OblivDGL: Oblivious Deep Graph Learning Made Practical

Yiming Qin*, Xiaoning Liu†, Shangqi Lai‡, Joseph K. Liu* and Xingliang Yuan‡
*Monash University †RMIT University ‡The University of Melbourne

***Abstract*—The rapid deployment of Graph Neural Networks (GNNs) within Machine Learning as a Service (MLaaS) has opened up new privacy concerns. Both sensitive graph data and proprietary GNN models are submitted to and processed in third-party MLaaS platform for training and inference. To alleviate those concerns, recent efforts have been made to design tailored secure computation protocols for GNNs. Unfortunately, existing designs cannot accomplish security, functionality, and scalability simultaneously. They leak a certain degree of graph information, offer limited support on secure training of GNNs, and can hardly scale to large graphs.**

In this paper, we design and implement OblivDGL, an oblivious deep graph learning framework for secure GNN inference and training. Our design is inspired by Deep Graph Library (DGL), an industrial practice for graph learning. At its core, we distill the secure computational patterns of GNNs into a generalized form, and empower both secure GNN training and inference through a novel *oblivious message passing paradigm*. To realize such a paradigm, we design a new oblivious graph representation, and realize it by function secret sharing (FSS), a lightweight secure computation technique with low latency. Compared to prior secure GNN systems, OblivDGL achieves the most stringent privacy guarantees, i.e., revealing no private information except for the total number of edges and nodes. One standout feature of OblivDGL is the scalability of processing encrypted graphs by carefully tailored parallelization designs. Extensive evaluations on four benchmark graph datasets demonstrate that, compared to state-of-the-art secure GNN frameworks, OblivDGL reduces communication cost by up to 1024x and achieves up to 14.3x speedup in runtime during secure training. For secure inference, OblivDGL achieves up to 754x lower communication and a 4.19x speedup.

I. Introduction

Machine learning as a service (MLaaS) has streamlined the AI development and deployment lifecycle [2], [3]. It allows a company (i.e., model developer) to upload customer data for AI model training, where the end-users can access such prebuilt prediction services by supplying their data through the MLaaS platform. With the rapid growth of deep graph learning, existing MLaaS platforms have embraced this advancement by integrating Graph Neural Networks (GNNs) into their services. For example, AWS SageMaker [4] has integrated the Deep Graph Library (DGL). Such an integration is advantageous for a wide range of graph-based applications, from drug discovery [5] to financial fraud detection [6]. However, current MLaaS platforms operate on cleartext graph data and GNN models, raising critical privacy concerns. The graph data are often collected from individuals, such as customer data used for training and user inputs provided during inference. These data usually contains private individual information, such as bank accounts and transaction details in fraud detection scenarios. Meanwhile, the proprietary GNN model is deemed the intellectual property of the company. A particular concern with GNNs is the privacy of graph structure data. This data encapsulates the connectivity of nodes and are particularly information-rich, making it highly valuable in sensitive domain applications. There is thus a pressing need to ensure the privacy of both graph data and model during GNN training and inference.

To alleviate these concerns, privacy-preserving machine learning (PPML) for GNNs has become a trending field. A number of works [7]–[11] have proposed PPML techniques to enable secure GNN training and/or inference. Those works proceed by representing a graph as two matrices: an *adjacency matrix* that captures the graph structure and a *feature matrix* that indicates the embeddings of each node. A secure training or inference protocol is then executed over the encrypted adjacency matrix, feature matrix, and trainable model weights to develop an encrypted GNN model or generate predictions securely. While PPML techniques seem promising, they all suffer from security and performance limitations that hinder their practical deployment. These limitations stem from the inherent constraints of the aforementioned matrix-based representation of graph data.

As illustrated in Table I, prior works cannot fully protect sensitive graph structure information revealed from the adjacency matrix. This information includes the number of edges connected to each node [7], [11], and the total number of nodes in the graph [10]. Even when encrypted, the dimensions of the adjacency matrix can still leak such information. Recent works such as GraphSC [12] primarily focus on secure graph processing rather than secure deep graph learning. However, the leakage of node degree hinders their direct applicability to GNNs. CoGNN [13] leverages collaborative learning to reduce overhead of GraphSC. However, this design choice inevitably reveals the size of the subgraph. RingSG [14] and GraphOS [15] similarly focus on secure graph processing: RingSG reveals the processed subgraph sizes, whereas GraphOS reveals the numbers of nodes and the number of edges. Their leakages can be exploited by model inversion attacks to steal the graph property [16] or even reconstruct the entire graph without authorization [17], [18]. Moreover, secure GNN training and inference require several building blocks beyond the scope of prior secure graph processing systems,

The conference version of this paper is published in NDSS 2027 [1].

	SecGNN	OblivGNN	GraphSC	GraphOS	CoGNN	RingSG	OblivDGL
Secure Training	✓	✗	✗	✗	✓	✗	✓
Secure Inference	✓	✓	✗	✗	✓	✗	✓
Message Passing Paradigm	✗	✗	✓	✗	✓	✓	✓
Unified Operation	✗	✗	✗	✗	✗	✗	✓
Parallelization	✗	✗	✓	✗	✗	✗	✓
Graph Structure Privacy	✗	✗	◐	◐	◐	◐	✓

TABLE I: Qualitative Comparison. Unified operation refers to whether a secure GNN system supports both training and inference using a consistent computational framework. For graph structure, SecGNN and OblivGNN protect graph structure by directly operating on an encrypted sparse adjacency matrix, yet reveal $|V|$. GraphOS reveals $|V|$ and $|E|$. CoGNN and RingSG employ oblivious message passing to protect the graph structure, yet still reveal $|V|$ of the subgraph.

such as oblivious message passing and aggregation and secure evaluation of non-linear activation functions.

Second, existing PPML designs for GNNs cannot unify secure computations over encrypted matrix-based graph representations across both forward and backward propagation. Therefore, most existing work can only support secure inference, but not training [7]–[10]. Some recent works support secure training but either require a different protocol dedicated to backward propagation [11], or resort to an entirely different training paradigm [13]. Third, existing works do not support parallel computation. Thus, they cannot scale to GNNs with large graph data. The communication and computation costs of the secure computation protocols will scale with graph size, naturally increasing the overall latency for executing the PPML protocol.

A. Summary of Techniques

In this paper, we design, implement, and evaluate OblivDGL, an oblivious deep graph learning framework for secure GNN training and inference (Table I). To overcome the above bottlenecks from matrix-based graph representations, our starting point is to leverage the Deep Graph Library (DGL) [19], an industrial practice for deep learning on graph-structured data. At its core, we distill the secure computational patterns of GNNs into a generalized form *oblivious message passing paradigm*. To realize such a paradigm, we propose an oblivious graph representation, and realize it by function secret sharing (FSS), a lightweight secure computation technique with low latency. One standout feature of OblivDGL is the scalability of GNNs with large graphs via extensive parallelization. Together, our framework makes oblivious deep graph learning practical, while achieving the most stringent security guarantees, i.e., leaking no private information except for the total number of edges and nodes ($|V| + |E|$) in the graph. We summarize our contributions as follows.

C1: Oblivious data structure for the most stringent security. To surpass the security leakage caused by matrix-based graph representations, we resort to the message passing paradigm in DGL to replace traditional matrix multiplication. However, trivially realizing message passing paradigm in an encrypted domain would inherently leak node degrees, i.e., the number of edges connected to a node. To ensure no sensitive graph information is revealed, we design a specialized data structure, i.e., *linear tuple for oblivious graph representation*. Such a data structure enables oblivious processing in a single linear scan. We thus can perform identical operations on both graph nodes and edges. Recent work such as GraphSC [12] adopts a tuple-based representation to secure graph computation. However, its use of if-else flow leaks node degree, making it unsuitable for direct application to GNNs. CoGNN [13] follows the same tuple design within GNNs and employs collaborative learning to mitigate the overhead. However, this design choice results in the leakage of the subgraph size. For the secure realization of OblivDGL's oblivious message passing paradigm, we resort to arithmetic FSS [20] for addition and multiplication, and distributed point function (DPF) [21], [22] and distributed comparison function (DCF) [20] for non-linear operations. As a result, our oblivious message passing paradigm guarantees that (1) no sensitive graph information, including the number of nodes or node degrees, is leaked during computation and (2) the message passing process is fully oblivious, ensuring that intermediate computations do not reveal any private information (See Section IV-C for details).

C2: Unifying secure GNN training and inference. OblivDGL's oblivious message passing paradigm unifies the computation of secure GNN training and inference. Unlike existing secure GNN training method [11], which requires heavy matrix-based gradient computation, we design two unified oblivious functions for both backward and forward propagation: an oblivious reduce function, and an oblivious message function. We observe that the oblivious message passing paradigm on the original graph data during forward propagation can be transformed to its reverse version - the same computation on the reverse graph. We can thus foster an oblivious message passing paradigm on reverse graph for backward propagation. As a result, our oblivious message passing paradigm in Section IV can serve for both training and inference, whereas the only difference is the input graphs for the function calls.

C3: Scaling for large graphs with parallelization. As mentioned above, existing secure deep graph learning designs fail to scale. To enable OblivDGL to support parallelization, the insight is that our oblivious message passing paradigm inherently involves computing dependencies between nodes and edges, making direct parallelization infeasible. Indeed, as we explain in Section IV-D, simply employing parallelization within the oblivious message passing paradigm can inadvertently reveal node degrees, thereby compromising graph privacy. We require a parallelism design that efficiently processes our oblivious message passing paradigm. To address this, we design a prefix-sum mechanism that enables efficient parallel processing while preserving the security guarantees of

oblivious message passing.

Evaluation results. We implement and evaluate OblivDGL on four benchmark datasets, Cora (>2k nodes and >1k edges), Citeseer (>3K nodes and >4K edges), Pubmed (>19K nodes and >44K edges) and Facebook (>22K nodes,>343K edges). We select the state-of-the-art frameworks SecGNN [11], OblivGNN [10], and CoGNN [13] as baselines for comparison. Across all settings, OblivDGL achieves the lowest runtime and communication costs while maintaining model accuracy. For secure training, it attains speedups of up to 14.3x (Pubmed) and reduces communication by up to 1024x (Citeseer) compared to CoGNN. For secure inference, OblivDGL cuts communication by up to 754x and runtime by up to 64x, completing inference on Pubmed in 97.75 seconds with 84.6% accuracy. On the larger Facebook dataset, OblivDGL completes training in 1927.73 s with 4.52 GB of communication, while inference takes 884.99 s and incurs 4.60 GB of communication.

II. Preliminaries

A. Graph Neural Network

Graph. We study a directed graph $G = (V, E)$, which consists of two sets – the set of nodes V and the set of edges E. The set of source (src) nodes and destination (dst) nodes are denoted by $u \in V$ and $v \in V$, respectively. An edge $e(u, v) \in E$ connecting a pair of nodes u and v indicates that the direction goes from u to v, and e is the ID associated with the edge. To specify multiple nodes, we use a 1-D integer tensor of node IDs called node-tensors. Likewise, we use a tuple of node-tensors $e(u, v)$ to specify multiple edges. Specifically, a $e(u, v)$ denotes an edge from u to v. Let $H \in \mathbb{R}^{|V|\times d_1}$ denote the node feature matrix. For each node $v \in V$, let $h_v \in \mathbb{R}^{d_1}$ be its corresponding feature vector. For edge $e(u, v)$, let $w_e \in \mathbb{R}^{1\times d_2}$ be a d_2-dimensional feature vector for edge $e(u, v)$.

DGL for GNNs. Deep graph library (DGL) [19] is a deep learning framework designed to unify different GNN variants into the *message passing paradigm*. For computation at each step $t+1$, the message passing paradigm consists of a message function ϕ and a reduce function ρ as follows.

Message function ϕ is defined on each edge to generate a message m_e by combining the edge feature with the features of its incident nodes:

$$m_e^{(t+1)} = \phi\left((h_v^{(t)}, h_u^{(t)}, w_e^{(t)}) : e(u, v) \in E\right). \quad (1)$$

Reduce function ρ aggregates the incoming messages m_e at each node:

$$h_v^{(t+1)} = \rho\left(m_e^{(t+1)} : e(u, v) \in E\right). \quad (2)$$

These two primitives are essential in GNNs. A series of GNN models can be expressed by them. Without loss of generality, we focus on the two-layer Graph Convolutional Network (GCN) model and how it corresponds to the message passing paradigm.

Forward pass. The original GCN computation is centered on the adjacency matrix $A \in \{0, 1\}^{|V|\times|V|}$, where $A_{u,v} =$ 1 indicates the existence of an edge $e(u, v) \in E$. H is a node feature matrix. The computation of the two-layer forward propagation of GCN is defined as:

$$Z = \text{Softmax}(\hat{A}\text{ReLU}(\hat{A}HW^{(1)})W^{(2)}), \quad (3)$$

$W^{(1)}$ and $W^{(2)}$ are the two trainable weight matrices, $\hat{A}$ is a normalized adjacency matrix. ReLU and Softmax denote non-linear activation functions. AH actually implements message passing by propagating information along graph edges. As a result, this multiplication can be reformulated as [19]:

$$h_v^{(t+1)} = \sum_{u\in N(v)} h_u^{(t+1)}, \quad (4)$$

where $\sum$ denotes the summation function. As shown in Figure 1, the above GCN under the message passing paradigm consists of the message function ϕ that updates the edge message m_e by copying the src node feature h_u, and the reduce function ρ updates the dst node feature h_v by summing its incoming edge messages m_e. The original GCN forward propagation can be directly reformulated within the message passing paradigm as:

$$Z = \text{Softmax}(\sum_{u\in N(v)} \frac{1}{c_{vu}}\text{ReLU}(\sum_{u\in N(v)} \frac{1}{c_{vu}} h_u W^{(1)})W^{(2)}), \quad (5)$$

where $\sum$ denotes summation. $N(v)$ is the set of neighbors of node v, and $\frac{1}{c_{vu}} = \frac{1}{\sqrt{|N(v)|}\sqrt{|N(u)|}}$ is the normalization factor of A (i.e., $\hat{A}$). For each element in the node feature vector, the activation function $\text{ReLU}(h_{V_i,j}) = \max(0, h_{V_i,j})$ is defined as [23], where $j \in [0, d_1]$ and the activation function Softmax [24] is to produce the probabilities of each class for all nodes $Z_{V_i} = \text{Softmax}(h_{V_i}) = \frac{e^{h_{V_i,j}}}{\sum_{j\in[1,K]} e^{h_{V_i,j}}}$, where Z_{V_i} denotes the inference result of node V_i, and K is the number of possible classification labels.

To train the GCN model, a forward pass is first performed to obtain node predictions. The cross-entropy loss is then computed by comparing each labeled node's prediction with its label vector, as follows:

$$L = -\frac{1}{|\mathcal{T}|}\sum_{V_i\in T}\sum_{j\in[1,K]} T_{V_i,j}\ln Z_{V_i,j}, \quad (6)$$

where $\mathcal{T}$ is the set of labeled nodes and $T_{V_i,j}$ is the classification label of node V_i for class j.

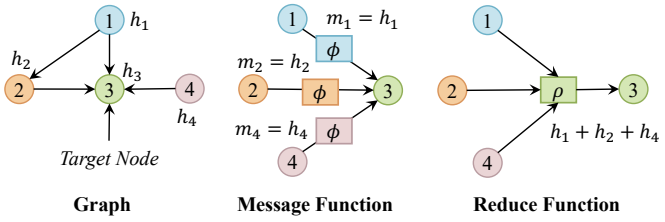


Fig. 1: GCN Message Passing Paradigm.

Backward pass. Each trainable weight $W_{i,j} \in W^{(1)} \cup W^{(2)}$ can be updated by its gradient:

$$W_{V_i,j} = W_{V_i,j} - \alpha \frac{\partial L}{\partial W_{V_i,j}}, \tag{7}$$

where α is a learning rate defining the magnitude to move towards the minimum in each iteration. To calculate Equation 7, we compute the vectors $\frac{\partial L_i}{\partial W_i}$ via message passing paradigm defined on the reverse graph (see the full proof in full version).

B. Function Secret Sharing

A two-party FSS scheme [20]–[22] splits a function g into two succinct function shares such that any function share does not reveal any information about the function g; when evaluation occurs at a given point x, the two evaluation results are combined to obtain $g(x)$. We identify the FSS variations Distributed Point Functions (DPFs) [21], Distributed Comparison Functions (DCFs) [20], and Arithmetic Function Secret Sharing [20] as natural fits for OblivDGL.

Definition 1: Distributed Point Functions (DPFs). Given a point function $g(x) \rightarrow \beta, \text{if } x = \alpha$, a DPF scheme parameterized by a finite field $\mathbb{Z}_{2^\ell}$, consists of two algorithms:

- $\mathsf{DPF.Gen}(\alpha, \beta) \rightarrow k_0, k_1$. Given inputs $\alpha, \beta \in \mathbb{Z}_{2^\ell}$, it outputs two DPF keys (k_0, k_1), that allow parties P_0, P_1 to jointly evaluate g, whose output is β only at input α.
- $\mathsf{DPF.Eval}(k_b, x) \rightarrow \mathfrak{b}_b$. Given a DPF key k_b and an input $x \in \mathbb{Z}_{2^\ell}$, party P_b locally evaluates its function share at x to obtain $\mathfrak{b}_b \in \mathbb{Z}_{2^\ell}$.

To evaluate the DPFs, all parties calculate and reveal $(\mathfrak{b}_0 + \mathfrak{b}_1) \bmod 2^\ell = \beta$ for $x = \alpha$; and $(\mathfrak{b}_0 + \mathfrak{b}_1) \bmod 2^\ell = 0$ for $x \neq \alpha$.

Similarly, DCF [20] scheme can be described as $\mathsf{DCF.Gen}(\alpha, \beta) \rightarrow k_b$ and $\mathsf{DCF.Eval}(k_b, x) \rightarrow \mathfrak{b}_b$, where $b \in \{0, 1\}$. More detailed, all parties calculate and reveal $(\mathfrak{b}_0 + \mathfrak{b}_1) \bmod 2^\ell = \beta$ if $x < \alpha$ and $(\mathfrak{b}_0 + \mathfrak{b}_1) \bmod 2^\ell = 0$ if $x \geq \alpha$.

Definition 2: Arithmetic Function Secret Sharing. An Arithmetic FSS [20] scheme securely evaluates an arithmetic gate $g^\diamond(x, y) = x \diamond y$, where $\diamond$ could be addition $+$ or multiplication $\times$.

- $\mathsf{FSS.Gen}(r_1^{\text{in}}, r_2^{\text{in}}, r^{\text{out}}) \rightarrow k_0, k_1$. Given two input masks $(r_1^{\text{in}}, r_2^{\text{in}})$ and an output mask r^{out}, it outputs two FSS keys (k_0, k_1).
- $\mathsf{FSS.Eval}(k_b, \hat{x}, \hat{y}) \rightarrow g_b^\diamond(\hat{x}, \hat{y})$. Given an arithmetic FSS key k_b and two masked inputs $\hat{x} = x + r_1^{\text{in}}$, $\hat{y} = y + r_2^{\text{in}}$, P_b evaluates the gate $g^\diamond$ to its secretly-shared gate evaluation $g_b^\diamond(\hat{x}, \hat{y}) = (x + r_1^{\text{in}}) \diamond (y + r_2^{\text{in}}) = [x \diamond y + r^{out}]_b$.

Given the above arithmetic FSS, we define the *arithmetic FSS addition gate* $\Pi_{\mathsf{Add}}(\hat{x}, \hat{y})$ as $\mathsf{Gen}(r_1^{\text{in}}, r_2^{\text{in}}, r^{\text{out}}) \rightarrow k_b$ and $\mathsf{Eval}(k_b, \hat{x}, \hat{y}) \rightarrow g_b(\hat{x}, \hat{y})$. We aim to calculate $g_0(\hat{x}, \hat{y}) + g_1(\hat{x}, \hat{y}) = x + y + r_{\text{out}}$.

Similarly, we define the *arithmetic FSS multiplication gate* $\Pi_{\mathsf{Mul}}(\hat{x}, \hat{y})$ as $\mathsf{Gen}(r_1^{\text{in}}, r_2^{\text{in}}, r^{\text{out}}) \rightarrow k_b$ and $\mathsf{Eval}(k_b, \hat{x}, \hat{y}) \rightarrow g_b(\hat{x}, \hat{y})$. We aim to calculate $g_0(\hat{x}, \hat{y}) + g_1(\hat{x}, \hat{y}) = x \cdot y + r_{\text{out}}$.

Inversion. We describe the bit inversion function for FSS Π_{Inverse}. While $\beta = 1$, $\mathsf{DPF.Eval}(k_b, x) \rightarrow \mathfrak{b}_b$ is an evaluation algorithm that given b, k_b and x outputs a share $\mathfrak{b}_b$. $\mathfrak{b}'_0 = 0 - \mathfrak{b}_0$, $\mathfrak{b}'_1 = 1 - \mathfrak{b}_1$. Specifically, $\mathfrak{b}' = \mathfrak{b}'_0 + \mathfrak{b}'_1 = 1 - (\mathfrak{b}_0 + \mathfrak{b}_1) = 1 - \mathfrak{b}$.

Oblivious Sort. Given an array of M elements, the oblivious sorting protocol outputs the elements in sorted order. Consequently, we instantiate Π_{Sort} using bitonic sort [25]. Although bitonic sort has a higher asymptotic complexity, $O(M \log^2 M)$, than alternatives such as bucket sort, $O(M \log M)$, prior work has demonstrated its practical efficiency in multi-threaded settings [26].

III. System Overview

A. System Architecture

The OblivDGL system consists of the entities (Figure 2):

- Two servers $\mathsf{S}_0, \mathsf{S}_1$ run OblivDGL's secure training to train a GNN model for the model developer. Servers retain a copy of the encrypted GNN model and pass it to OblivDGL's secure inference to produce an inference for users.
- Model developer M holding the training dataset runs OblivDGL to train a GNN model, while preserving training graph data privacy.
- Each user C sends requests to OblivDGL and receives inference results from OblivDGL, while keeping its query graph data private.

OblivDGL workflow. We detail the three components:

During *FSS key generation*, model developer M preprocesses its training graph data $(u, v, h), W^{(1)}, W^{(2)}$ into FSS keys, including arithmetic FSS keys $k_b^+, k_b^\times$ for addition and multiplication, as well as DPF and DCF keys k_b for inversion and comparison. When secure inference is invoked, each user C preprocesses its query data (u, v, h) to a set of FSS keys. Finally, the model developer M or user C can distribute these keys to server S_b for secure training and inference.

During *secure training* (OblivDGL-Training on the left side of Figure 2), upon receiving the above mentioned FSS keys, the secret shares of the graph $[u]_b, [v]_b, [h]_b$ and model weights $[W^{(1)}]_b$, $[W^{(2)}]_b$ from the model developer, the two servers $\mathsf{S}_0, \mathsf{S}_1$ jointly execute the secure forward pass, comprising the oblivious message passing paradigm (OblivMP) and the oblivious activation function (OblivAct). This is followed by the computation of the loss function (OblivLoss) and a secure backward pass (OblivDe). Finally, the updated model weights $[W^{(1)}]_b$ and $[W^{(2)}]_b$ are produced as outputs of the secure training. The trained model weights will be passed to secure inference for consuming (the encrypted Graph Model in the middle part of Figure 2).

During *secure inference* (OblivDGL-Inference on the right side of Figure 2), upon receiving the FSS keys and the secret shares of the graph $[u]_b, [v]_b, [h]_b$ from each user C, two servers S_b jointly run secure inference. This involves performing the oblivious message passing paradigm (OblivMP) and the oblivious activation function (OblivAct), utilizing the trained model weights $[W^{(1)}]_b$, $[W^{(2)}]_b$. Finally, the servers send the inference result to user C.

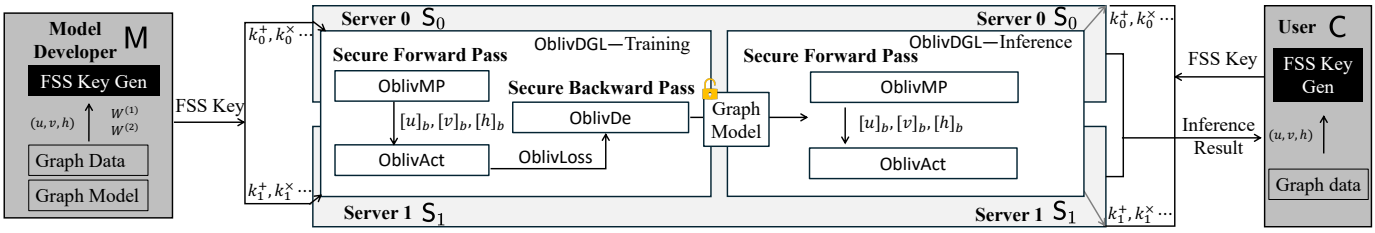


Fig. 2: System Architecture. OblivDGL consists of secure training and inference procedures. Both the model developer and the user invoke the FSS key generator to generate FSS keys for their graph data and send the keys to the servers. Then, for the training protocol, the servers sequentially execute the forward pass (oblivious message passing paradigm and activation function), loss function, and backward pass (stochastic gradient descent) to train a GNN model. For the inference protocol, the servers perform a secure forward pass and send the inference result from FSS protocols to the user.

OblivDGL functions. OblivDGL's system is executed via the following function calls.

- $\mathsf{GenKey}(u, v, \epsilon, h, W) \rightarrow k_b$: Generate a set of FSS keys given training/query graph data (u, v, ϵ, h) and initial model weight $(W^{(1)}, W^{(2)})$ held by the model developer/user.
- $\mathsf{OblivMP}(k_b, ([u]_b, [v]_b, [\epsilon]_b, [h]_b)) \rightarrow ([u']_b, [v']_b, [\epsilon']_b, [h']_b)$: Run oblivious message passing paradigm to propagate and aggregate input linear tuple $([u]_b, [v]_b, [\epsilon]_b, [h]_b)$ feature vector, to the output linear tuple $([u']_b, [v']_b, [\epsilon']_b, [h']_b)$, where the linear tuple is OblivDGL's newly defined data structure for oblivious graph representation (see Section IV-C).
- $\mathsf{OblivAct}(k_b, [h]_b) \rightarrow [h]_b$: Run oblivious activation function (ReLU and Softmax) on secretly-shared node feature $[h]_b$.
- $\mathsf{OblivLoss}(k_b, [h]_b) \rightarrow [h]_b$: Update the shared node features $[h]_b$ via oblivious cross-entropy loss given FSS keys k_b.
- $\mathsf{OblivDeLoss}(k_b, [h]_b) \rightarrow [h]_b$: Backpropagate the derivative of the cross-entropy loss on the secretly-shared node features $[h]_b$ and produce updated $[h]_b$ as outputs.
- $\mathsf{OblivDeAct}(k_b, [h]_b) \rightarrow [h]_b$: Backpropagate the derivative of the oblivious activation function on the shared node features $[h]_b$, and produce updated $[h]_b$ as outputs.

B. Threat Model and Security Guarantees

OblivDGL considers a semi-honest adversary that compromises the user and model developer M and one of the two untrusted servers S_b independently. This captures the standard two non-colluding servers model in prior works [10], [21], [22], [27]–[34], where only one server is corrupted by the adversary and the other one behaves honestly. Such an adversary will faithfully follow the prescribed protocol, but try to infer private information about the other party's inputs from the exchanged message transcripts. The non-colluding assumption is realistic, as these parties could be in different trust domains [35]–[40]. In recent years, a growing number of application providers , including Signal [36], [41]–[43], Fireblocks [37], Google [38], [44], [45], Apple [38], [44], [45], J.P. Morgan [39], and Meta [40], have adopted distributed-trust architectures to safeguard their users' sensitive data [46]. For example, Signal's secure value recovery project aims to enable users to securely back up their private keys through a distributed trust domain [36], [41]–[43], [46]. Likewise, MPC wallets platforms such as Fireblocks [37] protect digital assets by distributing users' private key across multiple parties and using MPC for signing [46]. Another real-world application is Apple and Google's ENPA system [38], [44], [45], which distributes device reports through a distributed trust domain to support privacy-preserving analytics for COVID-19 exposure notifications. These deployments collectively demonstrate the practical viability of distributed trust domains. In OblivDGL, the two servers can be hosted by different, reputable, and business-driven cloud service providers,thereby placing them in different trust domains. They are business competitors and well-established companies, and thus are unlikely to risk their commercial reputations to collude and behave maliciously.

We consider a model developer who constructs and trains a GNN model using a pair of untrusted servers. The model developer seeks to protect (i) the training graph feature and the labels of the node (during secure training only), (ii) the number of nodes and edges, and (iii) the model's weights and its architecture from being exposed to untrusted servers (and users during secure inference). The user participates only in the inference process and aims to protect the privacy of their individual query. Specifically, the user seeks to keep the following graph-related information private: (i) the inference features and labels of nodes, (ii) the number of nodes and edges, and (iii) the inference results themselves.

We clarify that all sensitive data intended for protection by the model developer/user is transmitted to the server during training and inference, and the server has access to all updated data throughout the execution phase. Furthermore, we emphasize that OblivDGL, differing from previous works, reveals only the total number of nodes and edges in the graph $(|V| + |E|)$, without leaking any additional information.

IV. Oblivious Message Passing

A. Overview

We design *oblivious message passing paradigm* to distill the secure computational patterns of GNNs training and inference into two unified oblivious functions: an oblivious reduce function, and an oblivious message function. To realize such a paradigm, we design a new oblivious data structure and realize it by FSS, a lightweight secure computation technique with low latency. We identify that these functions consist of

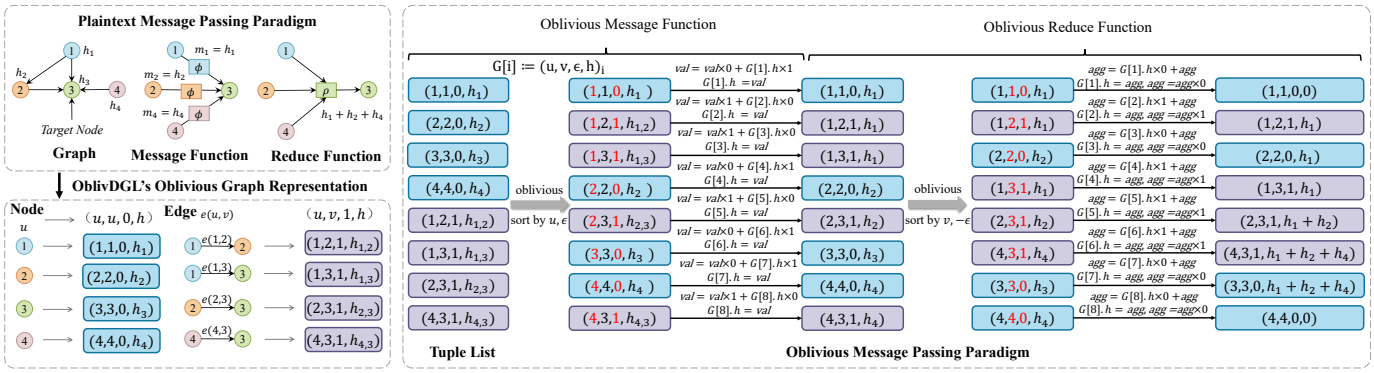


Fig. 3: The Oblivious Graph Representation and Oblivious Message Passing Paradigm. Both nodes and edges are alternatively represented as a linear tuple of graph, where node u or node v is described as $(u, u, 0, h)$; edge (u, v) is described as $(u, v, 1, h)$. Here, $h_{(\cdot)}$ represents a specific node or edge feature. All features are indistinguishable in appearance and share the same representation, denoted as $G[i].h$, $i \in [|V| + |E|]$.

a series of linear operations (e.g., aggregation) and non-linear operations (e.g., sorting). We thus resort to arithmetic FSS [20] for linear operations, as well as DPFs [21], [22] and DCFs [20] for non-linear operations. Naively implementing the message passing paradigm via FSS incurs two unique challenges.

Executing most stringent obliviousness (C1). On the one hand, the execution of message passing inherently leaks the following information about the graph structure. 1) In GraphSC [12], the message passing paradigm scatters and gathers information along edges. Due to the use of explicit if–else control flow, this process leaks the number of outgoing and incoming edges. 2) CoGNN employs collaborative learning to mitigate the overhead associated with message passing. Nevertheless, this design choice results in the leakage of the subgraph size. Therefore, we need data-obliviousness for message passing execution. More specifically, the data-oblivious algorithm does not disambiguate between edges and nodes. Unlike traditional graph computing approaches [12], [14], [47]. The data-oblivious guarantee ensures that gather and scatter operations are performed in an oblivious manner within conditional (if-else) statements. However, this guarantee is insufficient for deep graph learning, as it inadvertently exposes information such as node positions, edge positions, and node degrees. Such leakage has been exploited by attackers in model inversion attacks, including property inference [16] and graph reconstruction [17], [18].

Our idea is to eliminate conditional statements by applying the same operations to both nodes and edges, ensuring the most stringent obliviousness in the message passing paradigm. Specifically, such protocols must obscure whether the processor operates on nodes or edges. Intuitively, this can be achieved by ensuring that the algorithm executes both branches of a conditional statement, effectively concealing which branch is actually taken. This method reveals only the total number of nodes and edges ($|V| + |E|$). See Section IV-C for details.

Parallelizing oblivious message passing paradigm (C2). On the other hand, the inherent structure of GNNs necessitates intensive computation over large-scale graphs, making parallelization a natural solution for accelerating both training and inference. As the oblivious message passing paradigm is the cornerstone of our framework, we can directly parallelize its components, including message function, reduce function and prefix sum. However, parallelization of the entire paradigm would harm obliviousness due to the creation of data dependencies. It contradicts the aim of OblivDGL to achieve the most stringent obliviousness within a random linear order comprising nodes and edges.

We observe that within each processor block, oblivious message passing can be executed independently. After processing, each block only needs to update the features at the same corresponding positions, and this update process can support parallel execution. Finally, we observe that arithmetic FSS is still well-suited for implementing overall computations. See Section IV-D for details.

Notation	GNN	OblivDGL (This work)
Node	u	(u, v, ϵ, h)
Edge	$e(u, v)$	
Node Feature	h_u or h_v	h
Edge Feature	w_e	
Is Edge	—	ϵ

TABLE II: OblivDGL's Oblivious Graph Representation.

B. Oblivious graph representation via linear tuple.

To make message passing oblivious, we need to hide from the servers which node or edge of the graph an operation is applied to. That is, the processors of servers must apply the same operations uniformly across all nodes and edges [12]. To do this, we design an oblivious graph representation data structure that unifies both nodes and edges into linear tuples. Our oblivious graph structure ensures a message passing (1) remains indistinguishable between edges or nodes from each linear tuple; (2) executes the same operations (i.e., message/reduce function) over every tuple, regardless of its underlying

Algorithm 1: Oblivious Message Passing Paradigm

Inputs: The list of tuples $G(u, v, \epsilon, h)$.
Public parameters: The total number of $G : M = |V| + |E|$.

FSS Key Generation

$\mathsf{FSS.Gen}(r_1^{\text{in}}, r_2^{\text{in}}, r^{\text{out}}) \to k_b$ ▷ for addition.
$\mathsf{FSS.Gen}(r_1^{\text{in}}, r_2^{\text{in}}, r^{\text{out}}) \to k_b$ ▷ for multiplication.
$\mathsf{DPF.Gen}(0, 1) \to k_b$ ▷ for inversion.
$\mathsf{DCF.Gen}(0, 1) \to k_b$ ▷ for sorting.
Share $G := (u, v, \epsilon, h)$ as $G := ([u]_b, [v]_b, [\epsilon]_b, [h]_b)$.
Send $G := ([u]_b, [v]_b, [\epsilon]_b, [h]_b)$ and FSS key k_b to S_b.

Oblivious Message Function $\Pi_{\mathsf{OblivMF}}(u, \epsilon, h)$

Oblivious Message Function on Reverse Graph $\Pi_{\mathsf{OblivMF}}(v, \epsilon, h)$

$[t_1]_b, [t_2]_b \leftarrow 0$, $[val]_b \leftarrow 0$
$\Pi_{\mathsf{Sort}}([u]_b, [\epsilon]_b)$ or $\Pi_{\mathsf{Sort}}([v]_b, [\epsilon]_b)$ ▷ oblivious sort
for $i \leftarrow 1$ to M **do** ▷ oblivious propagate
  $[\neg G[i].\epsilon]_b \leftarrow \Pi_{\mathsf{Inverse}}([G[i].\epsilon]_b)$
  $[t_1]_b \leftarrow \Pi_{\mathsf{Mul}}([\text{val}]_b, [G[i].\epsilon]_b)$
  $[t_2]_b \leftarrow \Pi_{\mathsf{Mul}}([G[i].h]_b, [\neg G[i].\epsilon]_b)$
  $[val]_b \leftarrow \Pi_{\mathsf{Add}}([t_1]_b, [t_2]_b)$ ▷ ϕ is + operation in GCN
  $[G[i].h]_b \leftarrow [val]_b$
end for

Oblivious Reduce Function $\Pi_{\mathsf{OblivRF}}(v, -\epsilon, h)$

Oblivious Reduce Function on Reverse Graph $\Pi_{\mathsf{OblivRF}}(u, -\epsilon, h)$

$[t_3]_b \leftarrow 0$, $[\text{agg}]_b \leftarrow 0$
$\Pi_{\mathsf{Sort}}([v]_b, -[\epsilon]_b)$ or $\Pi_{\mathsf{Sort}}([u]_b, -[\epsilon]_b)$ ▷ oblivious sort
for $i \leftarrow 1$ to M **do** ▷ oblivious aggregate
  $[t_3]_b \leftarrow \Pi_{\mathsf{Mul}}([G[i].h]_b, [G[i].\epsilon]_b)$
  $[\text{agg}]_b \leftarrow \Pi_{\mathsf{Add}}([t_3]_b, [\text{agg}]_b)$ ▷ ρ is + operation in GCN
  $[G[i].h]_b = [\text{agg}]_b$
  $[\text{agg}]_b \leftarrow \Pi_{\mathsf{Mul}}([\text{agg}]_b, [G[i].\epsilon]_b)$
end for

type; (3) outputs a linear tuple preserving the exact size and format of the input.

Linear tuple: We design both nodes (u, v) and edges $e(u, v)$ as a linear tuple of the form (u, v, ϵ, h), where $\epsilon \in \{0, 1\}$:

- Node: each node u is described as $(u, u, 0, h)$, $\epsilon = 0$;
- Edge: each edge $e(u, v)$ is described as $(u, v, 1, h)$, $\epsilon = 1$;
- Feature vector: node/edge feature h_u, h_v, w_e is as feature h;
- Graph: G is a list of tuples $G[i] := (u, v, \epsilon, h)_i$, $i \in [|V| + |E|]$.

We illustrate OblivDGL's oblivious graph representation in Figure 3 (left side). Hence, by the above definition, the oblivious algorithm would hide both the graph structure, and the data on the graph's nodes and edges. Only the "size" of the graph $|V| + |E|$ is revealed.

C. Oblivious Message Passing Paradigm

OblivDGL expresses message passing paradigm as data-oblivious algorithms based on DGL [19]. It comprises the oblivious message function $\Pi_{\mathsf{OblivMF}}(u, \epsilon, h)$ and the oblivious reduce function $\Pi_{\mathsf{OblivRF}}(v, -\epsilon, h)$.

Oblivious message function. Oblivious message function takes every linear tuple (u, v, ϵ, h) as inputs, passing the feature h from source to destination, and produces (u, v, ϵ, h'), which consists of an oblivious sort and propagate operation. For clarity, we define the oblivious message function as $\Pi_{\mathsf{OblivMF}}(u, v, \epsilon, h)$.

1. Oblivious sort: On the input u and ϵ, $\Pi_{\mathsf{Sort}}(u, \epsilon)$ defines on operating the order such that each node appears before all edges originating from that node, where u and ϵ are represented in ascending order. We define the oblivious sorting algorithm as $\Pi_{\mathsf{Sort}}(\cdot)$. Specifically, $\Pi_{\mathsf{Sort}}(x)$ denotes an ascending sort, whereas $\Pi_{\mathsf{Sort}}(-x)$ denotes the descending sort. Oblivious sorting is fundamentally built upon a secure swap operation, which we realize using DCF.
   - $\mathsf{DCF.Gen}(\alpha = 0, \beta = 1) \to (k_0, k_1)$. On input the parameters (α and β), DCF key generator outputs a pair of keys (k_0, k_1). Each party P_b receives the key share k_b.
   - $[\mathfrak{b}]_b \leftarrow \mathsf{DCF.Eval}(k_b, [x_1]_b - [x_2]_b)$. On input the k_b and $([x_1]_b, [x_2]_b)$, each party P_b evaluates and outputs $[\mathfrak{b}]_b$.
   - Party P_b computes secret-shared values for the conditional swap Π_{Swap}: $[s]_b \leftarrow [\mathfrak{b}]_b \cdot ([x_1]_b - [x_2]_b) + [x_2]_b$, $[t]_b \leftarrow ([x_1]_b + [x_2]_b) - [s]_b$. Finally, return $([s]_b, [t]_b)$.

   The DCF scheme can be described as $\mathsf{DCF.Gen}(\alpha = 0, \beta = 1) \to k_b$ and $\mathsf{DCF.Eval}(k_b, x_1 - x_2) \to \mathfrak{b}_b$, where $b \in \{0, 1\}$. When $(\mathfrak{b}_0 + \mathfrak{b}_1) \bmod 2^\ell = 1$ if $x_1 - x_2 < 0$ (i.e. $x_1 < x_2$) and $(\mathfrak{b}_0 + \mathfrak{b}_1) \bmod 2^\ell = 0$ if $x_1 - x_2 \geq 0$ (i.e. $x_1 \geq x_2$). We adopt the bitonic sorting network [25], which sorts n elements with a comparison complexity of $O(n \log^2 n)$.
2. Propagate: On the input h_u, h_v and h_e, the function ϕ defines the process in which each edge updates the edge feature by combining the edge feature with the feature of its incident nodes by linear scanning. Note that the propagate operation ϕ must supply the corresponding user-defined function such that,

$$(u, v, \epsilon, \underbrace{h}_{h_e}) \leftarrow \phi(u, v, \epsilon, \underbrace{h}_{h_u, h_v, h_e}) \tag{8}$$

As for GCN, we will use the ϕ : *copy* operation, i.e., each edge data is updated by propagating node data to edge data. We need to perform the $\Pi_{\mathsf{Sort}}(u, \epsilon)$ operation, i.e., each node should appear before all edges originating from that node. In Figure 3, these two columns are highlighted in red for clarity. For a data-oblivious setting, the algorithm does not disambiguate between edges and nodes. As a result, we define a propagation value val, which is initialized 0. Then, we compute $val = val \times G[i].\epsilon + G[i].h \times \neg G[i].\epsilon$ and $G[i].h = val$, where val temporarily stores the h of this outputted tuple. We then compute the oblivious message function for the $i + 1$-th linear tuple, while using val to propagate the feature from i-th linear tuple to $i+1$. We iteratively compute the above function in every tuple by linear scanning as shown in Figure 3. OblivDGL's oblivious message function is formally described in Algorithm 1.

Oblivious reduce function. Oblivious reduce function takes every linear tuple (u, v, ϵ, h) as inputs, and produces aggregated feature (u, v, ϵ, h'), comprising an oblivious sort and aggregate operation.

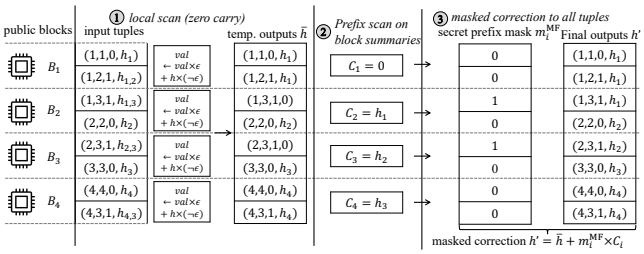


Fig. 4: Parallel Oblivious Message Function.

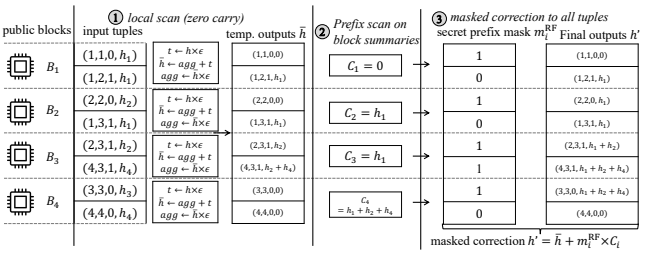


Fig. 5: Parallel Oblivious Reduce Function.

1. Oblivious sort: On the input v and $-\epsilon$, $\Pi_{\mathsf{Sort}}(v, -\epsilon)$ defines on operating the order such that the edges with the same destination node appear adjacent to each other. Further, each node should appear after the list of edges ending at that node. v and $-\epsilon$ are represented such that v is in ascending order, while $-\epsilon$ is in descending order.
2. Aggregate: On the input h_e and h_v, the function ρ defines the process in which each node updates the node feature by aggregating its incoming edge feature by linear scanning. Like the propagate operation, the aggregate operation must also supply the user-defined functions such that,

$$(u, v, \epsilon, \underbrace{h}_{h_v}) \leftarrow \rho(u, v, \epsilon, \underbrace{h}_{h_v, h_e}) \tag{9}$$

As for GCN, we will use ρ : *summation* operation, i.e., each node updates the node feature by summing its incoming messages. For a data-oblivious setting, we compute $G[i].h \times G[i].\epsilon$ and output $G[i].h$ if the tuple is an edge, 0 otherwise. We define an aggregation value agg as $G[i-1].h$, which is initialized 0. Then we compute $\text{agg} = G[i].h \times G[i].\epsilon + \text{agg}, G[i].h = \text{agg}$, and $\text{agg} = \text{agg} \times G[i].\epsilon$ to perform an aggregate operation. We perform the above function in every tuple by linear scanning as shown in Figure 3. OblivDGL's oblivious reduce function is formally defined in Algorithm 1.

Oblivious message passing on reverse graph. The backward pass of a GCN produces gradients from all the trainable weights following the chain rule. The backward computation of the original GCN at the (l-1)-th layer is defined as:

$$\mathcal{D}^{l-1} = (\hat{A}H^{l-1})^\top \Delta^l \tag{10}$$

$$\mathcal{G}^{l-1} = \hat{A}^\top \Delta^l (W^{l-1})^\top \tag{11}$$

$\Delta^l := \sigma'(Z^l) \odot \mathcal{G}^l$ and $Z^l = \hat{A}H^l W^l$, where $\odot$ represents element-wise multiplication and σ' is the derivation of activation function. $(\cdot)^\top$ represents the transposition of a matrix. $\mathcal{G}^l$ denotes the gradient of the loss with respect to the node feature H^l, Δ^l denotes the gradient with respect to the pre-activation Z^l, and $\mathcal{D}^{l-1}$ denotes the gradient of the loss with respect to the weight matrix W^{l-1}.

From the above Eq. 10 and Eq. 11, we observe that the computation of $\mathcal{G}^l$ is symmetric to H^l, while the only difference is the order of multiplication. That is, to derive $\mathcal{G}^{l-1}$, we can multiply the matrices in the order of $\hat{A}^\top$, Δ^l and $(W^{l-1})^\top$. Taking such an observation into consideration, we design the *reverse oblivious message passing* by only substituting the sorting operation in oblivious message passing. In this case, we can unify both forward-pass oblivious message passing and backward-pass reverse oblivious message passing with the same oblivious computations.

Specifically, for the reverse message function, the sort operation is defined on sort by v, which means each node should appear before all edges that are destinations from that node. For the reverse reduce function, the sort operation is defined on sorting by u, which means the edges with the same source nodes appear adjacent to each other. Further, each node should appear after the list of edges ending at that node. Specifically, we describe the derivative of the oblivious message function as $\Pi_{\mathsf{OblivMF}}(v, \epsilon, h)$ and the oblivious reduce function as $\Pi_{\mathsf{OblivRF}}(u, -\epsilon, h)$, as shown in Algorithm 1. Based on the oblivious message passing paradigm, we only modified the inputs for sorting (blue text in Algorithm 1), while the aggregate and propagate operations remain unchanged following the original oblivious message passing setup.

D. Parallelizing Oblivious Message Passing

We now describe how to parallelize the oblivious message passing paradigm. Recall that both oblivious message passing paradigms start with an oblivious sort. We observe that *oblivious sort* can be parallelized trivially. The oblivious sort is a $\log^2(|V| + |E|)$-depth circuit and therefore is parallelized at the circuit level directly.

After sorting, both propagate and aggregate operations can be viewed as segmented scans over the tuple list. The main challenge is that a segment may span multiple processor blocks. Consequently, a block locally scan cannot be corrected by updating only the first tuple of each block: if a block begins in the middle of a long edge segment, all tuples in its leading segment depend on the carry from the previous block. Our parallel protocol addresses this dependency by computing block summaries and applying masked carry corrections to all tuples in each block.

Let $M = |V| + |E|$ be the tuple-list length and let P be the public number of processors. We partition the sorted tuple list into P public consecutive blocks $B_j = [s_j, t_j]$, where $j \in [P]$ and the block boundaries depend only on M and P. After sorting, the propagate and aggregate phases are segmented scans. Each block first performs the sequential scan locally with zero initial carry, producing temporary outputs $[\bar{h}_i]$ for all $i \in B_j$. It also computes a block summary T_j, which describes how an arbitrary incoming carry affects the outgoing carry

Algorithm 2: Parallelizing Oblivious Message Function

Inputs: The list of tuples $G(u, v, \epsilon, h)$.
Public parameters: The total number of $G : M = |V| + |E|$.

1: $\Pi_{\mathsf{Sort}}([u]_b, [\epsilon]_b)$ ▷ sort by (u, ϵ); use (v, ϵ) for reverse graph
2: **for** $j \leftarrow 1$ to P **in parallel do**
3: $[val]_b \leftarrow [0]_b$, $[m]_b \leftarrow [1]_b$
4: **for** $i \leftarrow s_j$ to t_j **do**
5: $[\neg G[i].\epsilon]_b \leftarrow \Pi_{\mathsf{Inverse}}([G[i].\epsilon]_b)$
6: $[a_1]_b \leftarrow \Pi_{\mathsf{Mul}}([val]_b, [G[i].\epsilon]_b)$
7: $[a_2]_b \leftarrow \Pi_{\mathsf{Mul}}([G[i].h]_b, [\neg G[i].\epsilon]_b)$
8: $[val]_b \leftarrow \Pi_{\mathsf{Add}}([a_1]_b, [a_2]_b)$
9: $[\bar{h}_i]_b \leftarrow [val]_b$ ▷ local output under zero incoming carry
10: $[m]_b \leftarrow \Pi_{\mathsf{Mul}}([m]_b, [G[i].\epsilon]_b)$
11: $[m_i^{\mathsf{MF}}]_b \leftarrow [m]_b$ ▷ $m_i^{\mathsf{MF}} = \prod_{r=s_j}^{i} \epsilon_r$
12: **end for**
13: $[\alpha_j]_b \leftarrow [m]_b$, $[\beta_j]_b \leftarrow [val]_b$ ▷ $T_j(x) = \alpha_j x + \beta_j$
14: **end for**
15: $([C_1]_b, \ldots, [C_P]_b) \leftarrow \Pi_{\mathsf{Prefix}}(\{([\alpha_j]_b, [\beta_j]_b)\}_{j=1}^{P})$ ▷ $C_1 = 0$ and C_j is the carry entering block $\mathcal{B}_j$
16: **for** $j \leftarrow 1$ to P **in parallel do**
17: **for** $i \leftarrow s_j$ to t_j **do**
18: $[\Delta_i]_b \leftarrow \Pi_{\mathsf{Mul}}([m_i^{\mathsf{MF}}]_b, [C_j]_b)$
19: $[G[i].h]_b \leftarrow \Pi_{\mathsf{Add}}([\bar{h}_i]_b, [\Delta_i]_b)$ ▷ masked correction for every tuple
20: **end for**
21: **end for**

Algorithm 3: Parallelizing Oblivious Reduce Function

Inputs: The list of tuples $G(u, v, \epsilon, h)$.
Public parameters: The total number of $G : M = |V| + |E|$.

1: $\Pi_{\mathsf{Sort}}([v]_b, -[\epsilon]_b)$ ▷ sort by $(v, -\epsilon)$; use $(u, -\epsilon)$ for reverse graph
2: **for** $j \leftarrow 1$ to P **in parallel do**
3: $[\mathrm{agg}]_b \leftarrow [0]_b$, $[m]_b \leftarrow [1]_b$
4: **for** $i \leftarrow s_j$ to t_j **do**
5: $[m_i^{\mathsf{RF}}]_b \leftarrow [m]_b$ ▷ $m_i^{\mathsf{RF}} = \prod_{r=s_j}^{i-1} \epsilon_r$
6: $[a_3]_b \leftarrow \Pi_{\mathsf{Mul}}([G[i].h]_b, [G[i].\epsilon]_b)$
7: $[\bar{h}_i]_b \leftarrow \Pi_{\mathsf{Add}}([\mathrm{agg}]_b, [a_3]_b)$ ▷ local output under zero incoming carry
8: $[\mathrm{agg}]_b \leftarrow \Pi_{\mathsf{Mul}}([\bar{h}_i]_b, [G[i].\epsilon]_b)$
9: $[m]_b \leftarrow \Pi_{\mathsf{Mul}}([m]_b, [G[i].\epsilon]_b)$
10: **end for**
11: $[\alpha_j]_b \leftarrow [m]_b$, $[\beta_j]_b \leftarrow [\mathrm{agg}]_b$ ▷ $T_j(x) = \alpha_j x + \beta_j$
12: **end for**
13: $([C_1]_b, \ldots, [C_P]_b) \leftarrow \Pi_{\mathsf{Prefix}}(\{([\alpha_j]_b, [\beta_j]_b)\}_{j=1}^{P})$ ▷ $C_1 = 0$ and C_j is the carry entering block $\mathcal{B}_j$
14: **for** $j \leftarrow 1$ to P **in parallel do**
15: **for** $i \leftarrow s_j$ to t_j **do**
16: $[\Delta_i]_b \leftarrow \Pi_{\mathsf{Mul}}([m_i^{\mathsf{RF}}]_b, [C_j]_b)$
17: $[G[i].h]_b \leftarrow \Pi_{\mathsf{Add}}([\bar{h}_i]_b, [\Delta_i]_b)$ ▷ masked correction for every tuple
18: **end for**
19: **end for**

of this block. Since these summaries compose associatively, a fixed parallel prefix scan over $T_1, \ldots, T_P$ computes the carry $[C_j]$ entering each block. Finally, each block applies the incoming carry using secret prefix masks. For every tuple $i \in B_j$, the protocol computes

$$[h_i'] \leftarrow \Pi_{\mathsf{Add}}\left([\bar{h}_i], \Pi_{\mathsf{Mul}}([m_i], [C_j])\right), \tag{12}$$

where $[m_i]$ indicates whether tuple i belongs to the leading segment affected by C_j.

Prefix. Given the summary function T_i of each block j, the prefix operation computes the incoming carry C_i by composing the summary functions of all preceding blocks.

Parallel oblivious message function. We describe the parallel message function using GCN's copy operation as an example. The complete procedure is presented in Algorithm 2. After oblivious sorting by (u, ϵ) (Line 1 of Algorithm 2), the parallel oblivious message function proceeds as follows:

- **Line 3:** Each block initializes its local carry to zero and its prefix mask to one.
- **Lines 5-9: Local segmented scan.** Each block evaluates $[val_i] = \Pi_{\mathsf{Add}}\left(\Pi_{\mathsf{Mul}}([val_{i-1}], [\epsilon_i]), \Pi_{\mathsf{Mul}}([h_i], [1 - \epsilon_i])\right)$ with zero incoming carry, producing the temporary output $[\bar{h}_i] = [val_i]$. A node tuple with $\epsilon_i = 0$ resets the carry, whereas an edge tuple with $\epsilon_i = 1$ propagates it.
- **Lines 10-11: Prefix mask.** Each block computes $[m_i^{\mathsf{MF}}] = \prod_{r=s_j}^{i} [\epsilon_r]$, which identifies the leading edge segment whose outputs depend on the incoming carry.
- **Line 13: Block-summary computation.** Each block constructs a summary T_j describing how an arbitrary incoming carry affects its outgoing carry.
- **Line 15: Prefix scan.** A fixed parallel prefix scan over $T_1, \ldots, T_P$ computes the carry $[C_j]$ entering each block.
- **Lines 16–19: Masked correction.** Each block corrects its temporary outputs in parallel as

$$[h_i'] \leftarrow \Pi_{\mathsf{Add}}\left([\bar{h}_i], \Pi_{\mathsf{Mul}}([m_i^{\mathsf{MF}}], [C_j])\right), \qquad i \in B_j. \tag{13}$$

Thus, the incoming carry is applied only to tuples in the leading edge segment of the block.

Parallel oblivious reduce function. We describe the parallel reduce function in Algorithm 3 using GCN's summation operation as an example. After oblivious sorting by $(v, -\epsilon)$ (Line 1 of Algorithm 3), the parallel oblivious reduce function proceeds as follows:

- **Line 3:** Each block initializes its local aggregate to zero and its prefix mask to one.
- **Lines 4-10: Local segmented scan.** Each block performs the reduce scan locally with zero incoming aggregate. Edge tuples accumulate their messages, whereas a node tuple outputs the aggregate and resets the local carry. This produces the temporary output $[\bar{h}_i]$ for each tuple. During the local scan, each block computes the prefix masks, $[m_i^{\mathsf{RF}}] = \prod_{r=s_j}^{i-1} [\epsilon_r]$, (Line 5), while Line 9 updates the running mask for the next tuple.

- **Line 11: Block-summary computation.** Each block constructs $T_j(x) = \alpha_j x + \beta_j$, where $[\alpha_j] = [m]$ and $[\beta_j] = [agg]$, to describe how an arbitrary incoming aggregate affects the outgoing aggregate of the block.
- **Line 13: Prefix scan.** A fixed parallel prefix scan over the block summaries computes the incoming aggregate $[C_j]$ for each block.
- **Lines 14–19: Masked correction.** Each block applies its incoming aggregate to the temporary outputs:

$$[h'_i] \leftarrow \Pi_{\mathsf{Add}}\left([\bar{h}_i], \Pi_{\mathsf{Mul}}([m_i^{\mathsf{RF}}], [C_j])\right), \qquad i \in B_j. \quad (14)$$

  Thus, the incoming aggregate affects only the leading segment indicated by $[m_i^{\mathsf{RF}}]$.

Cost analysis. Under our standing assumption that ϕ and ρ have $O(1)$ time complexity. In this case, it is not hard to see that the parallel runtime of aggregate and propagate operations is $O(\frac{M}{P})$. The parallel oblivious message and reduce functions each have a per-core computational complexity $O(\frac{M}{P} + \log(M))$ where $M = |V| + |E|$, P is the number of processors. We provide the OblivDGL's performance on FSS. We focus on the performance of the oblivious message passing paradigm, as it comprises the majority of the computational cost in GNN inference. The complexity of oblivious message passing depends on the input size. Table III evaluates the cost of oblivious message passing using GCN as an example.

V. Implementing Secure GNN

A. Oblivious Activation Functions

After performing oblivious message passing and multiplication with the trainable weight matrix, an activation function is applied to compute the layer states (ReLU and Softmax). In this section, we design efficient protocols for oblivious activation in the FSS domain. For illustration, we focus on the two-layer GCN in Equation 5 when describing the activation design in OblivDGL.

Oblivious ReLU function. The ReLU function is defined as $\mathrm{ReLU}(x) = \max(x, 0)$. Its derivative is represented by $\mathrm{DReLU}(x) = \mathbb{1}[x > 0]$, which outputs 1 if $x > 0$ and 0 otherwise. This comparison operation is naturally supported within the FSS domain. Specifically, we observe that the Distributed Comparison Functions (DCFs) exhibit an inverse relationship relative to the ReLU operation. By leveraging this property, similar to Π_{Inverse}, we construct the oblivious ReLU function $\Pi_{\mathsf{OblivReLU}}$ and its derivative $\Pi_{\mathsf{OblivDeReLU}}$ using the inversion of DCF, as detailed below.

- $\mathsf{DCF.Gen}(0, 1) \rightarrow k_0, k_1$ is a DCF generation algorithm that given $\alpha = 0$ and $\beta = 1$ outputs a pair of keys (k_0, k_1). We refer to the output as the key share k_b for the party P_b.
- $\mathsf{DCF.Eval}(k_b, x) \rightarrow \mathfrak{b}_b$ is a DCF evaluation algorithm that given b, k_b and x outputs a share $\mathfrak{b}_b$.
- $0 - \mathfrak{b}_0 \rightarrow \mathfrak{b}'_0$, $1 - \mathfrak{b}_1 \rightarrow \mathfrak{b}'_1$. More detailed, $\mathfrak{b}' = \mathfrak{b}'_0 + \mathfrak{b}'_1 = 1 - (\mathfrak{b}_0 + \mathfrak{b}_1) = 1 - \mathfrak{b}$.
- $\Pi_{\mathsf{OblivDeReLU}} \leftarrow \mathfrak{b}'$, $\Pi_{\mathsf{OblivReLU}} \leftarrow \mathfrak{b}' \times h$.

Algorithm 4: Secure GNN Training

Inputs: The list of tuples $G(u, v, \epsilon, h)$.
Outputs: Trainable weight $W^{(1)}$, $W^{(2)}$.
Public parameters: The total number of $G : M = |V| + |E|$.

Forward Pass

First Layer

for $i \leftarrow 1$ to M **do**
- $\left[\hat{D}_u h\right]_b \leftarrow \Pi_{\mathsf{Mul}}\left(\left[(\hat{D}_u\right]_b, [h]_b\right)$
- $\left[\hat{h}\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_v\right]_b, \left[\hat{D}_u h\right]_b)$
- $\left[h^1\right]_b \leftarrow \Pi_{\mathsf{OblivMF}}([u]_b, [\epsilon]_b, \left[\hat{h}\right]_b)$
- $\left[h^2\right]_b \leftarrow \Pi_{\mathsf{OblivRF}}([v]_b, [-\epsilon]_b, [h^1]_b)$
- $\left[h_i^3\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[h_i^2\right]_b, \left[W^{(1)}\right]_b)$
- $\left[h_i^4\right]_b \leftarrow \Pi_{\mathsf{OblivReLU}}(\left[h_i^3\right]_b)$

end for

Second Layer

for $i \leftarrow 1$ to M **do**
- $\left[\hat{D}_u h^4\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_u\right]_b, \left[h^4\right]_b)$
- $\left[\hat{h}'\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_v\right]_b, \left[\hat{D}_u h^4\right]_b)$
- $\left[h^5\right]_b \leftarrow \Pi_{\mathsf{OblivMF}}([u]_b, [\epsilon]_b, \left[\hat{h}'\right]_b)$
- $\left[h^6\right]_b \leftarrow \Pi_{\mathsf{OblivRF}}([v]_b, [-\epsilon]_b, \left[h^5\right]_b)$
- $\left[h^7\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[h^6\right]_b, \left[W^{(2)}\right]_b)$
- $\left[h^8\right]_b \leftarrow \Pi_{\mathsf{OblivSoftmax}}(\left[h^7\right]_b)$

end for

Cross-entropy Loss

for $i \leftarrow 1$ to M **do**
- $\left[h_i^9\right] \leftarrow \Pi_{\mathsf{OblivLoss}}(\left[h_i^8\right])$

end for

Backward Pass

for $i \leftarrow 1$ to M **do**
- $\left[H^1\right]_b \leftarrow \Pi_{\mathsf{OblivDeLoss}}(h^9)$
- $\left[H^2\right]_b \leftarrow \left[(h^6)^\top\right]_b$, $\left[H_i^3\right]_b \leftarrow \left[(W^2)^\top\right]_b$
- $\left[H_i^4\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[H^1\right]_b, \left[H^3\right]_b)$
- $\left[H_i^5\right]_b \leftarrow \Pi_{\mathsf{OblivMF}}([v]_b, [\epsilon]_b, \left[H_i^4\right]_b)$
- $\left[H_i^6\right]_b \leftarrow \Pi_{\mathsf{OblivRF}}([u]_b, [-\epsilon]_b, \left[H_i^5\right]_b)$
- $\left[H_i^7\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[H_i^6\right]_b, \left[(\hat{D}_v)_i\right]_b)$
- $\left[H_i^8\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[H_i^7\right]_b, \left[(\hat{D}_u)_i\right]_b)$
- $\left[H_i^9\right]_b \leftarrow \Pi_{\mathsf{OblivDeReLU}}(\left[H_i^8\right]_b)$
- $\left[H^{10}\right]_b \leftarrow \left[(h^2)^\top\right]_b$
- $\left[H_i^{11}\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[H^1\right]_b, \left[H^2\right]_b)$
- $\left[H_i^{12}\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[H^9\right]_b, \left[H^{10}\right]_b)$
- $\left[H_i^{13}\right]_b \leftarrow \Pi_{\mathsf{Mul}}([\alpha]_b, \left[H_i^{11}\right]_b)$
- $\left[H_i^{14}\right]_b \leftarrow \Pi_{\mathsf{Mul}}([\alpha]_b, \left[H_i^{12}\right]_b)$
- $\left[W^{(1)}\right]_b \leftarrow \Pi_{\mathsf{Add}}(\left[W^{(1)}\right]_b, -\left[H_i^{14}\right]_b)$
- $\left[W^{(2)}\right]_b \leftarrow \Pi_{\mathsf{Add}}(\left[W^{(2)}\right]_b, -\left[H_i^{13}\right]_b)$

end for

	Communication Cost	Computational Cost
OblivGNN	$8 \times O(\|V\|^2)$	$(2 \times T_{\mathsf{Sub}} + 4 \times T_{\mathsf{Mul}} + 3 \times T_{\mathsf{Add}}) \times O(\|V\|^2)$
OblivDGL	$11 \times O(M)$	$(4 \times T_{\mathsf{DCF}} + T_{\mathsf{DPF}} + 4 \times T_{\mathsf{Mul}} + 2 \times T_{\mathsf{Add}}) \times O(M)$
Parallelization	$11 \times O(M)$	$(4 \times T_{\mathsf{DCF}} + T_{\mathsf{DPF}} + 4 \times T_{\mathsf{Mul}} + 2 \times T_{\mathsf{Add}}) \times O(\frac{M}{P})$

TABLE III: Theoretical Complexity: Communication and Computational Cost between OblivGNN and OblivDGL. $M = |V| + |E|$ is the length of tuple. T_{Sub} is the time cost of single subtraction, T_{Mul} is the time cost of single multiplication, T_{Add} is the time cost of single addition, T_{DPF} is the time cost of single DPF, T_{DCF} is the time cost of single DCF.

Algorithm 5: Secure GNN Inference

Inputs: The list of tuple $G(u, v, \epsilon, h)$
Outputs: Inference Results Z
Public Parameters: The total size of $G : M = |V| + |E|$

First layer

for $i \leftarrow 1$ to M **do**
  $\left[\hat{D}_u h\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_u\right]_b, [h]_b)$
  $\left[\hat{h}\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_v\right]_b, \left[\hat{D}_u h\right]_b)$
  $\left[h^1\right]_b \leftarrow \Pi_{\mathsf{OblivMF}}([u]_b, [\epsilon]_b, \left[\hat{h}\right]_b)$
  $\left[h^2\right]_b \leftarrow \Pi_{\mathsf{OblivRF}}([v]_b, [-\epsilon]_b, [h^1]_b)$
  $\left[h_i^3\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[h_i^2\right]_b, \left[W^{(1)}\right]_b)$
  $\left[h_i^4\right]_b \leftarrow \Pi_{\mathsf{OblivReLU}}(\left[h_i^3\right]_b)$
end for

Second layer

for $i \leftarrow 1$ to M **do**
  $\left[\hat{D}_u h^4\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_u\right]_b, \left[h^4\right]_b)$
  $\left[\hat{h}'\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[\hat{D}_v\right]_b, \left[\hat{D}_u h^4\right]_b)$
  $\left[h^5\right]_b \leftarrow \Pi_{\mathsf{OblivMF}}([u]_b, [\epsilon]_b, \left[\hat{h}'\right]_b)$
  $\left[h^6\right]_b \leftarrow \Pi_{\mathsf{OblivRF}}([v]_b, [-\epsilon]_b, \left[h^5\right]_b)$
  $\left[h^7\right]_b \leftarrow \Pi_{\mathsf{Mul}}(\left[h^6\right]_b, \left[W^{(2)}\right]_b)$
  $[Z]_b \leftarrow \Pi_{\mathsf{OblivSoftmax}}(\left[h^7\right]_b)$
end for

Secure Softmax function. As for GCN, we consider a classification task. Therefore, GCN required the Softmax function to normalize the probabilities of inference results. We compute the Softmax function securely inspired by Keller *et al.* [48] and Mohassel *et al.* [27] by replacing the softmax function by leveraging the ReLU function. To ensure obliviousness, branching conditions are replaced with conditional evaluation. Specifically, equations for all conditions are computed, and the result is selected using an oblivious selector: $(a\&condition)|(b\&\neg condition)$. We describe the oblivious Softmax function as $\Pi_{\mathsf{OblivSoftmax}}$.

B. Secure Training and Inference.

Secure training. As shown in Equation 5, each forward pass layer $l-1$ is described as $\sigma(\sum_{u \in N(v)} \frac{1}{c_{vu}} h_u W^{(l-1)})$. For the (l)-th layer, the tuple ($[u]_b, [v]_b, [\epsilon]_b, [h]_b$) is fed into the Π_{OblivMF} protocol to compute the updated feature vector $[h]_b$. Subsequently, the Π_{OblivRF} protocol aggregates all intermediate feature vectors $[h']_b$, followed by a secure matrix multiplication via Π_{Mul} with the weight matrix $[W^{(l-1)}]_b$, yielding $[h'W^{(l-1)}]_b$. Finally, the $\Pi_{\mathsf{OblivReLU}}$ or $\Pi_{\mathsf{OblivSoftmax}}$ protocol applies the activation to $[h'W^{(l-1)}]_b$ and outputs the activated feature vector $[h'']_b$.

The backward computation proceeds from layer $K-1$ to layer 0. For each layer l, the initialization of $[h]_b$ depends on the layer index. When $l = K-1$, which is the first layer processed in the backward pass, we initialize $[h]_b$ to the prediction error $[\hat{T} - T]_b$. This value corresponds to the gradient $[\mathcal{G}^K]_b$ with respect to the output representations. Each node computes this gradient locally from the loss function. Thus, no oblivious message passing is required at this layer, and we directly set $[h'']_b = [h]_b$.

For all subsequent layers ($l < K-1$), $[h]_b$ is initialized with the gradient $[\mathcal{G}^l]_b$, representing the gradient of the loss with respect to the node representation $[h]_b$. Given the incoming gradient message $[\mathcal{G}^l]_b$, each node first applies the local backward update by computing the gradient with respect to the pre-activation. If $l \neq 0$, the reverse message passing paradigm is performed to generate an outgoing gradient message. When $l = 0$, only the weight gradient $[\mathcal{D}^l]_b$ is generated, as no further gradient propagation across layers is required. After gradient aggregation and weight update, the updated weight $[W^{l'}]_b$ are assigned to $[W^l]_b$. The propagated gradient messages $[\mathcal{G}^{l-1}]_b$ are then used as inputs to the next backward message passing layer, while $[W^l]_b$ is used in the subsequent training epoch.

Remark. The backward pass of each layer l consists of (i) a local gradient update $\Delta^l = \sigma'(Z^l) \odot \mathcal{G}^l$, (ii) computing the weight gradient $\mathcal{D}^{l-1} = (\hat{A}H^{l-1})^\top \Delta^l$, and (iii) propagating gradients via reverse message passing $\mathcal{G}^{l-1} = \hat{A}^\top \Delta^l (W^{l-1})^\top$. The full protocol of secure GNN training is shown in Algorithm 4.

Secure inference. We consolidate the above components into the secure GNN inference protocol. Each layer performs OblivDGL's oblivious message passing followed by an oblivious activation (ReLU or Softmax). For an unlabeled node, the user provides its ID to the servers, which then execute secure forward propagation within OblivDGL and return an encrypted prediction for reconstruction. The detailed protocol for secure GNN inference is presented in Algorithm 5.

C. GNN Variants

Our framework builds upon DGL's message passing interface to provide native support for a wide spectrum of GNN operators, such as *u_add_v*, *u_sub_v*, and *mean*. By securely implementing the message and reduce functions within this abstraction, our design eliminates the need for protocol-level modifications when adapting to different GNN variants. This

Ideal Functionality	Input/Output Description	Proof
DPF $\mathcal{F}_\bullet$	$x \to [\mathbb{1}_{x=\alpha}]$	[22]
DCF $\mathcal{F}_<$	$x \to [\mathbb{1}_{x<\alpha}]$	[20]
Addition $\mathcal{F}_{\mathsf{Mul}}$	$[x],[y] \to [x+y]$	[20]
Multiplication $\mathcal{F}_{\mathsf{Mul}}$	$[x],[y] \to [xy]$	[20]
ReLU $\mathcal{F}_{\mathsf{ReLU}}$	$[x] \to [\mathrm{ReLU}(x)]$	[10]
Message Passing $\mathcal{F}_{\mathsf{OblivMP}}$	$[x] \to [\mathrm{OblivMP}(x)]$	§VI-A
Inverse $\mathcal{F}_{\mathsf{Inverse}}$	$[x] \to [\neg x], x \in \{0,1\}$	§VI-C
Sort $\mathcal{F}_{\mathsf{Sort}}$	$[G] \to [G']$	§VI-D
Softmax $\mathcal{F}_{\mathsf{Softmax}}$	$[x] \to [\mathrm{Softmax}(x)]$	§VI-E

TABLE IV: Definitions of Ideal Functionalities

abstraction enables modular and extensible protocol development for a broad class of secure GNN frameworks. We further validate the generality of the proposed approach by explicitly implementing and assessing its compatibility with additional representative GNN variants

- *GraphSAGE.* We insert a degree column in our linear tuple, perform oblivious message passing as usual, and then divide each node feature by its degree.
- *GIN.* For GIN, we perform the oblivious message function as in GCN, while the reduce function is defined as, $\mathrm{agg} = G[i].h \times \epsilon_i + \mathrm{agg} + G[i].h \times (\neg G[i].\epsilon), G[i].h = \mathrm{agg}$, and $\mathrm{agg} = \mathrm{agg} \cdot G[i].\epsilon$.

VI. Security Analysis

We prove the security of OblivDGL following the ideal/real-world paradigm [49]. We write $\overset{c}{\equiv}$ to denote computational indistinguishability. Table IV defines the ideal functionalities for our protocols and building blocks.

Definition 3 (Semi-honest security with leakage [49]). Let $\mathcal{F} = (\mathcal{F}_0, \mathcal{F}_1)$ be a two-party functionality and let Π be a protocol, both with input $\mathsf{in} = (\mathsf{in}_0, \mathsf{in}_1)$. Let $\mathsf{view}_i^{\Pi}(\mathsf{in})$ denote the view of party P_i in an execution of Π, and let $\mathsf{out}^{\Pi}(\mathsf{in})$ denote the output of the protocol. We say that Π securely realizes $\mathcal{F}$ with leakage $\mathcal{L}$ in the presence of static semi-honest adversaries if there exist PPT simulators $\mathsf{S}_0, \mathsf{S}_1$ such that, for every input in and every $i \in \{0,1\}$.

$$\{\mathsf{S}_i(\mathsf{in}_i, \mathcal{F}_i(\mathsf{in}), \mathcal{L}(\mathsf{in})), \mathcal{F}(\mathsf{in})\} \overset{c}{\equiv} \{\mathsf{view}_i^{\Pi}(\mathsf{in}), \mathsf{out}^{\Pi}(\mathsf{in})\}.$$

A. Proof Sketch of Oblivious Message Passing Paradigm

Theorem 1. Suppose Π_{Add}, Π_{Mul}, Π_{Inverse}, and Π_{Sort} securely realize $\mathcal{F}_{\mathsf{Add}}$, $\mathcal{F}_{\mathsf{Mul}}$, $\mathcal{F}_{\mathsf{Inverse}}$, and $\mathcal{F}_{\mathsf{Sort}}$, respectively. Then Π_{OblivMF} and Π_{OblivRF} securely realize $\mathcal{F}_{\mathsf{OblivMF}}$ and $\mathcal{F}_{\mathsf{OblivRF}}$ with leakage $\mathcal{L}_{\mathsf{OblivMP}} = M$ against static semi-honest adversaries.

Proof. By Definition 3, it suffices to simulate the view of a corrupted server S_b, for $b \in \{0,1\}$, given its input share, output share, and leakage M. Both Π_{OblivMF} and Π_{OblivRF} consist of an oblivious sorting followed by a linear scan over M tuples. The simulator first invokes the simulator of Π_{Sort} for the sorting phase. By the security of Π_{Sort}, the sorting transcript reveals only public metadata determined by M; sorting keys, comparison bits, swap decisions, tuple permutation, tuple types, graph data, source node, destination node remain hidden. At every tuple position, the real protocol executes the same sequence of secure arithmetic gates, independent of whether the tuple represents a node or an edge. In particular, the tuple-type bit $G[i].\epsilon$ is never opened and is used only as a secret-shared input to Π_{Inverse}, Π_{Mul}, and Π_{Add}.

The simulator emulates each primitive call using the corresponding primitive simulator and samples all intermediate values as random shares with the correct public dimensions. By the assumed security of the primitives, each simulated call is computationally indistinguishable from the real one. Since the protocols execute only polynomially many primitive calls, sequential composition implies that the simulated views of Π_{OblivMF} and Π_{OblivRF} are indistinguishable from the real views. Therefore, Π_{OblivMF} and Π_{OblivRF} securely realize their ideal functionalities with leakage M. □

B. Proof Sketch of Secure Training and Inference

Theorem 2. Suppose Π_{MP}, Π_{Inverse}, Π_{Add}, Π_{Mul}, Π_{ReLU}, Π_{Softmax} securely realize $\mathcal{F}_{\mathsf{MP}}$, Π_{Inverse}, Π_{Add}, Π_{Mul}, $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{F}_{\mathsf{Softmax}}$. Π_{Train} and $\Pi_{\mathsf{Inference}}$ securely realizes $\mathcal{F}_{\mathsf{Train}}$ and $\mathcal{F}_{\mathsf{Inference}}$ in the presence of static semi-honest adversaries.

Proof. Let in be arbitrary and $\{\Pi_{\mathsf{MP}}, \Pi_{\mathsf{Inverse}}, \Pi_{\mathsf{Add}}, \Pi_{\mathsf{Mul}}, \Pi_{\mathsf{ReLU}}, \Pi_{\mathsf{Softmax}}\}$ securely realize $\{\mathcal{F}_{\mathsf{MP}}, \Pi_{\mathsf{Inverse}}, \Pi_{\mathsf{Add}}, \Pi_{\mathsf{Mul}}, \mathcal{F}_{\mathsf{ReLU}}, \mathcal{F}_{\mathsf{Softmax}}\}$. By definition, $\mathcal{F}_{\mathsf{Train}} \overset{c}{\equiv} \mathsf{out}^{\Pi_{\mathsf{Train}}}$ and $\mathcal{F}_{\mathsf{Inference}} \overset{c}{\equiv} \mathsf{out}^{\Pi_{\mathsf{Inference}}}$. Observe that Π_{Train} and $\Pi_{\mathsf{Inference}}$ sequentially calls $\{\Pi_{\mathsf{MP}}, \Pi_{\mathsf{Inverse}}, \Pi_{\mathsf{Add}}, \Pi_{\mathsf{Mul}}, \Pi_{\mathsf{ReLU}}, \Pi_{\mathsf{Softmax}}\}$. By the sequential composition theorem, $\mathcal{F}(\mathsf{in})\} \overset{c}{\equiv} \mathsf{view}_i^{\Pi_{\mathsf{Train}}}(\mathsf{in})$ and $\mathcal{F}(\mathsf{in})\} \overset{c}{\equiv} \mathsf{view}_i^{\Pi_{\mathsf{Inference}}}(\mathsf{in})$ for $i \in \{0,1\}$ with leakage $|V| + |E|$. □

C. Security of Oblivious Inversion

Theorem 3. Suppose $\Pi_\bullet$ securely realize $\mathcal{F}_\bullet$. Π_{Inverse} securely realizes $\mathcal{F}_{\mathsf{Inverse}}$ in the presence of static semi-honest adversaries.

Proof. Let in be arbitrary and $\{\Pi_\bullet\}$ securely realize $\{\Pi_\bullet\}$. By definition, $\mathcal{F}_{\mathsf{Inverse}} \overset{c}{\equiv} \mathsf{out}^{\Pi_{\mathsf{Inverse}}}$. Observe that Π_{Inverse} calls $\Pi_\bullet$. By the sequential composition theorem, $\{\mathcal{F}(\mathsf{in})\} \overset{c}{\equiv} \mathsf{view}_i^{\Pi_{\mathsf{Inverse}}}(\mathsf{in})$ for $i \in \{0,1\}$. □

D. Proof Sketch of Oblivious Sorting

Theorem 4. Let $\Pi_<$ securely realize $\mathcal{F}_<$ on secret-shared sorting keys, and let Π_{Swap} securely realize $\mathcal{F}_{\mathsf{Swap}}$ on secret-shared tuples. Then Π_{Sort}, implemented as a fixed bitonic sorting network, securely realizes $\mathcal{F}_{\mathsf{Sort}}$ with leakage $\mathcal{L}_{\mathsf{Sort}} = M$, against static semi-honest adversaries corrupting at most one server, where $M = |V| + |E|$ is the tuple-list length.

Proof. By Definition 3, it suffices to simulate the view of a corrupted server S_b, for $b \in \{0,1\}$, given its input share, output share, and $\mathcal{L}_{\mathsf{Sort}} = M$.

The simulator reconstructs the bitonic sorting network from M. Since the network is fixed, its comparator schedule and memory accesses depend only on M, not on private graph data. For each comparator, the real protocol invokes $\Pi_<$ to

obtain a comparison bit $[\mathfrak{b}]$, which is never opened, and then invokes Π_{Swap} to conditionally swap all tuple fields.

The simulator emulates each comparator using the simulators of $\Pi_{<}$ and Π_{Swap}, and samples all intermediate comparison bits, swap decisions, and tuple values as random shares with the correct public dimensions. By the security of $\Pi_{<}$ and Π_{Swap}, each simulated comparator is computationally indistinguishable from the real one. Since the sorting network contains only polynomially many fixed comparators, sequential composition implies that the simulated sorting transcript is indistinguishable from the real transcript. Therefore, Π_{Sort} securely realizes $\mathcal{F}_{\mathsf{Sort}}$ with leakage M. □

E. Proof Sketch of Oblivious Softmax

Theorem 2. Suppose Π_{ReLU}, Π_{Mul} securely realize $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{F}_{\mathsf{Mul}}$. Π_{Softmax} securely realizes $\mathcal{F}_{\mathsf{Softmax}}$ in the presence of static semi-honest adversaries.

Proof. Let in be arbitrary and $\{\Pi_{\mathsf{ReLU}}, \Pi_{\mathsf{Mul}}\}$ securely realize $\{\Pi_{\mathsf{ReLU}}, \Pi_{\mathsf{Mul}}\}$. By definition, $\mathcal{F}_{\mathsf{Softmax}} \overset{c}{\equiv} \mathsf{out}^{\Pi_{\mathsf{Softmax}}}$. Observe that Π_{Softmax} sequentially calls Π_{ReLU}, Π_{Mul}. By the sequential composition theorem, $\mathcal{F}(\mathsf{in})\} \overset{c}{\equiv} \mathsf{view}_i^{\Pi_{\mathsf{Softmax}}}(\mathsf{in})$ for $i \in \{0,1\}$. □

F. Security Proof of Secure Training and Inference

In this section, we use a simulation framework [50] based on multi-party computation (MPC) to define OblivDGL's security guarantees. we consider a probabilistic polynomial-time (PPT) adversary denoted as $\mathcal{A}$, who statically corrupts at most one of the two servers. Under $\mathcal{A}$'s control, the corrupted server can act as a semi-honest entity. To keep the proof description simple, we first assume that the client behaves honestly.

To establish security within the simulation paradigm, it is necessary to define two distinct worlds: the real world, wherein honest participants execute the actual protocol, and an ideal world, where an ideal functionality $\mathcal{F}$ receives inputs from the parties and provides the outputs directly to the respective parties. $\mathcal{A}$ controls the behaviour of the corrupted server as well as observing its view during the protocol run. To make the ideal world view of $\mathcal{A}$ indistinguishable from the real world, we need to define a simulator $\mathcal{S}$ which imitates messages between the model owner, honest server, client and the corrupt server. If the ideal functionality $\mathcal{F}$ correctly models the leakage that our protocols claim to have and if $\mathcal{A}$ cannot distinguish between the two worlds, then we deem our protocols as being secure.

Ideal functionality $\mathcal{F}$. We define our ideal functionality $\mathcal{F}$ to be a graph neural network and provide the graph data training service as shown in Table IV. We allow the ideal functionality $\mathcal{F}$ to leak the following information: (i) the size of graph tuple $(|V| + |E|)$, (ii) the weight size, (iii) the number of FSS and DPF keys, (iv) the FSS and DPF key size.

Simulator construction. Let P_0 be the corrupted party. Upon receiving inputs $\mathcal{Q}$ and known leakage $\mathcal{L}(\mathcal{F}(\mathcal{Q}))$, the simulator $\mathcal{S}$ performs as follows:

1. On receiving $(\mathcal{L}(\mathcal{F}(\mathcal{Q})))$ from $\mathcal{F}$: $\mathcal{S}$ conducts FSS key generation.
2. On receiving $(k^{\times}, k^{+}, G)$ from $\mathcal{F}$: $\mathcal{S}$ samples random number G, considering the received the size of G. $\mathcal{S}$ stores it locally and forwards it to $\mathcal{A}$.
3. On receiving $k^{<}$ from $\mathcal{F}$: $\mathcal{S}$ samples random numbers for matrices after ReLU and Softmax, ensuring the output size aligns appropriately. Following this, $\mathcal{S}$ transmits the share of the sampled Graph tuple for P_0 to $\mathcal{A}$.

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 (i.e. Hybrid 0,$\cdots$,Hybrid 4).

Hybrid 0. This is the real-world distribution based on calls of $\mathcal{F}_{\mathsf{MP}}$, $\mathcal{F}_{\mathsf{Add}}$, $\mathcal{F}_{\mathsf{Mul}}$, $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{F}_{\mathsf{Softmax}}$.

Hybrid 1. For graph data, $\mathcal{S}$ simulates the shares generated by uniformly sampling share vectors based on the vector length information $\mathcal{L}$ ad model dimensions specified in algorithm. The indistinguishability from Hybrid 0 is trivial.

Hybrid 2. In this hybrid, we iteratively simulate the view of P_i received during each execution of Π_{MP}, Π_{Add}, Π_{Mul}, Π_{ReLU}, Π_{Softmax}. The simulation process calls the simulators of $\mathcal{F}_{\mathsf{MP}}$, $\mathcal{F}_{\mathsf{Add}}$, $\mathcal{F}_{\mathsf{Mul}}$, $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{F}_{\mathsf{Softmax}}$ respectively, to simulate the intermediate computation data and output data generated/received during each of Π_{MP}, Π_{Add}, Π_{Mul}, Π_{ReLU}, Π_{Softmax}. Because of secure realization of ($\mathcal{F}_{\mathsf{MP}}$, $\mathcal{F}_{\mathsf{Add}}$, $\mathcal{F}_{\mathsf{Mul}}$, $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{F}_{\mathsf{Softmax}}$), Hybrid 2 is indistinguishable from Hybrid 1.

Hybrid 3. This hybrid handles the simulation of the share redistribution after each execution Π_{Add}, Π_{Mul}, Π_{ReLU}, or Π_{Softmax}. The corrupted server P_i will receive $[z]_i + R$, where z denotes an intermediate value and R is a random mask. The indistinguishability between Hybrid 2 and Hybrid 3 is is guaranteed by the randomness of the shares received by P_i and irrelevance between the shares and P_i's data.

Hybrid 4. In this hybrid, we replace the result reconstruction phase with the simulated view. The tuple list $[G]$ and the model weight $[W]$ received from P_i are replace with the corresponding shares constructed by $\mathcal{S}$ during the result reconstruction phase. The received shares in the real world and simulated shares are in the same distribution. Thus, Hybrid 3 and Hybrid 4 are computationally indistinguishable. The proof completes.

VII. Evaluation

In this section, we demonstrate that OblivDGL represents a significant performance improvement over SOTA in secure GNN models, quantify its overhead across GNN variants, and measure the gains from our parallelization. We aim to answer the following questions:

- *What is accuracy for OblivDGL, and how does it compare to state-of-the-art secure GNNs systems and plaintext?*
- *What is the overhead introduced by OblivDGL for GNN variants, and how does it compare to prior secure GNN schemes?*
- *How much efficiency has been improved by the oblivious message passing paradigm compared with the baseline?*

Dataset	Node	Feature	Edge	# Class	∗ Train	∗ Test
Cora	2708	1433	5429	7	0.2	0.6
Citeseer	3327	3703	4732	6	0.2	0.6
Pubmed	19717	500	44338	3	0.05	0.8
Facebook	22470	128	342004	4	0.6	0.2

Class: the number of distinct ground-truth node labels.
∗ Train/Test: split ratio between training and test samples.

TABLE V: Graph Statistics for Cora, Citeseer, Pubmed (medium size graph) and Facebook (large graph).

Dataset	Plaintext	SecGNN	CoGNN	OblivGNN	OblivDGL
Cora	86.82	78 **(-8.82)**	85.93 **(-0.89)**	83.76 **(-2.52)**	85.33 **(-1.49)**
Citeseer	76.35	68.30 **(-8.05)**	74.14 **(-2.21)**	75.53 **(-0.82)**	75.42 **(-0.93)**
Pubmed	85.23	78.6 **(-6.63)**	84.72 **(-0.51)**	83.24 **(-1.99)**	84.60 **(-0.63)**
Facebook	86.09	–	–	–	85.35 **(-0.74)**

TABLE VI: Inference Accuracy.

A. Evaluation Setup

We evaluate the performance of our oblivious message passing paradigm and OblivDGL with Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz with 16 physical cores and 128 GB of RAM. The experiments were conducted in two phases: an offline phase and an online phase. The offline phase consists of FSS key generation. The online phase includes both secure training and secure inference. The inference refers to a forward pass, while the training involves an epoch of training. For all experiments, we report the runtime and communication cost. We evaluate OblivDGL on the following datasets and graph neural networks.

Baseline. For GCN-based scenarios, several specialized protocols have been proposed for secure training and secure inference. We therefore compare OblivDGL against the following representative frameworks: SecGNN [11], which supports both secure training and inference; OblivGNN [10], which focuses on secure inference; and CoGNN [13], which provides secure training and inference.

Datasets. We evaluate OblivDGL on four GNN node classification datasets, i.e., Cora [51], Citeseer [52], Pubmed [53], Facebook [54]. Existing secure GNN baselines have been evaluated only on three benchmark datasets: Cora, Citeseer, and Pubmed. For a fair comparison, we evaluate OblivDGL on the same datasets. Moreover, OblivDGL is the first secure GNN system evaluated on the Facebook dataset, which serves as a larger benchmark. Table V summarizes graph statistics and train-test splits: labeled nodes form the training set and the test set is used for inference. We adopt the same splits as prior work [13], [55].

Models. We evaluate three GNN variants: GCN [56], GraphSAGE [57], and GIN [58]. Since no specialized protocols exist for other GNN variants, we resort to microbenchmarks, as shown in Figure 13, to demonstrate the concrete performance impact of strengthening each component of OblivDGL across different GNN variants.

B. Accuracy Evaluation

We report the inference accuracy as well as the percentage point difference, defined as $\mathsf{Diff} = \mathsf{Acc}_{\mathrm{frame}} - \mathsf{Acc}_{\mathrm{Plain}}$ in Table VI. Due to fixed-point representation and non-linear approximation, all frameworks exhibit slight differences in accuracy. As shown, OblivDGL preserves model accuracy across all datasets, achieving performance comparable to plaintext models, with a mean absolute accuracy gap below 1.5 percentage points. The slight accuracy degradation can be attributed to the ReLU-based approximation used in the secure Softmax implementation, as discussed in prior work [27], [48]. Compared to CoGNN, our approach incurs only a marginal accuracy loss of less than 1 percentage point. In contrast, OblivDGL outperforms SecGNN by more than 6 percentage points in absolute accuracy. Furthermore, its accuracy is comparable to that of OblivGNN, confirming that the proposed protocol introduces negligible degradation in model utility. These results demonstrate that OblivDGL achieves strong privacy guarantees under secure computation, with negligible trade-offs in model utility.

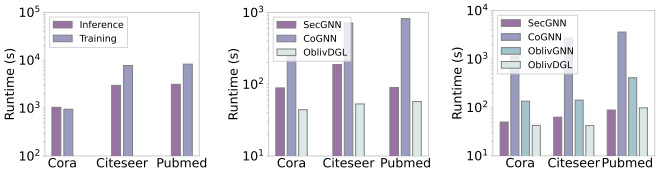
C. FSS Key Generation

Figures 6a and 7a report the runtime and communication costs of FSS key generation in the offline preprocessing phase of OblivDGL. For inference, offline preprocessing is performed between the user and the servers, whereas it is performed between the model developer and the servers for training. Specifically, OblivDGL incurs 723.77–2005.40 GB of offline communication for inference and 518.15–5282.83 GB for training. The corresponding preprocessing runtimes are 1038.78–3143.10s and 946.35–8280.62s, respectively. Among secure GNN frameworks, SecGNN incurs 5.3, 13.5, and 7.3 GB of offline communication across datasets, respectively. OblivGNN also relies on FSS preprocessing, but its adjacency-matrix-based graph updates incur prohibitive communication costs, reaching 459.89 TB on Cora. CoGNN has modest offline cost (1.18–3.27 GB), yet online training communication dominates, reaching 269.80 GB (Figure 7b). In contrast, OblivDGL cuts online communication by orders of magnitude, requiring only 0.176 GB, 0.198 GB, and 0.681 GB across datasets.

Since FSS key generation is independent of the private input data, the user for inference or the model developer for training can generate and distribute the required keys in advance, during periods of idle computational and network capacity, before the corresponding online execution begins. This shifts cryptographic operations out of online training and inference, eliminating online cryptographic overhead.

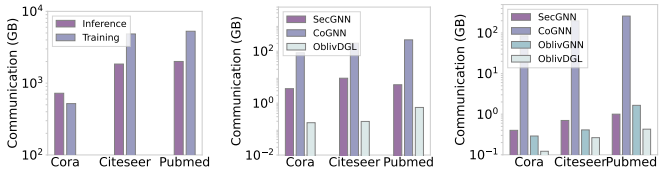
D. Runtime Evaluation

In this section, we present a detailed online runtime analysis, comparing the runtime overhead of OblivDGL with that of representative baseline frameworks as shown in Figure 6. Specifically, the inference is defined as a forward pass (see



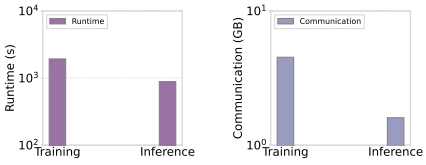
(a) Key Generation (b) Secure Training (c) Secure Inference

Fig. 6: Comparison of Runtime between OblivDGL, SecGNN, CoGNN, and OblivGNN.



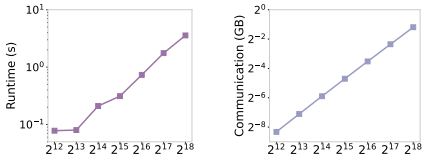
(a) Key Generation (b) Secure Training (c) Secure Inference

Fig. 7: Comparison of Communication between OblivDGL, SecGNN, CoGNN, and OblivGNN.



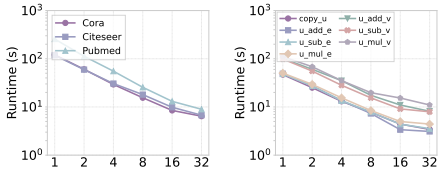
(a) Runtime on the Facebook. (b) Communication on the Facebook.

Fig. 8: Communication and Runtime on Facebook.



(a) Runtime of Oblivious Sorting. (b) Communication of Oblivious Sorting.

Fig. 9: Communication and runtime of oblivious sorting.



(a) Use GCN Datasets. (b) Use DGL Built-in Function.

Fig. 10: Parallel runtime comparison with parallelization enabled (thread varying from 2 to 32) vs disabled (thread=1).

Section V-B), whereas the training corresponds to one training epoch (see Section V).

Secure training. As shown in Table I, SecGNN [11] and CoGNN [13] are the only two state-of-the-art approaches that support secure training under the MPC setting. We therefore select them as baselines for comparison. Figure 6b compares the secure training runtime of OblivDGL with these baselines on Cora, Citeseer, and Pubmed, while the result on Facebook is reported in Figure 8. While runtime varies across datasets, the online training time of OblivDGL remains within 44.10−1927.73 s across all evaluated datasets. Across Cora, Citeseer and Pubmed, OblivDGL achieves substantial reductions in training runtime, yielding speedups of 6.3x, 13.4x, and 14.3x compared to CoGNN. When compared with SecGNN, which is based on arithmetic secret sharing, our protocol delivers speedups of 1.6x−3.5x.

Secure inference. In addition to CoGNN and SecGNN, we further compare OblivDGL with the most recent secure inference framework, OblivGNN [10]. Figure 6c summarizes the results on Cora, Citeseer, and Pubmed, while Figure 8 reports the result on Facebook. Across all evaluated datasets, the online inference time of OblivDGL ranges from 42.06 to 884.99 s. As a strong competitor, CoGNN exhibits significantly higher runtime overhead. In comparison, OblivDGL achieves substantial speedups over CoGNN, amounting to 27.21x on Cora, 64.12x on Citeseer, and 36.92x on Pubmed. Moreover, when compared with OblivGNN, which also adopts FSS primitives as its backend, OblivDGL still delivers notable runtime reductions, achieving speedups of 3.17x, 3.36x, and 4.19x on Cora, Citeseer, and Pubmed, respectively.

We identified that the most time-consuming operation during GNN training, particularly between the adjacency matrix and the feature matrix, involves sparse matrix multiplication in previous works. Consequently, by employing FSS primitives with sparsity-aware graph representations (e.g., linear tuples) and graph-structure protection mechanisms, OblivDGL significantly reduces this computational bottleneck.

Latency. OblivDGL is deployed on AWS infrastructure, leveraging the deployment support of DGL on AWS SageMaker. The latency results are summarized in Figure 11c. We assume a nationwide WAN latency of 35 ms to emulate a typical wide-area deployment. We note that OblivDGL is the first work to report end-to-end latency for both training and inference. In addition, the associated runtime constitutes a one-time cost during training and can be amortized across repeated inference requests, thereby yielding substantial improvements in online efficiency.

E. Communication Evaluation

To evaluate communication cost in OblivDGL for the online phase, we record the communication between servers as shown in Figure 7. Similar to the runtime evaluation, the communication cost evaluation considers different baselines for training and inference. During secure training, we compare OblivDGL against CoGNN and SecGNN, while during secure inference, we additionally include OblivGNN.

Secure training. Figure 7b compares the secure training communication of OblivDGL and two baseline frameworks on Cora, Citeseer, and Pubmed. Across these datasets, OblivDGL remains $<$ 0.7 GB of communication, with concrete values of 0.176 GB, 0.198 GB, and 0.681 GB, respectively. On the larger Facebook dataset, the communication cost is 4.52 GB, as shown in Figure 8. Compared to

Secure GNN System		Cora	Citeseer	Pubmed
CoGNN	*Message function*	1085.75	2507.27	2287.87
	Reduce function	51.16	148.24	155.51
RingSG	*Message function*	0.681	1.561	1.925
	Reduce function	5.835	15.97	10.729
OblivDGL	*Message function*	3.71	3.57	4.65
	Reduce function	3.02	3.54	3.60

TABLE VII: Comparison of Message Passing Runtime (s).

Secure GNN System	Cora	Citeseer	Pubmed
Adjacency Matrix	160.15	610.69	3082.02
Adjacency List	312.02	923.14	10014.92
GraphSC	1.45	3.71	5.86
CoGNN	0.61	1.49	2.62
RingSG	0.96	2.40	2.65
OblivDGL	0.09	0.23	0.32

TABLE VIII: Comparison of Message Passing Communication (GB).

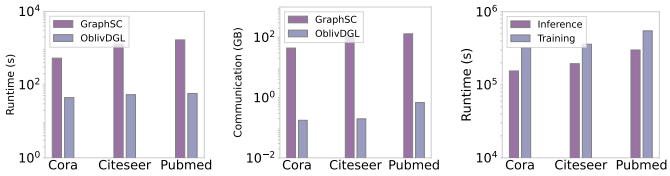
CoGNN, which relies on OT and HE, OblivDGL achieves up to 1024x lower communication on Citeseer, 494x on Cora, and 400x on Pubmed. Compared to SecGNN, OblivDGL demonstrates notable improvements, reducing communication by 20x on Cora, 46x on Citeseer, and 7.4x on Pubmed.

Secure inference. Similar to runtime evaluation, we further compare OblivDGL with OblivGNN. As shown in Figure 7c, the communication are 0.122, 0.264, and 0.427 GB for Cora, Citeseer, and Pubmed, respectively, while the cost for Facebook is 4.60 GB, as reported in Figure 8. Regarding CoGNN, OblivDGL achieves reductions of up to 692x on Cora, 754x on Citeseer, and 609x on Pubmed. Compared to SecGNN, OblivDGL also demonstrates improved communication efficiency, achieving 2.34x-3.27x reductions. For OblivGNN, OblivDGL achieves lower communication overhead, with reductions of 2.38x, 1.55x, and 3.81x, respectively.

The performance gains primarily stem from the adoption of non-interactive arithmetic FSS gates. Under this design, the communication cost is limited to revealing the masked expressions $x_1 + r^{\text{in}}$ and $\text{x}_2 + r^{\text{in}}$. This communication reduction directly translates into lower inference runtime, as shown in Figures 6.

F. Amortized Cost

Secure training in OblivDGL is performed by the cloud servers as a one-time task; thus, the runtime and communication costs of both the offline and online phases are incurred only once per trained model. For secure inference, the reported runtime and communication costs correspond to processing the entire test set rather than a single query. In practical deployments where the trained model serves repeated inference queries, the offline FSS key generation and online costs can be amortized across queries, reducing the amortized per-query cost.



(a) Comparison of Runtime. (b) Comparison of Communication. (c) Comparison of Latency across Datasets.

Fig. 11: Comparison of communication and runtime between OblivDGL and GraphSC in (a) and (b), and OblivDGL latency across datasets in (c).

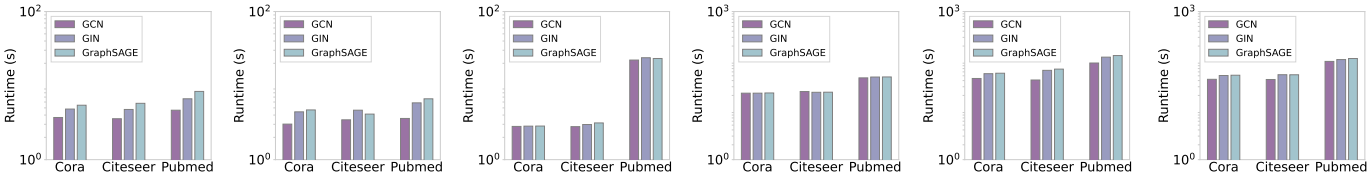
G. Improvements to Message Passing Paradigm

We demonstrate the efficiency and expressiveness of OblivDGL's message passing. Recent works CoGNN and RingSG also adopt the message passing paradigm, making them suitable baselines.

Message passing paradigm. In Table VII and VIII, we show benchmarks for the message passing paradigm. Our protocol achieves substantial runtime gains for the message function, with speedups ranging from 292.7x to 702.3x, while the reduce function attains speedups of 16.9x to 44.7x across the three evaluated datasets. The communication overhead of OblivDGL is lower than that of both adjacency matrix–based and message passing–based approaches at small scales. In RingSG, although the message function is faster, its reduce function is substantially slower than OblivDGL's. Consequently, the total message passing time of RingSG remains higher than that of OblivDGL. We note that CoGNN and RingSG reveal the size of the processed subgraph, whereas our protocol implements fully secure GNN training and inference without such leakage.

GNN variants. The proposed protocol is compatible with a range of GNN variants. In Figure 13, we benchmark different GNN variants, including GCN, GIN, and GraphSAGE. Our framework leverages DGL's message passing interface to support diverse GNN variants. By securely implementing the message and reduce functions within this abstraction, we avoid protocol-level changes when adapting to different architectures, enabling modular and extensible protocol development. To demonstrate generality, we implemented GraphSAGE with mean aggregation and GIN, with results reported in Figure 13. Both variants require only minor adjustments to the message and reduce functions, while the overall protocol remains unchanged (See Section V-C for details).

Oblivious sort. To further assess scalability, we evaluate the oblivious sorting primitive on synthetic graphs of increasing sizes, with $M = |V| + |E|$ ranging from 2^{12} to 2^{18}. As shown in Figure 9, the runtime increases from 0.078 s at $M = 2^{12}$ to 0.733 s at $M = 2^{16}$ and 3.569 s at $M = 2^{18}$, while the communication remains below 0.44 GB. Thus, even when the input size increases by 64x, the oblivious sorting remains within a few seconds.



(a) Message function. (b) Reduce function. (c) ReLU function. (d) Softmax function.

Fig. 12: Runtime Breakdown Per Full Batch for GraphSAGE, GIN, and GCN.

(a) Secure Training. (b) Secure Inference.

Fig. 13: Training and Inference Runtime for GraphSAGE, GIN, and GCN.

H. Comparison with GraphSC

Figure 11 presents the communication and computation cost of OblivDGL and GraphSC across three benchmark datasets. The results clearly demonstrate the superior efficiency of our framework in both dimensions. OblivDGL achieves three to four orders of magnitude reduction in communication volume compared to GraphSC. For example, on the Cora dataset, communication drops from 43.67 GB to 0.176 GB, a 244x reduction. Similar reductions are observed on Citeseer (548x) and Pubmed (189x), attributable to our use of FSS instead of communication-intensive OT. OblivDGL also significantly reduces training time per epoch. On Citeseer, the computation time is reduced from 1294.09s to 53.06s, yielding a 24x speedup. On Cora and Pubmed, we observe 12x and 29x speedups, respectively. These improvements stem from our lightweight protocol design and efficient oblivious primitives.

I. Scalability Evaluation of DGL's Built-in Functions

In this subsection, we examine the message passing paradigm as shown in Figure 10a. Figure 10a presents the runtime comparison of the message passing paradigm on Cora, Citeseer, and Pubmed with different thread counts. As the number of threads increases, runtime decreases significantly across all datasets. Notably, Pubmed, the largest dataset, starts with a higher runtime but achieves the greatest improvement, effectively leveraging parallelism, improving from 256.67 seconds to 8.95 seconds as thread count increases.

As shown in Figure 10b, we evaluate the performance of various DGL built-in functions, and our design demonstrates the ability to naturally support a wide range of GNN variants integrated into DGL [59]. All functions benefit significantly from threading, with runtime consistently decreasing as the number of threads increases. This highlights the scalability of our approach, which leverages parallelism to reduce computational overhead effectively. Functions with higher initial runtimes, such as *u_mul_v*, experience the most dramatic improvements with increased threading, demonstrating that our system efficiently handles computationally intensive operations. Additionally, as the number of threads increases, the performance differences between operations diminish. Overall, these results underscore the versatility and scalability of our design, which enables efficient execution of various GNN architectures by leveraging the built-in functions of DGL in a parallelized manner. This adaptability ensures that our system is well-suited for real-world applications requiring secure and efficient GNN computations.

J. Reliance on Function Secret Sharing

We adopt FSS due to its concrete efficiency in secure computation and communication. Existing secure GNN frameworks, such as CoGNN, SecGNN, and others, rely on customized protocols built upon heavyweight cryptographic primitives, including HE, OT, and ASS. In contrast, FSS provides a significantly more lightweight alternative, achieving substantially lower computation and communication overhead. Compared with CoGNN, which relies on expensive OT and HE, OblivDGL reduces communication by up to 1024x, 494x, and 400x on Citeseer, Cora, and Pubmed, respectively. In terms of computation, our protocols also provide 13x speedup on Citeseer, 10x on Pubmed, and 6.3x on Cora. We further present a comparative evaluation of OblivDGL and GraphSC (which adopts OT-based primitives) in terms of communication and computation. The detailed results are available in Section VII-H. Even when compared to SecGNN, which utilizes arithmetic secret sharing, OblivDGL demonstrates notable improvements, reducing communication by 20x on Cora, 46x on Citeseer, and 7.4x on Pubmed. In terms of computation, our protocol delivers speedups of 2x on Cora, 3.5x on Citeseer, and 1.1x on Pubmed.

VIII. Related Work

Secure deep graph learning. Most of the existing works on secure deep graph learning focus on designing cryptographic protocols for secure GNN inference [7]–[10]. Among those works, CryptoGCN [7] represents the first effort to address the challenge of secure GNN inference. Some follow-up works [8], [9] aim to optimize and adapt graph structures for scalability. However, these approaches suffer from significant performance bottlenecks when applied to realistic and large-scale networks due to the underlying heavy cryptographic primitives. Recently, OblivGNN [10] proposes to leverage function secret sharing (FSS) [20]–[22] to securely compute linear layers and approximate activation functions, which significantly reduces computational and communication overheads. Nonetheless, OblivGNN directly computes on graph matrices, thus revealing critical properties, such as the number of nodes, that are vulnerable to potential leakage. On the other hand, secure GNN training has been studied in one prior work [11]. This work utilizes a padding technique to protect

the graph information leakage during the training phase. However, it comes at the cost of significant computational overhead, limiting its scalability and efficiency.

Privacy-preserving machine learning. The problem of privacy-preserving machine learning (PPML) has been extensively studied [27]–[30], [33], [60]–[65]. Prior solutions leverage secret sharing (SS) [27], [60], [61] or garbled circuit [33], [62], [63] to enable PPML, while some follow-up works [64], [65] further integrate homomorphic encryption (HE) schemes to improve the runtime performance of PPML frameworks. However, those solutions can involve extensive runtime computation and communication costs, which limit the real-world deployment of those solutions. Recently, using function secret sharing [20]–[22] to enable PPML applications has attracted much attention [10], [28]–[30]. Existing work has shown that FSS-based solution can enable PPML with low computation and communication costs [10], [30], and can be further optimized with GPU acceleration [28], [29]. Despite these notable successes, FSS has not been specifically optimized or customized for the unique computational patterns and challenges of GNNs, leaving a significant gap in secure deep graph learning.

Secure graph processing. Many prior works have studied secure graph processing [12], [14], [15] rather than secure deep graph learning. Among these works, GraphSC [12] represents the first effort to address the challenges of secure graph analytics. From a security perspective, GraphSC provides weaker structural privacy guarantees. Although it is designed to conceal control flow, its execution algorithm still permits explicit if–else branching. Its application to GNNs still results in leakage of structural information such as node degrees and neighborhood sizes. In contrast, OblivDGL fully conceals both aspects, thereby providing stronger structural privacy guarantees within the secure deep graph learning. For functionality, GraphSC doesn't provide GNN-specific support for dense linear layers, activation and loss functions, gradient computation, optimizers, or weight updates. For performance, GraphSC uses oblivious transfer, while OblivDGL adopts function secret sharing for lower communication and computation overhead.

OblivDGL offers a unified design that supports training and inference within a single framework, while enabling parallelization and supporting GNN variants. OblivDGL's parallelization preserves stronger structural privacy, as it reveals only $|V|+|E|$. RingSG [14] reveals the sizes of the subgraphs during secure graph analytics.

GraphOS [15] focuses on oblivious graph processing, supporting key-value access and exact graph analytics queries e.g., graph traversal and shortest-path, which is built on a TEE-based system model with ORAM. In contrast, OblivDGL adopts a two-server FSS model under the non-collusion assumption and targets privacy-preserving graph learning, specifically DGL-style GNN training and inference, where the core building blocks include oblivious message passing paradigm, secure non-linear activations. Specifically, GraphOS relies on graph data in Omix++, a doubly-oblivious key-value store, to support isolated node/edge lookups in a TEE-based graph database. Its stated leakage profile reveals the number of nodes and the number of edges. In contrast, secure GNN training requires several building blocks beyond the scope of GraphOS, including feature-tensor message aggregation, message-reduce operations, non-linear activations, and reverse message passing for backpropagation. These capabilities cannot be obtained directly through GraphOS's key–value interface. Extending GraphOS to support them would therefore require the design of a new TEE-based secure GNN training system, rather than providing GNN training as a straightforward application of GraphOS. Overall, secure graph processing and OblivDGL target different applications: GraphOS is optimized for private graph queries and exact graph analytics, whereas OblivDGL is optimized for privacy-preserving DGL-style GNN training and inference.

Oblivious sort. Oblivious sorting ensures that the sequence of compare-and-swap operations and memory accesses is independent of the secret input values. Bitonic sort is the most practical oblivious sorting algorithm in multi-threaded settings, as demonstrated by prior work [12], [13], [25], [26], [66], [67]. Bitonic sort is based on a sorting network, ensuring that the sequence of element comparisons is deterministic and independent of the input values. Moreover, its independent compare-and-swap operations at each level of the sorting network can be executed in parallel, making it highly scalable. DBucket sort [68] is designed to accelerate oblivious sorting in multi-enclave TEE settings. It achieves $O(M \log M)$ asymptotic complexity, whereas bitonic sort incurs $O(M \log^2 M)$. Despite its higher asymptotic complexity, bitonic sort can outperform bucket sort in practice.

OBLIVIATOR [26] demonstrates that bitonic sort is faster in single-server, multi-threaded settings. Specially, Dbucket sort [68] is designed for with TEE with multiple TEE enclaves , whereas OblivDGL's oblivious bitonic sort uses FSS backend under two non-colluding servers without trusted hardware. Nevertheless, we highlight that naively realizing the DBucket sort in MPC would requires additional protocols for oblivious shuffle, oblivious routing, oblivious compaction, and share refreshing to preserve obliviousness. The computational and communication costs introduced by these additional protocols may offset the $O(\log M)$ asymptotic advantage over bitonic sort. Moreover, incorporating these additional protocols would effectively amount to designing a new MPC-compatible bucket-sort protocol, rather than directly reusing DBucket sort [68].

IX. More Discussions

Deployment scenarios and limitations. OblivDGL can be deployed as a privacy-preserving GNN service in an MLaaS setting. Following the outsourced multi-server settings considered in prior PPML and GNN systems, a graph owner secret-shares the graph structure, node features, and labels between two computation servers operated in distinct administrative trust domains, such as independent cloud providers. The servers maintain the model parameters and intermediate values

in secret-shared form throughout training and inference [10], [11], [27]. We assume that the servers are semi-honest and non-colluding and communicate through authenticated and confidential channels. A practical deployment must also provision sufficient computation, memory, storage, and network bandwidth for secret-shared tensors, FSS keys, and inter-server communication [10], [28]–[30].

This deployment model introduces non-negligible operational overhead. Establishing and operating independent trust domains may require third-party business relationships and separate mechanisms for availability, security, fault tolerance, logging, recovery, authentication, and billing [46]. In addition, OblivDGL's computation, storage, and communication costs grow with the graph size, while the FSS preprocessing material may impose substantial memory requirements. Supporting large or frequently updated graphs may further require re-sharing modified graph records and, when the graph dimensions or execution schedule change, regenerating and redistributing preprocessing material [10].

Cardinality leakage. Compared with existing secure GNN frameworks, OblivDGL provides the strongest obliviousness guarantee: it reveals only the total number of node and edge (i.e., $|V|+|E|$). A detailed comparison is presented in Table I. GraphSC uses a linear tuple representation and oblivious message passing, but permits explicit if-else control flow, which may expose node degrees and neighborhood sizes when instantiated for GNN workloads. CoGNN and RingSG realize the oblivious message passing to protect the structure privacy but reveal the $|V|$ of subgraph. SecGNN and OblivGNN operate on sparse adjacency matrix, yet reveal $|V|$.

Cardinality leakage in certain scenarios (i.e., finance) is common across existing secure GNN frameworks. This leakage is particularly relevant to dynamic graphs: repeated observations of $|V| + |E|$ across updates, may reveal changes of tuple size, such as variations in participating accounts or transaction relationships. When the model developer updates the training graph by adding participating accounts, transaction relationships, or both, the servers observe only the updated tuple-list size ($M' = M + n = |V| + |E| + n$), where n denotes the number of newly added tuples. Because OblivDGL uniformly encodes nodes and edges and processes them obliviously, the servers cannot determine whether the increase arises from new accounts, new transaction relationships, or a combination of both. Thus, the observed change in the tuple-list size cannot be directly attributed to node or edge growth, although the tuple-list size changes from M to M' remains visible. Other potential mitigations include padding and differentially private algorithms for graph processing [69].

X. Conclusion

We present OblivDGL, a framework that ensures security, scalability, and functionality for encrypted GNN training and inference in a semi-honest two-server setting. This work bridges the gap between current PPML models and plaintext GNN models, addressing critical challenges in securely handling graph-structured data. OblivDGL introduces a comprehensive set of algorithms that enable data-oblivious and low-interaction training and inference for a wide range of GNN variants. By adopting a highly optimized and scalable design, OblivDGL ensures robust privacy guarantees while maintaining practical efficiency, making it adaptable to real-world applications. Our evaluation shows that OblivDGL though offering higher security guarantees reduces the runtime cost by $6.3-14.3$x to $27.21-64.12$x compared to the SOTA designs for training and inference, respectively.

Acknowledgment

This research was supported in part by the Australian Research Council (No. FT240100043, No. LP250100307, No. DE260100019).

Ethical Considerations

We ensure that all parties involved in this project understand the design and secure computation techniques used for data handling. All secure computation techniques and system designs employed in OblivDGL are carefully documented to ensure transparency and ethical soundness. Our evaluation introduces no ethical concerns, as all experiments are conducted on publicly available benchmark datasets, neither of which involves the collection or processing of personally identifiable information (PII). Consequently, this research does not pose risks of adverse outcomes, either during the study or through its publication.

References

[1] Y. Qin, X. Liu, S. Lai, J. K. Liu, and X. Yuan, "Oblivdgl: Oblivious deep graph learning made practical," in *NDSS*, 2027.

[2] B. Wu, X. Yuan, S. Wang, Q. Li, M. Xue, and S. Pan, "Securing graph neural networks in mlaas: A comprehensive realization of query-based integrity verification," in *IEEE S&P*, 2024.

[3] E. Onose, "Machine learning as a service: What it is, when to use it and what are the best tools out there," https://neptune.ai/blog/machine-learning-as-a-service-what-it-is-when-to-use-it-and-what-are-the-best-tools-out-there, 2023.

[4] Amazon, "Sagemaker," https://aws.amazon.com/sagemaker/, 2017.

[5] W. Jin, J. M. Stokes, R. T. Eastman, Z. Itkin, A. V. Zakharov, J. J. Collins, T. S. Jaakkola, and R. Barzilay, "Deep learning identifies synergistic drug combinations for treating COVID-19," *Proceedings of the National Academy of Sciences*, vol. 118, no. 39, p. e2105070118, 2021.

[6] Y. Liu, X. Ao, Z. Qin, J. Chi, J. Feng, H. Yang, and Q. He, "Pick and choose: A gnn-based imbalanced learning approach for fraud detection," in *ACM WWW*, 2021.

[7] R. Ran, W. Wang, Q. Gang, J. Yin, N. Xu, and W. Wen, "Cryptogcn: Fast and scalable homomorphically encrypted graph convolutional network inference," in *NeurIPS*, 2022.

[8] R. Ran, N. Xu, T. Liu, W. Wang, G. Quan, and W. Wen, "Penguin: Parallel-packed homomorphic encryption for fast graph convolutional network inference," in *NeurIPS*, 2023.

[9] H. Peng, R. Ran, Y. Luo, J. Zhao, S. Huang, K. Thorat, T. Geng, C. Wang, X. Xu, W. Wen, and C. Ding, "Lingcn: Structural linearized graph convolutional network for homomorphically encrypted inference," in *NeurIPS*, 2023.

[10] Z. Xu, S. Lai, X. Liu, A. Abuadbba, X. Yuan, and X. Yi, "Oblivgnn: Oblivious inference on transductive and inductive graph neural network," in *USENIX Security*, 2024.

[11] S. Wang, Y. Zheng, and X. Jia, "Secgnn: Privacy-preserving graph neural network training and inference as a cloud service," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2923–2938, 2023.

[12] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *IEEE S&P*, 2015.

[13] Z. Zou, Z. Liu, J. Shan, Q. Li, K. Xu, and M. Xu, "Cognn: Towards secure and efficient collaborative graph learning," in *ACM CCS*, 2024.

[14] ——, "Ringsg: Optimal secure vertex-centric computation for collaborative graph processing," in *ACM CCS*, 2025.

[15] J. G. Chamani, I. Demertzis, D. Papadopoulos, C. Papamanthou, and R. Jalili, "Graphos: Towards oblivious graph processing," *VLDB*, 2023.

[16] Z. Zhang, M. Chen, M. Backes, Y. Shen, and Y. Zhang, "Inference attacks against graph neural networks," in *USENIX Security*, 2022.

[17] V. Duddu, A. Boutet, and V. Shejwalkar, "Quantifying privacy leakage in graph embedding," in *ACM MobiQuitous*, 2020.

[18] Z. Zhang, Q. Liu, Z. Huang, H. Wang, C. Lu, C. Liu, and E. Chen, "Graphmi: Extracting private graph data from graph neural networks," in *IJCAI*, 2021.

[19] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019.

[20] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function secret sharing for mixed-mode and fixed-point secure computation," in *EUROCRYPT*, 2021.

[21] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *ACM CCS*, 2016.

[22] ——, "Function secret sharing," in *EUROCRYPT*, 2015.

[23] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[24] W. Liu, Y. Wen, Z. Yu, and M. Yang, "Large-margin softmax loss for convolutional neural networks," in *ICML*, 2016.

[25] K. E. Batcher, "Sorting networks and their applications," in *SJCC*, 1968.

[26] A. Mavrogiannakis, X. Wang, I. Demertzis, D. Papadopoulos, and M. N. Garofalakis, "OBLIVIATOR: oblivious parallel joins and other operators in shared memory environments," in *USENIX Security*, 2025.

[27] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *IEEE S&P*, 2017.

[28] N. Jawalkar, K. Gupta, A. Basu, N. Chandran, D. Gupta, and R. Sharma, "Orca: Fss-based secure training and inference with gpus," in *IEEE S&P*, 2024.

[29] K. Gupta, N. Jawalkar, A. Mukherjee, N. Chandran, D. Gupta, A. Panwar, and R. Sharma, "SIGMA: secure GPT inference with function secret sharing," *PoPETs*, 2024.

[30] T. Ryffel, P. Tholoniat, D. Pointcheval, and F. R. Bach, "Ariann: Low-interaction privacy-preserving deep learning via function secret sharing," *PoPETs*, 2022.

[31] X. Liu, B. Wu, X. Yuan, and X. Yi, "Leia: A lightweight cryptographic neural network inference system at the edge," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 237–252, 2022.

[32] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi, "Secfloat: Accurate floating-point meets secure 2-party computation," in *IEEE S&P*, 2022.

[33] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: xnor-based oblivious deep neural network inference," in *USENIX Security*, 2019.

[34] E. Dauterman, M. Rathee, R. A. Popa, and I. Stoica, "Waldo: A private time-series database from function secret sharing," in *IEEE S&P*, 2022.

[35] J. Aas and T. Geoghegan, "Introducing isrg prio services for privacy respecting metrics," https://www.abetterinternet.org/post/introducing-prio-services/, 2020.

[36] Signal, https://signal.org/, 2024.

[37] Fireblocks, https://www.fireblocks.com/, 2023.

[38] Apple and Google, "Exposure notification privacy-preserving analytics (enpa) white paper," https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf, 2021.

[39] A. Polychroniadou, G. Asharov, B. E. Diamond, T. Balch, H. Buehler, R. Hua, S. Gu, G. Gimler, and M. Veloso, "Prime match: A privacy-preserving inventory matching system," in *USENIX Security*, 2023.

[40] J. Reyes, "Building the next generation of digital advertising with mpc," *Real World Crypto*, 2022.

[41] Y. Lindell, D. Cook, T. Geoghegan, S. Gran, R. Schmidt, E. Kret, D. Kaviani, and R. A. Popa, "The deployment dilemma: Merits & challenges of deploying mpc," https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html, 2023.

[42] J. Lund, "Technology preview for secure value recovery," https://signal.org/blog/secure-value-recovery/, 2019.

[43] G. Connell, V. Fang, R. Schmidt, E. Dauterman, and R. A. Popa, "Secret key recovery in a global-scale end-to-end encryption system," in *USENIX OSDI*, 2024.

[44] D. Kaviani, "Google & apple, privacy-preserving COVID-19 exposure notifications," https://mpc.cs.berkeley.edu/posts/Privacy-Preserving-COVID-19-Exposure-Notifications/, 2024.

[45] Apple and Google, "Privacy-preserving contact tracing," https://covid19.apple.com/contacttracing, 2020.

[46] D. Kaviani, S. Tan, P. G. Kannan, and R. A. Popa, "Flock: A framework for deploying on-demand distributed trust," in *USENIX OSDI*, 2024.

[47] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure graph analysis at scale," in *ACM CCS*, 2021.

[48] M. Keller and K. Sun, "Effectiveness of mpc-friendly softmax replacement," *CoRR*, vol. abs/2011.11202, 2020.

[49] O. Goldreich, *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[50] Y. Lindell, "How to simulate it - a tutorial on the simulation proof technique," Cryptology ePrint Archive, Paper 2016/046, 2016.

[51] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," *Information Retrieval*, vol. 3, no. 2, pp. 127–163, 2000.

[52] C. L. Giles, K. D. Bollacker, and S. Lawrence, "Citeseer: An automatic citation indexing system," in *ACM DL*, 1998.

[53] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad, "Collective classification in network data," *AI Magazine*, vol. 29, no. 3, pp. 93–106, 2008.

[54] B. Rozemberczki, C. Allen, and R. Sarkar, "Multi-scale attributed node embedding," *Journal of Complex Networks*, vol. 9, no. 2, 2021.

[55] A. Kolluri, T. Baluta, B. Hooi, and P. Saxena, "Lpgnet: Link private graph networks for node classification," in *ACM CCS*, 2022.

[56] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR (Poster)*, 2017.

[57] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017.

[58] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *ICLR*, 2019.

[59] Deep Graph Library, "dgl.function," https://docs.dgl.ai/api/python/dgl.function.html, 2023.

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

[61] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "Ezpc: Programmable and efficient secure two-party computation for machine learning," in *IEEE EuroS&P*, 2019.

[62] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: scalable provably-secure deep learning," in *ACM DAC*, 2018.

[63] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski, "Garbled neural networks are practical," Cryptology ePrint Archive, Paper 2019/338, 2019.

[64] C. Juvekar, V. Vaikuntanathan, and A. P. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security*, 2018.

[65] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference system for neural networks," in *USENIX Security*, 2020.

[66] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *USENIX NSDI*, 2017.

[67] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *VLDB*, 2020.

[68] N. Ngai, I. Demertzis, J. G. Chamani, and D. Papadopoulos, "Distributed & scalable oblivious sorting and shuffling," in *IEEE S&P*, 2024.

[69] S. Mazloom and S. D. Gordon, "Secure computation with differentially private access patterns," in *ACM CCS*, 2018.

[70] J.-H. He, S. Elagan, and Z. Li, "Geometrical explanation of the fractional complex transform and derivative chain rule for fractional calculus," *Elsevier Physics letters A*, 2012.

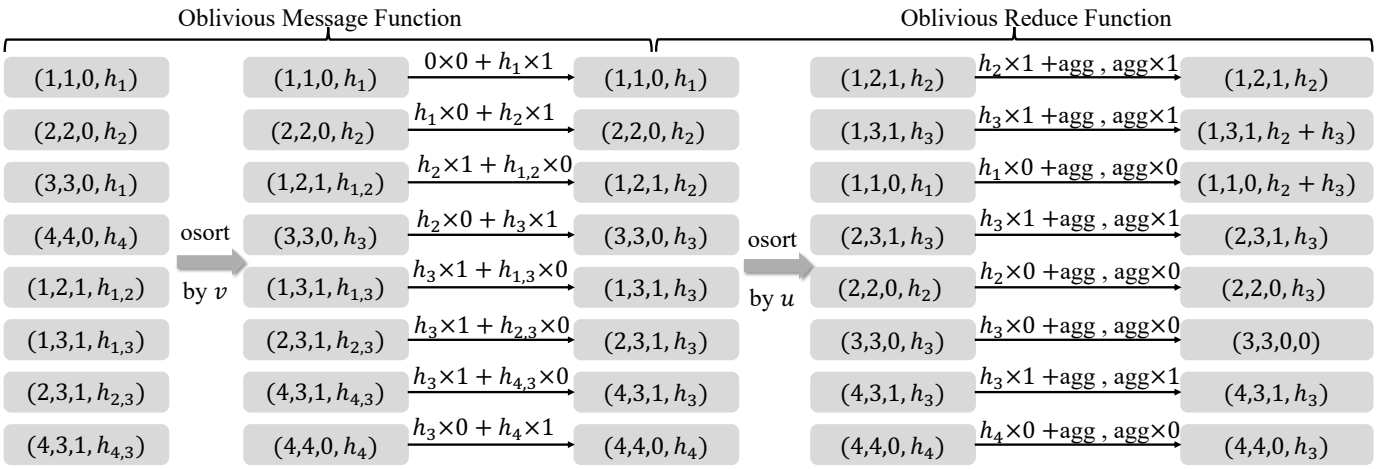


Fig. 14: Oblivious Reverse Message Passing Paradigm. In the reduce function, agg denotes the data of h_{i-1}, which is initialized to 0. We change the Π_{Sort} operation based on the oblivious message passing paradigm to achieve the reverse consequence.

APPENDIX A
MORE STOCHASTIC GRADIENT DESCENT DETAILS

First, we decompose Equation 5 as follows,

$$H^{(0)} = \psi\left(\frac{1}{c_{vu}} h_u\right) \tag{15}$$
$$H^{(1)} = H^{(0)} W^{(1)} \tag{16}$$
$$H^{(2)} = \mathrm{ReLU}(H^{(1)}) \tag{17}$$
$$H^{(3)} = \psi\left(\frac{1}{c_{vu}} H^{(2)}\right), \quad \psi = \rho(m_e), m_e = \phi(H^{(2)}) \tag{18}$$
$$H^{(4)} = H^{(3)} W^{(2)} \tag{19}$$
$$Z = \mathrm{Softmax}(H^{(4)}) \tag{20}$$
$$L = L(Z) \tag{21}$$

$$\begin{aligned}
\frac{\partial L}{\partial W^{(2)}} &= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W^{(2)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}}\frac{\partial H^{(4)}}{\partial W^{(2)}} \\
\frac{\partial L}{\partial W^{(1)}} &= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}}\frac{\partial H^{(4)}}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}}\frac{\partial H^{(4)}}{\partial H^{(3)}}\frac{\partial H^{(3)}}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}}\frac{\partial H^{(4)}}{\partial H^{(3)}}\frac{\partial H^{(3)}}{\partial H^{(2)}}\frac{\partial H^{(2)}}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}}\frac{\partial H^{(4)}}{\partial H^{(3)}}\frac{\partial H^{(3)}}{\partial H^{(2)}}\frac{\partial H^{(2)}}{\partial H^{(1)}}\frac{\partial H^{(1)}}{\partial W^{(1)}}
\end{aligned} \tag{22}$$

The cloud servers perform backward propagation to calculate each trainable weight's gradient $\frac{\partial L}{\partial W_{V_i,j}}$ by chain rule as shown in Equation 22. First we need to calculate $\frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}}$. According to Appendix V-A, we replace Softmax inspired by Keller *et al.* [48]. This is to say, the implement gradient descent modifying the gradient of L for h_i as follows:

$$\frac{\partial L}{\partial Z}\frac{\partial Z}{\partial H^{(4)}} = \frac{\partial L}{\partial h_i} = \begin{cases} -T_i, & \text{if } \sum_j \mathrm{ReLU}(h_j) = 0 \\ -T_i + \frac{\mathrm{ReLU}(h_i)}{\sum_j \mathrm{ReLU}(h_j)}, & \text{otherwise} \end{cases} \tag{23}$$

which can be securely calculated by using oblivious if in Section V-A. Similarly, The $\frac{\partial H^{(2)}}{\partial H^{(1)}}$(i.e.ReLU function's derivative) is:

$$\frac{\partial H^{(2)}}{\partial H^{(1)}} = \frac{\partial \mathrm{ReLU}(h)}{\partial h} = \begin{cases} 0, & \text{if } h \le 0 \\ 1, & \text{if } h > 0 \end{cases} \tag{24}$$

which can be securely calculated using the inverse DCF function in Section V-A. Finally, the $\frac{\partial H^{(3)}}{\partial H^{(2)}}$ (i.e. message passing paradigm's derivative) can be calculated as follows:

$$\frac{\partial H^{(3)}}{\partial H^{(2)}} = \frac{\partial \psi}{\partial H^{(2)}} = \frac{\partial \rho}{\partial m_e}\frac{\partial \phi}{\partial H^{(2)}} \tag{25}$$

According backward's feature, $\frac{\partial \rho}{\partial m_e}$is reverse reduce function and $\frac{\partial \phi}{\partial H^{(2)}}$ is the reverse message function. The detailed gradient of the message passing paradigm can refer to [19].

A. The Gradient of Message Passing Paradigm

This section derives gradients step by step, showing how matrix-based backpropagation with normalization and activation can be equivalently expressed through reverse tuple message passing.

According backward pass, $\frac{\partial \rho}{\partial m_e}$ is reverse reduce function and $\frac{\partial \phi}{\partial H^{(2)}}$ is the reverse message function. The detailed gradient of the message passing paradigm can refer to [19]. Without loss of generality, we use GCN as a simple case. Suppose the message function ϕ *: copy* , reduce function ψ*: sum*. We could verify that the computation is equivalent to multiplying the node embedding matrix X with the adjacency matrix A:

$$X^{(0)} = AX \tag{26}$$
$$X^{(1)} = X^{(0)}W^{(1)} \tag{27}$$
$$X^{(2)} = \mathrm{ReLU}(X^{(1)}) \tag{28}$$
$$X^{(3)} = AX^{(2)} \tag{29}$$
$$X^{(4)} = X^{(3)}W^{(2)} \tag{30}$$
$$Z = \mathrm{Softmax}(X^{(4)}) \tag{31}$$
$$L = L(Z) \tag{32}$$

$$\begin{aligned}
\frac{\partial L}{\partial W^{(2)}} &= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W^{(2)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial X^{(4)}}\frac{\partial X^{(4)}}{\partial W^{(2)}} \\
\frac{\partial L}{\partial W^{(1)}} &= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial X^{(4)}}\frac{\partial X^{(4)}}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial X^{(4)}}\frac{\partial X^{(4)}}{\partial X^{(3)}}\frac{\partial X^{(3)}}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial X^{(4)}}\frac{\partial X^{(4)}}{\partial X^{(3)}}\frac{\partial X^{(3)}}{\partial X^{(2)}}\frac{\partial X^{(2)}}{\partial W^{(1)}} \\
&= \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial X^{(4)}}\frac{\partial X^{(4)}}{\partial X^{(3)}}\frac{\partial X^{(3)}}{\partial X^{(2)}}\frac{\partial X^{(2)}}{\partial X^{(1)}}\frac{\partial X^{(1)}}{\partial W^{(1)}}
\end{aligned} \tag{33}$$

We can compute the gradient according to the chain rule as shown in Equation 33. The gradient of the activation function can be computed by Equation 23, 24. Then, given the gradient $\frac{\partial X^{(3)}}{\partial X^{(2)}}$, we can compute as follow,

$$\frac{\partial X^{(3)}}{\partial X^{(2)}} = A^\top \tag{34}$$

$A^\top$ denotes the transposition of A, which is equivalent to the reverse of the graph. Another way to understand it is that according to the chain rule multi-variable case [70], the gradient of a variable is the sum of the partial derivative of each component. For a graph, a node is used by all its outgoing edges during forward propagation, so the gradients need to be summed over the reversed edges.

Taking inspiration from Equation 34 and DGL [19], its gradient can be computed via *an oblivious message passing paradigm defined on the reverse graph* $\tilde{G}$. For ease of exposition, we perform an oblivious reverse message passing paradigm for GCN as shown in Figure 14.

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

This artifact implements the privacy-preserving deep graph learning framework proposed in our paper, "OblivDGL: Oblivious Deep Graph Learning Made Practical." This artifact provides the code needed to validate the experimental results reported in the paper, including those for secure training, secure inference, and breakdowns. The source code is publicly available, with detailed setup instructions provided in https://doi.org/10.5281/zenodo.23682406.

1) How to access: The artifact is available in an anonymous repository: https://doi.org/10.5281/zenodo.23682406, which contains the source code, documentation needed to validate the experimental results reported in the paper.

2) Hardware dependencies: The artifact was evaluated on an x86_64 server, equipped with an Intel Xeon E-2288G CPU and 128 GB of RAM.

3) Software dependencies: The artifact has been tested on Ubuntu 22.04 and requires a CPU with AES-NI and AVX2 support. Software dependencies include GCC $\geq$ 11 with C++20 support, CMake $\geq$ 3.25, and the GMP, OpenSSL, TBB, NTL, and gf2x libraries. All dependencies are listed in `README.md` and can be installed using a single `apt-get install` command.

4) Benchmarks: We evaluate the artifact on four benchmark datasets: Cora, Citeseer, Pubmed, and Facebook. Cora, Citeseer, and Pubmed are citation networks commonly used to evaluate GNNs for node classification. Facebook is a page-to-page network in which nodes represent verified Facebook pages and edges represent mutual likes.

B. Artifact Installation & Configuration

Our documentation provides detailed instructions for setting up the required environment and installing the artifact. The main steps are as follows:

- Get the artifact from https://doi.org/10.5281/zenodo.23682406.
- Install CMake and the required libraries by following the instructions in the `Install dependencies` section of `README.md`.

C. Evaluation

1) Experiment (E1): [Effectiveness of Secure Training]: Experiment (E1) evaluates the effectiveness of our proposed secure GNN training and provides a breakdown of the results across four datasets.

[How to] All preparation steps are described in Artifact Appendix A. The commands required to run the functionality tests for Experiment (E1) are provided in `README.MD`.

[Results] The experiment should run successfully on all four datasets and output the training time and its breakdown for the proposed secure GNN training approach.

2) Experiment (E2): [Effectiveness of Secure Inference]: Experiment (E2) evaluates the effectiveness of our proposed secure GNN Inference and provides a breakdown of the results across four datasets.

[How to] All preparation steps are described in Artifact Appendix A. The commands required to run the functionality tests for Experiment (E2) are provided in `README.md`.

[Results] The experiment should run successfully on all four datasets and output the inference time and its breakdown for the proposed secure GNN Inference approach.

3) Experiment (E3): [Effectiveness of Oblivious Sorting]: Experiment (E3) evaluates the effectiveness of our proposed secure Oblivious Sorting.

[How to] All preparation steps are described in Artifact Appendix A. The commands required to run the functionality tests for Experiment (E3) are provided in `README.md`.

[Results] The experiment should run successfully on all four datasets and output the oblivious sorting times.