. We note that the edge servers perform parallel processing by data pipelining, so ML inference becomes faster. We consider that the edge servers in Fig. 1 could employ model parallelization for faster ML inference, but the crucial question of how to provide privacy to both data and the model should be addressed, which is the focus of this paper.

In this paper, we design a privacy-preserving hierarchical model-distributed inference, `privateMDI` protocol. `privateMDI` uses hierarchical ML inference demonstrated in Fig. 1. Similar work has been explored in the literature [4]–[6] but did not consider model-distributed inference, which is one of our novelties. Furthermore, we structure `privateMDI` in two phases: offline and online. The offline and online phases are designed in a way that the communication overhead of the online phase is small. Indeed, the communication cost of the online phase to/from the cloud server is zero in `privateMDI`. This significantly reduces the ML inference time of `privateMDI`. Our `privateMDI` design uses additive secret sharing and linearly homomorphic encryption to handle linear calculations in the ML inference, and garbled circuit and three-party oblivious transfer are used to handle non-linear functions (such as ReLU). The following are our contributions.

- We design an ML inference protocol `privateMDI` that uses model-distributed inference to speed up ML inference and employs a hierarchical setup to reduce the communication overhead while providing privacy to both data and model. To the best of our knowledge, `privateMDI` is the first privacy-preserving model-distributed inference protocol in a hierarchical ML setup.
- `privateMDI` consists of offline and online phases. We designed these phases in a way that most of the data exchange (hence communication) is done in the offline phase, which is done anytime before the online phase as it is independent of the client's data. Thus, the communication overhead of the online phase is reduced. Indeed, there is no communication to/from the cloud server in the online phase, and the amount of communication between the client and edge servers is minimized.
- Our `privateMDI` design uses additive secret sharing and linearly homomorphic encryption to handle linear calculations in the ML inference, and garbled circuit and three-party oblivious transfer are used to handle non-linear functions. `privateMDI` uses additive secret sharing with homomorphic encryption in the offline phase, which reduces the number of computations in the online phase significantly. Our novel three-party OT design, inspired by PROXY-OT protocol from [9], reduces the complexity of the existing OT protocols [3], [10] and provides information-theoretic security.
- We implemented `privateMDI` as well as baselines using ACCESS computing platform [11] and our in-lab computers. The experimental results demonstrate that `privateMDI` significantly improves the ML inference time as compared to the baselines.

## II. Related Work

Privacy-preserving ML inference protocols can be roughly classified into two categories; (i) client/server protocols and (ii) hierarchical protocols as in Fig. 1.

**Client/server protocols.** CryptoNets [12] pioneered the client/server protocols by using leveled homomorphic encryption (HE) and polynomial approximations for activation functions. CryptoDL [13] improved upon CryptoNets with better function approximations, while LoLa [14] focused on reducing latency in HE-based inference. In contrast, nGraph-HE [15] avoids such approximations by passing feature maps to the data owner for clear text processing, raising concerns about ML model parameter privacy.

To address the challenges of ML model exposure and using approximation for the activation functions, subsequent studies combine HE with multi-party computation (MPC) techniques like oblivious transfer (OT), garbled circuits (GC), and secret sharing. For instance, MiniONN [1] and CrypTFlow2 [16] integrate HE with MPC to enhance performance and accuracy. CrypTFlow2 uses OT or HE for linear layers and specialized MPC protocols for ReLU and Maxpool to handle non-linear activation functions. Similarly, Gazelle [2], Delphi [3], Auto-Privacy [17], and MP2ML [18] employ a combination of HE and garbled circuits to optimize both computational efficiency and privacy, using secret sharing for linear layers and GC for non-linear layers. Meanwhile, Cheetah [19] streamlines the process by eliminating costly HE operations, such as rotations in linear layer evaluations, and instead uses efficient OT-based protocols for handling non-linearities.

In parallel, a growing body of work aims to sidestep the high complexity of traditional cryptographic techniques like HE and GC by discretizing DNNs [20], [21]. For example, XONN [21] simplifies the secure computation landscape by leveraging binary neural networks (BNNs), which operate on binary values (+1 or -1). This approach significantly reduces the cryptographic burden by focusing primarily on efficient XNOR operations. Similarly, COINN [22] utilizes advanced quantization techniques to lower both communication and computational overhead at the expense of accuracy loss.

Despite these advancements, client/server models often encounter high latency issues due to the bottleneck between the client and cloud server, especially when the participating

parties are geographically distant—a common scenario in real-world applications. This latency can significantly affect the performance and feasibility of ML inference. Thus, we consider a hierarchical model in our setup, which differentiates us from client/server protocols. SECO [23] builds upon Delphi and splits an ML model into two parts, where one of them is processed at the client while the other part is processed at the cloud server, but it does not support model splitting and distribution over multiple (edge) servers.

**Hierarchical protocols:** Hierarchical protocols for ML inference with the participation of three or more parties have been explored in [4], [5], [10], [24] usually assuming that a client has data and multiple servers (edge and cloud) privately access or hold the ML model. ABY$^3$ [24] is a three-party framework that efficiently transitions between arithmetic, binary, and Yao's three-party computation (3PC), using a three-server model that tolerates a single compromised server. Cryptflow [25] introduces a three-party protocol built upon SecureNN [5], tolerating one corruption and optimizing convolution for reduced communication in non-linear layers. Falcon [6] combines techniques from SecureNN and ABY$^3$ to improve protocol efficiency. SSNet [26] introduces a private inference protocol using Shamir's secret sharing.

*Our work in perspective.* As compared to existing hierarchical protocols summarized above, our approach (i) uses additive secret sharing with HE to provide privacy for ML model parameters, which reduces the number of computations in the online phase significantly (for example, as compared to SSNet [26]); (ii) eliminates the need for continuous and interactive communication with the client during the online phase, thus reducing the communication overhead; (iii) uses model-distributed inference, which enables full utilization of available edge servers and their computing power, and (iv) works with any non-linear ML inference functions.

## III. System Model & Preliminaries

**Notations.** We define $\mathbb{Z}_q$ as a finite field of size $q$, and $\mathbf{x} \in \mathbb{Z}_q^n$ as a vector of size $n$ over the field $\mathbb{Z}_q$. Similarly, $\mathbf{x} \in \{0,1\}^n$ is defined as a binary vector of length $n$. We will denote vectors by bold lowercase letters, while matrices are denoted by bold uppercase letters.

**Setup.** We consider a three-party ML inference, which includes a cloud server (model owner), client (data owner), and edge servers (compute nodes), Fig. 1. Clients and edge servers are directly connected, where high-speed device-to-device links can be used. Client and cloud servers can also communicate using infrastructure-based links such as Wi-Fi or cellular, which is typically considered a bottleneck in today's communication networks. This paper aims to reduce the communication cost between clients and the remote cloud server.

Our system supports multiple clients, as shown in Fig. 1, but we will focus on only one client in the rest of the paper for the sake of clarity and as the multiple client extension is straightforward. The edge servers are divided into clusters. Each cluster consists of $T+2$ edge servers, two of which are garbler and evaluator servers, which are needed to implement the garbled circuit operations, detailed later in this section. Each cluster is responsible for processing a set of layers according to model-distributed inference (model-distribution algorithm), i.e., each cluster ($T+2$ edge servers) computes a part of the model. We will present the details of cluster operation in Section IV.

**Threat Model.** We consider that all the participants are semi-honest, i.e., they follow the protocols, but they are curious. Edge servers in each cluster may collude; we consider that maximum $T$ edge servers (out of $T+2$) collude to obtain data. We assume that the evaluator and the garbler servers do not collude. Also, clients, edge servers, and the cloud server do not collude. Our aim is to design a privacy-preserving model distributed inference mechanism at the edge servers where (i) the client and edge servers do not learn anything about the model[1] and (ii) the cloud server and edge servers do not learn anything about the client's data.

**ML Model and Model-Distribution.** We consider that the cloud server stores a pre-trained ML model. We assume that the ML model can be partitioned, which applies to most of the ML models used in today's ML applications [8]. The ML model could have any linear and/or non-linear operations. Our mechanism is designed to work with any such operations.

Different clusters of edge servers may have different and time-varying computational capacities. Thus, a cluster with higher computing power should process more layers than the others. We follow a similar approach of AR-MDI proposed in [8] to achieve such an adaptive model-distributed inference. AR-MDI [8] is an ML allocation mechanism that determines the set of layers $\Lambda_n(k)$ that should be activated at edge server $n$ for processing data $A_k$. AR-MDI allocates $\lfloor \rho_n(k) \rceil$ layers to edge server $n$ for data $A_k$, where $\lfloor . \rceil$ rounds $\rho_n(k)$ to the closest integer that is a feasible layer allocation. AR-MDI determines $\rho_n(k)$ as $\rho_n(k) = W \frac{1/\gamma_n(k)}{\sum_{m=0}^{N-1} 1/\gamma_m(k)}$, where $W$ is the total number of parameters in the ML model, $\gamma_n(k)$ is the per parameter computing delay. AR-MDI performs layer allocation decentralized as each worker can determine their share of layers by calculating $\lfloor \rho_n(k) \rceil$. The per parameter computing delay $\gamma_n(k)$ can be measured by each worker and shared with other workers. We note that we will use the AR-MDI's model allocation mechanism over our clusters of edge servers and by providing privacy for both data and model.

**Three-Party Oblivious Transfer.** Oblivious Transfer (OT) protocols [27], [28] typically consider two-party setup with a "sender" and "receiver". The idea is that the sender has two binary string inputs $\mathbf{k}_0$ and $\mathbf{k}_1$, and the receiver would like to learn $\mathbf{k}_i$, $i \in \{0,1\}$, but (i) the sender should not learn which input is selected, and (ii) the receiver learns only $\mathbf{k}_i$ and gains no information about $\mathbf{k}_{1-i}$. In our `privateMDI` design, we need to use three-party OT [9] as detailed in Section IV.

In particular, we design a novel three-party OT inspired by PROXY-OT introduced in [9] providing information-theoretic privacy. In our three-party protocol shown in Fig. 3, the three

[1]We note that the client knows the number of non-linear layers and their dimensions in the ML mode, which is needed in our `privateMDI` design, but not the whole architecture of the ML model, nor the specific weights of the ML model.

parties are the client, evaluator server, and garbler server. The client has a one-bit input $i$, and the garbler server has the input labels $\mathbf{k}_0$ and $\mathbf{k}_1$. Additionally, the evaluator server and the client generate random sample $b$ using a pseudorandom generator with the same seed, and the evaluator and garbler server generate random samples $\mathbf{u}_0$ and $\mathbf{u}_1$ similarly. Here, $b, i \in \{0,1\}, \mathbf{u}_j, \mathbf{k}_j \in \{0,1\}^\kappa, j = 0, 1$. First, the client sends its masked input $i \oplus b$ to the garbler server. Next, the garbler server sends the masked labels to the client. The client sends $\mathbf{k}_i \oplus \mathbf{u}_b$ to the evaluator server. The evaluator server can unmask the received label to obtain $\mathbf{k}_i$. The privacy proof of our three-party OT protocol is provided in Appendix A of [29].

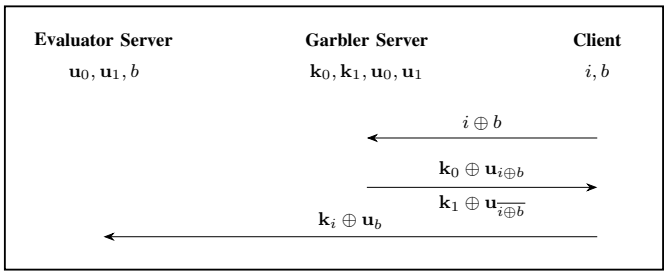



Fig. 3: Our novel three-party OT protocol.

**Garbled Circuit.** Garbled Circuit (GC) is a technique for encoding a boolean circuit $C$ and its inputs $\mathbf{x}$ and $\mathbf{y}$ in a way that allows an evaluator to compute the output $C(\mathbf{x}, \mathbf{y})$ without revealing any information about $C$ or $\mathbf{x}$ and $\mathbf{y}$ other than the output itself. This process involves a garbling scheme consisting of algorithms for encoding and evaluating the circuit, ensuring completeness (the output matches the actual computation) and privacy (the evaluator learns nothing beyond the output and the size of the circuit).

A garbling scheme [30], [31] is a tuple of algorithms $\texttt{GS} = \texttt{(Garble,Eval)}$ with the following syntax:

- $(\tilde{C}, \{\mathbf{k}_0^j, \mathbf{k}_1^j\}_{j\in[2n]}) \leftarrow \texttt{GS.Garble}(1^\lambda, C)$. Given a security parameter $\lambda$ and a boolean circuit $C$ as input, $\texttt{Garble}$ produces a garbled circuit $\tilde{C}$ and a set of labels $\{\mathbf{k}_0^j, \mathbf{k}_1^j\}_{j\in[2n]}$. Here, $\mathbf{k}_i^j$ represents assigning the value $i \in \{0,1\}$ to the $j$-th input label.
- Given a garbled circuit $\tilde{C}$ and labels $\{\mathbf{k}_{\mathbf{x}_j}^j\}_{j\in[n]}$ corresponding to an input $\mathbf{x} \in \{0,1\}^n$, and labels $\{\mathbf{k}_{\mathbf{y}_j}^{n+j}\}_{j\in[n]}$ corresponding to an input $\mathbf{y} \in \{0,1\}^n$, $\texttt{Eval}$ outputs a string $C(\mathbf{x}, \mathbf{y})$.

*Correctness:* For correctness, we require that for every circuit $C$ and inputs $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ the output of $\texttt{Eval}$ must equal $C(\mathbf{x}, \mathbf{y})$. Formally, $\mathbb{P}\left[C(\mathbf{x}, \mathbf{y}) = \texttt{Eval}\left(\tilde{C}, \{\mathbf{k}_{\mathbf{x}_j}^j, \mathbf{k}_{\mathbf{y}_j}^{n+j}\}_{j\in[n]}\right)\right] = 1$.

*Security:* For security, we require that a simulator $\mathcal{S}_{\text{GS}}$ exists such that for any circuit $C$ and inputs $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$, we have $\left(\tilde{C}, \{\mathbf{k}_{\mathbf{x}_j}^j, \mathbf{k}_{\mathbf{y}_j}^{n+j}\}_{j\in[n]}\right) \approx_c \mathcal{S}_{\text{GS}}\left(1^\lambda, 1^{|C|}, C(\mathbf{x}, \mathbf{y})\right)$.

Our $\texttt{privateMDI}$ design involves three parties: the garbler with input $\mathbf{x}$, the evaluator without any input, and the client with input $\mathbf{y}$. For the evaluator to compute $C(\mathbf{x}, \mathbf{y})$ without any party gaining information about other parties' inputs, conventional two-party OT is substituted with our three-party OT protocol.

**Linearly Homomorphic Encryption.** We use Linearly homomorphic encryption (LHE) in our $\texttt{privateMDI}$ design, which enables certain computations on encrypted data without the need for decryption [32], [33]. In LHE, operations on ciphertexts correspond to linear operations on the plaintexts they encrypt. A linearly homomorphic encryption scheme is characterized by four algorithms, collectively denoted as $\texttt{HE} = \texttt{(KeyGen,Enc,Dec,Eval)}$, which can be described as

- $\texttt{HE.KeyGen}$ is an algorithm that generates a public key $\texttt{pk}$ and a secret key $\texttt{sk}$ pair.
- $\texttt{HE.Enc(pk,}m\texttt{)}$ encrypts the message $m$ using the public key $\texttt{pk}$ and outputs a ciphertext $ct$. The message space is a finite field $\mathbb{Z}_q$.
- $\texttt{HE.Dec(sk,}ct\texttt{)}$ decrypts the ciphertext $ct$ using the secret key $\texttt{sk}$ and outputs the message $m$.
- $\texttt{HE.Eval(pk,}ct_1\texttt{,}ct_2\texttt{,}f\texttt{)}$ outputs the encrypted $f(m_1, m_2)$ using the public key $\texttt{pk}$ on the two ciphertexts $ct_1$ and $ct_2$ encrypting messages $m_1$ and $m_2$, where $f$ is a linear function.

Let $ct_1 = \texttt{HE.Enc}(pk, m_1)$, $ct_2 = \texttt{HE.Enc}(pk, m_2)$, $ct' = \texttt{HE.Eval}(pk, ct_1, ct_2, f)$. We require $\texttt{HE}$ to satisfy the following properties:

- *Correctness:* $\texttt{HE.Dec}$ outputs $m$ using $\texttt{sk}$ and a ciphertext $ct = \texttt{HE.Enc(pk,}m\texttt{)}$.
- *Homomorphism:* $\texttt{HE.Dec}$ outputs $f(m_1, m_2)$ using $\texttt{sk}$ and a ciphertext $ct' = \texttt{HE.Eval}(pk, ct_1, ct_2, f)$.
- *Semantic security:* Given a ciphertext $ct$ and two messages of the same length, no attacker should be able to tell which message was encrypted in $ct$.
- *Function privacy:* Given a ciphertext $ct$, no attacker can tell what homomorphic operations led to $ct$. Formally, $(ct_1, ct_2, ct') \approx_c \mathcal{S}_{\text{FP}}(1^\lambda, m_1, m_2, f(m_1, m_2))$.

**Additive Secret Sharing.** An $m$-of-$m$ additive secret sharing scheme over a finite field $\mathbb{Z}_q$, splits a secret $\mathbf{x} \in \mathbb{Z}_q$ into a $m$-element vector $([\mathbf{x}]_1, \ldots, [\mathbf{x}]_m) \in \mathbb{Z}_q^m$. The scheme consists of a pair of algorithms $\texttt{ADD} = \texttt{(Shr,Re)}$, where $\texttt{Shr}$ corresponds to share, and $\texttt{Re}$ refers to reconstruct;

- $(\mathbf{a}_1, \ldots, \mathbf{a}_m) \leftarrow \texttt{ADD.Shr}(\mathbf{x}, m)$. On inputs a secret $\mathbf{x} \in \mathbb{Z}_q$ and number of shares $m$, $\texttt{Shr}$ samples $(m-1)$ values $\mathbf{a}_1, \ldots, \mathbf{a}_{m-1}$ and computes $\mathbf{a}_m = \mathbf{x} - \sum_{j=1}^{m-1} \mathbf{a}_j$. Then it outputs the shares $(\mathbf{a}_1, \ldots, \mathbf{a}_m)$.
- The $\texttt{ADD.Re}$ algorithm on input a sharing $(\mathbf{a}_1, \ldots, \mathbf{a}_m)$, computes and outputs $\mathbf{x} = \sum_{j=1}^{m} \mathbf{a}_j$.

*Correctness:* Let $\mathbf{x} \in \mathbb{Z}_q$ be a secret. Then: $\texttt{ADD.Re}(\texttt{ADD.Shr}(\mathbf{x})) = \mathbf{x}$.

*Security:* Let $(\mathbf{a}_1^1, \ldots, \mathbf{a}_m^1) \leftarrow \texttt{ADD.Shr}(\mathbf{x}_1, m)$ and $(\mathbf{a}_1^2, \ldots, \mathbf{a}_m^2) \leftarrow \texttt{ADD.Shr}(\mathbf{x}_2, m)$ be the secret shares of two secrets, $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively. Here, the $j$-th party possesses the shares $\mathbf{a}_j^1$ and $\mathbf{a}_j^2$. For any $j$, the shares $\mathbf{a}_j^1$ and $\mathbf{a}_j^2$ are identically distributed. Consequently, each share is indistinguishable from a random value to the $j$-th party, ensuring no information about $\mathbf{x}_1$ or $\mathbf{x}_2$ is leaked.

## IV. Privacy-Preserving Model-Distributed Inference (privateMDI) Design

The objective of our privateMDI protocol is to compute the inference result of the client's input data $\mathbf{x} \in \mathbb{Z}_q^n$ using an ML model $\mathbf{M}(.)$ and give output $\mathbf{M}(\mathbf{x})$ in a privacy-preserving manner. The ML model consists of $L$ layers and is represented by $\mathbf{M} = (\mathbf{M}_1, \ldots, \mathbf{M}_L)$ by abuse of notation.

**Algorithm 1** Offline privateMDI operation

**Input:** The cloud server has the ML model parameters $\mathbf{M} = (\mathbf{M}_1, \ldots, \mathbf{M}_L)$.

1: The client generates (pk, sk) using HE.KeyGen.
2: The client generates random sample $\mathbf{r}_1$ over $\mathbb{Z}_q^n$, and sends HE.Enc(pk, $\mathbf{r}_1$) to the cloud server.
3: The cloud server initializes $\mathbf{c}_1^{enc} \leftarrow$ HE.Enc(pk, $\mathbf{r}_1$).
4: **for** $j \in [1, \ldots, P]$ **do**
5:  **for** $l \in [l_{start}^j, \ldots, l_{end}^j]$ **do**
6:  Cloud server generates random sample $\mathbf{s}_l \in \mathbb{Z}_q^n$, secret shares $([\mathbf{M}_l]_1$,..., $[\mathbf{M}_l]_{T+1}) \leftarrow$ ADD.Shr $(\mathbf{M}_l, T+1)$ and $([\mathbf{s}_l]_1, \ldots, [\mathbf{s}_l]_{T+1}) \leftarrow$ ADD.Shr$(\mathbf{s}_l, T+1)$.
7:  The cloud server sends $[\mathbf{M}_l]_v$ and $[\mathbf{s}_l]_v$ to the edge server $v$, $v \in [v_1^j, \ldots, v_{T+1}^j]$.
8:  The cloud server encrypts: $\mathbf{M}_l^{enc} \leftarrow$ HE.Enc(pk, $\mathbf{M}_l$), $\mathbf{s}_l^{enc} \leftarrow$ HE.Enc(pk, $\mathbf{s}_l$).
9:  The cloud server computes $\mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$.
10:  **if** $l$ only has linear operations **then**
11:  The cloud server determines $\mathbf{c}_{l+1}^{enc} \leftarrow \mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$.
12:  **else**
13:  The cloud server sends $\mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$ to the client.
14:  The client sends HE.Enc(pk, $\mathbf{r}_{l+1}$) to the cloud server.
15:  The cloud server determines $\mathbf{c}_{l+1}^{enc} \leftarrow$ HE.Enc(pk, $\mathbf{r}_{l+1}$).
16:  The client, garbler, and evaluator server run Algorithm 2.
17:  **end if**
18:  **end for**
19: **end for**
20: **if** $L$ only has linear operations **then**
21:  The cloud server sends $\mathbf{c}_{L+1}^{enc}$ to the client.
22: **end if**

Our privateMDI protocol is divided into two phases: offline and online, similar to some previous work [1]–[4], [10], [24]. The goal behind having two phases is to move the computationally intensive aspects to the offline phase and make the online phase, which calculates the ML model output, faster. Also, the offline phase is designed to minimize the communication overhead between the client and the cloud server, which is the bottleneck link in the online phase. The offline phase (i) exchanges keys between the client and cloud server, (ii) shares the secret ML model with the edge servers, and (iii) exchanges the garbled circuits and two out of three inputs of the garbled circuits labels. We note that the offline phase does not use and is independent of the client's data $\mathbf{x}$, but makes the system ready for the online phase, i.e., computing the ML inference $\mathbf{M}(\mathbf{x})$.

We note that an ML model consists of linear and non-linear layers. In our privateMDI design, we use additive secret sharing, linear homomorphic encryption for linear operations, garbled circuit, and three-party oblivious transfer for non-linear operations. As we mentioned earlier, our privateMDI design is generic enough to work with any non-linear functions. Next, we will describe the details of the offline and online parts of the privateMDI.

### A. *Offline* privateMDI

In this section, we describe the details of the offline privateMDI protocol. The overall operation is summarized in Algorithm 1.

The cloud server has the ML model parameters $\mathbf{M} = (\mathbf{M}_1, \ldots, \mathbf{M}_L)$. In the initialization phase of the algorithm (lines 1-2), the client generates a pair of (pk, sk) using HE.KeyGen and a random sample $\mathbf{r}_1$ over $\mathbb{Z}_q^n$, and sends HE.Enc(pk, $\mathbf{r}_1$) to the cloud server. We note that $\mathbf{r}_l, \mathbf{s}_l \in \mathbb{Z}_q^n$ are random samples of length $n$ generated by the client and cloud server for ML model layer $l$, respectively. The cloud server initializes $\mathbf{c}_1^{enc}$ with the received ciphertext $\mathbf{r}_1$ from the client according to $\mathbf{c}_1^{enc} \leftarrow$ HE.Enc(pk, $\mathbf{r}_1$).

Next, the offline protocol shares encrypted model parameters and keys (garbles circuit labels) for each cluster of edges, where there are $P$ clusters. We note that there are $T+2$ edge servers in each cluster. Each cluster $j$ processes the set of layers $[l_{start}^j, \ldots, l_{end}^j]$ that are assigned to cluster $j$ employing the adaptive model-distribution algorithm AR-MDI described in Section III, where $l_{start}^j$ and $l_{end}^j$ are the first and last assigned layers to cluster $j$.

**Algorithm 2** Garbled circuit and oblivious transfer operation in a cluster in the *offline* phase.

**Input:** Circuit $C$.

1: The garbler server runs the GS.Garble algorithm on circuit $C$ and outputs $\tilde{C}$ and the garbled labels.
2: The garbler server sends $\tilde{C}$ to the evaluator server.
3: The garbler server, evaluator server, and client run the three-party OT algorithm.
4: The evaluator server receives the garbled labels of $\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l$ and $\mathbf{r}_{l+1}$.

The cloud server generates random sample $\mathbf{s}_l$ over $\mathbb{Z}_q^n$, calculates $T+1$ additive secret shares of the model parameters $\mathbf{M}_l$ and the random samples $\mathbf{s}_l$ (line 6). We note that $\mathbf{s}_l$ provides privacy for the ML model parameters against client and edge servers. Then, it sends each share ($[\mathbf{M}_l]_v$ and $[\mathbf{s}_l]_v$) to the edge server $v$ of cluster $j$ (line 7). We note that $v$ is any edge server in cluster $j$ such that $v \in [v_1^j, \ldots, v_{T+1}^j]$ excluding the evaluator server. The cloud server encrypts $\mathbf{M}_l$ and $\mathbf{s}_l$ and determines the enycrpted versions $\mathbf{M}_l^{enc} \leftarrow$

HE.Enc(pk, $\mathbf{M}_l$), $\mathbf{s}_l^{enc} \leftarrow$ HE.Enc(pk, $\mathbf{s}_l$) (line 8). The cloud server computes $\mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$ by executing the HE.Eval algorithm on $\mathbf{M}_l^{enc}$, $\mathbf{c}_l^{enc}$, and $\mathbf{s}_l^{enc}$ (line 9).

Next, the offline privateMDI determines $\mathbf{c}_{l+1}^{enc}$, which is a secret share needed in the online part. Depending on whether a layer of the ML model has only linear or non-linear components, the calculation of $\mathbf{c}_{l+1}^{enc}$ differs. If a layer only has **linear** components, the cloud server determines $\mathbf{c}_{l+1}^{enc} \leftarrow \mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$ (line 11).

The following steps (lines 13-16) are performed if layer $l$ has **non-linear** components. First, the cloud server sends $\mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$, computed in the linear part, to the client. Then the client sends a new random sample HE.Enc(pk, $\mathbf{r}_{l+1}$) to the cloud server, and the cloud server determines $\mathbf{c}_{l+1}^{enc}$ with HE.Enc(pk, $\mathbf{r}_{l+1}$). Finally, the garbled circuit is used to compute the non-linear activation function privately.

In particular, cluster $j$'s garbler and evaluator servers and the client run Algorithm 2. Using this algorithm, the garbler server in the cluster runs the GS.Garble algorithm on the input circuit $C$ and sends the output to the evaluator server (lines 1-2 of Algorithm 2). Next, the client, garbler, and evaluator server run the three-party OT algorithm described in Section III (line 3 of Algorithm 2). At the end, the evaluator receives the labels corresponding to $\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l$ and $\mathbf{r}_{l+1}$, two out of three inputs of the garbled circuit, where $\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l$ is the decrypted value of $\mathbf{M}_l^{enc} \cdot \mathbf{c}_l^{enc} + \mathbf{s}_l^{enc}$. Note that the circuit $C$ is a boolean circuit. Assuming that the non-linearity is due to the ReLU function as an example (noting that our algorithm works with any non-linear functions), the circuit $C$ computes the results in the following order.

1) $\mathbf{M}_l \cdot \mathbf{x}_l = (\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l) + (\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l)$, where the second term (i.e., $(\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l)$) is calculated in the online phase.
2) $\mathbf{x}_{l+1} = \text{ReLU}(\mathbf{M}_l \cdot \mathbf{x}_l)$.
3) $\mathbf{x}_{l+1} - \mathbf{r}_{l+1}$.

In the last step of Algorithm 1, the cloud server sends the ciphertext $\mathbf{c}_{L+1}^{enc}$ to the client (line 21) if the last layer is linear, as it's needed for the client in the online phase to decrypt the inference result. If the last layer is non-linear, the client already possesses the decrypted $\mathbf{c}_{L+1}^{enc}$ according to line 14.

### *B. Online* privateMDI

The online phase of privateMDI is responsible for computing the ML inference function $\mathbf{M}(\mathbf{x})$ given the client's data $\mathbf{x}$. We note that there is no communication between edge servers and the cloud server or the client, except for the beginning of the algorithm, when the client sends the input data to the first cluster to start the process, and at the end when the last cluster sends the result of the inference to the client. The online phase is summarized in Algorithm 3.

First, the client sends a secret share of its input $\mathbf{x}$ masked by its random sample $\mathbf{r}_1$ to the garbler server of the first cluster, denoted as $\mathbf{x}_1 - \mathbf{c}_1$, where $\mathbf{x}_1 \leftarrow \mathbf{x}$ and $\mathbf{c}_1 \leftarrow \mathbf{r}_1$ (line 1). Then, each layer $l$ assigned to cluster $j$ is processed with input $\mathbf{x}_l - \mathbf{c}_l$, where $\mathbf{c}_l$ is the decrypted value of $\mathbf{c}_l^{enc}$, determined in the offline phase. We note that the input of every layer is denoted by $\mathbf{x}_l - \mathbf{c}_l$, where $\mathbf{x}_l$ is the output of the previous $l-1$ layers.

The cluster $j$ performs linear computations of layer $l$ (if there is any) on input $\mathbf{x}_l - \mathbf{c}_l$ to produce the output $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$.

First, the garbler server sends the input of the current layer $\mathbf{x}_l - \mathbf{c}_l$ to the other $T$ computing edge servers. Next, each computing edge server $v$, including the garbler server, uses $[\mathbf{M}_l]_v$ and $[\mathbf{s}_l]_v$ and computes $[\mathbf{M}_l]_v(\mathbf{x}_l - \mathbf{c}_l) - [\mathbf{s}_l]_v$. Note that each computing edge server makes its output private from the garbler server by adding $[\mathbf{s}_l]_v$ to its computation. Then, the garbler server runs the ADD.Re algorithm on the $T$ secret shares received from the computing edge servers and its own share $[\mathbf{M}_l]_{T+1}(\mathbf{x}_l - \mathbf{c}_l) - [\mathbf{s}_l]_{T+1}$, and obtains $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$, which is the input of the next layer, denoted as $\mathbf{x}_{l+1} - \mathbf{c}_{l+1}$.

Note that for a purely linear layer, if we expand $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$, we get: $\mathbf{M}_l \cdot \mathbf{x}_l - \mathbf{M}_l \cdot \mathbf{c}_l - \mathbf{s}_l = \mathbf{M}_l \cdot \mathbf{x}_l - (\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l)$. According to the definitions of $\mathbf{x}_{l+1}$ and $\mathbf{c}_{l+1}$, we have: $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l = \mathbf{M}_l \cdot \mathbf{x}_l - (\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l) = \mathbf{x}_{l+1} - \mathbf{c}_{l+1}$, which represents the input of the next layer, layer $l+1$.

**Algorithm 3** Online privateMDI operation.

**Input:** The client's input data $\mathbf{x}$ and the cloud server's model parameters $\mathbf{M}$.

1: The client sends a secret share of its input $\mathbf{x}$ masked by its random share $\mathbf{r}_1$ to the garbler server of the first cluster, denoted as $\mathbf{x}_1 - \mathbf{c}_1$, where $\mathbf{x}_1 \leftarrow \mathbf{x}$ and $\mathbf{c}_1 \leftarrow \mathbf{r}_1$.
2: **for** $j \in [1, \ldots, P]$ **do**
3:  **for** $l \in [l_{start}^j, \ldots, l_{end}^j]$ **do**
4:  The cluster $j$ performs linear operations on $\mathbf{x}_l - \mathbf{c}_l$ and determines $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$
5:  **if** $l$ has non-linear components **then**
6:  The garbler and evaluator server run Algorithm 4, and outputs $\mathbf{x}_{l+1} - \mathbf{c}_{l+1}$, where $\mathbf{c}_{l+1} = \mathbf{r}_{l+1}$.
7:  **end if**
8:  **end for**
9:  The garbler server sends $\mathbf{x}_{l+1} - \mathbf{c}_{l+1}$ to the garbler server of the next cluster.
10: **end for**
11: The garbler server of the last cluster sends $\mathbf{x}_{L+1} - \mathbf{c}_{L+1}$ to the client.
12: The client unmasks the above secret share and obtains the inference result $\mathbf{M}(\mathbf{x}) = \mathbf{x}_{L+1}$.

**Output:** Result of the inference $\mathbf{M}(\mathbf{x})$ on the client side.

If $l$ has non-linear components, the garbler and evaluator server run Algorithm 4. This algorithm requires three sets of inputs: $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$, calculated by the garbler server, $\mathbf{r}_{l+1}$, a random sample generated by the client to ensure the layer output's privacy, and $\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l$, calculated by the cloud server and sent to the client in the offline phase.

The evaluator server obtains two sets of labels in the offline phase via the three-party OT protocol in Fig. 3. The garbler server sends the last set of labels corresponding to its output in the linear part, denoted as $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$ (line 1 of Algorithm 4). We don't need OT to exchange the labels of this

input. This justifies the garbler server's choice as the cluster's head server, responsible for communicating with the other edge servers of the cluster. Otherwise, it would increase the amount of communication. Then, with all the required labels and circuits, the evaluator server runs the `GS.Eval` algorithm and calculates $C(\mathbf{M}_l \cdot \mathbf{x}_l) - \mathbf{r}_{l+1}$, denoted by $\mathbf{x}_{l+1} - \mathbf{r}_{l+1}$ (line 2 of Algorithm 4). Finally, the evaluator server sends the result $(\mathbf{x}_{l+1} - \mathbf{r}_{l+1})$ back to the garbler server (line 3 of Algorithm 4).

**Algorithm 4** Handling non-linearity in the *online* phase.

**Input:** The garbled labels of $\mathbf{M}_l \cdot \mathbf{c}_l + \mathbf{s}_l$ and $\mathbf{r}_{l+1}$ stored by the evaluator server.

1: The garbler server sends the corresponding labels of the input $\mathbf{M}_l(\mathbf{x}_l - \mathbf{c}_l) - \mathbf{s}_l$ to the evaluator server.
2: The evaluator server runs `GS.Eval` and calculates $C(\mathbf{M}_l \cdot \mathbf{x}_l) - \mathbf{r}_{l+1}$, denoted by $\mathbf{x}_{l+1} - \mathbf{r}_{l+1}$.
3: The evaluator server sends the result back to the garbler server.

**Output:** $\mathbf{x}_{l+1} - \mathbf{r}_{l+1}$.

Cluster $j$ performs all the linear and non-linear parts of all the layers assigned to it in Algorithm 3. Ultimately, the garbler server forwards the cluster's result either to the next cluster if $l^j_{end}$ is the final layer allocated to cluster $j$ (line 9) or the client (line 11) if $l^j_{end}$ is the ML model's final layer $L$, i.e., $l^j_{end} = L$. Finally, the client unmasks the secret share $\mathbf{x}_{L+1} - \mathbf{c}_{L+1}$ and obtains the inference result $\mathbf{M}(\mathbf{x}) = \mathbf{x}_{L+1}$.

## V. Privacy and Communication Analysis

In this section, we analyze `privateMDI` in terms of its privacy guarantee and communication overhead.

### A. Privacy

**Definition 1.** *A cryptographic inference protocol $\mathbf{\Pi}$ involves a cloud server with model parameters $\mathbf{M} = (\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_l)$, a client with an input vector $\mathbf{x}$, and $P$ clusters, each containing $T$ computing edge servers, a garbler server, and an evaluator server. The protocol is considered secure if it satisfies the following conditions:*

*Correctness: The protocol ensures that for any given set of model parameters $\mathbf{M}$ held by the cloud server and any input vector $\mathbf{x}$ provided by the client, the client receives the correct prediction $\mathbf{M}(\mathbf{x})$ after executing the protocol.*

*Security: Consider the following scenarios:*

***Compromised cloud server:** A semi-honest cloud server, even if partially compromised, should not learn the client's private input $\mathbf{x}$. This is formally captured by the existence of an efficient simulator $\mathcal{S}_{\text{CS}}$ such that $\text{View}^{\mathbf{\Pi}}_{\text{CS}} \approx_c \mathcal{S}_{\text{CS}}(\mathbf{M})$, where $\text{View}^{\mathbf{\Pi}}_{\text{CS}}$ denotes the cloud server's view during the protocol execution.*

***Compromised client:** A semi-honest client, even if compromised, should not learn the cloud server's model parameters $\mathbf{M}$. This is formally captured by the existence of an efficient simulator $\mathcal{S}_{\text{C}}$ such that $\text{View}^{\mathbf{\Pi}}_{\text{C}} \approx_c \mathcal{S}_{\text{C}}(\mathbf{x}, \mathbf{y})$, where $\text{View}^{\mathbf{\Pi}}_{\text{C}}$ denotes the client's view during the protocol execution, including the input, randomness, and protocol transcript, and $\mathbf{y}$ is the output of the inference.*

***Compromised edge server:** Semi-honest edge servers within a cluster should not gain information about the client's input $\mathbf{x}$ or the cloud server's model parameters $\mathbf{M}$. This is formally captured by the existence of an efficient simulator $\mathcal{S}_{\text{ES}}$ for each edge server $i$, such that $\text{View}^{\mathbf{\Pi}}_{\text{ES},i} \approx_c \mathcal{S}_{\text{ES},i}$, where $\text{View}^{\mathbf{\Pi}}_{\text{ES},i}$ denotes the view of the edge server during the protocol execution. The edge server can be a computing edge server, a garbler server, or an evaluator server, $i \leftarrow C, G, E$.*

**Theorem 1.** `privateMDI` *is secure according to Definition 1 assuming the use of secure garbled circuits, linearly homomorphic encryption, and three-party OT.*

*Proof.* We use simulation-based, hybrid arguments to prove Theorem 1 as detailed in Appendix B of [29]. □

### B. Communication Overhead

We analyze the communication overhead of `privateMDI` as compared to Delphi [3], SecureNN [5], and Falcon [6] as summarized in Table I. In this context, $T$ represents the number of colluding edge servers in a cluster, $N$ is the number of bits in a single input data $x \in \mathbb{Z}_q$, $\kappa$ denotes the length of the garbled circuit labels, $N_{enc}$ is the number of bits in homomorphically encrypted data, and $|GC|$ indicates the number of bits required to transmit the garbled circuits. Additionally, $h$ denotes the dimension of the square input matrix to a layer, $g$ is the dimension of the kernel corresponding to the layer, and $i$ and $o$ represent the number of input and output channels, respectively. $p$ is the smaller field size in SecureNN [5].

Each row in Table I shows the number of communication rounds and the amount of data (bits) exchanged between the following pairs: the cloud server and the client, the client and the edge server, and two edge servers. We exclude the communication overhead between the cloud server and edge servers, as they are usually connected via high-speed links. The critical insight from this table is that `privateMDI` offloads as much communication as possible to the edge servers in the offline phase. In contrast to Delphi, which follows a client-server model, `privateMDI` shifts the bottleneck (sending the GCs) to the edge servers and eliminates online phase communication between the client and the cloud server. SecureNN does not separate the protocol into online and offline phases and has high communication overhead. Only Falcon has better communication overhead as compared to `privateMDI`, but it is limited in the sense that it (i) only supports a few non-linear functions, (ii) has higher linear computation overhead, (iii) does not support model-distributed inference, and (iv) cannot tolerate collusion among multiple parties due its use of replicated secret sharing. Section VI confirms our analysis and shows that `privateMDI` reduces ML inference time as compared to Delphi, SecureNN, and Falcon. More details are provided in Appendix C of [29].

## VI. Evaluation

In this section, we evaluate the performance of our protocol `privateMDI` in a real testbed. We will first describe our experimental setup and then provide our results.

TABLE I: Communication overhead analysis. CS: Cloud Server (Model Owner), C: Client (Data Owner), and ES: Edge Server

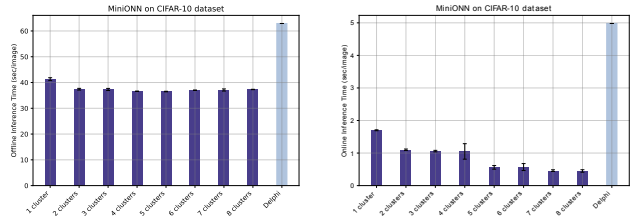
| | Layer Component | Delphi Offline | | Delphi Online | | SecureNN | | Falcon | | privateMDI Offline | | privateMDI Online | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Rds. | Comm. | Rds. | Comm. | Rds. | Comm. | Rds. | Comm. | Rds. | Comm. | Rds. | Comm. |
| CS-C | linear | 2 | $2N_{enc}$ | 2 | $2N$ | - | - | - | - | 2 | $2N_{enc}$ | - | - |
| | non-linear | 3 | $\|GC\| + 2N\kappa$ | 1 | $N\kappa$ | - | - | - | - | - | - | - | - |
| C-ES | linear | - | - | - | - | - | - | - | - | - | - | - | - |
| | non-linear | - | - | - | - | - | - | - | - | 6 | $2N(2\kappa+1)$ | - | - |
| ES-ES | linear | - | - | - | - | 2 | $(h^2g^2i + 2g^2oi + h^2o)N$ | 1 | $(h^2o)N$ | - | - | 2 | $(h^2i + h^2o)N$ |
| | non-linear | - | - | - | - | 10 | $(8\log p + 24)N$ | $5 + \log N$ | $0.5N$ | 1 | $\|GC\|$ | 1 | $N\kappa$ |

**Experimental Setup.** To run our experiments, we used a desktop computer with an Intel Core i7-8700 CPU at 3.20GHz with 16GB of RAM as the cloud server and different instances of Jetstream2 [34] from ACCESS [11] as the edge servers and the client. Specifically, for the first four clusters (of edge servers) of privateMDI, we used m3.quad as the computing edge server, m3.xl as the garbler server, and m3.2xl as the evaluator server. For clusters 4 to 6, we used m3.quad as the computing edge server, m3.large as the garbler server, and m3.xl as the evaluator server. For clusters 7 and 8, we used m3.quad as the computing edge server, m3.medium as the garbler server, and m3.large as the evaluator server. We used an instance of m3.medium as the client. All the devices are connected via TCP connections. We compare privateMDI with baselines Delphi [3], SecureNN [5], and Falcon [6].

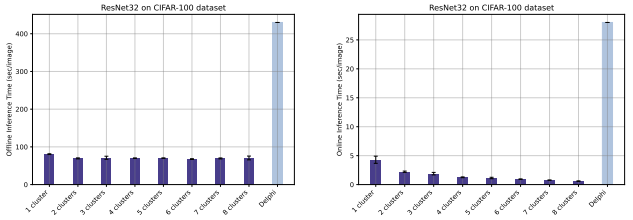
To ensure a fair comparison with Delphi, we utilized the same instance types for the client (m3.medium on Jetstream2) and the cloud server (a desktop computer equipped with an Intel Core i7-8700 CPU at 3.20 GHz with 16 GB of RAM) as described in the privateMDI setup. Additionally, we conducted experiments with various Jetstream2 instance types to verify that Delphi's computational setup does not negatively impact their latency. For our comparison with Falcon, we employed three m3.2xl instances on Jetstream2.

We used the following ML architectures and datasets to run the experiments: (i) The ML model in Figure 13 of MiniONN [1] on the CIFAR-10 dataset; (ii) ResNet32 model introduced in [35] with CIFAR-100 dataset; (iii) The ML model in Figure 12 of MiniONN [1] on MNIST dataset, with Maxpool being replaced by Meanpool; and (iv) VGG16 model introduced in [36] on Tiny ImageNet dataset.

**Results.** Fig. 4 presents the ML inference time of privateMDI as compared to Delphi in both offline and online phases. Fig. 4(a) shows the delay of the offline protocols for both privateMDI and Delphi for the CIFAR-10 dataset on MiniONN architecture. As seen, privateMDI significantly improves the offline delay as compared to Delphi, thanks to reducing the communication overhead. As seen, the increasing number of clusters does not affect the offline delay in privateMDI, because model-distributed inference only affects the delay in the online phase. Indeed, Fig. 4(b) shows that the ML inference time of privateMDI is significantly lower than Delphi in the same setup and decreases with the increasing number of clusters thanks to model distribution.




(a) Offline. CIFAR-10 on MiniONN (b) Online. CIFAR-10 on MiniONN




(c) Offline. CIFAR-100 on ResNet32 (d) Online. CIFAR-100 on ResNet32

Fig. 4: ML inference time of privateMDI in online and offline phases with increasing number of clusters.

Fig. 4(c) and (d) show the ML inference time in the offline and online phases for the CIFAR-100 dataset and ResNet32 architecture. As seen, the improvement of privateMDI as compared to Delphi is more pronounced in this setup as the dataset and the ML model are larger in this setup, and privateMDI performs better in this setup, thanks to model parallelization and reducing communication overhead.

TABLE II: ML inference time of privateMDI in the online phase as compared to Falcon and SecureNN.

| | MiniONN | AlexNet | VGG16 |
|---|---|---|---|
| privateMDI | 0.09 s | 2.73 s | 3.63 s |
| Falcon | 0.02 s | 2.11 s | 5.12 s |
| SecureNN | 0.44 s | - | - |

Table II shows the ML inference time of privateMDI in the online phase as compared to SecureNN and Falcon for MiniONN, AlexNet, and VGG16 architectures. MNIST dataset is used with the MiniONN, and Tiny ImageNet dataset is used with the AlexNet and VGG16. privateMDI has 8 clusters for AlexNet and VGG16 and 7 clusters for MiniONN. As seen, privateMDI improves over SecureNN in MiniONN. However, Falcon's improved communication cost, as discussed in Section V-B, gives it an edge over smaller ML models like MiniONN. As the models grow larger, the performance

gap narrows. While privateMDI shows improvements but does not outperform Falcon in AlexNet, it demonstrates significant improvement in the VGG16 setup due to the benefits of model-distributed inference, which become more pronounced with larger ML models and datasets. Further results of privateMDI on the VGG16 ML model for different numbers of clusters are provided in Appendix D of [29].

## VII. Conclusion

This paper designed privacy-preserving hierarchical model-distributed inference, privateMDI protocol to speed up ML inference in a hierarchical setup while providing privacy to both data and ML model. Our privateMDI design (i) uses model-distributed inference at the edge servers, (ii) reduces the amount of communication to/from the cloud server to reduce ML inference time, and (iii) uses additive secret sharing with HE, which reduces the number of computations. The experimental results demonstrated that privateMDI significantly reduced the ML inference time as compared to the baselines.

## References


[1] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 619–631.

[2] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: a cryptographic inference service for neural networks," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[4] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 19–38.

[5] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *Proceedings on Privacy Enhancing Technologies*, 2019.

[6] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," *arXiv preprint arXiv:2004.02229*, 2020.

[7] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. N. Walters, "Pipeline parallelism for inference on heterogeneous edge computing," *arXiv preprint arXiv:2110.14895*, 2021.

[8] P. Li, E. Koyuncu, and H. Seferoglu, "Adaptive and resilient model-distributed inference in edge computing systems," *IEEE Open Journal of the Communications Society*, vol. 4, pp. 1263–1273, 2023.

[9] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *Proceedings of the 1st ACM Conference on Electronic Commerce*, 1999, pp. 129–139.

[10] 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 *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018.

[11] "Nsf access computing resources," 2024. [Online]. Available: https://access-ci.org/

[12] 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 *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[13] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *arXiv preprint arXiv:1711.05189*, 2017.

[14] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *International Conference on Machine Learning*. PMLR, 2019, pp. 812–821.

[15] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "ngraph-he: a graph compiler for deep learning on homomorphically encrypted data," in *Proceedings of the 16th ACM international conference on computing frontiers*, 2019, pp. 3–13.

[16] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

[17] Q. Lou, S. Bian, and L. Jiang, "Autoprivacy: Automated layer-wise parameter selection for secure neural network inference," *Advances in Neural Information Processing Systems*, vol. 33, pp. 8638–8647, 2020.

[18] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "Mp2ml: A mixed-protocol machine learning framework for private inference," in *Proceedings of the 15th international conference on availability, reliability and security*, 2020, pp. 1–10.

[19] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure {Two-Party} deep neural network inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 809–826.

[20] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón, "Quotient: two-party secure neural network training and prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1231–1247.

[21] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "{XONN}:{XNOR-based} oblivious deep neural network inference," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1501–1518.

[22] S. U. Hussain, M. Javaheripi, M. Samragh, and F. Koushanfar, "Coinn: Crypto/ml codesign for oblivious inference via neural networks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3266–3281.

[23] S. Chen and A. Khisti, "Seco: Secure inference with model splitting across multi-server hierarchy," *arXiv preprint arXiv:2404.16232*, 2024.

[24] P. Mohassel and P. Rindal, "Aby3: A mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 35–52.

[25] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 336–353.

[26] S. Duan, C. Wang, H. Peng, Y. Luo, W. Wen, C. Ding, and X. Xu, "Ssnet: A lightweight multi-party computation scheme for practical privacy-preserving machine learning service in the cloud," *arXiv e-prints*, pp. arXiv–2406, 2024.

[27] M. Naor and B. Pinkas, "Oblivious transfer and polynomial evaluation," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, 1999, pp. 245–254.

[28] R. Rivest, "Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer," *Unpublished manuscript*, 1999.

[29] F. J. Dehkordi, Y. Keshtkarjahromi, and H. Seferoglu, "Privacy-preserving model-distributed inference at the edge," *arXiv preprint arXiv:2407.18353*, 2024.

[30] A. C.-C. Yao, "How to generate and exchange secrets (extended abstract)," in *IEEE Annual Symposium on Foundations of Computer Science*, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:52818943

[31] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 784–796.

[32] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[33] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[34] D. Y. Hancock, J. Fischer, J. M. Lowe, W. Snapp-Childs, M. Pierce, S. Marru, J. E. Coulter, M. Vaughn, B. Beck, N. Merchant, E. Skidmore, and G. Jacobs, "Jetstream2: Accelerating cloud computing via jetstream," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437359.3465565

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.