CoGNN: Towards Secure and Efficient Collaborative Graph Learning

Zhenhua Zou
Tsinghua University
& Zhongguancun Laboratory
Beijing, China
zou-zh21@mails.tsinghua.edu.cn

Zhuotao Liu*
Tsinghua University
& Zhongguancun Laboratory
Beijing, China
zhuotaoliu@tsinghua.edu.cn

Jinyong Shan
Sudo Technology Co.,LTD
Beijing, China
jnngshan@gmail.com

Qi Li
Tsinghua University
& Zhongguancun Laboratory
Beijing, China
qli01@tsinghua.edu.cn

Ke Xu
Tsinghua University
& Zhongguancun Laboratory
Beijing, China
xuke@tsinghua.edu.cn

Mingwei Xu
Tsinghua University
& Zhongguancun Laboratory
Beijing, China
xumw@tsinghua.edu.cn

ABSTRACT

Collaborative graph learning represents a learning paradigm where multiple parties jointly train a graph neural network (GNN) using their own proprietary graph data. To honor the data privacy of all parties, existing solutions for collaborative graph learning are either based on federated learning (FL) or secure machine learning (SML). Although promising in terms of efficiency and scalability due to their distributed training scheme, FL-based approaches fall short in providing provable security guarantees and achieving good model performance. Conversely, SML-based solutions, while offering provable privacy guarantees, are hindered by their high computational and communication overhead, as well as poor scalability as more parties participate.

To address the above problem, we propose CoGNN, a novel framework that simultaneously reaps the benefits of both FL-based and SML-based approaches. At a high level, CoGNN is enabled by (i) a novel message passing mechanism that can obliviously and efficiently express the vertex data propagation/aggregation required in GNN training and inference and (ii) a two-stage Dispatch-Collect execution scheme to securely decompose and distribute the GNN computation workload for concurrent and scalable executions. We further instantiate the CoGNN framework, together with customized optimizations, to train Graph Convolutional Network (GCN) models. Extensive evaluations on three graph datasets demonstrate that compared with the state-of-the-art (SOTA) SML-based approach, CoGNN reduces up to 123x running time and up to 522x communication cost per party. Meanwhile, the GCN models trained using CoGNN have nearly identical accuracies as the plaintext global-graph training, yielding up to 11.06% accuracy improvement over the GCN models trained via federated learning.

CCS CONCEPTS

• **Security and privacy → Cryptography**; • **Computing methodologies → Machine learning**.

KEYWORDS

Collaborative Graph Learning, Secure Multi-party Computation

*Zhuotao Liu is the corresponding author.

1 INTRODUCTION

Recent years have witnessed the success of deploying Graph Neural Network (GNN) [19, 25, 48] in areas like financial fraud detection [51, 53] and drug discovery [22, 58]. As a special category of neural network, GNN takes as input a graph, composed of vertices and the edges connecting vertices. Commonly, the goal of a GNN is to classify vertices, which is called vertex-level (or node-level) prediction. Other tasks include classifying edges or the whole graph. In vertex-level prediction, each vertex is a sample with its features, while each edge connects two vertices and is assigned a weight. The edge weight implicates how the two vertices are related to each other. The key feature distinguishing GNN from the other machine learning models is that it considers the edges (i.e., relations) between different samples during forward and backward computation, while the other models predict on each sample independently. Specifically, the computation of GNN consists of two stages. First, it propagates the features or hidden-layer representation of each sample to influence its neighbors, via the graph edges. This process is called *message passing* [15, 49, 56]. Second, it performs NN computation on the result from the previous stage, as in a traditional NN model.

In this paper, we study the collaborative learning and inference of GNN, where multiple graph data owners jointly train a global GNN model by contributing their *private* graph data. These graph owners have exclusively different sample sets (i.e., vertices), but there could be *inter-edges* connecting the vertices (or graphs) held by different owners. For example, different banks have different sets of accounts, but their transaction graphs are interleaved into a global transaction graph across banks by inter-bank transactions. Interleaved communication graphs of multiple mobile carriers are similar. Jointly analyzing the collection of inter-related graphs is critical to a number of use cases. For instance, to detect fraud accounts across multiple inter-related financial systems [47] or mobile carriers [21], collaboratively training GNNs using data from multiple organizations results in supreme model performances.

However, as data privacy becomes increasingly important (either due to commercial interests or legal compliance (e.g., GDPR [1]), honoring data privacy in collaborative GNN is challenging. The community has proposed significant research in this regard. The first category is based on Federated Learning (FL) [29]. In particular, these FL-based approaches [43, 52, 54, 57] have the participants locally execute graph message passing and NN computation, and coordinate them by performing gradient/model weight aggregation. The FL-based approaches are efficient because all the parties compute their local graph concurrently. They do not support message passing across the inter-edges because passing vertex embeddings/hidden-layer representations on inter-edges in plaintext raises privacy concerns. However, ignoring inter-edge messaging passing results in non-trivial performance degradation of the GNN model, as shown in [52]. In addition, the intermediate data disclosed during federated training (such as the global gradient) raises further privacy concerns, which has been exploited by existing work [36, 41]. Some recent FL-based approaches [13, 32] enhance their privacy (security) guarantee by incorporating cryptographic techniques like secure multi-party computation (MPC) [11, 37] and homomorphic encryption (HE) [23], where they focus on protecting the model/gradient aggregation process. However, they fail to provide an end-to-end provable privacy guarantee, because the aggregated intermediate results (like the global gradient) are either still revealed in plaintext to the clients [32] (introducing privacy vulnerabilities as stated above) or hidden via TEE hardware [13] (vulnerable to various attacks [24, 40]).

On the other hand, another branch of prior approaches leverages secure machine learning (SML) to provide end-to-end security guarantees. These approaches [26, 50] work in an outsourced setting, where all graph owners outsource their data, in the secret-shared form, to several third-party computing servers. The servers jointly train the GNN model and only send back the trained model, without disclosing any intermediate results. The data privacy is provably guaranteed via the underlying MPC protocols [11, 27, 37]. However, these approaches are limited by two drawbacks: (i) *poor efficiency*: the computation/communication cost in the outsourced setting is high, mainly caused by expensive cryptographic operations like oblivious sort [5, 26] or secure array accesses [8]. For instance, experimental results of [5] show that oblivious sort takes up more than 70% of the overall running time of graph algorithms; (ii) *poor scalability*: unlike the distributed computation in FL where each party processes its local graph only, each party in SML-based solutions has to process the entire global graph, resulting in poor scalability as the number of graph owners increases (i.e., as the scale of the global graph increases). Therefore, scaling SML-based approaches to compute large global graphs is challenging.

In summary, in supporting collaborative GNN training and inference, existing solutions face critical limitations. FL-based approaches, while offering distributed computation, lack provable privacy guarantees and suffer from limited model performance. Conversely, SML-based have desirable security properties, but at the cost of significant overhead and limited scalability. In this paper, we seek to answer the following research question: *can we combine the benefits of both categories of approaches: achieving provable privacy guarantees and high model performance (in line with the SML-based approaches), while adopting a fully distributed computation scheme (in line with the FL-based approaches) to enable supreme efficiency and scalability.*

Towards this end, we propose CoGNN, a novel framework to enable secure and efficient collaborative GNN learning. CoGNN is empowered by two fundamental designs: (i) a novel *oblivious message passing paradigm* to support the vertex data propagation/aggregation in GNN training and inference, which reduces the communication cost by half compared with the state-of-the-art (SOTA) (see § 5.1); (ii) a *two-stage Dispatch-Collect* execution scheme built upon the message passing paradigm to securely and efficiently decompose and distribute the workload of GNN computation across all parties for scalable computation (see § 5.2). In particular, our oblivious message passing paradigm stores vertices and edges as separated lists and uses efficient and oblivious interaction between the two lists to express vertex data propagation/aggregation. At the same time, the two-stage Dispatch-Collect decomposes GNN computation (including message passing) into Dispatch tasks and Collect tasks and distributes these tasks to different parties for concurrent and scalable execution. Each task is a secure computation workload executed by a pair of parties. The task distribution or assignment leverages the secret topology information held by each party to avoid computationally expensive cryptographic operations (e.g., oblivious sort or secure array access).

We instantiate the CoGNN framework to train Graph Convolutional Network (GCN) models and provide detailed construction of all the Dispatch and Collect tasks to enable GCN computations (see § 6). The instantiation includes further optimizations to improve the concrete efficiency of computing GCN (see § 6.6).

Our implementation of CoGNN includes $\sim$ 6800 lines of C++ code. We evaluate it extensively on commonly used graph datasets (see § 8). Compared to the state-of-the-art SML-based approach in training GCN, CoGNN reduces the running duration and per-party communication significantly. The advantages are enlarged as the number of graph owners increases, indicating better scalability of CoGNN. In particular, when there are 5 graph owners, CoGNN reduces the running duration by 123x, and reduces per-party communication by 522x. Moreover, CoGNN has a model performance comparable to plaintext global-graph training and surpasses the FL-based approach significantly. Compared to the FL-based approach, CoGNN has an accuracy elevation up to 3.87% in the two-party setting, and up to 11.06% in the five-party setting.

Contributions. The primary contributions of this paper are the design, implementation and evaluation of CoGNN, a novel framework to enable collaborative GNN learning in a provably secure, efficient and scalable manner. Concretely, our contributions are summarized as follows:

- We propose CoGNN, a secure, efficient and scalable framework for collaborative GNN learning among multiple graph owners. Inside CoGNN, we design a novel oblivious message-passing mechanism that halves the communication of the SOTA approach. Based on this, we introduce a two-stage Dispatch-Collect scheme to securely and efficently execute the workload of GNN training and inference.
- We instantiate the CoGNN framework with concrete protocols to train Graph Convolutional Networks (GCNs). Meanwhile, we propose customized optimizations tailored for GCNs to further improve its learning efficiency.
- We implement the CoGNN framework and evaluate it extensively on multiple datasets used in GCN learning. Experimental results show that, compared to the SOTA SML-based approach,

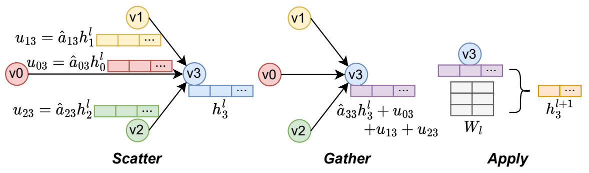
v1
$u_{13} = \hat{a}_{13}h_1^l$
$u_{03} = \hat{a}_{03}h_0^l$
v0
v3
$u_{23} = \hat{a}_{23}h_2^l$
v2
h_3^l
Scatter
v1
v0
v3
v2
$\hat{a}_{33}h_3^l + u_{03} + u_{13} + u_{23}$
Gather
v3
W_l
h_3^{l+1}
Apply

Figure 1: GCN forward pass as vertex-centric computation.

CoGNN reduces up to 123x running time and up to 522x communication cost per party. The improvement is enlarged as the number of parties increases, demonstrating the superior scalability of CoGNN. Meanwhile, the GCN models trained using CoGNN have nearly identical accuracies as the plaintext global-graph training, yielding up to 11.06% accuracy improvement over the GCN models trained via federated learning.

2 BACKGROUND & PREVIOUS WORKS

2.1 Graph

We study a directed graph $G = (V, E)$, where V (E) is the vertex (edge) list. Each vertex $v \in V$ is a tuple (v.id, v.data), where v.id is a unique vertex identifier and v.data is the vertex data. Each edge $e \in E$ is (e.src, e.dst, e.data), where e.src (e.dst) is the identifier of the source (destination) vertex. We call edge e the outgoing edge of e.src and the incoming edge of e.dst. The number of outgoing (incoming) edges of a vertex is called its outgoing (incoming) degree. An undirected graph can be converted to an equivalent directed graph by decomposing each undirected edge into two directed edges in opposite directions. For simplicity, we use v_i to denote v that v.id $= i$, and use e_{ij} to denote e that e.src $= i$, e.dst $= j$.

2.2 Graph Neural Network

The input of GNN is a directed graph $G = (V, E)$, where V is the sample list and E connects the samples. Specifically, v.data contains the vertex features, while e.data stores the edge weight. We introduce the computation of GNN from a general perspective, leveraging the vertex-centric (or think-like-a-vertex) abstraction [33, 34]. In particular, a forward/backward GNN layer can be abstracted as the following three consecutive operations (i.e., the GAS model):
Scatter. For $\forall v \in V$, propagate the vertex data to each of its outgoing edges e and outputs an *update* u:

$$u \leftarrow \text{Scatter}(v.\text{data}, e.\text{data}), e.\text{src} = v.\text{id}$$

Gather. For $\forall v \in V$, aggregate the update data u originating from all of its incoming edges e with v.data:

$$v.\text{data} \boxplus \{u\} \leftarrow \text{Gather}(v.\text{data}, \{u \mid u.\text{dst} = v.\text{id}\})$$

Apply. Perform NN computation (like weight multiplication, non-linear activation) on v.data $\boxplus \{u\}$ to obtain the updated v.data$'$:

$$v.\text{data}' \leftarrow \text{Apply}(v.\text{data} \boxplus \{u\})$$

The Scatter-Gather operations are called *message passing*. Different GNN models vary in their constructions of the three operations.

2.3 Gragh Convolutional Network

We elaborate on the most used GNN model, i.e., Graph Convolutional Network (GCN), and how it corresponds to the GAS model.

Forward Pass. The computation of GCN centers around the adjacency matrix $A \in \{0, 1\}^{|V| \times |V|}$, which expresses E ($a_{ij} = 1$ if and only if $e_{ij} \in E$). The features in V are expressed as a feature matrix X. The computation of the l-th forward layer of GCN is:

$$H_{l+1} = \sigma(\hat{A}H_l W_l). \tag{1}$$

H_l is the l-th layer hidden representation of all the vertices, where $H_0 = X$. W_l is the trainable weight matrix of the l-th layer. $\hat{A}$ is the normalized version of A, where $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, $\tilde{A} = A + I$. $\tilde{D}$ is a diagonal matrix such that $\tilde{d}_{ii} = \Sigma_j \tilde{a}_{ij}$ (i.e., each diagonal cell storing the degree of a vertex). $\hat{A}H_l$ actually corresponds to the message passing process. $\sigma(.)$ denotes a non-linear activation function, e.g., $ReLU$ and $Softmax$. $Softmax$ is typically used at the last layer to produce the probabilities of each class for all vertices.
Forward Pass as GAS. In the GAS model, the execution of $\hat{A}H_l$ is decomposed into two steps, which correspond to the Scatter and Gather phases respectively. Figure 1 is an example showing the related GAS computations for v_3 at the forward layer l. At the beginning of this GAS iteration, for each vertex v_i, v_i.data stores its hidden representation h_i^l, which corresponds to a row in H_l. The e_{ij}.data for edge e_{ij} is its weight $\hat{a}_{ij}$, which is an element in $\hat{A}$.

During Scatter, we multiply the hidden representation h_i^l of each vertex v_i with the weight $\hat{a}_{ij}$ on its corresponding outgoing edge to create the update $u_{ij} = \hat{a}_{ij}h_i^l$ for its destination vertex v_j. During *Gather*, we add up the hidden representation h_j^l of the destination vertex v_j with all of its incoming updates. The merge (aggregate) operation, i.e., $\boxplus$, of GCN is a weighted sum. For example, in Figure 1, $h_3^l \boxplus u_{03} \boxplus u_{13} \boxplus u_{23} = \hat{a}_{33}h_3^l + u_{03} + u_{13} + u_{23}$.

During Apply, the result of each vertex from Gather is first multiplied with the weight matrix W_l, and then passed to the activation function σ. The final output of Apply for v_j is h_j^{l+1}, i.e., the hidden representation of the next layer. By combining all h_j^{l+1} from each v_i, we obtain H_{l+1} in Equation (1).
Backward Pass. The backward pass of a GCN produces gradients for all the trainable weights following the chain rule. The backward computation of GCN at the $(l-1)$-th layer is defined as:

$$D_{l-1} = (\hat{A}H_{l-1})^T(\sigma'(Z_{l-1}) \odot \hat{A}^T G_l), \tag{2}$$

$$G_{l-1} = (\sigma'(Z_{l-1}) \odot \hat{A}^T G_l)W_{l-1}^T. \tag{3}$$

D_l denotes the gradient w.r.t. W_l, while G_l denotes the gradient w.r.t. $\hat{A}H_l$. σ' is the derivation of σ. $\odot$ represents elementwise multiplication, while $(\cdot)^T$ represents the transposition of a matrix. $Z_l = \hat{A}H_l W_l$ is computed during the forward pass. When using cross-entropy loss, for a K-layer GCN, $D_{K-1} = (\widehat{A}H_{K-1})^T(\hat{Y} - Y)$, where Y is the ground-truth prediction results. $G_{K-1} = (\hat{Y} - Y)W_{K-1}^T$.
Backward Pass as GAS. From Equation (3) we can see that the computation of G_l is symmetric to H_l. Specifically, to derive G_{l-1} from G_l, we first multiply it with $\hat{A}^T$, and then perform element-wise multiplication with $\sigma'(Z_{l-1})$ before multiplying W_{l-1}^T. As a result, when expressing GCN backward layer $l-1$ in the GAS model, the input data of each vertex v_j is its corresponding row in G_l, i.e., g_j^l. The Scatter and Gather phases execute $\hat{A}^T$, while the Apply phase performs the multiplication with $\sigma'(Z_{l-1})$ and W_{l-1}^T. The output of Apply for v_j is its corresponding row in G_{l-1}, i.e., g_i^{l-1}.

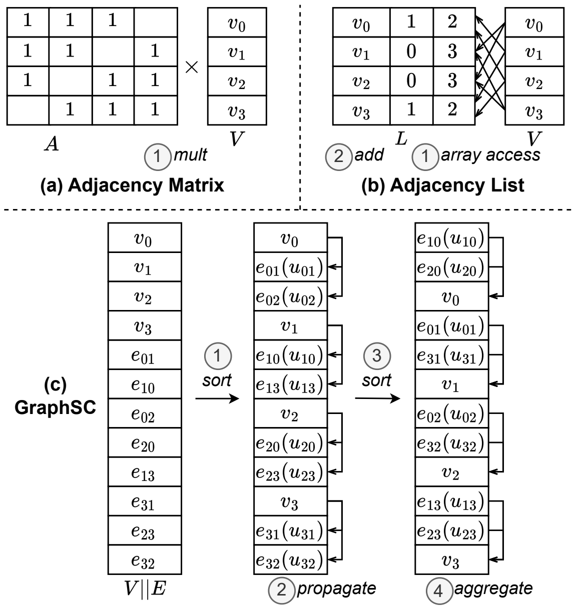

Figure 2: Prior approaches of oblivious message passing.

The byproduct of Apply in each vertex v_j is a gradient matrix D^j_{l-1} for W_{l-1}. It is derived by multiplying v_j's corresponding column of $(\hat{A}H_{l-1})^T$ with its row in $\sigma'(Z_{l-1}) \odot \hat{A}^T G_l$. Note that, D_{l-1} is actually the sum of the gradient matrices D^j_{l-1} generated in all vertices v_j. To obtain D_{l-1}, at the end of the corresponding backward GAS iteration, we add an aggregation step to average the gradients produced in each vertex. D_{l-1} can then be applied to update the weight matrix W_{l-1}.

2.4 Previous Works

The main technical challenge in achieving provably secure GNN learning in SML-based approaches is to efficiently perform oblivious message passing. In this section, we overview and compare prior approaches of oblivious message passing. Without loss of generality, we discuss a simplified case of message passing, i.e., the Scatter operation copies v.data to all $\{u \mid u.\text{src} = v.\text{id}\}$ and the Gather operation adds up v.data and all $\{u \mid u.\text{dst} = v.\text{id}\}$. In the form of adjacency matrix, it equals computing AH, where $A \in \{0,1\}^{|V|\times|V|}$. Suppose that A is held by P_0, while H is additively secret-shared between P_0 and P_1. The size of each element of H is m bits. For fair comparison, we assume that all costly operations, e.g., multiplication, array access and sort, have been preprocessed offline and focus on comparing the optimal online efficiency.

Adjacency Matrix-Based. The naive approach is to execute AH directly as secure matrix multiplication, as shown in (a) of Figure 2. H corresponds to the vertex list V. Utilizing the matrix multiplication triple [39] prepared offline, the online communication overhead (send + receive) of each party is $2m|V|^2 + 2m|V|$. However, [39] shows that the offline duration is over 10 times the online duration.

Adjacency List-Based. For a sparse adjacency matrix A, SecGNN [50] stores A as an adjacency list L and avoids the huge effort of preprocessing matrix multiplication triples by retrieving vertex data from the vertex list using secure array access [8]. As shown in (b) of Figure 2, each row of L records a vertex and the indices of all the vertices connected to it. SecGNN appends dummy indices to each row to make the degree of each vertex match the highest degree in the original graph. The message passing process specified by AH is then decomposed into two steps: (1) for each neighbor of each vertex, run an oblivious array access operation [8] to V to extract the corresponding vertex data; (2) for each vertex, sum up its neighbors' data obtained from the previous step. Since the online communication of each array access is $m|V|$, the method of [50] has a total online communication of at least $m|E||V|$, i.e., when no dummy indices are added.

GraphSC-Based. Recently, Entrada [26] introduced GraphSC [42], a vertex-centric secure graph processing paradigm, to secure GNN computation. As shown in (c) of Figure 2, the GraphSC paradigm works by storing vertices and edges in the same list $V||E$, where E expresses the graph topology defined in A. Executing AH takes four steps: (1) sort $V||E$ to place the edges right after their source vertices; (2) obliviously propagate the data of each vertex to its outgoing edges to generate updates; (3) sort $V||E$ again to place the edges right before their destination vertices; (4) obliviously aggregate the updates of each edge to their destination vertices. The sort operations in steps (1) and (3) can be specified by a permutation π and executed using permutation correlations online [12]. Each sort takes $m(|E| + |V|)$ online communication. The original propagate and aggregate schemes provided in [42] are both $O(n\log(n))$, where $n = |E| + |V|$. Recent work [7] shows that they can be replaced by prefix adder network-based oblivious computation to achieve linear complexity. In our case, each of propagate and aggregate takes $2(|E| + |V|)$ OTs and requires at least $4m(|E| + |V|)$ online communication. Taken together, the overall communication cost of executing AH using the GraphSC paradigm is at least $10m(|E|+|V|)$. In § A.4, we show GraphSC surpasses the two prior approaches significantly and is the state-of-the-art approach for secure GNN computation, mainly due to its linear complexity w.r.t. $|E| + |V|$.

3 PRELIMINARIES

3.1 Cryptographic Primitives

Fixed-Point Encoding and Secret Share. In CoGNN, we encode all the graph data as a Fixed-Point representation over the ring, $\mathbb{Z}_L$, where $L := 2^l$. The 2-out-of-2 additive secret share of a fixed-point number x over $\mathbb{Z}_L$ is straightforward. We denote the i-th share as $\langle x\rangle_i$, $i \in \{0,1\}$. By randomly sampling $\langle x\rangle_0 \in \mathbb{Z}_L$, we get $\langle x\rangle_1 = x - \langle x\rangle_0 \pmod L$. We use $\langle x\rangle$ to denote that x is in secret-shared form. For simplicity, we omit the $\langle\cdot\rangle$ symbol in most of our texts, while including it in the formal specifications of protocols, ideal functionality, and theorems/lemmas.

Secure Computation on Secret-Shared Data. We consider a general-purpose semi-honest two-party secure computation (2PC) protocol over 2-out-of-2 additively secret-shared data, e.g., [14, 44]. Apart from arithmetic operations, e.g., addition, subtraction, multiplication and division, we require logic operations, including comparison and two-way multiplexer (MUX$_2$) [44].

$\mathcal{F}_{\text{CoGNN}}$: $(\{E_i\}, \{E_{i,j}\}, \{V_i\}, \{W_l\}, \text{alg}) \rightarrow (\{V_i'\}, \{W_l'\})$

$\mathcal{F}_{\text{CoGNN}}$ interacts with N parties, $(P_0, P_1, ..., P_{N-1})$.

Input:

(1) For $i \in [N]$, P_i sends $(E_i, E_{i,j}, V_i)$ to $\mathcal{F}_{\text{CoGNN}}$;
(2) All P_i agree on a GNN specification, alg, which defines the computation details, e.g., GNN model, training epoch, loss function, and learning rate. They send alg to $\mathcal{F}_{\text{CoGNN}}$.
(3) All P_i agree on the initial model weight of each layers, i.e., $\{W_l\}, l \in [K]$. They send $\{W_l\}$ to $\mathcal{F}_{\text{CoGNN}}$.

Compute:

(1) $\mathcal{F}_{\text{CoGNN}}$ concatenates the received subgraph data as a global graph G, where $G = (V, E)$, $V := \cup V_i$ and $E := (\cup E_i) \cup (\cup E_{i,j})$;;
(2) $\mathcal{F}_{\text{CoGNN}}$ runs GNN training/inference on G according to the specification of alg. It stores the obtained vertex embedding in the updated vertex lists $\{V_i'\}, i \in [N]$ and stores the trained model weights in $\{W_l'\}, l \in [K]$.

Output:

(1) For $i \in [N]$, $\mathcal{F}_{\text{CoGNN}}$ sends V_i' and $\{W_l'\}, l \in [K]$ to P_i;
(2) Additionally, $\mathcal{F}_{\text{CoGNN}}$ sends the subgraph sizes to P_i, i.e., $\mathcal{L} := \{|E_j|, |V_j|\}, \forall j \in [N] \backslash \{i\}$ representing the numbers of intra-edges and vertices of each subgraph.

Functionality 1: $\mathcal{F}_{\text{CoGNN}}$

3.2 Oblivious Vector Operations

In this segment, we introduce the two cryptographic primitives for oblivious vector operations used in our oblivious message passing discussed in § 5.1: *oblivious extended permutation (OEP)* [38] and *oblivious group aggregation (OGA)* [7].

$\mathcal{F}_{\text{OEP}}$ performs an extended permutation π to T_{in}:

$$\langle T_{\text{out}} \rangle = \langle \pi(T_{\text{in}}) \rangle \leftarrow \mathcal{F}_{\text{OEP}}(\langle T_{\text{in}} \rangle, Id_{\text{in}}, Id_{\text{out}}) \quad (4)$$

π is defined by a pair of identifier lists $(Id_{\text{in}}, Id_{\text{out}})$, corresponding to T_{in} and T_{out} respectively. Elements in Id_{in} are unique, while each element of Id_{out} equals one element of Id_{in}. π permutes Id_{in} to Id_{out}. Each vector element in T_{out} (i.e., $T_{\text{out}}[x]$) is a random reshare of a specific vector element in T_{in} (i.e., $T_{\text{in}}[\pi(x)]$). We extend the definition of $\mathcal{F}_{\text{OEP}}$ to support that some identifiers in Id_{out} might not have an equal identifier in Id.in, and $\mathcal{F}_{\text{OEP}}$ sets the corresponding elements in $\langle T^{\text{out}} \rangle$ to 0.

$\mathcal{F}_{\text{OGA}}$ aggregates the elements in T_{in} using the operation $\boxplus$:

$$\langle T_{\text{out}} \rangle \leftarrow \mathcal{F}_{\text{OGA}}(\langle T_{\text{in}} \rangle, \mathcal{G}, \boxplus) \quad (5)$$

Specifically, $\mathcal{G}$ is a group identifier vector of the same length as T_{in}, and can be split into segments, each containing elements of the same value. $\mathcal{F}_{\text{OGA}}$ aggregates T_{in} elements with the same group identifier (i.e., in the same segment) and stores the result in the first element of the corresponding segment, producing T_{out}. For the rest of elements in each segment of T_{out}, their values are undefined.

4 THE IDEAL FUNCTIONALITY OF COGNN

In this section, we introduce the ideal functionality of CoGNN (i.e., $\mathcal{F}_{\text{CoGNN}}$, shown in Functionality 1) to formally capture its input setting, computation goal, threat model, and privacy guarantee.

There are N participating parties, i.e., $(P_0, P_1, ..., P_{N-1})$. P_i locally holds the subgraph $G_i = (V_i, E_i)$. The sets of vertex identifiers held by different parties are exclusively different. Each edge $e \in E_i$ is called an intra-edge since both e.src and e.dst are in V_i.

There could be inter-edges between two subgraphs G_i and G_j, $i \neq j$. We use $E_{i,j}$ to denote the edge list directed from G_i to G_j. For $e \in E_{i,j}$, both P_i and P_j learn (e.src, e.dst, e.data). Nevertheless, we suppose that only P_i, the source party of all the edges in $E_{i,j}$, needs to maintain it. CoGNN requires that the edge lists (E_i, $E_{i,j}$, $j \in [N] \backslash \{i\}$) held by each party P_i, $i \in [N]$ are ordered by placing the edges with the same destination vertex as a contiguous *segment* in E_i ($E_{i,j}$) (i.e., these edges are located at adjacent positions).

At the input stage, for $i \in [N]$, P_i sends $(E_i, E_{i,j}, V_i)$ to $\mathcal{F}_{\text{CoGNN}}$. Additionally, they agree on a GNN specification alg and the initial model weights $\{W_0, W_1, ..., W_{K-1}\}$, and send them to $\mathcal{F}_{\text{CoGNN}}$. In particular, alg specifies GNN computation details such as the GNN model, training epochs, loss function, and learning rate.

During computation, $\mathcal{F}_{\text{CoGNN}}$ first concatenates the received separate subgraphs into a global graph G. The concatenation is based on inter-edges: i.e., $G = (V, E)$, $V := \cup V_i$, $E := (\cup E_i) \cup (\cup E_{i,j})$. Then, $\mathcal{F}_{\text{CoGNN}}$ performs GNN training/inference according to the specification of alg. The computation result consists of two parts: (i) the updated vertex set lists, i.e., $\{V_0', V_1', ..., V_{N-1}'\}$, which store the embedding (prediction values) of each vertex; (ii) the updated model weights, i.e., $\{W_0', W_1', ..., W_{K-1}'\}$, trained on the global graph.

At the output stage, for all $i \in [N]$, $\mathcal{F}_{\text{CoGNN}}$ sends the vertex embedding V_i' and trained model weights $\{W_0', W_1', ..., W_{K-1}'\}$ to P_i. Additionally, $\mathcal{F}_{\text{CoGNN}}$ sends the subgraph size, i.e., numbers of vertices and edges, of each party to P_i.

Threat Model. In CoGNN, we assume that each party is semi-honest, i.e., these parties are protocol compliant, while being curious of deriving other parties' private data. In addition, we assume that these parties do not collude with each other. The rationale behind this threat model is that CoGNN primarily considers the non-outsourced computation setting, where the graph owners themselves act as the computing parties while keeping their raw data local. Therefore, colluding would cause privacy risks to their own data, while behaving maliciously may eventually degrade the overall collaboration gain. In contrast, prior SML-based approaches require the data owners to outsource their raw data to third-party servers. Thus, it is essential for these approaches to ensure security when these third parties collude with each other or even act arbitrarily. In § 9, we discuss how to extend CoGNN to other security models.

Private Data. The private data of party $P_i, \forall i \in [N]$ includes: (i) intra-edge-related information, i.e., (e.src, e.dst, e.data) of each edge $e \in E_i$; (ii) (v.id, v.data) for $v \in V_i$. In addition, all the intermediate data computed using the aforementioned private data from the other subgraphs must be kept secret throughout the process.

Public Data. The public data *between each pair of parties* (P_i and P_j, $i, j \in [N], i \neq j$) includes: (i) inter-edge-related information, i.e., (e.src, e.dst, e.data) of each edge $e \in E_{i,j} \cup E_{j,i}$; (ii) the sizes of subgraphs (i.e., $|V_i|$, $|V_j|$, $|E_i|$, $|E_j|$).

5 THE COGNN FRAMEWORK

In this section, we present the CoGNN framework to securely realizing $\mathcal{F}_{\text{CoGNN}}$ with a fully distributed and secure computation scheme. Specifically, we begin with a novel oblivious message passing mechanism that securely and efficiently expresses the Scatter-Gather of GNN. Based on that, we present how CoGNN securely

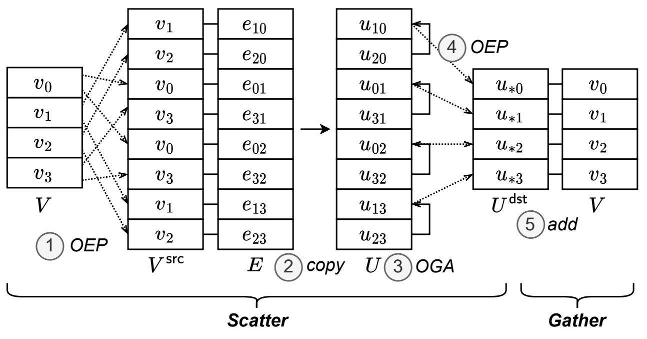

Figure 3: Our oblivious message passing.

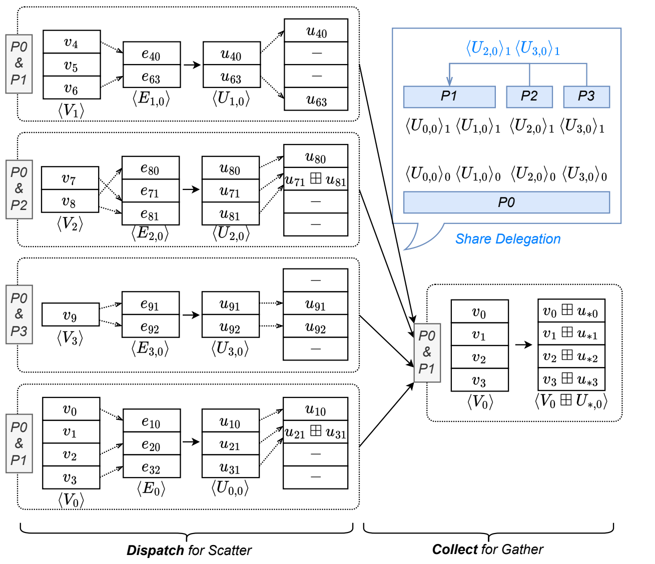

Figure 4: The first-stage Dispatch-Collect in CoGNN: for Scatter and Gather

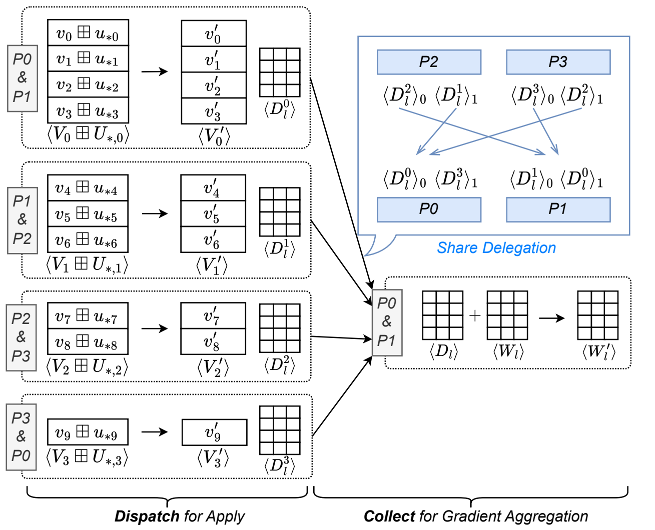

Figure 5: The second-stage Dispatch-Collect in CoGNN: for Apply and Gradient Aggregation

decomposes and distributes the vertex-centric computation of GNN models across all parties by executing each GNN forward/backward layer in the two stages of Dispatch-Collect.

5.1 Our Oblivious Message Passing

We introduce the principle of our oblivious message passing (Scatter-Gather) method using the same setting as in § 2.4, i.e., when the Scatter operation copies v.data to all $\{u \mid u.\text{src} = v.\text{id}\}$ and the Gather operation adds up v.data and all $\{u \mid u.\text{dst} = v.\text{id}\}$.

Instead of storing all the vertices and edges in one list like in GraphSC, we place them in V and E respectively, as shown in Figure 3. Based on that, our oblivious message passing scheme runs in five steps: (1) using OEP (our protocol that realizes the primitive $\mathcal{F}_{\text{OEP}}$) to duplicate each vertex to all of its outgoing edges to create a source vertex list V^{src}; (2) copy V^{src} to obtain the updates U; (3) aggregate updates that share the same destination vertices using OGA (our protocol that realizes the primitive $\mathcal{F}_{\text{OGA}}$); (4) Use OEP to place each update at the same vector position as its destination vertex, creating U^{dst}; (5) locally add updates to vertices.

Steps (1) to (4) correspond to Scatter, while step (5) expresses Gather. Among them, steps (2) and (5) are local. For steps (1) and (4), our OEP protocol works in an offline-online paradigm, where we prepare permutation correlations in batch offline (for multiple online calls), and consume these correlations online. The online communication of each invocation of our OEP protocol is the same as the size of its input vector. As a result, steps (1) and (4) take online $m(|E| + |V|)$ communication in total. Our OGA protocol takes the prefix adder network paradigm, resulting in $4m|E|$ communication. Taken together, our message passing approach has $m(5|E| + |V|)$ online communication, which is less than half of that of the state-of-the-art approach based on GraphSC. More importantly, it enables the following distributed and secure computation scheme in CoGNN. Due to page limit, the detailed protocol constructions of OEP and OGA are deferred to § A.1 and § A.2, respectively.

5.2 Iteration as Two-Stage Dispatch-Collect

CoGNN executes each forward/backward layer (i.e., a GAS iteration) of a GNN model using two stages of Dispatch-Collect. The high-level principle of the Dispatch-Collect-based CoGNN design can be summarized as: (i) using the Dispatch phase to securely distribute the heavy part of vertex-centric GNN computation workload across all the parties; (ii) using the Collect phase to perform lightweight aggregation across different parties when necessary. In particular, each Dispatch task involves two worker parties and the computed data is secret-shared between the two parties. During Collect, the output shares produced by each Dispatch task are delegated to specific aggregator parties, which securely aggregate the data and feed the result shares back to the Dispatch worker parties.

We use a four-party example presented in Figure 4 and Figure 5 as the illustrative example when describing our protocol below.

5.2.1 The First-Stage Dispatch-Collect. In the first-stage Dispatch-Collect, the Dispatch phase processes the Scatter workload in a fully distributed manner, while the Collect phase performs Gather. CoGNN distributes the Scatter workload based on the data partition inherent to its input setting. Specifically, the Scatter from V_j to $E_{j,i}, j \in [N]\backslash\{i\}$ is encapsulated as a Dispatch task and is handled by (P_j, P_i) using 2PC. Accordingly, V_j, $E_{j,i}$ and the update

$U_{j,i}$ produced by Scatter have to be secret-shared between (P_j, P_i). As for the Scatter from V_i to E_i, we have (P_i, P_{i+1}) secret-share $(V_i, E_i, U_{i,i})$ and execute the corresponding Scatter computation as a Dispatch task. When $i = N - 1$, $i + 1$ wraps around to 0.

The example of Figure 4 illustrates the Scatter-Gather computations related to the update of V_0. In Figure 4, the updates targeting V_0 come from four sources, i.e., V_1, V_2, V_3, V_0. To compute these updates, the Scatter workload is decomposed into four Dispatch tasks, handled by (P_0, P_1), (P_1, P_2), (P_2, P_3), (P_0, P_1) respectively.

The Collect phase, conducting Gather, aims to merge the secret-shared updates $U_{j,i}, \forall j \in [N]$ produced by different Dispatch tasks with V_i. Since different $U_{j,i}$ is secret-shared between a different pair of parties (P_j, P_i), at the beginning of the Collect phase, we have $P_j, j \in [N]\backslash\{i, i+1\}$ delegate its share of $U_{j,i}$ to P_{i+1}. As a result, (P_i, P_{i+1}) serve as the delegator parties handling the Collect task, i.e., performing the Gather computation for V_i on behalf of the rest of the parties. Nevertheless, all the delegated data is still secret-shared between (P_i, P_{i+1}) and neither of them obtains the plaintext data. The output produced by the Collect task, i.e., $V_i \boxplus U_{*,i}$ representing V_i merged with all the updates from all P_j, is also secret-shared between (P_i, P_{i+1}). In the example of Figure 4, $\langle U_{1,0}\rangle, \langle U_{2,0}\rangle, \langle U_{3,0}\rangle, \langle U_{0,0}\rangle$ are secret-shared among P_0 and different parties. To apply them to update $\langle V_0\rangle$, we have P_2, P_3 delegate their shares to P_1. After that, P_0 and P_1 merge $\langle U_{1,0}\rangle, \langle U_{2,0}\rangle, \langle U_{3,0}\rangle, \langle U_{0,0}\rangle$ with $\langle V_0\rangle$ consecutively, and obtain the merged $\langle V_0 \boxplus U_{*,0}\rangle$.

5.2.2 The Second-Stage Dispatch-Collect. In the second-stage Dispatch-Collect, the Dispatch phase distributedly processes the Apply workload to perform NN computation on the vertex data of each subgraph and (optionally) produces gradients, while the Collect phase aggregates gradients generated by different subgraphs.

The secret-shared results produced by the Collect phase of the first-stage Dispatch-Collect are directly supplied to the Dispatch task of the second-stage Dispatch-Collect. Specifically, the Apply workload for V_i is encapsulated as a Dispatch task and assigned to (P_i, P_{i+1}). The input of the Apply computation, i.e., $V_i \boxplus U_{*,i}$, is inherently secret-shared between (P_i, P_{i+1}) due to the Collect of the previous stage of Dispatch-Collect. While the Apply computation aims to perform NN computation on the vertex data, the computation details depend on specific GNN models and the forward or backward layer that is executed. The main output of each Dispatch task is the updated vertex data V_i' secret-shared between (P_i, P_{i+1}). In the example of Figure 5, there are four Dispatch tasks, corresponding to the Apply computation of $V_0 \boxplus U_{*,0}$, $V_1 \boxplus U_{*,1}$, $V_2 \boxplus U_{*,2}$, and $V_3 \boxplus U_{*,3}$ respectively.

When a backward GNN layer (suppose it is layer l) is executed, the by-product of each Dispatch task is a gradient D_l^i generated from V_i, also secret-shared between (P_i, P_{i+1}). The goal of the Collect phase is to aggregate all $D_l^i, \forall i \in [N]$ and produce a global gradient D_l for W_l. Still, since different D_l^i is secret-shared between a different pair of parties (P_i, P_{i+1}), we have all parties $P_i, i \in [N]\backslash\{0, 1\}$ delegate their shares of D_l^i to P_0 and P_1. In particular, $\langle D_l^i\rangle_1$ is delegated to P_0, while $\langle D_l^i\rangle_0$ is delegated to P_1. (P_0, P_1) then handle the Collect task to aggregate these gradients and apply D_l to optimize W_l. All the data involved in the Collect task is secret-shared between (P_0, P_1).

Note that, here we select (P_0, P_1) as the delegator parties of the Collect task of this Dispatch-Collect stage. In practice, we can select a different pair of parties as the delegator parties of different Collect tasks to better balance the overhead among all the parties.

5.3 Cascade Iterations by Share Redistribution

The last problem for us is how to cascade iterations, i.e., how to organize the output from the current iteration to serve later iterations. From the above construction, the Dispatch phase of the second-stage Dispatch-Collect secretly shares the updated V_i' between (P_i, P_{i+1}). However, in the next GAS iteration, V_i' is assigned to V_i and is treated as the input of the Dispatch tasks (in the first-stage Dispatch-Collect) executed by $(P_i, P_j), j \in [N]\backslash\{i\}$. In other words, V_i' has to be secret-shared between $(P_i, P_j), j \in [N]\backslash\{i\}$ again. Thus, we have (P_i, P_{i+1}) randomize their shares of V_i' and have P_{i+1} send its share $\langle V_i'\rangle_1 + R_j$ to $P_j, j \in [N]\backslash\{i, i+1\}$, while having P_i keep $\langle V_i'\rangle_0 - R_j$. We call this process *share redistribution.*

In a backward layer, the optimized W_l' is secret-shared between (P_0, P_1) after the Collect phase of the second-stage Dispatch-Collect. To utilize W_l' in a later forward layer's Dispatch tasks (in the second-stage Dispatch-Collect), we have to perform share redistribution again to make W_l' secret-shared between all pairs of $(P_i, P_{i+1}), i \in [N]$. Specifically, we have (P_0, P_1) randomize their shares of W_l' and have P_1 send its share $\langle W_l'\rangle_0 + R_i$ to $P_i, i \in [N]\backslash\{0, 1\}$, while having P_0 send its share $\langle W_l'\rangle_1 - R_{i-1}$ to $P_i, i \in [N]\backslash\{0, 1\}$.

6 INSTANTIATE GCN UNDER COGNN

In this section, we provide the details of how we instantiate CoGNN to support the training and inference of Graph Convolutional Network (GCN) models. Specifically, we elaborate on the design of each Dispatch task and Collect task, and how we feed the outcomes of previous tasks to later tasks. The specification of each Dispatch task and Collect task is shown in Protocol 1. The thorough protocol description, i.e., CoGNN-GCN, is provided in Protocol 2.

Notations. We introduce some key notations in protocol description. We use V.id to represent the vertex identifier list of a vertex list, and use E.src/E.dst to represent the source/destination vertex identifier list of an edge list. In addition, we use V.deg to denote the degree list of a vertex list. Accordingly, we use E.srcDeg/E.dstDeg to denote the list of degrees corresponding to the source/destination vertices of an edge list. $X^{-\frac{1}{2}}$ means taking the reciprocal of each element in the list X, and then taking the square root of it.

6.1 Graph Data Secret Share

Each P_i breaks its V_i into 2-out-of-2 additive secret shares, keeps one share as secret, and distributes the other share to all the other parties. At the same time, E_i and $E_{i,j}, \forall j \in [N]\backslash\{i\}$ are also broken into secret shares. However, unlike V_i, E_i is only shared with its neighboring (next) party P_{i+1}. $E_{i,j}$ is only shared with party P_j.

Initially, V_i stores $H_0^i = X_i$, i.e., the features of vertices. For simplicity, the input preparation process is omitted in Protocol 2.

6.2 The First-Stage Dispatch-Collect

6.2.1 Dispatch Task. A Dispatch task in the first-stage Dispatch-Collect (denoted as Dispatch-1) is executed by two parties $(P_i, P_j), i, j \in$

```
1  Dispatch-1($i, j, \langle V\rangle, \langle E\rangle, V^{\text{dst}}$.id) $\longrightarrow \langle U^{\text{dst}}\rangle$          # for Scatter
2      $\langle V^{\text{src}}\rangle \leftarrow$ OEP($\langle V\rangle, V$.id, $E$.src)
3      $\langle U\rangle \leftarrow E.\text{srcDeg}^{-\frac{1}{2}} \odot \langle V^{\text{src}}\rangle \odot E.\text{dstDeg}^{-\frac{1}{2}}$
4      $\langle U'\rangle \leftarrow$ OGA($\langle U\rangle, E$.dst, $+$)
5      $\langle U^{\text{dst}}\rangle \leftarrow$ OEP($\langle U'\rangle, E$.dst, $V^{\text{dst}}$.id)
6      return $\langle U^{\text{dst}}\rangle$
7  Collect-1($i, j, \langle V\rangle, \{\langle U_k^{\text{dst}}\rangle\}$) $\longrightarrow \langle V \boxplus U_*^{\text{dst}}\rangle$          # for Gather
8      $\langle V'\rangle \leftarrow \langle V\rangle \odot V.\text{deg}^{-1}$
9      $\langle V \boxplus U_*^{\text{dst}}\rangle \leftarrow \langle V'\rangle + \sum_{k=0}^{N-1}\langle U_k^{\text{dst}}\rangle$
10     return $\langle V \boxplus U_*^{\text{dst}}\rangle$
11 Dispatch-2($i, j, l$, isForward, $\langle V\rangle, \langle W_l\rangle, ..., \langle Z\rangle, \langle AH\rangle$)
                              $\longrightarrow (\langle V'\rangle, ..., \langle Z\rangle, \langle \hat{Y}\rangle, \langle D_l\rangle)$ # for Apply
12     if isForward:
13         $\langle Z\rangle \leftarrow \langle V\rangle \cdot \langle W_l\rangle$
14         if $l \neq K-1$: $\langle V'\rangle \leftarrow ReLU(\langle Z\rangle)$
15         else: $\langle \hat{Y}\rangle \leftarrow Softmax(\langle Z\rangle)$; $\langle V'\rangle \leftarrow \langle \hat{Y}\rangle - V$.y
16     else:
17         if $l = k-1$: $\langle M\rangle \leftarrow \langle V\rangle$
18         else: $\langle M\rangle \leftarrow ReLU'(\langle Z\rangle) \odot \langle V\rangle$
19         $\langle D_l\rangle \leftarrow \langle AH\rangle \cdot \langle M\rangle$
20         if $l \neq 0$: $\langle V'\rangle \leftarrow \langle M\rangle \cdot \langle W_l\rangle$
21     return $\langle V'\rangle, ..., \langle Z\rangle, \langle \hat{Y}\rangle, \langle D_l\rangle$
22 Collect-2($i, j, \langle W_l\rangle, \{\langle D_l^k\rangle\}, \{|V_k|\}$) $\longrightarrow \langle W_l'\rangle$     # for Gradient
23     $\langle D_l\rangle \leftarrow \frac{\sum_{k=0}^{N-1}(|V_k|\cdot\langle D_l^k\rangle)}{\sum_{k=0}^{N-1}|V_k|}$          # Aggregation
24     $\langle W_l'\rangle \leftarrow Opt(\langle W_l\rangle, \langle D_l\rangle)$          # Application
25     return $\langle W_l'\rangle$
```

Protocol 1: Dispatch tasks and Collect tasks in the first-stage and the second-stage Dispatch-Collect.

$[N], i \neq j$ and performs Scatter for GCN. The inputs of Dispatch-1 include a vertex list $\langle V\rangle$ and an edge list $\langle E\rangle$. The output is an update list $\langle U^{\text{dst}}\rangle$. All these data is secret-shared between (P_i, P_j).

Based on our message passing mechanism introduced in § 5.1, in this Dispatch-1, the Scatter for GCN is executed in four steps.

(1) Use OEP to construct a source vertex list $\langle V^{\text{src}}\rangle$ corresponding to E.src, as line 2 of Protocol 1. Both V.id and E.src are supplied by P_i.
(2) $\langle V^{\text{src}}\rangle$ is then used to perform element-wise computation with $\langle E\rangle$, to generate an update list of the same length as $\langle E\rangle$, as line 3 of Protocol 1. E.srcDeg and E.dstDeg are supplied by either P_i or P_j, depending on who holds them. $\odot$ represents element-wise multiplication between the two lists.
(3) Merge the updates in $\langle U\rangle$ that share the same destination vertices. According to CoGNN's requirement of the input edge list order (see § 4), all the updates targeting the same destination vertex are already placed in a contiguous segment. So we invoke OGA for this, as line 4 of Protocol 1. The operation to merge two updates is an addition, i.e., $+$.
(4) Use OEP to extract the merged updates and place them at the same positions as their destination vertices, as line 5 of Protocol 1. Note that V^{dst} represents the destination vertex list of E. V^{dst}.id is supplied by either P_i or P_j, depending on who owns it. Some identifiers in V^{dst}.id might not have an equal index in E.dst, and OEP sets their corresponding elements in $\langle U^{\text{dst}}\rangle$ to 0.

The Dispatch-1 tasks for a forward layer and a backward layer of GCN are the same. The only difference is in the data stored in the vertex list. In a forward layer, the vertex list stores the features or hidden representation of each vertex. In a backward layer, the vertex list stores a back-propagated intermediate gradient value. See Protocol 2 for the detailed inputs fed to each Dispatch-1 task.

6.2.2 Collect Task. In an N-party setting, the Collect-1 tasks are delegated to N pairs of parties, i.e., (P_0, P_1), (P_1, P_2),...,(P_{N-1}, P_0). Specifically, the Collect-1 task of (P_i, P_{i+1}) deals with the Gather of V_i. For V_i, suppose that the parties have already delegated their shares of update lists to P_{i+1}. Now (P_i, P_{i+1}) hold N secret-shared update lists, i.e., $\langle U_{0,i}^{\text{dst}}\rangle$, $\langle U_{1,i}^{\text{dst}}\rangle$,...,$\langle U_{N-1,i}^{\text{dst}}\rangle$. The goal of the Collect-1 task is to merge all these update lists with $\langle V_i\rangle$.

Thus, the inputs of the Collect-1 task are composed of a set of update lists $\{\langle U_k^{\text{dst}}\rangle\}$ and a vertex list $\langle V\rangle$. The output of the vertex list which has merged all the update lists, i.e., $\langle V \boxplus U_*^{\text{dst}}\rangle$.

For GCN, the Collect-1 task is executed in two steps:

(1) Scale the vertex list according to the degree of each vertex, as line 8 of Protocol 1, where V.deg is supplied by P_i.
(2) Add all the update lists to the vertex list, as line 9 of Protocol 1.

The Collect-1 tasks for forward and backward layers are the same. See Protocol 2 for the inputs fed to each Collect-1 task.

6.3 The Second-Stage Dispatch-Collect

6.3.1 Dispatch Task. In the second-stage Dispatch-Collect, each Dispatch-2 task performs the Apply computation for a $V_i, i \in [N]$ and is handled by a pair of parties (P_i, P_{i+1}). Since the Apply computations for different forward/backward layers differ, the Dispatch tasks are also different. Suppose that the GCN has K layers of NN computations. In total, there are five situations for the Dispatch-2 task, i.e., when the Dispatch-2 task is in a forward layer rather than the last layer (i.e., layer $K-1$), in the last forward layer, in the first backward layer (i.e., layer $K-1$). in the last backward layer (i.e., layer 0) and in the rest of the backward layers.

The main inputs of the Dispatch-2 task include a vertex list merged with updates, denoted as $\langle V\rangle$, and a weight matrix, denoted as $\langle W_l\rangle$. The main output of the Dispatch-2 task is the updated vertex list $\langle V'\rangle$. We discuss the above situations as follows:

- When the Dispatch-2 task is in a forward layer rather than the last layer, the Dispatch task executes weight multiplication and *ReLU* activation, as lines 13 and 14 of Protocol 1.
- When the Dispatch-2 task is in the last forward layer, it executes weight multiplication and *Softmax* normalization, as lines 13, 15 of Protocol 1. Here the additional output produced by the Dispatch task is the prediction list $\langle \hat{Y}\rangle$ for V. V.y denotes the label list for V. $\langle \hat{Y}\rangle - V$.y is the gradient of the cross-entropy loss w.r.t. $\langle Z\rangle$.
- When the Dispatch-2 task is in a backward layer rather than the first and the last backward layer, it executes lines 18, 19 and 20 of Protocol 1. Note that both $\langle Z\rangle$ and $\langle AH\rangle$ are intermediate computation results from forward layers and they serve as additional inputs to the Dispatch-2 task here.

Protocol CoGNN-GCN

Input. $P_i, \forall i \in [N]$ provides E_i, $E_{i,j}$ and $V_i = H_0^i = X_i$. Negotiated alg $= (K, maxEp)$, $\{W_l \mid l \in [K]\}$.
Output. $P_i, \forall i \in [N]$ gets $V_i' = P_i$ and $\{W_l' \mid l \in [K]\}$.

```
1  for $ep \in [maxEp]$:
2    for $l \in [K]$: # ............ Forward Pass, iterate from layer 0 to $K-1$
3      for $i \in [N]$: $\langle V_i\rangle \leftarrow \langle H_l^i\rangle$    # Set the layer input (vertex data)
4      for $i, j \in [N], i \neq j$:    # Scatter vertex embedding using Dispatch-1
5        $\langle U_{i,j}^{\text{dst}}\rangle \leftarrow$ Dispatch-1$(i, j, \langle V_i\rangle, \langle E_{i,j}\rangle, V_j.\text{id})$; if $j = i+1$: $\langle U_{i,i}^{\text{dst}}\rangle \leftarrow$ Dispatch-1$(i, i+1, \langle V_i\rangle, \langle E_i\rangle, V_i.\text{id})$
6      for $i \in [N]$:
7        for $j \in [N]\backslash\{i, i+1\}$: $P_j$ sends $\langle U_{j,i}^{\text{dst}}\rangle_1$ to $P_{i+1}$    # Delegate secret shares
8        $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle \leftarrow$ Collect-1$(i, i+1, \langle V_i\rangle, \{\langle U_k^{\text{dst}}\rangle\})$    # Gather using Collect-1
9        if $i \neq K-1$: $\langle V_i'\rangle, \langle Z_l^i\rangle \leftarrow$ Dispatch-2$(i, i+1, l, true, \langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle, \langle W_l\rangle)$    # Forward NN computation using Dispatch-2
10       else: $\langle V_i'\rangle, \langle Z_l^i\rangle, \langle P_i\rangle \leftarrow$ Dispatch-2$(i, i+1, l, true, \langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle, \langle W_l\rangle)$    # Forward prediction using Dispatch-2
11       for $j \in [N]\backslash\{i, i+1\}$: $P_{i+1}$ sends $\langle V_i'\rangle_1 + R_j$ to $P_j$    # Redistribute secret shares
12       $\langle H_{l+1}^i\rangle \leftarrow \langle V_i'\rangle$; $\langle \hat{A}H_l^i\rangle \leftarrow \langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle$    # Set the layer output
13   for $l \in [K]$ (reverse): # ............ Backward Pass, iterate from layer $K-1$ to 0
14     for $i \in [N]$:    # Set the layer input (vertex data)
15       if $l = K-1$: $\langle V_i\rangle \leftarrow \langle H_K^i\rangle$; else: $\langle V_i\rangle \leftarrow \langle G_l^i\rangle$
16     for $i, j \in [N], i \neq j$:    # Scatter intermediate gradients using Dispatch-1
17       $\langle U_{i,j}^{\text{dst}}\rangle \leftarrow$ Dispatch-1 $(i, j, \langle V_{i,j}\rangle, \langle E_{i,j}\rangle, V_j.\text{id})$; if $j = i+1$: $\langle U_{i,i}^{\text{dst}}\rangle \leftarrow$ Dispatch-1 $(i, i+1, \langle V_i\rangle, \langle E_i\rangle, V_i.\text{id})$
18     for $i \in [N]$:
19       for $j \in [N]\backslash\{i, i+1\}$: $P_j$ sends $\langle U_{j,i}^{\text{dst}}\rangle_1$ to $P_{i+1}$    # Delegate secret shares
20       $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle \leftarrow$ Collect-1$(i, i+1, \langle V_i\rangle, \{\langle U_k^{\text{dst}}\rangle\})$    # Gather using Collect-1
21       if $l = K-1$: $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle \leftarrow \langle V_i\rangle$    # For layer $K-1$, $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle$ is directly set to $\langle V_i\rangle$
22       if $i \neq 0$: $\langle V_i'\rangle, \langle D_l^i\rangle \leftarrow$ Dispatch-2 $(i, i+1, l, false, \langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle, \langle W_l\rangle, \langle Z_l^i\rangle, \langle(\hat{A}H_l^i)^T\rangle)$
23       else: $\langle D_l^i\rangle \leftarrow$ Dispatch-2$(i, i+1, l, false, \langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle, \langle W_l\rangle, \langle Z_l^i\rangle, \langle(\hat{A}H_l^i)^T\rangle)$    # Gradient computation using Dispatch-2
24     for $i \in [N]\backslash\{0,1\}$: $P_i$ sends $\langle D_l^i\rangle_0$ to $P_1$ and sends $\langle D_l^{j-1}\rangle_1$ to $P_0$    # Delegate secret shares of gradients
25     $\langle W_l'\rangle \leftarrow$ Collect-2$(0, 1, \langle W_l\rangle, \{\langle D_l^k\rangle\}, \{|V_k|\}), k \in [N]$    # Gradient aggregation and application using Collect-2
26     for $i \in [N]\backslash\{0,1\}$: $P_1$ sends $\langle W_l'\rangle_0 + R_i$ to $P_i$; $P_0$ sends $\langle W_l'\rangle_1 - R_{i-1}$ to $P_i$    # Redistribute secret shares of the new weight
27     for $i \in [N]\backslash\{0,1\}$: $\langle G_{l-1}^i\rangle \leftarrow \langle V_i'\rangle$; $\langle W_l\rangle \leftarrow \langle W_l'\rangle$    # Set the layer output
```

Protocol 2: The construction of CoGNN-GCN.

- When the Dispatch-2 task is in the first backward layer, it executes lines 17, 19 and 20 of Protocol 1.
- When the Dispatch-2 task is in the last backward layer, it executes lines 18, 19 of Protocol 1.

In Protocol 1, we use ... to separate the required inputs/outputs and the optional inputs/outputs of Dispatch-2. The data stored in $\langle V\rangle$ and $\langle V'\rangle$ has different meanings in different GCN forward/backward layers. See Protocol 2 for the details of how CoGNN prepares inputs for Dispatch-2 tasks of different layers.

6.3.2 Collect Task. In a backward layer, the second-stage Dispatch-Collect has a Collect phase (corresponding to the Collect-2 task). The goal of the Collect-2 task is to aggregate the gradient $\langle D_l^i\rangle$ produced on different $V_i, i \in [N]$, and to optimize the weight matrix $\langle W_l\rangle$ using the aggregated gradient. Suppose that all the secret shares of different $\langle D_l^i\rangle$ have been delegated to (P_0, P_1) according to the specification in § 5.2.2. The inputs of the Collect-2 task include a set of secret-shared gradients $\{\langle D_l^k\rangle\}, k \in [N]$ and the weight matrix $\langle W_l\rangle$. The output is the optimized weight matrix $\langle W_l'\rangle$.

The Collect-2 task consists of two steps:

(1) Obtain the global gradient by performing a weighted sum for all $\{\langle D_l^k\rangle\}$, as line 23 of Protocol 1.
(2) Optimize $\langle W_l\rangle$ using $\langle D_l\rangle$ as line 24 of Protocol 1. Here *Opt* abbreviates the weight optimizing process.

6.4 End-to-End Training and Inference

Training. As shown in line 3 Protocol 2, at the beginning of each forward layer l, the vertex data $\langle V_i\rangle$ of P_i is set to the hidden representation $\langle H_l^i\rangle$. For layer 0, $H_0^i = X_i$. $\langle V_i\rangle$ are fed to the Dispatch-1 tasks to compute update vectors $\langle U_{i,j}^{\text{dst}}\rangle, j \in [N]$. After that, the Collect-1 task aggregates all $\langle U_{j,i}^{\text{dst}}\rangle, j \in [N]$ to $\langle V_i\rangle$, and obtain $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle$. Then, the Dispatch-2 task performs NN-related computations on $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle$ to obtain $\langle V_i'\rangle$. If it is the last layer $K-1$, $\langle H_K^i\rangle = \langle V_i'\rangle$ stores $\langle \hat{Y}_i\rangle - V_i.y$. Note that $\langle V_i \boxplus U_{*,i}^{\text{dst}}\rangle$ is recorded as $\langle \hat{A}H_l^i\rangle$. Both $\langle \hat{A}H_l^i\rangle$ and $\langle Z_l^i\rangle$ are stored as forward layer l's results and used in the corresponding backward layer's computation.

The backward computation iterates from layer $K-1$ to layer 0. At the beginning of computing each backward layer, the data assigned to $\langle V_i\rangle$ depends on the layer index l. If it is layer $K-1$, i.e., the first backward layer, $\langle V_i \rangle$ gets $\langle H_K^i \rangle$, i.e., $\langle \hat{Y}_i \rangle - V_i$.y. Otherwise, $\langle V_i \rangle$ gets $\langle G_l^i \rangle$. Note that, when $l = K-1$, there is no need to execute Dispatch-1 tasks and the Collect-1 task on $\langle V_i \rangle$. We directly assign $\langle V_i \rangle$ to $\langle V_i \boxplus U_{*,i}^{\text{dst}} \rangle$, as line 21 of Protocol 2. For Dispatch-2 tasks in backward layers, if $l \neq 0$, each Dispatch-2 task produces $\langle V_i' \rangle$ and a gradient $\langle D_l^i \rangle$. If $l = 0$, each Dispatch-2 task only produces a gradient $\langle D_l^i \rangle$. Lines 24 to 26 in Protocol 2 show how the shares of all $\langle D_l^i \rangle$ are delegated to (P_0, P_1) for gradient aggregation and weight optimization in the Collect-2 task, and how the aggregated result is redistributed back to all $P_i, i \in [N] \backslash \{0, 1\}$. Line 27 of Protocol 2 assigns $\langle V_i' \rangle$ to $\langle G_{l-1}^i \rangle$ and assigns $\langle W_l' \rangle$ to $\langle W_l \rangle$. $\langle G_{l-1}^i \rangle$ is used for computing the next backward layer, while $\langle W_l \rangle$ is used in the next-epoch computation.

Inference. In CoGNN, performing a GCN inference process requires only a single run of the **Forward** part in Protocol 2. The weight matrices $\{W_l\}, l \in [K]$ can be either held by all parties in plaintext, or secretly shared as generated by the CoGNN training process, i.e., each pair of parties (P_i, P_{i+1}) hold a different two-party secret share of $\{W_l\}$.

Analysis. In total, running each forward layer requires N^2 Dispatch-1 tasks, N Collect-1 tasks and N Dispatch-2 tasks. Running the first backward layer ($l = K - 1$) requires N Dispatch-2 tasks and one Collect-2 task, while running each of the rest of the backward layers needs N^2 Dispatch-1 tasks, N Collect-1 tasks, N Dispatch-2 tasks and one Collect-2 task. In computing one forward or backward layer, tasks of the same type are executed distributedly (across all parties) and concurrently in CoGNN, making it scalable as more parties are involved. In § 8.2.2, we show that CoGNN is highly scalable: when the number of graph owners increases, the per-party running duration and communication grows modestly.

6.5 Result Reconstruction

Once the GCN training or inference process finishes, the parties jointly reconstruct the secret-shared computation outputs. For all forward layers, the final output is the updated vertex data $\langle V_i' \rangle$ storing the prediction values $\langle \hat{Y}_i \rangle$. For each backward layer l, the output is the updated model weight $\langle W_l' \rangle$. Both $\langle V_i' \rangle$ and $\langle W_l' \rangle$ are secret-shared between $(P_i, P_{i+1}), i \in [N]$. As a result, P_{i+1} sends $\langle V_i' \rangle_1$ and $\{\langle W_l' \rangle_1\}, l \in [K]$ to P_i for reconstructing V_i' and $\{W_l'\}, l \in [K]$ respectively. This part is omitted in Protocol 2.

6.6 Further Optimizations

Optimization w.r.t. the Input Dimension. The first optimization factor we consider here is the complexity of CoGNN w.r.t. the input dimension (or element dimension) of each layer. Our consideration is out of the fact that the input and output dimensions of different GCN layers are quite uneven. For example, for a typical two-layer GCN, the feature dimension could be as high as hundreds or even thousands, while the hidden representation dimensions are typically less than one hundred, and the number of classes is typically less than ten. Since the overall computation/communication overhead of a GCN layer is linear to its input dimension, the overhead of GCN layers with a high input dimension is also high.

However, we find that much overhead caused by the high input dimensions can be well circumvented by slightly changing Protocol 2. Concretely, in our current specification of CoGNN-GCN, the execution of each forward layer (as specified by Equation (1)) begins with performing message passing (i.e., $\hat{A}H_l$) using Scatter-Gather, and then handles NN-related computation, i.e., applying W_l and σ. The problem is that the dimensions of the elements participating in messaging passing are extremely high. For example, at layer 0, we perform message passing for each sample's high-dimension features. This problem can be alleviated by the following transformation to Equation (1):

$$H_{l+1} = \sigma(\hat{A}(H_l W_l)). \tag{6}$$

In other words, we can advance part of the computation conducted during the Apply phase, i.e., weight matrix multiplication, to the front of the Scatter phase (denoted as **PreScatter**), when the output dimension is lower than the input dimension. Such transformation largely slashes the overhead of Scatter-Gather for forward layers. We do not apply the transformation to backward layers since the output dimensions are typically higher than that of the input.

The problem here is that in backward layers, to derive the gradient w.r.t. the weight matrix (see Equation (2)), we need $\hat{A}H_{l-1}$, which is not computed due to our aforementioned transformation. Our workaround is the following transformation to Equation (2):

$$D_{l-1} = H_{l-1}^T(\hat{A}^T(\sigma'(Z_{l-1}) \odot \hat{A}^T G_l)). \tag{7}$$

In CoGNN, this transformation corresponds to two executions of Scatter-Gather (i.e., Dispatch-1 and Collect-1) for each backward layer. As long as the corresponding forward layer's input dimension is higher than two times its output dimension, the transformation still brings optimization.

Optimization w.r.t. $|E|$. In CoGNN, the overhead of the Scatter phase is linear to $|E|$. Though this complexity is inherent to GNN computation (since we have to do message passing on each edge), we can advance some costly computations to the front of Scatter or delay them to the back of Gather, in order to make their overhead linear to $|V|$. The exact benefit is that the concrete overhead of the overall message passing process is largely reduced. Specifically, in GCN, we make the following further transformation to Equation (6).

$$H_{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}}(\tilde{A}(\tilde{D}^{-\frac{1}{2}}(H_l W_l)))). \tag{8}$$

Previously, in Equation (6), $\hat{A}$ is a weighted matrix. These weights are represented as edge weights under the CoGNN computation model. As a result, the vectorized Scatter computation includes the multiplication with the edge weights at each vector slot, which is costly due to linear complexity w.r.t. $|E|$. After the above transformation, since $\tilde{D}^{-\frac{1}{2}}$ is a diagonal matrix, multiplying $\tilde{D}^{-\frac{1}{2}}$ to $H_l W_l$ means multiplying each row of $H_l W_l$ with a scaler, which only has a linear cost w.r.t. the length H_l, i.e., $|V|$. On the other hand, since $\tilde{A}$ is an unweighted adjacency matrix containing elements valued at $\{0, 1\}$, the whole Scatter-Gather phase now contains no multiplications. To achieve this tranformation in CoGNN, we advance the inner multiplication with $\tilde{D}^{-\frac{1}{2}}$ to the front of the Scatter phase (i.e., right after executing $H_l W_l$ in **PreScatter**), and delay the outer multiplication with $\tilde{D}^{-\frac{1}{2}}$ to the back of the Gather phase (denoted as **PostGather**).

Experimental results in § 8.2.2 show that these two optimizations bring $13 \sim 21$x improvement to the per-epoch duration of CoGNN, for various numbers of graph owners.

7 SECURITY THEOREM

Theorem 1. *Our CoGNN-GCN protocol (as specified in Protocol 2) securely realizes the functionality $\mathcal{F}_{CoGNN}$ in the ($\mathcal{F}_{Dsp\text{-}1}$, $\mathcal{F}_{Dsp\text{-}2}$, $\mathcal{F}_{Clc\text{-}1}$, $\mathcal{F}_{Clc\text{-}2}$)-hybrid model against a semi-honest, non-uniform adversary $\mathcal{A}$ corrupting one party at a time. Formally, for every PPT, semi-honest and non-uniform adversary $\mathcal{A}$ that corrupts one party P_i ($i \in [N]$), there exists a PPT, non-uniform simulator $\mathcal{S}$ corrupting the same party in the ideal world of $\mathcal{F}_{CoGNN}$, which satisfies:*

$$REAL^{\mathcal{F}_{Dsp\text{-}1},\mathcal{F}_{Dsp\text{-}2},\mathcal{F}_{Clc\text{-}1},\mathcal{F}_{Clc\text{-}2}}_{CoGNN\text{-}GCN,\mathcal{A}}(\kappa, \{E_i\}, \{E_{i,j}\}, \{V_i\}, \{W_l\}, alg)$$
$$\overset{c}{\equiv} IDEAL_{\mathcal{F}_{CoGNN},\mathcal{S}}(\kappa, \{E_i\}, \{E_{i,j}\}, \{V_i\}, \{W_l\}, alg),$$

where $\text{REAL}^{\mathcal{F}_{\text{Dsp-1}},\mathcal{F}_{\text{Dsp-2}},\mathcal{F}_{\text{Clc-1}},\mathcal{F}_{\text{Clc-2}}}_{\text{CoGNN-GCN},\mathcal{A}}$ represents a joint distribution over the view of the adversary (the corrupted party's input, randomness, protocol transcript) and the protocol output, when P_i and $P_j, \forall j \in [N]$ interact in the ($\mathcal{F}_{\text{Dsp-1}}, \mathcal{F}_{\text{Dsp-2}}, \mathcal{F}_{\text{Clc-1}}, \mathcal{F}_{\text{Clc-2}}$)-hybrid CoGNN on inputs ($\{E_i\}, \{E_{i,j}\}, \{V_i\}, \{W_l\}$, alg) and computational security parameter κ. ($\mathcal{F}_{\text{Dsp-1}}$, $\mathcal{F}_{\text{Dsp-2}}$, $\mathcal{F}_{\text{Clc-1}}$, $\mathcal{F}_{\text{Clc-2}}$) are the functionalties for (Dispatch-1, Dispatch-2, Collect-1, Collect-2) respectively. $\text{IDEAL}_{\mathcal{F}_{\text{FedGraph}},\mathcal{S}}(\kappa, \{E_i\}, \{E_{i,j}\}, \{V_i\}, \{W_l\}, \text{alg})$ represents a joint distribution over the simulated view of the corrupted party and the functionality output, when P_i and $P_j, \forall j \in [N]$ interact with $\mathcal{F}_{\text{CoGNN}}$ on inputs ($\{E_i\}, \{E_{i,j}\}, \{V_i\}, \{W_l\}$, alg) and computational security parameter κ; and $\overset{c}{\equiv}$ means the two distributions are computationally indistinguishable (in κ).

We provide the detailed proof of Theorem 1 in § A.7. Intuitively, Theorem 1 (i.e., the security of Protocol 2) holds in two intuitions: (i) the pairwise computations are all constructed using provably secure gadgets, including generic 2PC, OEP and OGA; (ii) the multi-party coordination is based on share delegation/redistribution, where all the messages received by each party are independent and uniformly random. More specifically, the security of CoGNN-GCN stems from the security of ($\mathcal{F}_{\text{Dsp-1}}$, $\mathcal{F}_{\text{Dsp-2}}$, $\mathcal{F}_{\text{Clc-1}}$, $\mathcal{F}_{\text{Clc-2}}$). Among them, the security of ($\mathcal{F}_{\text{Dsp-2}}$, $\mathcal{F}_{\text{Clc-1}}$, $\mathcal{F}_{\text{Clc-2}}$) is straightforward since their realizations include only simple 2PC operations, e.g., addition, multiplication, ReLU and Softmax, as shown in Protocol 1. On the other hand, the security of $\mathcal{F}_{\text{Dsp-1}}$ relies on the secure realizations of $\mathcal{F}_{\text{OEP}}$ and $\mathcal{F}_{\text{OGA}}$. Thus, in § A.7, we first prove two lemmas regarding the security of OEP and OGA, respectively. Afterwards, we construct a simulator $\mathcal{S}$ for $\mathcal{F}_{\text{CoGNN}}$. Finally, we prove Theorem 1 by showing that the real-world and ideal-world distributions are computationally indistinguishable via a hybrid distribution construction.

8 IMPLEMENTATION & EVALUATION

8.1 Implementation

The core component of our CoGNN prototype[1] is a GNN execution engine (4044 LoC C++) fulfilling the CoGNN framework. We further realize the OEP and OGA protocols (843 LoC C++) and integrate them into the engine for efficient Scatter-Gather computation. Based on that, we implement and optimize GCN (1922 LoC C++) to evaluate the concrete efficiency of CoGNN. Specifically, we use the RLWE-based HE and polynomial vector encoding provided by troy [4, 31] (A GPU-accelerated HE library) to support the batch HE operation of OEP[2] and the matrix multiplication between secret-shared matrices. The OT-related operations required by OEP and OGA are supported by the libOTe [2] library's implementation of Silent OT [10]. The rest of the computations on secret-shared data, e.g., vector-scaler multiplication, $Softmax$ and $ReLU$, are implemented based on the SCI library [3]. We replace SCI's OT implementation with libOTe's for better efficiency.

[1]https://github.com/InspiringGroup-Lab/CoGNN

[2]We use HE for the offline phase of OEP. See § A.1 for the details.

8.2 Evaluation

Our evaluations center around the following:

- **Efficiency and Scalability.** In § 8.2.2, we conduct an extensive comparison between CoGNN and the SOTA SML-based approach [26] to demonstrate CoGNN's efficiency improvement, as well as its scalability for an increasing number of graph owners. The comparison includes end-to-end duration/communication measurements on various datasets;
- **Accuracy.** In § 8.2.3, we show that CoGNN yields model accuracy comparable to plaintext GNN, while significantly surpassing prior FL-based GNN. This part of the experimental results reveals the importance of considering inter-edges during GNN collaboration.

Additionally, in § 8.2.4, we present the breakdowns of the optimized CoGNN's running durations to quantify the effectiveness of our message passing mechanism. Due to page limit, we defer several supporting evaluation results to appendix, including the comparison between our message passing approach and prior art (§ A.4), and CoGNN's duration breakdowns before optimizations and communication statistics for training/inference (§ A.5).

8.2.1 Setup. **Testbed.** Our testbed is a Linux server with two Intel (R) Xeon (R) Gold 6348 CPUs at 2.60GHz. The network environment is simulated using Linux network namespace and the tc command. We place different parties in different network namespaces connected via a virtual bridge, and use the tc command to set up specific bandwidth and latency. In particular, all our efficiency tests run in a simulated LAN environment with $4Gbps$ bandwidth and $1ms$ latency. A computation task assigned to each pair of parties (i.e., P_i and P_j, $i \neq j$) runs in a single thread.

Dataset. We evaluate CoGNN on three graph datasets commonly used for vertex-level prediction tasks, i.e., Cora [35], CiteSeer [16], PubMed [46]. The detailed information on the three datasets is provided in Table 3. During GNN training, we consider the semi-supervised setting, where only a relatively small proportion of vertices have labels but all vertices have features and participate in the training process. We perform batch gradient descent using the full graph dataset for every training iteration (epoch). The vertices with labels are treated as the training set, while the rest of the vertices are split into validation and test sets. Considering the number of vertices varies a lot for the three datasets, we use different train/valid/test splits for them, as shown in the rightmost column of Table 3. To distribute the whole graph dataset evenly to all the graph owners, we use random graph partition, i.e., uniform-randomly assigning each vertex to one of all the graph owners.

To evaluate the *efficiency and scalability* of CoGNN, we split each dataset into 5 partitions, each partition corresponding to a graph owner. As the number of graph owners increases, more partitions are involved in collaborative training, and the scale of the global graph also increases. To evaluate the *accuracy* of CoGNN, we split all three datasets into different numbers of partitions, and involve all partitions in training. In other words, the scale of the global graph is fixed, while the number of graph owners changes. In this setting, as the number of graph owners increases, the number of inter-edges also increases. We record the detailed numbers of inter-edges for different numbers of graph owners in Table 5.

Parameter. For the cryptographic part, we set the ring length L of fixed-point encoding to 2^{64} and the scaler for encoding a floating point number to 2^{13}. The parameters set for the RLWE-based HE scheme are $\{N = 8192, p = 2^{44}, q \approx 2^{180}\}$. It means each encrypted number is encoded over the ring of $\mathbb{Z}_{2^{44}}$ (i.e., the plaintext space is $\mathbb{Z}_{2^{44}}$). We use the extend and truncate operations provided by the SCI library to switch between the rings of $\mathbb{Z}_{2^{64}}$ and $\mathbb{Z}_{2^{44}}$. For the model structure, we use a 2-layer GCN and set the hidden layer dimension to 16. The first layer is activated by $ReLU$, while the second layer is normalized by $Softmax$. We use cross-entropy loss and full-batch gradient descent. For the model training part, we set the number of epochs to 90. The learning rates for Cora/CiteSeer/PubMed are set to 0.5/0.4/8.0 to guarantee that CoGNN and all the baseline methods have converged (but not overfitted) after the 90 epochs. The model weights are randomly initialized using the Glorot's method [17]. For FedGNN and CoGNN, we take the average of their accuracy over the test set of each party as their final test accuracy.

8.2.2 *Scalability & Efficiency.*

Since the source code and evaluation details of the SOTA SML-based approach [26] (denoted as GraphSC here since it is based on the GraphSC paradigm) are unavailable, for fair comparison, we re-produce it using the same MPC, OT and HE libraries as used in our CoGNN prototype. We use our OEP protocol for the shuffle operations in GraphSC, and use prefix network construction to realize the *propagate* operation for Scatter and *aggregate* operation for Gather. All these realization details try to achieve the best efficiency for GraphSC.

End-to-end Efficiency and Scalability. We compare CoGNN and GraphSC under different numbers of graph owners. According to our evaluation setup, as the number of graph owners increases, the scale (i.e., numbers of edges and vertices) of the global graph increases linearly. The number of computing parties in CoGNN is the same as the number of graph owners, while the number of computing parties in GraphSC remains unchanged and is always 2. We set the same number of threads between each pair of parties for CoGNN and GraphSC. Figure 6 shows the running duration and communication (send + receive) of each computing party in one epoch of full-batch training. The three lines in each subfigure represent GraphSC, CoGNN, and CoGNN with optimizations (i.e., CoGNN-Opt) respectively. CoGNN shows significant elevation over GraphSC in both efficiency and scalability. Specifically, compared to GraphSC when there are 2 graph owners, CoGNN (CoGNN-Opt) reduces the running duration by 1.9 ~ 2.0x (29 ~ 39x), and reduces the per-party communication of by 2.2 ~ 2.4x (88 ~ 164x). The duration/communication improvements are enlarged as the number of graph owners increases. In particular, when the number of graph owners increases from 2 to 5, the per-party duration (communication) of GraphSC is 4.3 ~ 4.9 (4.5 ~ 5.0) times higher. In contrast, the per-party duration (communication) grows to 1.1 ~ 1.2x (1.8 ~ 1.9x) in CoGNN, and grows to 1.4 ~ 1.5x (1.2 ~ 1.5x) in CoGNN-Opt. The per-party duration/communication of CoGNN grows much slower than GraphSC, indicating CoGNN's better scalability as more graph owners are involved. Compared to GraphSC when there are 5 graph owners, CoGNN (CoGNN-Opt) reduces the running duration by 6.7 ~ 8.5x (97 ~ 123x), and reduces the per-party communication by 5.8 ~ 5.9x (323 ~ 522x).

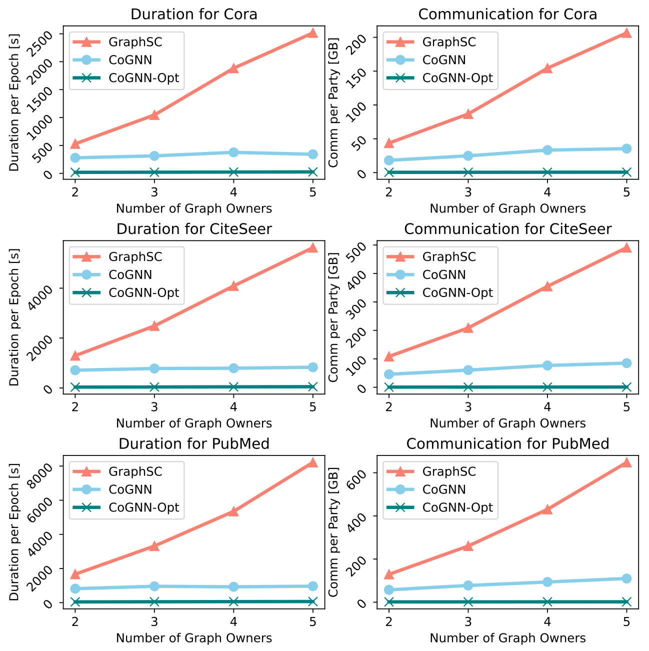

Figure 6: Duration and communication per epoch of each computing party for different numbers of graph owners.

8.2.3 *Model Accuracy.*

We compare the accuracy of CoGNN and the FL-based approach (abbreviated as FedGNN) to concretely show the benefits of global GCN training *with* inter-edge message passing. The accuracy baseline for us is plaintext GCN training on the global graph, denoted as PlainGNN. For FedGNN, we use the common scheme named *FedAvg* [52], which is equivalent to the gradient aggregation in CoGNN. Both the PlainGNN and FedGNN schemes we compare use the same training hyperparameters as CoGNN. FedGNN uses the same graph partitions as CoGNN.

We compare the accuracy of PlainGNN, CoGNN and FedGNN for different numbers of parties. According to our dataset setup, the number of inter-edges increases as the number of parties increases (detailed in Table 5), meaning that more edges are ignored by FedGNN. We are also interested in vertices connected by inter-edges. We call them *border vertices* and call their accuracy as *border accuracy*. Figure 7 shows that, (i) the accuracy and border accuracy of CoGNN remain stable and close to the PlainGNN accuracy as the number of parties/inter-edges increases for all datasets. In contrast, the overall accuracy of FedGNN fluctuates and is consistently

Dataset	OEP Pre	PreScatter	Scatter OEP (1)	Scatter Comp	Scatter OGA	Scatter OEP (2)	Gather	PostGather	Apply	Total
Cora	0.33	9.82	0.13	-	1.10	0.01	0.01	4.20	17.29	32.88
CiteSeer	0.35	25.64	0.13	-	1.06	0.01	0.01	4.52	34.18	65.89
PubMed	2.69	29.11	0.26	-	2.61	0.05	0.01	14.51	44.84	94.08

Table 1: Execution duration $[s]$ breakdown for each full-batch training epoch of CoGNN with optimization.

Dataset	OEP Pre	PreScatter	Scatter OEP (1)	Scatter Comp	Scatter OGA	Scatter OEP (2)	Gather	PostGather	Apply	Total
Cora	0.09	6.97	0.08	-	0.61	0.01	0.01	2.80	7.12	17.68
CiteSeer	0.08	22.62	0.08	-	0.61	0.01	0.01	3.37	7.50	34.27
PubMed	0.55	17.04	0.17	-	1.37	0.03	0.01	12.71	20.93	52.79

Table 2: Execution duration $[s]$ breakdown for each full-batch inference of CoGNN with optimization.

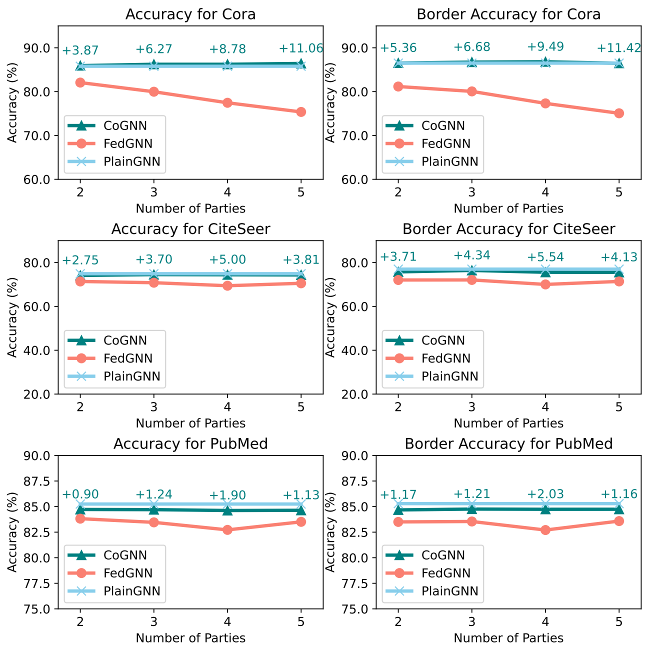

Figure 7: Accuracy for different numbers of parties. (+) represents the accuracy difference between CoGNN and FedGNN.

lower than CoGNN and PlainGNN. Specifically, from 2-party to 5-party, the accuracy (border accuracy) gap between CoGNN and FedGNN on Cora increases from 3.87% to 11.06% (5.36% to 11.52%); (ii) the accuracy gap between CoGNN and FedGNN is more significant in border vertices for all three datasets. The results reveal that in a GNN collaborative setting where the inter-edges are more intense, training on the global graph using CoGNN brings more accuracy elevation over FedGNN. Overall, collaboratively training GNN using CoGNN achieves better model performance. The computing/communication overhead in FedGNN is certainly small since it trains models in plaintext. Yet it lacks a provable privacy guarantee and promising model performance. In § A.6, we provide the detailed accuracy statistics under the two-party setting and five-party setting in Table 11 and 12, respectively.

8.2.4 Training/Inference Duration Breakdown. Besides accuracy, we provide the duration breakdowns of CoGNN-Opt in the two-party setting in Table 1 and 2 to show the exact overhead in GCN training/inference. The **OEP Pre** column is the offline OEP duration amortized for each training epoch/inference. The **PreScatter** and **PostGather** columns correspond to the optimizations of CoGNN in § 6.6. The **Scatter Comp** is not applicable here since CoGNN-Opt distributes the vector-scaler multiplication operations to **PreScatter** and **PostGather**. The **Total** column sums up all offline and online durations. The message passing (summing up **OEP Pre**, **Scatter OEP**, **Scatter OGA**, **Gather**) consumes only 2 ∼ 6% (2 ∼ 5%) of the total training (inference) duration, indicating that our message passing paradigm and OEP/OGA protocols are highly efficient. The heavyweight parts of CoGNN are in **PreScatter**, **PostGather** and **Apply**, because they perform expensive vector-scaler multiplication and NN computations (linear and non-linear). See § A.5 for the detailed communication overhead breakdown.

9 FUTURE WORK

Other Types of GNN Models. As a generic framework for collaborative GNN learning, CoGNN provides a high-level abstraction for secure, efficient and scalable GNN training/inference. Nevertheless, instantiating specific GNN models under CoGNN requires dedicated construction of all Dispatch tasks and Collect tasks to express specialized forward/backward computation. Further optimizations like our optimizations for GCN might also be needed to bring out concrete efficiency for computing specific GNN models. We leave the support of other types of GNN models, such as GraphSAGE [19] and graph attention network (GAT) [48], to future work.

Mini-Batch Training. CoGNN is currently designed for full-batch training across the whole global graph. In the real world, there is a need for mini-batch training to support GNN learning on a large graph dataset. Supporting mini-batch training in CoGNN requires an additional design of collaborative mini-batch sampling without violating the privacy of each graph data owner. The sampling process selects out the mini-batch-related graph data, which can be then fed to the CoGNN training/inference process. This is also a promising future direction for extending CoGNN.

Other Scenarios of Graph Collaboration. The graph collaboration setting that CoGNN deals with, i.e., exclusively different vertex sets with inter-edges, is one of the most popular settings of graph data collaboration. However, there are still other scenarios. For example, one party holds vertex features and labels, while another party has the graph topology [20, 55]. They want to put the features/labels and graph topology together to jointly train a GNN model. Such GNN collaboration can be handled by CoGNN by having the two parties secretly share the features and perform GNN training thereafter. A more complex scenario might be that multiple graph owners have overlapped vertex sets and different graph topologies. The vertex features that they hold are also different from each other. Adapting CoGNN to these scenarios to support efficient GNN collaboration is also included in our future work.

Other Types of Gradient Aggregation Schemes. The way how CoGNN-GCN aggregates the gradients of different subgraphs (in Collect-2) is equivalent to the FedAvg [9] scheme. Thus, in accuracy evaluation, we compare CoGNN with the FedAvg-based FL approach. Supporting other aggregation schemes, like FedOpt [6] and FedProx [30], is part of the future extension of CoGNN.

Other Security Models. CoGNN currently deals with semi-honest and non-colluding adversaries. We have introduced the rationale for this security model in § 4. We further discuss the possibility of extending CoGNN to other stronger security models (e.g., malicious or colluding) and the extra overhead that might be introduced thereafter. In the n-party setting (i.e., an arbitrary number of parties), to extend CoGNN to malicious security, the main difficulty is in protecting the delegated Collect phase. This goal is similar to verifiable secure aggregation in FL, and can borrow techniques from recent works (like [18, 45]). This increases computation/communication costs but does not necessarily sacrifice scalability. For colluding security, we need to combine CoGNN with an MPC scheme over t-out-of-n secret sharing. This would sacrifice scalability since the same task has to be processed by more parties.

In the setting of a fixed number of parties, like prior SML-based approaches, extending CoGNN to a stronger security model is relatively lightweight. For example, for the 2-party setting (like in [50]), CoGNN requires no share delegation/redistribution, and realizing malicious security mainly needs to protect OEP and OGA. For the 3-party or 4-party setting, we can incorporate the CoGNN message-passing mechanism into a maliciously secure 3PC or 4PC scheme (similar to [5, 26]), but the scalability is degraded.

10 CONCLUSION

This paper presents CoGNN, a secure, efficient and scalable framework for collaborative GNN learning among multiple graph owners. CoGNN introduces a new oblivious message passing paradigm that halves the communication required by the SOTA approach. Based on it, we build a two-stage Dispatch-Collect execution scheme to securely and efficiently express GNN training and inference as distributed vertex-centric computation. By instantiating GCN under the CoGNN framework and performing comprehensive comparisons with prior approaches, we show that CoGNN reduces up to 123x running time and up to 522x communication cost per party when compared to the SML-based SOTA approach, while achieving good scalability as the number of graph owners increases. Meanwhile, the GCN models trained under CoGNN have performance comparable with plaintext global-graph training, and yield up to 11.06% accuracy improvement over GCN models trained via federated learning.

REFERENCES

[1] 2020. Complete guide to GDPR compliance. https://gdpr.eu/ Accessed: 2022-12-08.

[2] 2023. libOTe. A fast, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe

[3] 2023. Secure and Correct Inference (SCI) Library. https://github.com/mpc-msri/EzPC/tree/master/SCI

[4] 2023. troy := seal-cuda. GPU implementation of BFV, CKKS and BGV homomorphic encryption schemes. https://github.com/lightbulb128/troy

[5] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 610–629. https://doi.org/10.1145/3460120.3484560

[6] Muhammad Asad, Ahmed Moustafa, and Takayuki Ito. 2020. FedOpt: Towards communication efficiency and privacy preservation in federated learning. *Applied Sciences* 10, 8 (2020), 2864.

[7] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2022. Secure Parallel Computation on Privately Partitioned Data and Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 151–164. https://doi.org/10.1145/3548606.3560695

[8] Marina Blanton, Ahreum Kang, and Chen Yuan. 2020. Improved Building Blocks for Secure Multi-party Computation Based on Secret Sharing with Honest Majority. In *Applied Cryptography and Network Security*, Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi (Eds.). Springer International Publishing, Cham, 377–397.

[9] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1175–1191.

[10] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 291–308. https://doi.org/10.1145/3319535.3354255

[11] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 496–511. https://doi.org/10.1109/EuroSP.2019.00043

[12] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 342–372.

[13] Fahao Chen, Peng Li, Toshiaki Miyazaki, and Celimuge Wu. 2021. Fedgraph: Federated graph learning with intelligent sampling. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1775–1786.

[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.

[15] Jiarui Feng, Yixin Chen, Fuhai Li, Anindya Sarkar, and Muhan Zhang. 2022. How powerful are k-hop message passing graph neural networks. *Advances in Neural Information Processing Systems* 35 (2022), 4776–4790.

[16] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. 1998. CiteSeer: An Automatic Citation Indexing System. In *Proceedings of the Third ACM Conference on Digital Libraries* (Pittsburgh, Pennsylvania, USA) *(DL '98)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/276675.276685

[17] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 9)*, Yee Whye Teh and Mike Titterington (Eds.). PMLR, Chia Laguna Resort, Sardinia, Italy, 249–256. https://proceedings.mlr.press/v9/glorot10a.html

[18] Changhee Hahn, Hodong Kim, Minjae Kim, and Junbeom Hur. 2023. VerSA: Verifiable Secure Aggregation for Cross-Device Federated Learning. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2023), 36–52. https://doi.org/10.1109/TDSC.2021.3126323

[19] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[20] Xinlei He, Jinyuan Jia, Michael Backes, Neil Zhenqiang Gong, and Yang Zhang. 2021. Stealing Links from Graph Neural Networks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2669–2686. https://www.usenix.org/conference/usenixsecurity21/presentation/he-xinlei

[21] Jinzhang Hu, Ruimin Hu, Zheng Wang, Dengshi Li, Junhang Wu, Lingfei Ren, Yilong Zang, Zijun Huang, and Mei Wang. 2023. Collaborative Fraud Detection: How Collaboration Impacts Fraud Detection. In *Proceedings of the 31st ACM International Conference on Multimedia (MM '23)*. Association for Computing Machinery, New York, NY, USA, 8891–8899. https://doi.org/10.1145/3581783.3613780

[22] Dejun Jiang, Zhenxing Wu, Chang-Yu Hsieh, Guangyong Chen, Ben Liao, Zhe Wang, Chao Shen, Dongsheng Cao, Jian Wu, and Tingjun Hou. 2021. Could graph neural networks learn better molecular representation for drug discovery? A comparison study of descriptor-based and graph-based models. *Journal of cheminformatics* 13, 1 (2021), 1–23.

[23] Lindell Katz. 2007. *11.3 The Paillier Encryption Scheme," in Introduction to modern cryptography.* CRC Press. 385–394 pages.

[24] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 971–985. https://doi.org/10.1145/3373376.3378486

[25] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *The International Conference on Learning Representations (ICLR Poster)*.

[26] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2023. Entrada to Secure Graph Convolutional Networks. Cryptology ePrint Archive, Paper 2023/1807. https://eprint.iacr.org/2023/1807 https://eprint.iacr.org/2023/1807.

[27] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively secure 4PC for secure training and inference. In *NDSS*.

[28] Richard E Ladner and Michael J Fischer. 1980. Parallel prefix computation. *Journal of the ACM (JACM)* 27, 4 (1980), 831–838.

[29] Qi Li, Zhuotao Liu, Qi Li, and Ke Xu. 2023. martFL: Enabling Utility-Driven Data Marketplace with a Robust and Verifiable Federated Learning Architecture *(CCS '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3576915.3623134

[30] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2020. Federated optimization in heterogeneous networks. *Proceedings of Machine learning and systems* 2 (2020), 429–450.

[31] Xuanqi Liu, Zhuotao Liu, Qi Li, Ke Xu, and Mingwei Xu. 2024. Pencil: Private and Extensible Collaborative Learning without the Non-Colluding Assumption. In *NDSS*.

[32] Yanjun Liu, Hongwei Li, Xinyuan Qian, and Meng Hao. 2023. ESA-FedGNN: Efficient secure aggregation for federated graph neural networks. *Peer-to-peer Networking and Applications* 16, 2 (2023), 1257–1269.

[33] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.* 8, 3 (nov 2014), 281–292. https://doi.org/10.14778/2735508.2735517

[34] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[35] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. 2000. Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval* 3, 2 (2000), 127–163. http://dblp.uni-trier.de/db/journals/ir/ir3.html#McCallumNRS00

[36] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. 2019. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 691–706.

[37] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35–52. https://doi.org/10.1145/3243734.3243760

[38] Payman Mohassel and Saeed Sadeghian. 2013. How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 557–574.

[39] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 19–38. https://doi.ieeecomputersociety.org/10.1109/SP.2017.12

[40] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1466–1482. https://doi.org/10.1109/SP40000.2020.00057

[41] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 739–753.

[42] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. 377–394. https://doi.org/10.1109/SP.2015.30

[43] Yang Pei, Renxin Mao, Yang Liu, Chaoran Chen, Shifeng Xu, Feng Qiang, and Blue Elephant Tech. 2021. Decentralized federated graph neural networks. In *International Workshop on Federated and Transfer Learning for Data Sparsity and Confidentiality in Conjunction with IJCAI*.

[44] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 325–342. https://doi.org/10.1145/3372297.3417274

[45] M. Rathee, C. Shen, S. Wagh, and R. Popa. 2023. ELSA: Secure Aggregation for Federated Learning with Malicious Actors. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1961–1979. https://doi.org/10.1109/SP46215.2023.10179468

[46] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (Sep. 2008), 93. https://doi.org/10.1609/aimag.v29i3.2157

[47] Connie Diaz De Teran. 2023. Collaboration Is Key in the Fight Against Anti-Money Laundering. https://www.paymentsjournal.com/collaboration-is-key-in-the-fight-against-anti-money-laundering/ Accessed: 2023-09-12.

[48] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *The International Conference on Learning Representations (ICLR)*.

[49] Clement Vignac, Andreas Loukas, and Pascal Frossard. 2020. Building powerful and equivariant graph neural networks with structural message-passing. *Advances in neural information processing systems* 33 (2020), 14143–14155.

[50] S. Wang, Y. Zheng, and X. Jia. 2023. SecGNN: Privacy-Preserving Graph Neural Network Training and Inference as a Cloud Service. *IEEE Transactions on Services Computing* 16, 04 (jul 2023), 2923–2938.

[51] Yanling Wang, Jing Zhang, Shasha Guo, Hongzhi Yin, Cuiping Li, and Hong Chen. 2021. Decoupling Representation Learning and Classification for GNN-Based Anomaly Detection. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 1239–1248. https://doi.org/10.1145/3404835.3462944

[52] Zhen Wang, Weirui Kuang, Yuexiang Xie, Liuyi Yao, Yaliang Li, Bolin Ding, and Jingren Zhou. 2022. FederatedScope-GNN: Towards a Unified, Comprehensive and Efficient Package for Federated Graph Learning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 4110–4120. https://doi.org/10.1145/3534678.3539112

[53] Bin Wu, Xinyu Yao, Boyan Zhang, Kuo-Ming Chao, and Yinsheng Li. 2023. Split-GNN: Spectral Graph Neural Network for Fraud Detection against Heterophily. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23)*. Association for Computing Machinery, New York, NY, USA, 2737–2746. https://doi.org/10.1145/3583780.3615067

[54] Chuhan Wu, Fangzhao Wu, Yang Cao, Yongfeng Huang, and Xing Xie. 2021. FedGNN: Federated graph neural network for privacy-preserving recommendation. *arXiv preprint arXiv:2102.04925* (2021).

[55] Fan Wu, Yunhui Long, Ce Zhang, and Bo Li. 2022. LINKTELLER: Recovering private edges from graph neural networks via influence analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2005–2024.

[56] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing graph neural networks with message passing data flow graph. *Proceedings of Machine Learning and Systems* 4 (2022), 515–528.

[57] Ke Zhang, Carl Yang, Xiaoxiao Li, Lichao Sun, and Siu Ming Yiu. 2021. Subgraph federated learning with missing neighbor generation. *Advances in Neural Information Processing Systems* 34 (2021), 6671–6682.

[58] Zehong Zhang, Lifan Chen, Feisheng Zhong, Dingyan Wang, Jiaxin Jiang, Sulin Zhang, Hualiang Jiang, Mingyue Zheng, and Xutong Li. 2022. Graph neural network approaches for drug-target interactions. *Current Opinion in Structural Biology* 73 (2022), 102327.

A APPENDIX

A.1 $\mathcal{F}_{\mathsf{OEP}}$ Instantiation

When the input size is smaller than the output size, [38] discusses a construction of OEP by combining two oblivious permutation networks and one vector scan. In total, the OEP protocol of [38] requires $O((|T_{\text{in}}|+|T_{\text{out}}|)\log(|T_{\text{in}}|+|T_{\text{out}}|))$ invocations of 1-out-of-2 OT, resulting in high communication overhead. As we will describe below, our OEP protocol achieves $O(|T_{\text{in}}|)$ online complexity and $O(|T_{\text{in}}|+|T_{\text{out}}|)$ total complexity.

By carefully looking into the scenario of GNN computation, we find two important patterns of invoking $\mathcal{F}_{\mathsf{OEP}}$: (i) the dimension of vector elements to be permuted is typically high, especially when the GAS iteration corresponds to the first forward layer and the element dimension equals the number of features; (ii) since the graph is undirected, all of the forward layers and backward layers share exactly the same graph topology, resulting in the same set of permutations. In other words, we need a dedicated realization of $\mathcal{F}_{\mathsf{OEP}}$ in GNN computation, to efficiently deal with large-size elements and repeated execution of the same permutations on different inputs.

```
Protocol Two-party OEP
Input. $P_i$ provides $\pi \leftarrow (Id_{\text{in}}, Id_{\text{out}})$, $\{\langle T^k_{\text{in}}\rangle_0\}$ and $P_j$ provides $\{\langle T^k_{\text{in}}\rangle_1\}$, where $|\langle T^k_{\text{in}}\rangle| = a$, $|\{\langle T^k_{\text{in}}\rangle\}| = m$, and $|\langle T^k_{\text{in}}\rangle[x]| = l_k$.
Output. $P_i$ gets $\{\langle T^k_{\text{out}}\rangle_0\}$ and $P_j$ gets $\{\langle T^k_{\text{out}}\rangle_1\}$, such that $T^k_{\text{out}} = \pi(T^k_{\text{in}})$, $|\langle T^k_{\text{out}}\rangle| = b$, and $|\langle T^k_{\text{out}}\rangle[x]| = l_k$.
1  Offline.
2      for $k \in [m]$: $P_j$ ($P_i$) randomly samples $R_k \leftarrow\$ \mathbb{Z}^a_{2^{l_k}}$ ($S_k \leftarrow\$ \mathbb{Z}^b_{2^{l_k}}$) to obtain a batch of random vectors $\{R_k\}$ ($\{S_k\}$), $k \in [m]$;
3      $P_j$ ($P_i$) concatenates $\{R_k\}$ ($\{S_k\}$) as $\widehat{R}$ ($\widehat{S}$), where $\widehat{R}[x] = ||R_k[x], \forall k \in [m], \forall x \in \mathbb{Z}_{|R_k|}$ ($\widehat{S}[x] = ||S_k[x], \forall k \in [m], \forall x \in \mathbb{Z}_{|S_k|}$);
4      $P_j$ performs per-element HE encryption to $\widehat{R}$ to obtain $[\widehat{R}]_j$ and sends it to $P_i$;
5      $P_i$ performs the extended permutation $\pi$ on $[\widehat{R}]_j$, and gets $\pi([\widehat{R}]_j) = [\pi(\widehat{R})]_j$;
6      $P_i$ performs HE cipher-plaintext subtraction to obtain $[\pi(\widehat{R}) - \widehat{S}]_j$. Then, $P_i$ sends $[\pi(\widehat{R}) - \widehat{S}]_j$ to $P_j$;
7      $P_j$ decrypts $[\pi(\widehat{R}) - \widehat{S}]_j$ and breaks $\pi(\widehat{R}) - \widehat{S}$ as $\{\pi(R_k) - S_k\}$. Then, $P_i$ stores $\{S_k\}$ while $P_j$ stores $\{R_k\}$ and $\{\pi(R_k) - S_k\}$.
8  Online.
9      for $k \in [m]$:
10         $P_j$ performs $\langle T^k_{\text{in}}\rangle_1 - R_k$ and sends the result to $P_i$;
11         $P_i$ performs $\langle T^k_{\text{in}}\rangle_0 + \langle T^k_{\text{in}}\rangle_1 - R_k$ to obtain $T^k_{\text{in}} - R_k$. Then, $P_i$ performs $\pi$ on $T^k_{\text{in}} - R_k$ to get $\pi(T^k_{\text{in}} - R_k)$;
12         $P_i$ performs $\pi(T^k_{\text{in}} - R_k) + S_k$ and sets $\langle T^k_{\text{out}}\rangle_0$ to the result. $P_j$ sets $\langle T^k_{\text{out}}\rangle_1$ to $\pi(R_k) - S_k$.
```

Protocol 3: Two-party OEP

Our OEP protocol works in an offline-online fashion. In the offline phase, we generate permutation correlations for multiple online OEP invocations in a batch, aided by homomorphic encryption (HE) techniques supporting batch operations. In the online phase, we consume the permutation correlations to efficiently permute different inputs. The core idea behind this design is to pack elements and fully utilize the rich batch slots provided by HE techniques to amortize the communication/computation cost of permuting each element in the batch.

The specification of OEP is provided in Protocol 3. Its inputs include the extended permutation π defined by a pair of identifier vectors $(Id_{\text{in}}, Id_{\text{out}})$ and a batch of secret shared vectors $\{\langle T^k_{in}\rangle\}$ to permuted. The batch size is m. Id_{in} and each vector in $\{\langle T^k_{in}\rangle\}$ are of length a. $|Id_{\text{out}}|$ defines the output length b. Different input vectors could have different element sizes, denoted as l_k. Note that $C_{\text{zero}} \in \{0,1\}^{|Id_{\text{out}}|}$ is an indicator vector indicating whether each element in Id_{out} (T_{out}) has an equal element in Id_{in} (T_{in}).

In the offline phase, P_j samples a batch of random vectors, denoted as $\{R_k\}$, of the same size as the input vectors. At the same time, P_i samples a batch of random vectors, $\{S_k\}$, of the same size as the output vectors. They then concatenate/pack $\{R_k\}$ and $\{S_k\}$ to obtain $\hat{R}$ and $\hat{S}$ respectively. See lines 2, 3 in Protocol 3. After that, P_j encrypts each element of $\hat{R}$ separately to obtain $[\hat{R}]_j$ and sends it to P_i. Upon receiving $[\hat{R}]_j$, P_i performs the extended permutation π to it and gets $[\pi(\hat{R})]_j$, which is of the same length as the output vectors. Further, P_i performs cipher-plaintext subtraction to $[\pi(\hat{R})]_j$ and $\hat{S}$ to derive $[\pi(\hat{R}) - \hat{S}]_j$, which is then sent back to P_j. Finally, P_j decrypts $[\pi(\hat{R}) - \hat{S}]_j$ and unpack it into $\{\pi(R_k) - S_k\}$. P_i stores $\{\hat{S}_k\}$ as a batch of permutation correlations, while P_j stores $\{\hat{R}_k\}$ and $\{\pi(R_k) - S_k\}$. See lines 4-7 of Protocol 3.

In the online phase, for each secret-shared input vector $\langle T^k_{in}\rangle$ in the batch, P_j calculates $\langle T^k_{in}\rangle_1 - R_k$ and sends it to P_i. P_i adds $\langle T^k_{in}\rangle_1 - R_k$ to $\langle T^k_{in}\rangle_0$ to obtain $T^k_{in} - R_k$. Permuting $T^k_{in} - R_k$ results in $\pi(T^k_{in} - R_k)$, which is of the same length as the output vector.

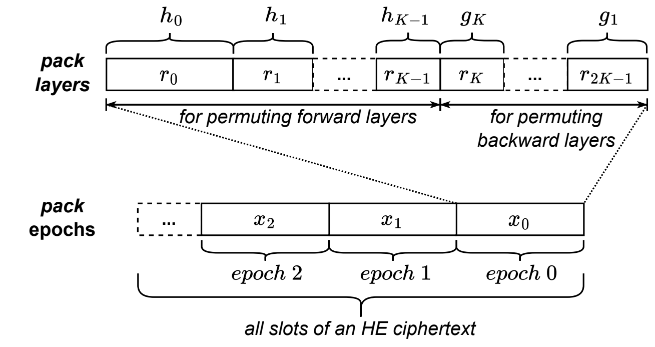

Figure 8: Hierarchical packing in batch OEP preprocessing.

Then, P_i sets $\langle T^k_{out}\rangle_0$ to $\pi(T^k_{in} - R_k) + S_k$, while P_j sets $\langle T^k_{out}\rangle_1$ to $\pi(R_k) - S_k$. See lines 10-12 of Protocol 3.

Zero Out Unmatched Elements. We have mentioned in § 3.2 that the definition of $\mathcal{F}_{\text{OEP}}$ extends the definition of OEP in [38] by allowing the elements in Id_{out} to be not in Id_{in}. We call these elements in Id_{out} (T_{out}) unmatched elements. To handle them, we add a dummy element valued 0 to T_{in} and have P_i adjust π to permute this dummy element to all the unmatched elements in T_{out}. The additional overhead introduced to our OEP protocol is negligible since the input size turns to $|T_{\text{in}}| + 1$, while the output size stays unchanged.

Hierarchical Packing. The reason why we design Protocol 3 to support packing vectors of different element sizes in a batch is that the input dimensions to different forward/backward layers of GNN are different. Figure 8 shows our design of hierarchical packing for preprocessing the permutations in GNN computation. Specifically, we first pack the forward and backward layers of each epoch. The size of each packed element equals the input dimension of the corresponding GNN layer. For instance, in Figure 8, the sizes of $r_0, ..., r_{K-1}$ equal the sizes of $h_0, ..., h_{K-1}$ respectively, which represent the hidden representations fed to layer $0, ..., K-1$. The packing of backward layers is symmetric to the forward layers. As all the layers of each epoch is packed, we pack consecutive epochs

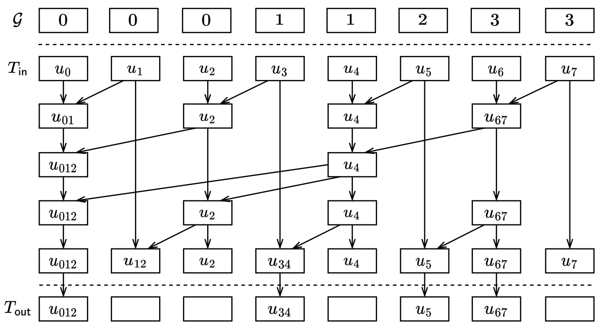

Figure 9: An example of OGA based on prefix adder network.

```
Protocol Two-party OGA
----------------------------------------------------------------
Input. $P_i$ provides $(\mathcal{G}, \langle T_{\text{in}}\rangle_0)$ and $P_j$ provides $\langle T_U\rangle_1$. $|\mathcal{G}| = |\langle T_{\text{in}}\rangle| = n$. $\mathcal{G}$ can be divided into segments, each of which contains elements of the same value. $P_i$ and $P_j$ agree on the merge operation $\boxplus$.
Output. $P_i$ gets $\langle T_{\text{out}}\rangle_0$ and $P_j$ gets $\langle T_{\text{out}}\rangle_1$. The first element of each segment of $T_{\text{out}}$ stores the aggregation result of the corresponding segment in $T_{\text{in}}$. The other elements of each segment of $T_{\text{out}}$ store 0.
----------------------------------------------------------------
Function Merge$(c, \langle x_1\rangle, \langle x_2\rangle)$
return $\mathsf{MUX}_2(c, \langle x_1\rangle, \langle x_1 \boxplus x_2\rangle)$
----------------------------------------------------------------
Function Aggregate$(\boldsymbol{b}, \langle \boldsymbol{v}\rangle)$
$n \leftarrow |\langle \boldsymbol{v}\rangle|$;
if $n = 1$ then return $\langle \boldsymbol{v}\rangle$;
for $k \in \{0, ..., \lfloor n/2\rfloor - 1\}$ :
    $\boldsymbol{b}'[k] \leftarrow \boldsymbol{b}[2k] \wedge \boldsymbol{b}[2k+1]$;
    $\langle \boldsymbol{v}'[k]\rangle \leftarrow$ Merge$(\boldsymbol{b}[2k], \langle \boldsymbol{v}[2k]\rangle, \langle \boldsymbol{v}[2k+1]\rangle)$;
$\langle \boldsymbol{w}\rangle \leftarrow$ Aggregate$(\boldsymbol{b}', \langle \boldsymbol{v}'\rangle)$;
for $k \in \{0, ..., \lfloor n/2\rfloor - 1\}$ :
    $\langle \boldsymbol{v}[2k]\rangle \leftarrow \langle \boldsymbol{w}[k]\rangle$;
for $k \in \{0, ..., \lceil n/2\rceil - 2\}$ :
    $\langle \boldsymbol{v}[2k+1]\rangle \leftarrow$ Merge$(\boldsymbol{b}[2k+1], \langle \boldsymbol{v}[2k+1], \langle \boldsymbol{v}[2k+2]\rangle)$;
return $\langle \boldsymbol{v}\rangle$.
----------------------------------------------------------------
1 for $k \in \{0, ..., n-2\}$:
      $P_i$ calculates $\boldsymbol{b}[k] \leftarrow \begin{cases} 0, & \mathcal{G}[k] \neq \mathcal{G}[k+1], \\ 1, & \mathcal{G}[k] = \mathcal{G}[k+1]; \end{cases}$
2 $P_i$ sets $\boldsymbol{b}[n-1]$ to 0;
3 $\langle T_{\text{out}}\rangle \leftarrow$ Aggregate$(\boldsymbol{b}, \langle T_{\text{in}}\rangle)$.
```

Protocol 4: Two-party OGA

until all the slots of an HE ciphertext are consumed. Supposing the number of packed epochs in a ciphertext is C, we have the batch size $m = C \cdot 2K$.

Cost Analysis. The main computation cost of the offline phase of OEP is from HE-related operations, i.e., a encryptions by P_j, b cipher-plaintext subtractions by P_i, and b decryptions by P_j. Correspondingly, the communication cost of the offline phase of OEP is from transferring $a + b$ HE ciphertexts. These offline computation/communication costs are amortized for each online OEP invocation in the batch. Each online invocation of OEP contains plaintext computation only and the communication is of the same size as $\langle T^k_{\text{in}}\rangle$, i.e., $a \cdot l_k$.

A.2 $\mathcal{F}_{\text{OGA}}$ Instantiation

Our construction of the OGA protocol follows the prefix adder network paradigm [28], which is a commonly used scheme for parallel oblivious computation. We illustrate the idea of OGA using the example in Figure 9, in which we oblivious merge an 8-element T_{in}. In Figure 9, when two arrows direct to the same element, it means that the element is the result of the conditional merging of the two source elements. If the groups of the two source elements are the same, we merge. Otherwise, we copy the first one of the two source elements to the destination element. When only one arrow directs to an element, it means a copy operation. The overall computation is oblivious since the arrow topology (i.e., how the arrows are positioned) depends purely on the length of T_{in} and is irrelevant to the content in $\mathcal{G}$. From Figure 9 we can see that the prefix adder network has a recursive structure, where the forward layers fold like a tree to the root, and the backward layers unfold back to the leaves. In the following, we explain it in detail, with the help of our detailed OGA protocol specification in Protocol 4.

As shown in Protocol 4, the OGA protocol is based on two core functions, i.e., Merge and Aggregate. The Merge function conditionally aggregates two secret-shared elements based on the boolean flag c provided by P_i. If $c = 1$, the merge result is returned. If $c = 0$, a random reshare of the first element is returned. Ignoring the merge operation $\boxplus$, Merge requires one invocation of OT.

The Aggregate function is a recursive function that expresses the prefix adder network. It aims to obliviously aggregate the elements in $\langle \boldsymbol{v}\rangle$ according to the boolean indicator vector $\boldsymbol{b}$, which is provided by P_i and is of the same length as $\langle \boldsymbol{v}\rangle$. If $\boldsymbol{b}[i]$ is 1, it indicates that $\boldsymbol{v}[i]$ and $\boldsymbol{v}[i+1]$ are in the same group. If $\boldsymbol{b}[i]$ is 0, they are not in the same group. In detail, the Aggregate function is composed of three consecutive parts: (1) conditionally merge the even-indexed elements with the subsequent odd-indexed elements that are directly following them, and output $\langle \boldsymbol{v}'\rangle$; (2) invoke Aggregate recursively on $\langle \boldsymbol{v}'\rangle$ and write the results back to the even-indexed elements of $\langle \boldsymbol{v}\rangle$; (3) conditionally merge the odd-indexed elements of $\langle \boldsymbol{v}\rangle$ with the subsequent even-indexed elements that are directly following them. The correctness of the function Aggregate can be derived inductively. In particular, supposing that the (2) part correctly merges $\langle \boldsymbol{v}'\rangle$, it is not difficult to infer that after executing parts (1) and (2), all the even-indexed elements in $\langle \boldsymbol{v}\rangle$ have reached their merge goal, i.e., merging all the subsequent elements that are of the same group as them. Therefore, after executing part (3), all the odd-indexed elements in $\langle \boldsymbol{v}\rangle$ would also reach their merge goal.

At the entrance of the OGA protocol, P_i locally calculates the boolean indicator vector $\boldsymbol{b}$ from $\mathcal{G}$. See lines 1-2 of Protocol 4. Then, P_i and P_j jointly execute the function Aggregate on $\boldsymbol{b}$ and $\langle T_{\text{in}}\rangle$. The output of Aggregate is assigned to $\langle T_{\text{out}}\rangle$. See line 3 of Protocol 4.

Cost Analysis. From the property of prefix network, our OGA protocol requires $2n$ ($n = \langle T_{\text{in}}\rangle$) executions of Merge, which corresponds to $2n$ invocations of MUX_2 and $\boxplus$. Each invocation of MUX_2 requires one OT, where P_i is the receiver and P_j is the sender.

A.3 Information about Evaluation Datasets

Table 3, 4, and 5 present some statistics about the datasets used for our evaluations. $|E|$ is the number of directed edges. The rightmost column of Table 3 is the train/valid/test split for the three datasets.

Dataset	$\|V\|$	$\|E\|$	Feature	Class	Split
Cora	2,708	10,556	1,433	7	0.2,0.2,0.6
CiteSeer	3,327	9,104	3,703	6	0.2,0.2,0.6
PubMed	19,717	88,648	500	3	0.05,0.15,0.8

Table 3: Dataset information.

	N=2	N=3	N=4	N=5
$\|V\|$ of Cora	1,084	1,626	2,167	2,708
$\|E\|$ of Cora	1,728	3,956	7,752	10,556
$\|V\|$ of CiteSeer	1,332	1,997	2,662	3,327
$\|E\|$ of CiteSeer	1,422	3,298	6,334	9,104
$\|V\|$ of PubMed	7,888	11,831	15,774	19,717
$\|E\|$ of PubMed	14,580	32,988	57,318	88,648

Table 4: Total number of edges (vertices) for different numbers of graph owners (in efficiency and scalability evaluation).

Dataset	N=2	N=3	N=4	N=5
Cora	5,404 (51%)	7,184 (68%)	8,028 (76%)	8,552 (81%)
CiteSeer	4,668 (51%)	6,040 (66%)	6,996 (77%)	7,354 (81%)
PubMed	44,278 (50%)	59,248 (67%)	66,424 (75%)	71,202 (80%)

Table 5: Total number of inter-edges for different numbers of graph owners (in accuracy evaluation). (%) represents the percentage of inter-edges in all edges.

Dataset	Adj Matrix	Adj List	Graph-SC	CoGNN	CoGNN -Opt
Cora	160.15	312.02	1.45	0.61	0.03
CiteSeer	610.69	923.14	3.71	1.49	0.02
PubMed	3082.02	10,014.92	5.86	2.62	0.27

Table 6: Online Communication [GB] of oblivious message passing for each training epoch.

A.4 Message Passing Efficiency

We compare the message passing cost of CoGNN with prior methods (mentioned in § 2.4 and § 5.1) by measuring the communication of message passing for the three graph datasets respectively. For the fairness of comparison, we use the same comparison setting as in § 2.4, i.e., computing AH, where A is the $\{0, 1\}$ adjacency matrix of the whole dataset and is held by P_0, and H is the secret-shared data vector. We compare online communication only and ignore offline preprocessing costs. Considering the diversity of implementation schemes, we provide estimation results here, based on the communication analysis in § 2.4 and § 5.1. In particular, we estimate the communication cost of full-batch training one epoch by summing up the costs of all invocations of message passing in this epoch. The estimation results are provided in Table 6. Straightforward oblivious message passing solutions based on adjacency matrix and adjacency list all have quite high communication overhead, making them impractical for the graph scale of the tested datasets. GraphSC provides state-of-the-art performance among existing methods. CoGNN without further optimizations yields 2.2 $\sim$ 2.5x improvement over GraphSC, while CoGNN with further optimizations (i.e., CoGNN-Opt) achieves 22 $\sim$ 155x improvement over GraphSC. It is worth noting that, since the hidden layer dimension of the GCN model is fixed and CoGNN-Opt's message passing communication is independent of the feature dimension, a higher feature dimension results in a higher improvement ratio of CoGNN-Opt w.r.t. GraphSC. Consequently, among the three datasets, CiteSeer has the highest feature dimension and also the highest improvement ratio, while PubMed with the lowest feature dimension has the lowest improvement ratio.

A.5 Efficiency & Communication Breakdown

In this section, we present the duration/communication breakdown for one training epoch and one full-graph inference of CoGNN in the two-party setting to show the efficiency brought by our message passing paradigm and the OEP/OGA protocols design. Additionally, we compare the durations of CoGNN with and without further optimizations to show how vital these optimizations are in pushing CoGNN into practical performance. *Note that, these experiments run on the same dataset setup as used in our accuracy evaluation (in § 8.2.3), i.e., evenly partitioning each of the three datasets and involving the whole dataset in training, which is different from our dataset setup for our efficiency and scalability evaluation (in § 8.2.2). See § 8.2.1 for the details.*

The duration breakdowns for the original CoGNN design and the optimized CoGNN design are shown in Table 7 and Table 1 respectively. In both tables, the **OEP Pre** column represents the offline OEP preprocessing duration amortized for each training epoch, and the **Total** column sums up all offline and online durations. Benefiting from our extremely efficient message passing design, for both CoGNN and CoGNN-Opt, the message passing effort, i.e., sum of OEP and OGA, takes up only a small proportion of the overall running duration, i.e., $\sim$ 1% in CoGNN and 2 $\sim$ 6% in CoGNN-Opt. The overhead of CoGNN without further optimizations is dominated by the phase of vectorized Scatter computation, where we perform vector-scaler multiplication for each secret-shared vertex hidden representation. This part of overhead for CoGNN is linear to the number of edges and the feature dimension, resulting in high computation complexity. In CoGNN-Opt, our further optimizations (i) perform weight matrix multiplication before the Scatter phase (i.e., **PreScatter**) to make the dimension of vertex data involved in message passing independent of the feature dimension; (ii) decompose the vector-scaler multiplication operations in Scatter computation and distribute them to the **PreScatter** phase and the **PostGather** phase, making the number of vector-scaler multiplications independent of the number of edges. Finally, the total number of vector-scaler multiplications is linear to the relatively low number of vertices and the hidden layer dimension. As shown in Table 7 and Table 1, the total duration of a training epoch of CoGNN-Opt yields 35 $\sim$ 41x[3] improvement over the original CoGNN for the three datasets.

Table 9 shows the offline/online per-epoch communication of CoGNN and CoGNN-Opt for various datasets, when there are two parties. For both CoGNN and CoGNN-Opt, the offline communication is relatively small compared to the online communication, indicating the low cost of CoGNN's OEP protocol design. Summing up the offline and online phases, the per-epoch communication of CoGNN ranges from 87 to 273 GB among the three datasets, while the per-epoch communication of CoGNN-Opt ranges from 0.82 to

[3]The improvement ratio is different from the results in § 8.2.2 since their dataset setups are different, which leads to different numbers of edges and vertices of the global graph.

Dataset	OEP Pre	PreScatter	Scatter OEP (1)	Scatter Comp	Scatter OGA	Scatter OEP (2)	Gather	PostGather	Apply	Total
Cora	5.81	-	0.33	1077.20	5.78	0.14	50.45	-	21.71	1161.42
CiteSeer	14.62	-	0.68	2529.73	13.52	0.34	145.32	-	48.88	2753.10
PubMed	15.58	-	0.81	3499.99	18.15	0.51	158.14	-	57.60	3750.77

Table 7: Execution duration $[s]$ breakdown for each full-batch training epoch of CoGNN without further optimization.

Dataset	OEP Pre	PreScatter	Scatter OEP (1)	Scatter Comp	Scatter OGA	Scatter OEP (2)	Gather	PostGather	Apply	Total
Cora	5.85	-	0.29	1079.70	5.61	0.15	51.16	-	11.17	1153.93
CiteSeer	14.46	-	0.64	2492.89	13.41	0.33	148.24	-	27.17	2697.13
PubMed	15.33	-	0.74	3387.87	17.79	0.51	155.51	-	32.01	3609.76

Table 8: Execution duration $[s]$ breakdown for each full-batch inference of CoGNN without further optimization.

Dataset	CoGNN Offline	CoGNN Online	CoGNN-Opt Offline	CoGNN-Opt Online
Cora	1.18	85.81	0.07	0.75
CiteSeer	2.96	199.85	0.07	1.33
PubMed	3.27	269.98	0.55	3.78

Table 9: Offline and online communication [GB] of each full-batch training epoch.

Dataset	CoGNN Offline	CoGNN Online	CoGNN-Opt Offline	CoGNN-Opt Online
Cora	1.18	84.45	0.02	0.48
CiteSeer	2.96	198.33	0.02	0.89
PubMed	3.27	260.32	0.11	1.91

Table 10: Offline and online communication [GB] of each full-batch inference.

4.32 GB. Thus, the further optimizations we propose for our GCN implementations yield 63 ~ 144x improvement in communication cost in the two-party setting.

Table 2, 8 and 10 are the corresponding duration/communication breakdowns for one full-graph (batch) inference of CoGNN. We run the evaluation by supposing that all graph data (including the model weights) is secret-shared.

Data-set	Plain GNN	Plain GNN Border	Co-GNN	Co-GNN Border	Fed-GNN	Fed-GNN Border
Cora	85.79	86.5	85.93 **(+3.87)**	86.52 **(+5.36)**	82.06	81.17
Cite-Seer	74.91	77.01	74.14 **(+2.75)**	75.76 **(+3.71)**	71.39	72.05
Pub-Med	85.25	85.29	84.72 **(+0.90)**	84.67 **(+1.17)**	83.82	83.5

Table 11: Accuracy and border-accuracy in the two-party setting. (+) represents the accuracy difference between CoGNN and FedGNN.

A.6 Accuracy Comparison

Table 11 and 12 present the detailed accuracies of PlainGNN, CoGNN and FedGNN under the 2-party setting and 5-party setting, respectively.

A.7 Security Proof

In this section, we present a detailed proof for Theorem 1. We first prove two lemmas on the security of OEP and OGA respectively.

Data-set	Plain GNN	Plain GNN Border	Co-GNN	Co-GNN Border	Fed-GNN	Fed-GNN Border
Cora	85.79	86.5	86.42 **(+11.06)**	86.48 **(+11.42)**	75.37	75.06
Cite-Seer	74.91	77.01	74.36 **(+3.81)**	75.53 **(+4.13)**	70.56	71.40
Pub-Med	85.25	85.29	84.64 **(+1.13)**	84.74 **(+1.16)**	83.51	83.58

Table 12: Accuracy and border-accuracy in the five-party setting. (+) represents the accuracy difference between CoGNN and FedGNN.

After that, we construct a simulator $\mathcal{S}$ for $\mathcal{F}_{\text{CoGNN}}$. Finally, we carry out the proof of Theorem 1 using hybrid distribution construction.

A.7.1 Security Lemma and Proof Sketch for $\mathcal{F}_{OEP}$ and $\mathcal{F}_{OGA}$.

We have the following security lemmas for $\mathcal{F}_{\text{OEP}}$ and $\mathcal{F}_{\text{OGA}}$.

LEMMA 2. *Denote the κ-security homomorphic encryption functionality as $\mathcal{F}_{HE}$. OEP securely realizes the functionality $\mathcal{F}_{OEP}$ in the $\mathcal{F}_{HE}$-hybrid model against a semi-honest, non-uniform adversary $\mathcal{A}$ corrupting either P_i or P_j. Formally, given any PPT, semi-honest and non-uniform adversary $\mathcal{A}$ that corrupts either P_i or P_j, there exists a PPT, non-uniform simulator $\mathcal{S}$ corrupting the same party in the ideal world of $\mathcal{F}_{OEP}$, which satisfies:*

$$REAL^{\mathcal{F}_{HE}}_{OEP,\mathcal{A}}(\kappa, \langle T_{in} \rangle, \pi) \overset{c}{\equiv} IDEAL_{\mathcal{F}_{OEP},\mathcal{S}}(\kappa, \langle T_{in} \rangle, \pi).$$

LEMMA 3. *Denote the functionality performing the MUX_2 and merge computation $\boxplus$ as $\mathcal{F}_{MUX_2}$ and $\mathcal{F}_{\boxplus}$. OGA securely realizes the functionality $\mathcal{F}_{OGA}$ in the $(\mathcal{F}_{MUX_2}, \mathcal{F}_{\boxplus})$-hybrid model against a semi-honest, non-uniform adversary $\mathcal{A}$ corrupting either P_i or P_j. Formally, for every PPT, semi-honest and non-uniform adversary $\mathcal{A}$ that corrupts either P_i or P_j, there exists a PPT, non-uniform simulator $\mathcal{S}$ corrupting the same party in the ideal world of $\mathcal{F}_{OGA}$, which satisfies:*

$$REAL^{\mathcal{F}_{MUX_2},\mathcal{F}_{\boxplus}}_{OGA,\mathcal{A}}(\kappa, \langle T_{in} \rangle, \mathcal{G}) \overset{c}{\equiv} IDEAL_{\mathcal{F}_{OGA},\mathcal{S}}(\kappa, \langle T_{in} \rangle, \mathcal{G})$$

.

The notations used are similar to Theorem 1.

Proof Sketch for OEP. The overall proof idea is to construct two different simulators, $\mathcal{S}_i$ and $\mathcal{S}_j$, when P_i or P_j is corrupted respectively. After that, we prove that the real-world distribution and the simulated distribution are computationally indistinguishable in the $\mathcal{F}_{\text{HE}}$-hybrid model.

During the preprocessing phase, $\mathcal{S}_i$ accesses P_i's random tape to generate a set of random vectors $\{S_k\}$ and batch them into $\widehat{S}$. After that, $\mathcal{S}_i$ samples a vector of a random numbers, each of which is of the same length as the ciphertext of $\mathcal{F}_{\mathsf{HE}}$. The random vector is indistinguishable from the $[\widehat{R}]_j$ received in the real world from the property of $\mathcal{F}_{\mathsf{HE}}$. Thereafter, $\mathcal{S}_i$ applies the extended permutation π held by P_i to the random vector and subtracts it with $\widehat{S}$. The result is indistinguishable from the $[\pi(\widehat{R}) - \widehat{S}]_j$ sent to P_j in the real world. Thus, the view of $\mathcal{S}_i$ in the offline phase of OEP is indistinguishable from the real-world view. During the online phase, $\mathcal{S}_i$ simulates the received $\langle T_{in} \rangle_1 - R_k$ by randomly sampling a vector of equal length. The indistinguishability preserves since each element in $\langle T_{in} \rangle_1 - R_k$ is uniformly distributed in the secret share ring.

The construction of $\mathcal{S}_j$ is easier. The locally generated plaintext data of P_j can be simulated directly by accessing the random tape of P_j, while the $[\widehat{R}]_j$ in the preprocessing phase can be simulated by invoking the simulator of $\mathcal{F}_{\mathsf{HE}}$. Additionally, the received $[\pi(\widehat{R}) - \widehat{S}]_j$ can be simulated by uniformly sampling a vector of b slots and of the same slot (ciphertext) size as $[\pi(\widehat{R}) - \widehat{S}]_j$. The indistinguishability is preserved from the security property of $\mathcal{F}_{\mathsf{HE}}$. Thus, the real-world distribution is also indistinguishable from the ideal world.

Proof Sketch for OGA. The security of the OGA protocol stems from the obliviousness of the prefix adder network-based execution. In other words, the order in which the Merge function applies to the vector elements in $\langle T_{\mathsf{in}} \rangle$ is determined solely by the length of $\langle T_{\mathsf{in}} \rangle$, independent of the element values or $\mathcal{G}$. This order leaks nothing about $\mathcal{G}$ to P_j and nothing about the element values of $\langle T_{\mathsf{in}} \rangle$ to both P_i and P_j. Thus, the order of Merge function execution can be easily simulated based on the length of $\langle T_{\mathsf{in}} \rangle$. On the other hand, the Merge function can be simulated by invoking the simulators of $\mathcal{F}_{\mathsf{MUX}_2}$ and $\mathcal{F}_{\boxplus}$. As a result, the simulated views of both $\mathcal{S}_i$ and $\mathcal{S}_j$ are indistinguishable from the real world.

A.7.2 Simulator Construction. Since each party in $\mathcal{F}_{\mathsf{CoGNN}}$ is at an equal place, they share the same simulator, $\mathcal{S}$. We construct $\mathcal{S}$ from the perspective of P_i.

Graph Data Secret Share. During setup, $\mathcal{S}$ accesses V_i, E_i and $E_{i,j}$ to simulate the secret-sharing of graph data. In particular, the secret shares received from $P_j, j \in [N] \backslash \{i\}$ are simulated based on the vector length information, i.e., $\mathcal{L} := \{|E_j|, |V_j|\}, \forall j \in [N] \backslash \{i\}$, received from $\mathcal{F}_{\mathsf{CoGNN}}$, and the model details specified in alg.

Iteration. As shown in Protocol 2, the iteration of CoGNN is composed of calls of Dispatch-1, Dispatch-2, Collect-1, Collect-2 on various inputs. The output of one task directly serves as the input of another task. So the inputs and outputs of calls of the four tasks finally constitute the view of the adversary, which can be simulated based on the iteration input (shares of V_i, E_i, $E_{i,j}$, $E_{j,i}$, W_l) by invoking the simulators of these tasks' corresponding functionalities, according to the computation procedure in Protocol 2. As for the share delegation and redistribution phase, since the secret shares sent and received by party P_i are uniformly randomized, the corresponding simulation is also straightforward.

Result Reconstruction. The secret shares received from $P_j, j \in [N] \backslash \{i\}$ during the reconstruction phase can be simulated by subtracting the locally held shares from the updated graph data (i.e., V_i') and weights (i.e., $\{W_l'\}$) received from $\mathcal{F}_{\mathsf{CoGNN}}$.

A.7.3 Security Proof. Based on the construction of $\mathcal{S}$, we prove that the real-world distribution and the simulated distribution are computationally indistinguishable using a hybrid distribution construction.

Hyb0. This is the real-world distribution based on calls of $\mathcal{F}_{\mathsf{Dsp\text{-}1}}$, $\mathcal{F}_{\mathsf{Dsp\text{-}2}}$, $\mathcal{F}_{\mathsf{Clc\text{-}1}}$, $\mathcal{F}_{\mathsf{Clc\text{-}2}}$.

Hyb1. For graph data secret sharing, $\mathcal{S}$ simulates the shares generated by P_i by accessing the random tape of P_i and simulates the received shares by uniformly sampling share vectors based on the vector length information $\mathcal{L}$ and model dimensions specified in alg. The indistinguishability from Hyb0 is trivial.

Hyb2. In this hybrid, we iteratively simulate the view of P_i received during each execution of Dispatch-1, Collect-1, Dispatch-2, and Collect-2 tasks. The simulation process calls the simulators of $\mathcal{F}_{\mathsf{Dsp\text{-}1}}$, $\mathcal{F}_{\mathsf{Clc\text{-}1}}$, $\mathcal{F}_{\mathsf{Dsp\text{-}2}}$, and $\mathcal{F}_{\mathsf{Clc\text{-}2}}$ respectively, to simulate the intermediate computation data and output data generated/received during each of Dispatch-1, Collect-1, Dispatch-2, and Collect-2 tasks. Because of the secure realizations of ($\mathcal{F}_{\mathsf{Dsp\text{-}1}}$, $\mathcal{F}_{\mathsf{Clc\text{-}1}}$, $\mathcal{F}_{\mathsf{Dsp\text{-}2}}$, $\mathcal{F}_{\mathsf{Clc\text{-}2}}$), Hyb2 is indistinguishable from Hyb1. Note that according to Lemma 2 and Lemma 3, we know that Dispatch-1 securely realizes $\mathcal{F}_{\mathsf{Dsp\text{-}1}}$.

Hyb3. This hybrid deals with the simulation of the share delegation process before Collect-1 and Collect-2. Before each execution of Collect-1, P_i receives $\langle U^{\mathsf{dst}}_{j,i-1} \rangle_1$ from all $P_j, j \in [N] \backslash \{i, i-1\}$. Since these shares are uniformly random and independent of all the data held by P_i, they can be simulated by uniformly random samples. Similarly, when $i = 1$ or $i = 0$, the $\langle D_l^j \rangle_{1-i}, \forall j \in [N] \backslash \{0, 1\}$ received by P_i before each execution of Collect-2 can be simulated by uniformly random samples. The indistinguishability between Hyb2 and Hyb3 is guaranteed by the randomness of the shares received by P_i and irrelevance between the shares and P_i's data.

Hyb4. We handle the simulation of the share redistribution process after Dispatch-2 and Collect-2 in this hybrid. After each execution of Dispatch-2 in a forward layer, P_i receives $\langle V_j' \rangle_1 + R_i$ from $\forall P_{j+1}, j \in [N] \backslash \{i, i-1\}$. Since these shares are uniformly random and independent of all the data held by P_i, they can be simulated by uniformly random samples. On the other hand, when $i \neq 0, 1$, the $\langle W_l' \rangle_1 - R_{i-1}$ received from P_0 and the $\langle W_l' \rangle_0 + R_i$ received from P_1 after each execution of Collect-2 can be also simulated by uniformly random samples. The indistinguishability between Hyb3 and Hyb4 is guaranteed by the randomness of the shares received by P_i and irrelevance between the shares and P_i's data.

Hyb5. In this hybrid, we replace the result reconstruction phase with the simulated view. The vertex shares $\langle V_i' \rangle_1$ and the model weight shares $\{\langle W_l' \rangle_1\}$ received from P_{i+1} are replaced with the corresponding shares constructed by $\mathcal{S}$ during the result reconstruction phase. The received shares in the real world and the simulated shares are in the same distribution. Thus, Hyb4 and Hyb5 are computationally indistinguishable.

Hyb6. This is the simulated view. Since all the three phases (i.e., graph data secret share, iteration and result reconstruction) of CoGNN are simulated in Hyb5, Hyb5 is identically distributed as Hyb6. The proof completes.