OblivGNN: Oblivious Inference on Transductive and Inductive Graph Neural Network

Zhibo Xu, *Monash University and CSIRO's Data61;* Shangqi Lai, *CSIRO's Data61;* Xiaoning Liu, *RMIT University;* Alsharif Abuadbba, *CSIRO's Data61;* Xingliang Yuan, *The University of Melbourne;* Xun Yi, *RMIT University*

https://www.usenix.org/conference/usenixsecurity24/presentation/xu-zhibo

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

OblivGNN: Oblivious Inference on Transductive and Inductive Graph Neural Network

Zhibo Xu[1,2], Shangqi Lai[2], Xiaoning Liu[3], Alsharif Abuadbba[2], Xingliang Yuan[4], and Xun Yi[3]

[1]*Monash University*, [2]*CSIRO's Data61*, [3]*RMIT University*, [4]*The University of Melbourne*

Abstract

Graph Neural Networks (GNNs) have emerged as a powerful tool for analysing graph-structured data across various domains, including social networks, banking, and bioinformatics. In the meantime, graph data contains sensitive information, such as social relations, financial transactions, and chemical structures, and GNN models are IPs of the model owner. Thus, deploying GNNs in cloud-based Machine Learning as a Service (MLaaS) raises significant privacy concerns.

In this paper, we present a comprehensive solution to enable secure GNN inference in MLaaS, named OblivGNN. OblivGNN is designed to support both *transductive* (static graph) and *inductive* (dynamic graph) inference services without revealing either graph data or GNN models. In particular, we adopt a lightweight cryptographic primitive, i.e., function secret sharing, to achieve low communication and computation overhead during inference. Furthermore, we are the first to propose a secure update protocol for the inductive setting, which can obliviously update the graph without revealing which parts of the graph are updated. Particularly, our results with three widely-used graph datasets (Cora, Citeseer, and Pubmed) show that OblivGNN can achieve comparable accuracy to an Additive Secret Sharing-based baseline. Nonetheless, our design reduces the runtime cost by up to 38% and the communication cost by $10\times$ to $151\times$, highlighting its practicality when processing large graphs with GNN models.

1 Introduction

A growing emphasis has been placed on the development of Graph Neural Networks (GNNs) due to their power of analysing complex graph-structured data. GNNs are adopted in various domains (e.g., social networks [17], banking [67], bioinformatics [68]). Due to the popularity of GNNs, Machine Learning as a Service (MLaaS) has started to support GNN training and inference service [1, 4, 35]. Enterprises deploy GNN models in MLaaS for known benefits like scalability, reduced infrastructure maintenance, and convenient deployment.

Despite great advantages, deploying GNNs in an untrusted environment raises privacy concerns. On the one hand, collecting training graphs and building a GNN model often requires a large amount of human, computing, and economic resources [2, 66], and thus a model owner expects strong protection of its training graphs and GNN model against unauthorised access. On the other hand, GNNs' inference needs both model parameters, graph structure and node feature information, and sometimes graph data is sensitive by nature [67], such as users' financial transactions, and private friendships.

To ensure data confidentiality for machine learning in MLaaS, previous efforts have been made to construct privacy-preserving machine learning (PPML) protocols over encrypted data and models [16, 19, 28, 38–40, 44–46, 49, 51, 52, 56, 58]. However, those studies are designed for Convolutional Neural Networks and Recurrent Neural Networks, and are not tailored for GNNs. Although recent studies [14, 50, 59] designed to secure graph data in GNNs, they lack support for *full protection* of graph structures and feature information under the diverse settings of GNN deployment.

Practical GNN applications [17, 18, 27] demand that the inference should be conducted with both static graphs (*transductive*) and dynamic graphs (*inductive*). In this paper, we consider the node classification task which is the most representative task in GNN applications. In the transductive setting, the graph used in training and inference is the same. The classification labels for all nodes are generated during training, and the client directly queries the node ID to get the inference result. However, in the inductive setting, the inference graph is different from the training graph. Both the graph structure and node features can be updated before inference.

On the one hand, existing work reveals certain information about graph structures, such as the maximum node degree in SecGNN [59] and the range of node degree in CryptoGCN [50]. On the other hand, they do not support the inference in the inductive setting, where the graph needs to be securely updated. Specifically, directly updating the graph (e.g., adding new nodes or modifying the node features) will reveal the access to the encrypted graph. Such access could be exploited to infer the graph structure, which has been demonstrated in leakage attacks against encrypted graph processing protocols [21].

Another issue is that the existing PPML designs for GNNs suffer from high computational and communication overhead given the large size of the graph and the high dimension of node features. For example, some work [14, 50] adopts heavy cryptographic primitives like FHE which can hardly scale for large matrix multiplication. Therefore, it is challenging to enable privacy-preserving inference over encrypted graphs and GNN models under transductive and inductive settings in a secure and efficient manner.

1.1 Summary of Techniques

We propose OblivGNN, which brings the following capabilities: 1) full protection on the graph structure data during inference; 2) enabling oblivious inference under *transductive* and *inductive* settings; 3) ensuring reduced, constant inter-party communication during secure computation.

Full Protection on Graph Structure during Inference. As mentioned above, existing secure GNN designs [14, 50, 59] reveal the access pattern or partial graph information to achieve practical performance. To fully protect the graph structure in the GNN inference process, we observe that the native matrix multiplication can naturally hide the graph's structural information. However, the existing techniques [5, 23] for secure matrix multiplication encounter high computational and communication overhead.

To enable a low-cost secure matrix multiplication process, we resort to a lightweight cryptographic primitive, named function secret sharing (FSS) [6]. FSS provides an efficient way to conduct oblivious additions/multiplications with low computation and communication costs. Moreover, this technique can be used to realise practical oblivious secure activation functions [52], which makes it a perfect candidate for our application scenario. However, there are two main challenges when combining FSS primitives in GNN. First, FSS schemes, e.g., Distributed Point Function (DPF) [6] and arithmetic FSS gates [13] (Section 2.2) are designed to enable dedicated functionalities. This requires meticulous combinations of those schemes for specific GNN functions, e.g., matrix multiplication, and activation functions (Section 4.2.3), in a seamless and secure way. Moreover, unlike ASS-based designs that rely on independent randomness, the FSS-based design follows a circuit fashion that requires the gates (FSS keys) to be connected correctly for specific functions. Hence, it is also challenging to design a proper mechanism (Section 4.2.1) for generating/storing FSS keys correctly.

In OblivGNN, we carefully integrate different FSS functionalities together to enable secure and efficient matrix multiplication. Furthermore, the secure matrix multiplication protocol is customised to seamlessly integrate with the FSS-based secure activation functions, which guarantees a secure and access-pattern hiding (*oblivious*) inference process.

Support *Transductive* and *Inductive* Inference. The aforementioned design enables OblivGNN to conduct inference under the transductive setting, i.e., inference on static graphs. However, it is also crucial to support inferences on a graph with updated node information (*inductive* setting) to enable full GNN functionality in real-world applications [17, 18, 27].

Dynamic graphs are crucial in real-world application scenarios (i.e., *inductive* inference setting). However, the graph update operations entitle accessing individual nodes and nodes' features to perform the update. Such operations result in access pattern leakage if unprotected and can be exploited to recover/perturb the graph structure and reveal/perturb graph attribute information. Therefore, to protect the above leakage when supporting the dynamic graph in GNN inference, we develop secure oblivious update algorithms for OblivGNN. The proposed algorithms leverage the specific instantiation of FSS (DPF [6]) to update server-side sub-graphs into existing graphs without revealing which parts of the graph are updated. Our design ensures that the updated graph can be used as the input of the inference process directly, and thus enable OblivGNN to support oblivious GNN inference service on dynamic graphs.

Contributions. We present OblivGNN, a lightweight and oblivious GNN inference service. OblivGNN provides:

- **Fully support secure GNN inference:** We have devised the first oblivious GNN inference framework that supports both *transductive* and *inductive* settings, facilitating the protection of the inference and dynamic graph update operations (node insertion and graph update).
- **Obliviousness with Semi-honest Security:** We formally prove that the OblivGNN offers obliviousness under the semi-honest security assumption. Our proof successfully demonstrates the protection of the graph data, the graph update and clients' enquiry. A detailed exposition can be found in Appendix A.
- **Efficiency:** We have devised effective and oblivious protocols tailored specifically for fully oblivious GNN models to support both static and dynamic graphs. In three datasets, OblivGNN achieves a lower runtime cost compared to traditional ASS-based secure GNN, and it is scalable towards large dataset. To be specific, the baseline uses 11.05 minutes, while the OblivGNN uses only 6.83 minutes in the largest graph dataset (Pubmed with the size of 19717 nodes and 500 features). In the inter-server communication cost, OblivGNN achieved an average improvement of 93× among the three datasets.

2 Preliminary

This section introduces the graph neural networks and cryptographic tools used in OblivGNN. The notations are summarised in Table 1.

2.1 Graph Neural Networks

Node Classification via GNNs. Graph neural network (GNN) is a neural network architecture specifically designed to analyse graph-structured data. There are various graph analytics tasks enabled by GNNs. OblivGNN considers the most widely used one, i.e., graph convolutional networks (GCNs) [33] for node classification task. Specifically, let $\mathcal{G} = (\mathbf{A}, \mathbf{F})$ be an attributed graph, where $\mathbf{A} \in \{0,1\}^{n\times n}$ denotes the graph structure, and $\mathbf{F} \in \mathbb{R}^{n\times c}$ denotes the node features. Suppose the graph contains a number of n nodes with c-dimension of the feature vector, e.g., the $\mathbf{F}[i] \in \mathbb{R}^{1\times c}$ is the feature vector for node i. Given the nodes i labelled with respective $\mathbf{Z}_1$, a GNN node classification task aims to classify a node with a label based on both graph structure and node features.

GNN model formulation. We now formulate the GCN model architecture for node classification. Intuitively, GCN is organised in a pipeline, where each layer aggregates the features of a current node and its adjacent nodes (neighbours), followed by a non-linear activation function. A two-layer feed-forward GCN for node class predictions is formally defined as:

$$\mathbf{Z}_1 = \text{Softmax}(\hat{\mathbf{A}}\text{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})\mathbf{W_1}), \tag{1}$$

where $\hat{\mathbf{A}}$ is an $n \times n$ normalised adjacency matrix corresponding to graph structure; $\hat{\mathbf{F}}$ is an $n \times c$ normalised feature matrix. The weight matrices $\mathbf{W_0}, \mathbf{W_1}$ contain trainable parameters of GCN model. Softmax and ReLU are two non-linear activation functions. We follow standard GCN inference and refer the reader to the work from Kipf and Welling [33] for the details of training and normalization.

Transductive & Inductive GNNs. At a high level, GNN for node classification comprises two learning settings: transductive and inductive. The training process for both transductive and inductive settings is identical. The main difference is attributed to the dynamism of the underlying graphs. In the *transductive setting*, the graph used during both training and inference remains unchanged. In an MLaaS setting, the server can pre-generate the inference results for all nodes at the training time, and directly return inference results to any consequent queries at the inference time.

In the *inductive setting*, the graphs used for training and inference are different. This is because of the appearance of new nodes/graphs that would connect to existing nodes in the inductive inference. In an MLaaS setting, when a client submits new nodes or existing nodes with updated features for queries, the server should update the corresponding adjacency matrix and feature matrix with these newly submitted nodes before proceeding with the inference. The dynamism of the inductive setting makes it better suit real-world scenarios where the graph is employed during training to involve updates, such as new node insertions or graph updates, before the inference process [17, 18, 27].

b	Party number $b \in \{0,1\}$
P_b	Party P_0, P_1
m	Total number of output classes
h_l	Size of weight matrix in layer l
L	Number of classes
$\mathbf{A}^*$	Adjacency matrix for subgraph
$\mathbf{F}^*$	Feature matrix for subgraph
$\mathbf{A}'$	Masked adjacency matrix
$\mathbf{F}'$	Masked feature matrix
$\hat{\mathbf{A}} = \{\hat{A}_0, \ldots\}$	Normalised adjacency matrix
$\hat{\mathbf{F}} = \{\hat{f}_0, \ldots\}$	Normalised feature matrix
$\mathbf{N} = \{\text{N}[0][0], \ldots\}$	Padded neighbouring nodes matrix
$\mathbf{E} = \{\text{E}[0][0], \ldots\}$	Padded neighbouring edge matrix
$\mathbf{AF} = \{af_0, \ldots\}$	Aggregated matrix
$\mathbf{Z}_l = \{z_l[0], \ldots\}$	Result on layer l before activation

Table 1: Notation

2.2 Function Secret Sharing

A two-party function secret sharing (FSS) [6, 13] splits a function f into succinct function shares. It guarantees that each function share reveals no information about f. When the function shares are evaluated at a given point x, the combination of the evaluations is equal to $f(x)$. OblivGNN resorts to two FSS constructions 1) distributed point functions (DPF [6]) and 2) arithmetic FSS [13]. We now give their formal definitions.

DPF for Equality Test. $\mathsf{DPF.Equa}(\llbracket x \rrbracket)$ aims to evaluate whether a secret value x inputted to a point function equals to 0. Such that,

$$\mathsf{Eval}^{=}(k_0^{=}, x') + \mathsf{Eval}^{=}(k_1^{=}, x') = \begin{cases} y = 1 & \text{if } x' = \gamma \\ 0 & \text{otherwise.} \end{cases}$$

During key generation, the client takes random value $\gamma \in \mathbb{Z}_{2^\ell}$ and runs $\mathsf{KeyGen}^{=}(1^\lambda, \alpha = \gamma, \beta = 1)$ to produce k_b for two parties. During evaluation, given a secret value $\llbracket x' \rrbracket_b = \llbracket \gamma \rrbracket_b + \llbracket x \rrbracket_b$ is additively masked by $\gamma \in \mathbb{Z}_{2^\ell}$. Parties exchange $\llbracket x' \rrbracket_b$ to reveal the masked x'. Each party P_b evaluates the DPF $\mathsf{Eval}^{=}(k_b, x')$ to produce secret shared $\llbracket y \rrbracket_b$, which would be reconstructed to have $y = 1$ only if $x' = \gamma$ (i.e., $x = 0$).

DPF for Comparison. $\mathsf{DPF.Comp}(\llbracket x \rrbracket)$ works in a similar way. Such that,

$$\mathsf{Eval}^{<}(k_0^{<}, x') + \mathsf{Eval}^{<}(k_1^{<}, x') = \begin{cases} y = 1 & \text{if } x' \le \gamma \\ 0 & \text{if } x' > \gamma \end{cases}$$

Given a point function compares if x is less/equal to 0. the client runs $\mathsf{DPF.Comp}$'s key generation on $\alpha = \gamma$ and $\beta = 1$ to produce k_b. During DPF evaluation, party P_b evaluates the DPF $\mathsf{Eval}(k_b, x')$ on masked x and produces secret shared $y = 1$ if $x' \le \gamma$ (i.e., $x \le 0$). Otherwise, it is evaluated to $y = 0$ if $x' > \gamma$ (i.e., $x > 0$). Note that in both constructions, the DPF key generation is independent of the computation; it thus can be generated offline at random and stored in a key pool. For

Algorithm 1: FSS Multiplication $\mathsf{FSS.Mul}(x'_1, x'_2)$

```
Multiplication Gate (KeyGen^×, Eval^×)
KeyGen^×$(1^\lambda, r_1^{in}, r_2^{in}, r^{out})$
1) Sample random number $\tau_0, \tau_1 \in \mathbb{Z}_{2^\ell}^{in}$, where
   $\tau_0 + \tau_1 = r_1^{in}$.
2) Sample random number $\rho_0, \rho_1 \in \mathbb{Z}_{2^\ell}^{in}$, where
   $\rho_0 + \rho_1 = r_2^{in}$.
3) Sample random number $\xi_0, \xi_1 \in \mathbb{Z}_{2^\ell}^{in}$, where
   $\xi_0 + \xi_1 = r_1^{in} \cdot r_2^{in} + r^{out}$.
4) Let $k_b = \tau_b \parallel \rho_b \parallel \xi_b$, where $b \in \{0,1\}$.
Return: $(k_0^\times, k_1^\times)$.
Eval^×$(b, k_b^\times, x'_1, x'_2)$
1) Parse $k_b^\times = \tau_b \parallel \rho_b \parallel \xi_b$.
Return: $b \cdot (x'_1 \cdot x'_2) - x'_1 \cdot \rho_b - x'_2 \cdot \tau_b + \xi_b$.
```

ease of demonstration, we omit the security parameter in the key generation KeyGen routine.

FSS Multiplication Gate The FSS multiplication gate aims to secretly calculate the arithmetic function $g^\times(x_1, x_2) := x_1 \cdot x_2$ with masks r_1^{in}, r_2^{in}, i.e., $(x_1 + r_1^{in}) \cdot (x_2 + r_2^{in})$. The two routines in FSS multiplication gate are denoted as $\mathsf{KeyGen}^\times$ and $\mathsf{Eval}^\times$. We define the FSS multiplication evaluation process such that, $[\![x_1 \cdot x_2 + r_{out}]\!]_b \leftarrow \mathsf{FSS.Mul}(x'_1, x'_2)$. The formal algorithm is in Algorithm 1.

FSS Addition Gate. The FSS addition gate aims to secretly calculate masked $g^+(x_1, x_2) := x_1 + x_2$ with masks r_1^{in}, r_2^{in}, i.e., $(x_1 + r_1^{in}) + (x_2 + r_2^{in})$ non-interactively. The two underlying routines in FSS addition gate are denoted as KeyGen^+ and Eval^+. We define the FSS addition evaluation process such that, $[\![x_1 + x_2 + r_{out}]\!]_b \leftarrow \mathsf{FSS.Add}(x'_1, x'_2)$. The formal algorithm is in Algorithm 2.

2.3 Additive Secret Sharing

Additive secret sharing (ASS) [3] *shares* a secret value $x \in \mathbb{Z}_{2^\ell}$ into two additive secret shares $[\![x]\!]_0 + [\![x]\!]_1 \equiv x \pmod{2^\ell}$ by sampling random values $[\![x]\!]_0 \in_R \mathbb{Z}_{2^\ell}$ and $[\![x]\!]_1 = x - [\![x]\!]_0$. Let P_0 and P_1 be two parties involved in the computation. Each of them holds secret shares $[\![x]\!]_b, b \in \{0,1\}$. To *reveal* a secret, P_{1-b} sends its share $[\![x]\!]_{1-b}$ to P_b who computes $x = [\![x]\!]_0 + [\![x]\!]_1$. Given two secret values x and y shared among parties. Arithmetic operations are evaluated by ASS as follows. Addition $[\![z]\!]_b = [\![x]\!]_b + [\![y]\!]_b \pmod{2^\ell}$ can locally computed by P_b. Multiplication $[\![z]\!] = [\![x]\!] \cdot [\![y]\!] \pmod{2^\ell}$ is assisted by Beaver's multiplication technique [5].

3 System Overview

In this section, we introduce OblivGNN's system architecture, threat model and security guarantees.

Algorithm 2: FSS Addition $\mathsf{FSS.Add}(x'_1, x'_2)$

```
Addition Gate (KeyGen^+, Eval^+)
KeyGen^+$(1^\lambda, r_1^{in}, r_2^{in}, r^{out})$
1) Sample random number $\xi_0, \xi_1 \in \mathbb{Z}_{2^\ell}^{in}$, where
   $\xi_0 + \xi_1 = r^{out} - (r_1^{in} + r_2^{in})$.
2) Let $k_b^+ = \xi_b$, where $b \in \{0,1\}$.
Return: $(k_0^+, k_1^+)$.
Eval^+$(b, k_b^+, x'_1, x'_2)$
1) Parse $k_b^+ = \xi_b$.
Return: $x'_1 + \xi_0$ and $x'_2 + \xi_1$.
```

3.1 System Architecture

Figure 1 shows OblivGNN's system architecture, which has three entities: *model owner*, *client*, and two cloud *servers*.

- Model owner possesses a GNN model and wishes to leverage OblivGNN to enable an oblivious inference without revealing the model (i.e., training graph and model weights).
- Client is the querier who requests an oblivious node classification by making queries to OblivGNN. It submits node ID (transductive) and/or graph (inductive) to OblivGNN and receives inference results without revealing its private query.
- Two cloud servers in different trust domains obliviously execute the inference on both transductive and inductive GNNs, without seeing the model from the model owner and the query from the client in cleartext.

OblivGNN Workflow. OblivGNN has two settings of oblivious GNN inferences: Oblivious *Transductive* and Oblivious *Inductive*. Oblivious *Transductive* offloads the graph and model weights to the cloud servers in secret-shared format and only supports client queries to infer an existing node. Oblivious *Inductive*, on the other hand, supports updating the graph that has been already offloaded to the clouds. Both settings support OfflineGen, OblivAgg, and OblivAct, while Oblivious *Inductive* further supports OblivUpdate. Only OfflineGen is invoked by the model owner, all others are executed by the two cloud servers. Both settings are under OblivGNN's system framework and can be smoothly switched between each other. From a high-level point of view, Oblivious *Transductive* and Oblivious *Inductive* operate as follows.
Oblivious *Transductive*:

- OfflineGen: Given a ring $\mathbb{Z}_{2^\ell}$, mask and secret share the adjacency matrix $\mathbf{\hat{A}}$, feature matrix $\mathbf{\hat{F}}$, model weights $\mathbf{W_0}$ and $\mathbf{W_1}$ to two servers, and generate randomnesses.
- OblivAgg: Each party evaluates the matrix multiplications over the shared matrices, i.e., $[\![\mathbf{\hat{A}}]\!]_b \times [\![\mathbf{\hat{F}}]\!]_b \times [\![\mathbf{W_0}]\!]_b$ for the first layer, and $[\![\mathbf{\hat{A}}]\!]_b \times [\![\mathrm{ReLU}(\mathbf{\hat{A}\hat{F}W_0})]\!]_b \times [\![\mathbf{W_1}]\!]_b$ in the second layer, without revealing them.
- OblivAct: Each party uses the shares from the OblivAgg to perform oblivious ReLU, Softmax and Argmax.

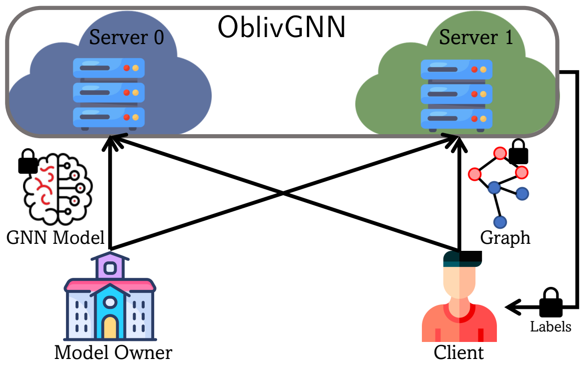


Figure 1: System Architecture of OblivGNN.

Oblivious *Inductive* works in a similar way to Oblivious *Transductive* yet additionally supports oblivious update OblivUpdate, therefore we only describe the above here. Oblivious *Inductive*:

- OblivUpdate: Upon receiving the new node information $[\![\hat{\mathbf{A}}^*]\!]_b$ and $[\![\hat{\mathbf{F}}^*]\!]_b$, update the shares on each party to include the newly added nodes and updated node features.

3.2 Threat Model

We consider our system to be a secure cloud-based GNN inference service that protects both data privacy and data access pattern (*obliviousness*). Similar to the prior outsourced PPML schemes [16, 39, 46, 52], two servers execute the proposed protocol collaboratively and compute the inference result in secret shares. During the process, each server only observes computations over random numbers and cannot learn any model information, inputs or inference results [41].

We now describe the threat model and security scope of OblivGNN. OblivGNN operates in a semi-honest two-party setting following the works [28, 38, 44, 46, 51, 52], guaranteeing its users' data privacy if the adversary controls only one server. In particular, we assume that several non-colluding adversaries compromise one server at most. These adversaries do not deviate from the protocol but will try to infer input/model information based on their processed data/pattern. Prior works [16, 46, 52] indicate that two-party collusion can be prevented by making it costly or infeasible by placing two servers in different clouds or countries/jurisdictions. The laws/conflict of interest stop collusion [29] effectively. Furthermore, even in a semi-honest non-colluding adversary model, the compromised server can still observe the access pattern when accessing the graph (i.e., updating the graph matrices and the client inquiry). As stated in [21], the graph access pattern can be exploited even in an encrypted graph.

3.3 Security Guarantees

The GNN model owner outsources the GNN model and graph data to two servers for GNN inference service. During the process, the adversary can obtain the entire view of the OblivGNN protocol execution on the compromised server and analyse it. Nonetheless, all private information from the GNN model owners, including the adjacency matrix, feature matrix and model weights will not be revealed during inference.

In OblivGNN, the access pattern is protected (therefore named oblivious) from potential semi-honest adversarial attacks when accessing and updating the original graph. The access pattern protection also ensures the security of clients' interactions, e.g., the clients' inquiries when accessing and retrieving the inference outcomes.

4 PROTOCOL

We start by describing the strawman approach in Section 4.1. We then describe our OblivGNN in Section 4.2. We show and discuss our protocols in Algorithm 3, 5 and the complete protocols are shown in Algorithm 7 and Algorithm 8.

4.1 Strawman Approach

A naive solution to enable oblivious GNN inference can be achieved with the arithmetic secret sharing scheme. In particular, we leverage the following steps to obliviously evaluate Equation 1 from Section 2.1 to enable the *transductive* inference (Oblivious *Transductive*):

- OfflineGen($\mathbb{Z}_{2^\ell}, \hat{\mathbf{A}}, \hat{\mathbf{F}}, \mathbf{W_0}, \mathbf{W_1}$): The model owner shares the adjacency matrix $\hat{\mathbf{A}}$, feature matrix $\hat{\mathbf{F}}$, model weights $\mathbf{W_0}$ and $\mathbf{W_1}$ over the given ring $\mathbb{Z}_{2^\ell}$. Then, the model owner sends the shares of $[\![\hat{\mathbf{A}}]\!]_b$, $[\![\hat{\mathbf{F}}]\!]_b$, $[\![\mathbf{W_0}]\!]_b$, $[\![\mathbf{W_1}]\!]_b$ to Server b.
- OblivAgg($\mathbb{Z}_{2^\ell}, [\![\hat{\mathbf{A}}]\!]_b, [\![\hat{\mathbf{F}}]\!]_b, [\![\mathbf{W_0}]\!]_b, [\![\mathbf{W_1}]\!]_b$): In the background, the model owner continuously generates beaver's multiplication triples, shares them over the given ring $\mathbb{Z}_{2^\ell}$, and sends them to Server b. With triples, two servers jointly compute the matrix multiplication with ASS additions [3] and Beaver's Triples [5] based multiplications.
- OblivAct($\mathbb{Z}_{2^\ell}$): The strawman approach leverages the approximation techniques [32] to approximate the Rectified Linear Unit (ReLU) and softmax to a polynomial. Then, two servers evaluate the activation functions with ASS and beaver's triples over $\mathbb{Z}_{2^\ell}$.

We can further enable oblivious *inductive* inference (Oblivious *Inductive*):

- OblivUpdate($\mathbb{Z}_{2^\ell}, [\![\hat{\mathbf{A}}]\!]_b, [\![\hat{\mathbf{F}}]\!]_b$): When the client or the model owner performs updates on the graph by inserting nodes and updating features, they directly modify the adjacency matrix $\hat{\mathbf{A}}$ and feature matrix $\hat{\mathbf{F}}$. Then, those matrices are shared over $\mathbb{Z}_{2^\ell}$ and sent to the servers.

The above design is fully oblivious. Because all the above-mentioned operations (i.e., matrix multiplications, polynomial

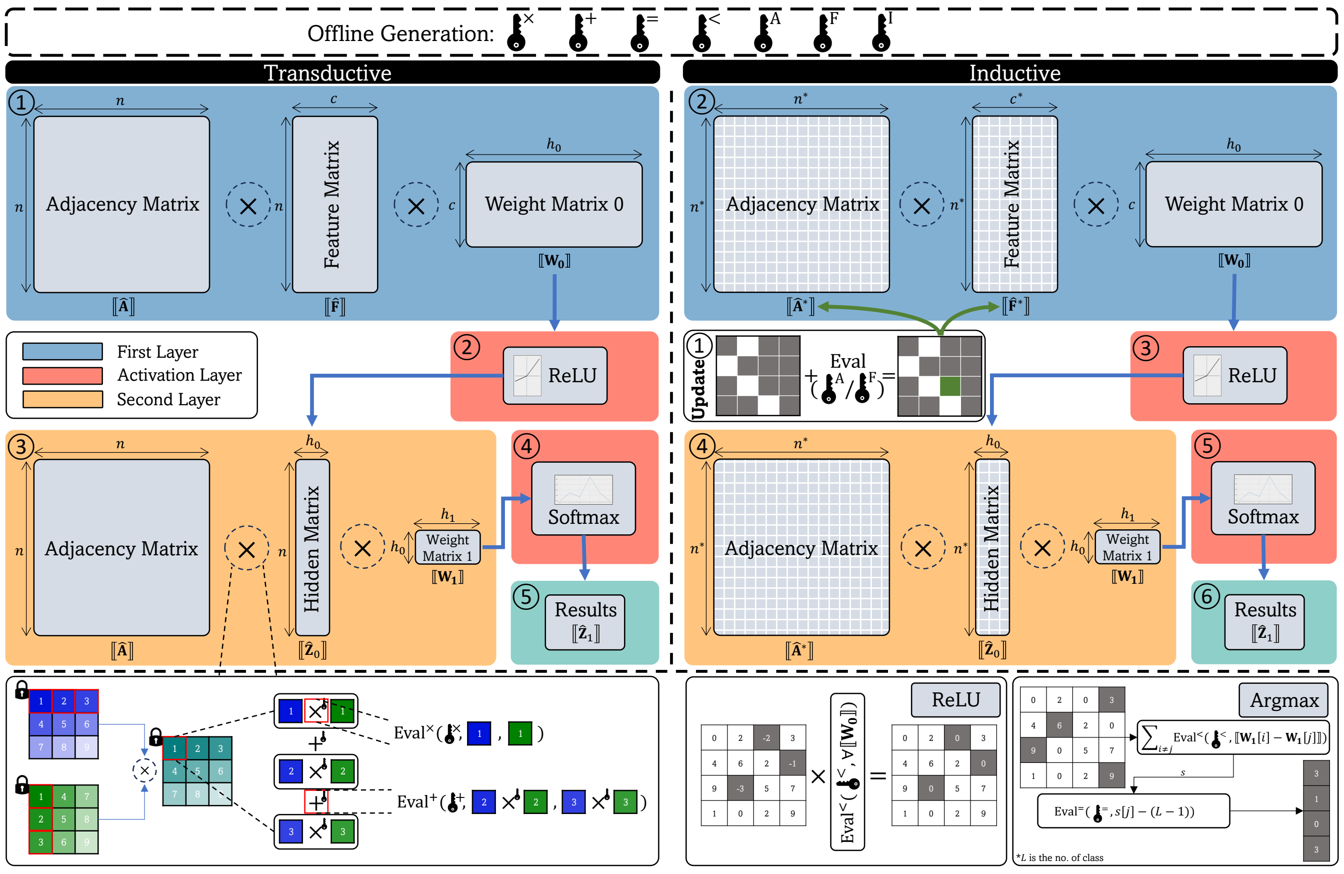


Figure 2: Protocol Overview of OblivGNN

evaluations, and graph updates) always execute the same process regardless of the input in the view of two servers. However, we noticed two problems: 1) the runtime/communication costs were significant due to adopting Beaver's Triple in aggregation. 2) to update obliviously, the communication cost of re-uploading the updated graph is significant. To address the above issues, we propose the OblivGNN approach.

4.2 OblivGNN Approach

To address the above two problems: we use Arithmetic FSS to save the significant runtime and communication costs due to adopting Beaver's Triple in aggregation (1). We adopt DPF to update obliviously with reduced communication cost (2).

Our focus has been primarily on executing secure GNN inference on the static graph (i.e., transductive setting). However, it is important to note that nodes and features may require updates in practice. We will discuss how OblivGNN approach facilitates the oblivious graph updates, limits the graph information leakage and substantially improves efficiency. Details are elaborated in the subsequent sections, and an illustration is provided in Figure 2.

Overview. The OblivGNN's oblivious inference operates as follows. In the *offline phase*, the model owner generates randomnesses to mask the normalised adjacency matrix, normalised feature matrix, and model weights. The randomnesses are also used to form the FSS keys pool, for all multiplication and addition pairs in oblivious matrix multiplication between the above matrices. The graph matrices are then split into secret shares using ASS from Section 2.3. In the inductive setting, the client or the model owner prepares the DPF keys for graph update, according to the updated subgraph's adjacency and feature matrix on the client side.

In the *online phase*, the model owner secretly shares the masked normalised adjacency matrix, feature matrix and the model weight matrices to both servers. Each party then performs oblivious aggregation by revealing the masked matrices and securely aggregating the neighbouring features for each node. The aggregated results then go through the first layer activation function, oblivious ReLU. As shown in Figure 2, the aggregation process will repeat with the oblivious softmax activation function for the second layer.

After two layers, the inference results are stored as secret shares in both parties. The client can then enquire the result(s) with specific node(s) ID. We will elaborate on each other the above protocol in the following sections.

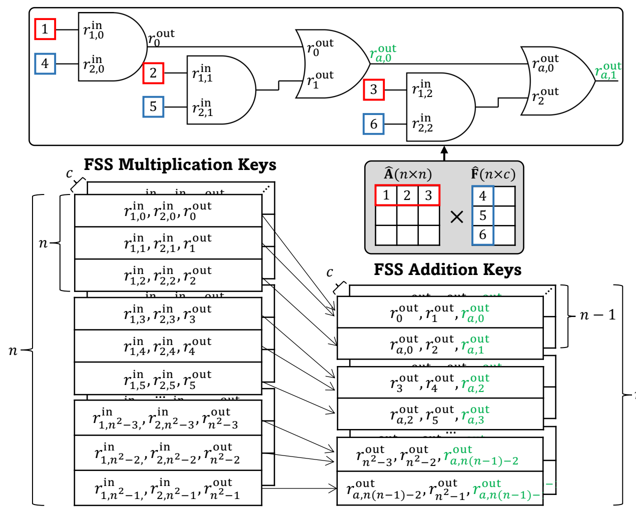

FSS Multiplication Keys
FSS Addition Keys
$\hat{\mathbf{A}}(n\times n)$
$\hat{\mathbf{F}}(n\times c)$
$r^{in}_{1,0}, r^{in}_{2,0}, r^{out}_0$
$r^{in}_{1,1}, r^{in}_{2,1}, r^{out}_1$
$r^{in}_{1,2}, r^{in}_{2,2}, r^{out}_2$
$r^{out}_0, r^{out}_1, r^{out}_{a,0}$
$r^{out}_{a,0}, r^{out}_2, r^{out}_{a,1}$
n
c
$n-1$

Figure 3: FSS Multiplicative and addition gates Chaining between Adjacency and Feature Matrix in OblivGNN

4.2.1 Oblivious Aggregation

The oblivious aggregation entitles the collection of information from nearby nodes. To protect the access pattern in aggregation, we employ matrix multiplication which linear accesses all nodes to naturally hide the access pattern.

To build secure matrix multiplication algorithms for oblivious aggregation, we leverage the FSS multiplication and addition gate, inspired by the works of Boyle et al. [7, 13].

FSS Circuit Chaining. To enable secure matrix multiplication, we should enable the secure dot product operation between each row and column of the matrix. While this is easy to be achieved with ASS, we realise that it is not straightforward to realise with FSS. This is because the FSS-based arithmetic operations consider a specific operation as a gate, which has randomnesses associated with the input and output wires of the gate (see Section 2.2). As a result, when calculating multiple arithmetic operations with FSS, it generates a circuit structure, where the randomness associated with the output wire of a gate, will be the randomness used by the input wire of the next gate.

In Figure 3, we provide an illustrative example for the dot product between one row and one column of the matrix. As shown in the figure, the dot product process will be converted to a circuit with FSS multiplication and addition gates. For each gate, three randomnesses $(r_1^{in}, r_2^{in}, r^{out})$ are selected and associated with the input/output wires of the gate, respectively (see Algorithm 1 and 2). For multiplication randomnesses, they can be selected independently since the multiplications in dot product are independent. However, the randomnesses for addition gates are correlated because they sum up the multiplication results to get the final result. More specifically, the randomnesses associated with the input wires of the addition gate are the randomnesses associated with the output wires of the previous multiplication gate or addition gate.

Without loss of generality, in the given example, when computing the dot product of the first row of $\hat{\mathbf{A}}$ and the first column of $\hat{\mathbf{F}}$, the multiplication of $\hat{\mathbf{A}}[0][i] \times \hat{\mathbf{F}}[i][0]$ for $i \in [0,2]$ are done with three independent sets of the randomnesses $(r_{1,i}^{in}, r_{2,i}^{in}, r_i^{out})$. When summing the multiplication outputs for the final dot product result, the circuit however requires to use the output wire randomnesses of prior gates that cascade to the addition gate. This covers two scenarios: The first one is that the addition gate is adding two outputs from multiplication gates ($\hat{\mathbf{A}}[0][0] \times \hat{\mathbf{F}}[0][0] + \hat{\mathbf{A}}[0][1] \times \hat{\mathbf{F}}[1][0]$ in our example), so it leverages r_0^{out} and r_1^{out} as the input wire randomnesses.

Following the above procedure, one can obtain a list of chained FSS keys that enable dot product operations for fixed-size vectors (the rows and columns of matrices) and thus can be readily used to support matrix multiplication.

The total number of arithmetic FSS keys depends on the size of the dataset. In Figure 2, there will be $n \times (nc + ch_0 + nh_0 + h_0h_1)(\mathrm{O}(n^2c))$ number of keys generated for FSS multiplication gate. And $(n-1) \times (nc + ch_0 + nh_0 + h_0h_1)(\mathrm{O}(n^2c))$ number of keys generated for FSS addition gate.

FSS Key Pool Generation. From the above, we observe that FSS keys are input-independent and solely rely on uniform randomnesses. This observation allows a similar pooling technique for Beaver's Triple [5], i.e., given the matrix sizes, the model owner can generate a large number of FSS keys following the above chaining process. Moreover, the model owner can distribute them to two servers in an offline phase for efficient matrix multiplications, while performing on-the-fly key pooling like in [46] to meet the query demands.

Aggregation functions. With the above FSS keys and pooling mechanism, two servers can conduct oblivious aggregation with matrix multiplication. In the aggregation function, two servers will have shared matrices $[\![\mathbf{X}]\!]_b$ and $[\![\mathbf{Y}]\!]_b$ and want to multiply them together. To bootstrap the computation, the model owner sends the FSS keys for one matrix multiplication to two servers. Also, he/she extracts the input wire randomnesses of multiplication gates to form two mask matrices $\mathbf{R_1}$ and $\mathbf{R_2}$ with the same sizes as $[\![\mathbf{X}]\!]_b$ and $[\![\mathbf{Y}]\!]_b$. These two mask matrices will be shared as additive secret share matrices $[\![\mathbf{R_1}]\!]_b$ and $[\![\mathbf{R_2}]\!]_b$ and sent to two servers to mask the corresponding elements in $\mathbf{X}$ and $\mathbf{Y}$, respectively.

Upon receiving the shared mask matrices and the FSS keys, two server first compute $[\![\mathbf{X}']\!]_b = [\![\mathbf{X}]\!]_b + [\![\mathbf{R_1}]\!]_b$ and $[\![\mathbf{Y}']\!]_b = [\![\mathbf{Y}]\!]_b + [\![\mathbf{R_2}]\!]_b$. Then, they exchange their local shares of $[\![\mathbf{X}']\!]_b$ and $[\![\mathbf{Y}']\!]_b$ to recover the masked $\mathbf{X}'$ and $\mathbf{Y}'$. With masked matrices, two servers can directly compute FSS-based multiplications and additions without interactions, and get $\mathbf{Z}' = \mathbf{X} \times \mathbf{Y} + \mathbf{R_{out}}$ at the end of the execution. In Algorithm 3, we provide the oblivious aggregation process in details. Note that the oblivious aggregation can be used to support the linear layer aggregation functions in each layer of GNN.

Algorithm 3: Oblivious Aggregation OblivAgg

```
Input: Shared matrices $[\![\mathbf{X}]\!]_b$ ($m \times n$ elements) and
       $[\![\mathbf{Y}]\!]_b$ ($n \times c$ elements)
Output: Masked aggregated results $\mathbf{Z}'$
Model Owner (FSS Key Pooling):
1) Generate $\mathbf{R_1}$ with $m \times n$ randomnesses, $\mathbf{R_2}$ with
   $n \times c$ randomness, and $\mathbf{R_{out}}$ with $m \times c$ randomness.
2) For $i \in [0, m-1]$,
3)   For $j \in [0, c-1]$,
4)     Initialise temp $r^{\times}_{out}$ and $r^{+}_{out}$ arrays.
5)     KeyGen$^{+}(1^{\lambda}, r^{\times}_{out}[0], r^{\times}_{out}[1], r^{+}_{out}[0])$.
6)     For $k \in [0, n-1]$,
7)       KeyGen$^{\times}(1^{\lambda}, \mathbf{R_1}[i][k], \mathbf{R_2}[k][j], r^{\times}_{out}[j])$.
8)       KeyGen$^{+}(1^{\lambda}, r^{+}_{out}[k-1], r^{\times}_{out}[k+1], r^{+}_{out}[k])$, if
   $k \in [1, n-3]$.
9)     KeyGen$^{+}(1^{\lambda}, r^{+}_{out}[n-3], r^{\times}_{out}[n-1], \mathbf{R_{out}}[i][j])$.
10) Share $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_{out}}$ as $[\![\mathbf{R_1}]\!]_b$, $[\![\mathbf{R_2}]\!]_b$ and
   $[\![\mathbf{R_{out}}]\!]_b$.
Send $[\![\mathbf{R_1}]\!]_b$, $[\![\mathbf{R_2}]\!]_b$ and $[\![\mathbf{R_{out}}]\!]_b$ and FSS keys to $P_b$.
Party $P_b$:
1) Mask the shares of $[\![\mathbf{X}]\!]_b$ as $[\![\mathbf{X}']\!]_b = [\![\mathbf{X}]\!]_b + [\![\mathbf{R_1}]\!]_b$
   and $[\![\mathbf{Y}]\!]_b$ as $[\![\mathbf{Y}']\!]_b = [\![\mathbf{Y}]\!]_b + [\![\mathbf{R_2}]\!]_b$.
2) Exchange shares with $P_{1-b}$ to recover $\mathbf{X}'$ and $\mathbf{Y}'$.
3) For $i \in [0, m-1]$,
4)   For $j \in [0, c-1]$,
5)     Set vector $[\![\mathbf{Temp}]\!]_b$.
6)     For $k \in [0, n-1]$,
7)       $[\![\mathbf{Temp}[k]]\!]_b$= FSS.Mul($\mathbf{X}'[i][k], \mathbf{Y}'[k][j]$).
8)     Exchange shares with $P_{1-b}$ to recover Temp.
9)     For $k \in [0, n-1]$,
10)      $[\![\mathbf{Z}'[i][j]]\!]_b$=FSS.Add($\mathbf{Z}'[i][j], \mathbf{Temp}[k]$).
11)      Exchange shares with $P_{1-b}$ to recover $\mathbf{Z}'[i][j]$.
return $\mathbf{Z}' = \mathbf{Z} + R_{out}$.
```

4.2.2 Oblivious Graph Update

DPF Update Key Generation. During the update phase, some entries of the adjacent and feature matrices will be updated to new values. We note that if we directly update the matrix, it reveals the access pattern on the graph matrices and has the potential to disclose the underlying graph structure.

OblivGNN employs the DPF to hide the access pattern in graph updates. In particular, two sets of the key will be generated to implement two modifications, i.e., the connections relations change on the adjacency matrix and the feature updates on the feature matrix. Algorithm 4 demonstrates how to use the above two information with their location information. The algorithm extracts the updated node location (v_{i^*}) from the updated subgraphs $\mathbf{A}^*$ and $\mathbf{F}^*$. Then, it generates n DPF keys to update the v_{i^*} location of each row on the adjacent matrix, and c DPF keys to update the v_{i^*} location of each column on the feature matrix. Those keys will be sent to two servers to let them update their local shares of the adjacent matrix and the feature matrix, respectively.

Algorithm 4: DPF Update Key Generation UpdateKeyGen

```
Input: Updated subgraph $\mathbf{A}^*$ and $\mathbf{F}^*$
Output: $k^{\mathrm{A}}_b$, $k^{\mathrm{F}}_b$
1) Initialise $k^{\mathrm{A}}_b, k^{\mathrm{F}}_b \leftarrow \{\}$.
2) Extract the target node $v_{i^*}$ for the update.
3) For $i \in [0, n-1]$, ;                 // loop column
4)   $k^{\mathrm{A}}_b[i] \leftarrow$KeyGen($v_{i^*}, \mathbf{A}^*[v_{i^*}][i] - \mathbf{A}[v_{i^*}][i]$).
5) For $i \in [0, c-1]$, ;                 // loop column
6)   $k^{\mathrm{F}}_b[i] \leftarrow$KeyGen($v_{i^*}, \mathbf{F}^*[v_{i^*}][i] - \mathbf{F}[v_{i^*}][i]$).
```

Algorithm 5: Oblivious Graph Update OblivUpdate

```
Input: New node adjacency and feature information $\hat{\mathbf{A}}$
       and $\hat{\mathbf{F}}$, Graph Update DPF keys $k^{\mathrm{A}}_b$ and $k^{\mathrm{F}}_b$
Output: Shares of updated adjacency matrix $[\![\hat{\mathbf{A}}]\!]_b$ and
        updated feature matrix $[\![\hat{\mathbf{F}}]\!]_b$
Party $P_b$:
1) For $i \in [0, n-1]$, ;                 // loop column
2)   For $j \in [0, n-1]$, ;                // loop row
3)     $[\![\hat{\mathbf{A}}[j][i]]\!]_b = [\![\hat{\mathbf{A}}[j][i]]\!]_b$+Eval($k^{\mathrm{A}}_b[i], j$).
4) For $i \in [0, c-1]$, ;                 // loop column
5)   For $j \in [0, n-1]$, ;                // loop row
6)     $[\![\hat{\mathbf{F}}[j][i]]\!]_b = [\![\hat{\mathbf{F}}[j][i]]\!]_b$+Eval($k^{\mathrm{F}}_b[i], j$).
```

Graph Update. With the DPF keys, two servers can easily update the graph with Algorithm 5. Specifically, to update the adjacency matrix, each server evaluates the given n DPF keys with inputs from $[0, n-1]$, and set the corresponding cell in each row to the evaluate output. By doing this, the connection of the updated point is changed while the access pattern of this update is hidden as the update scans the entire adjacency matrix. Similarly, each server can update the feature matrix by evaluating the given c DPF keys with inputs from $[0, n-1]$, and accumulates the evaluate output to corresponding cell in each column. Since the keys for the feature matrix update are generated by leverage the difference between new feature value and old one ($\mathbf{F}^*[v_{i^*}][i] - \mathbf{F}[v_{i^*}][i]$). The update process will replace the old feature with the new one, without revealing which node is updated.

Node Insertion. When inserting a new node, the model owner/client can directly shares the new node into two servers. Then, two servers can directly expand the adjacency (from $n \times n$ to $(n+1) \times (n+1)$) and feature (from $n \times c$ to $(n+1) \times c$) matrices and attach the shares to the expanded row/column. This process will not affect OblivGNN's security because the matrix size is not protected in OblivGNN.

However, since the matrix is expanded to fit the new node, the FSS pooling requires to be expanded accordingly to support the matrix multiplication with new matrix size. Fortunately, as stated in Section 4.2.1, the matrix multiplication can be seem as a series of independent dot products, thus we do not need to re- generate the FSS pooling to meet the computation requirement. Instead, we design a FSS key pooling

```
Algorithm 6: FSS Pooling Expansion PoolExpand
  Input: Expanded $\mathbf{R_1}$ with $(m+1)\times(n+1)$
         randomnesses, $\mathbf{R_2}$ with $(n+1)\times c$ randomness,
         and $\mathbf{R'_{out}}$ with $(m+1)\times c$ randomness;
         Original $\mathbf{R_{out}}$ with $m\times c$ randomness.
  Model Owner:
  1) For $i\in[0,m-1]$,
  2)   For $j\in[0,c-1]$,
  3)     Initialise temp $r^{\times}_{out}$ values.
  4)     $\mathsf{KeyGen}^{\times}(1^{\lambda},\mathbf{R_1}[i][n],\mathbf{R_2}[n][j],r^{\times}_{out})$.
  5)     $\mathsf{KeyGen}^{+}(1^{\lambda},\mathbf{R_{out}}[i][j],r^{\times}_{out},\mathbf{R'_{out}}[i][j])$.
  6) For $i\in[0,c-1]$,
  7)   Initialise temp $r^{\times}_{out}$ and $r^{+}_{out}$ arrays.
  8)   $\mathsf{KeyGen}^{+}(1^{\lambda},r^{\times}_{out}[0],r^{\times}_{out}[1],r^{+}_{out}[0])$.
  9)   For $j\in[0,n]$,
  10)    $\mathsf{KeyGen}^{\times}(1^{\lambda},\mathbf{R_1}[m][j],\mathbf{R_2}[j][i],r^{\times}_{out}[j])$.
  11)    $\mathsf{KeyGen}^{+}(1^{\lambda},r^{+}_{out}[j-1],r^{\times}_{out}[j+1],r^{+}_{out}[j])$, if
         $j\in[1,n-2]$.
  12)  $\mathsf{KeyGen}^{+}(1^{\lambda},r^{+}_{out}[n-2],r^{\times}_{out}[n],\mathbf{R'_{out}}[m][i])$.
  13) Share $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R'_{out}}$ as $[\![\mathbf{R_1}]\!]_b$, $[\![\mathbf{R_2}]\!]_b$ and
      $[\![\mathbf{R'_{out}}]\!]_b$.
  Send $[\![\mathbf{R_1}]\!]_b$, $[\![\mathbf{R_2}]\!]_b$ and $[\![\mathbf{R'_{out}}]\!]_b$ and FSS keys to $P_b$.
```

expansion algorithm (see Algorithm 6) to assist this process.

For the existing nodes (i.e., nodes labeled with 0 to $m-1$), the algorithm calculates the FSS key for the multiplications of newly added nodes and features, and then create a FSS key for additions that sums the previous output at $\mathbf{Z}'[i][j]$ (masked by original $\mathbf{R_{out}}[i][j]$) and the new multiplication result into the same place that masked by expanded output mask matrix $\mathbf{R'_{out}}[i][j]$ (line 1-5). For the new node, the algorithm re-runs the pooling algorithm for it (line 6-12), which generates the FSS keys for value in $\mathbf{Z}[m][i], i\in[0,c-1]$.

Following the graph update, the servers initiate a single secured forward propagation, as outlined in Equation 1, using the updated $\hat{\mathbf{A}}$ and $\hat{\mathbf{F}}$. This process culminates in the generation of inference results for all nodes, including the newly inserted node, and stored within the result matrix $\mathbf{Z}$.

Remark. We can hide the graph size by padding the adjacency and feature matrix to 2^{ℓ} and use DPF to obliviously write the sub-graph information to the padded matrices. Moreover, a prominent feature of the update phase is that we use the DPF key to achieve a sub-linear complexity for both the communication and computation costs. In specific, the DPF key can compress a vector of n elements to $\lceil\log_2 n\rceil$ size and allow the vector to be recovered with $\lceil\log_2 n\rceil$ evaluations. In the experiment, we demonstrate that the update cost increases in the logarithm scale while the matrix size increases linearly.

4.2.3 Oblivious Activation Functions

Taking inspiration from Théo et al. [52], we employ the DPF equality test and comparison protocol to construct activation functions (i.e., ReLU, Softmax and Argmax). We provide a concise description of the DPF equality test (DPF.Equa) and the DPF comparison protocol (DPF.Comp) in Section 2.2. Subsequently, we demonstrate how these protocols can be employed in constructing activation functions.

We note that DPF.Comp has an opposite relationship (i.e., outputs 1 when input ≤ 0) to ReLU, therefore, we describe the oblivious bit flip protocol OblivBitFlip as follows.

1) $[\![\mathfrak{b}]\!]_0=\mathsf{Eval}(k_0,[\![z]\!]_0)$, $[\![\mathfrak{b}]\!]_1=\mathsf{Eval}(k_1,[\![z]\!]_1)$
2) $[\![\mathfrak{b}']\!]_0=0-[\![\mathfrak{b}]\!]_0$, $[\![\mathfrak{b}']\!]_1=1-[\![\mathfrak{b}]\!]_1$
3) $\mathfrak{b}'=[\![\mathfrak{b}']\!]_0+[\![\mathfrak{b}']\!]_1=1-([\![\mathfrak{b}]\!]_0+[\![\mathfrak{b}]\!]_1)=1-\mathfrak{b}$

Oblivious ReLU. We now use the aforementioned DPF.Comp to build the oblivious ReLU activation function. We describe the formal OblivReLU protocol on a $n\times c$ masked matrix $[\![\mathbf{X}]\!]_b$ as follows.

1) Each P_b exchanges its local share and recovers $\mathbf{X}$.
2) Each P_b receives $n\times c$ DPF.Comp keys.
3) Each P_b invoke received DPF keys to run DPF.Comp against each element in the aggregated matrix.
4) Each P_b invoke OblivBitFlip on DPF output $[\![\mathfrak{b}'[i][j]]\!]_b, i\in[0,n-1], j\in[0,c-1]$.
5) Each P_b outputs shared matrix that $[\![\mathbf{X}'[i][j]]\!]_b\leftarrow [\![\mathfrak{b}'[i][j]]\!]_b\times X[i][j]$

Oblivious Softmax. We develop the oblivious softmax function OblivSoftmax, inspired from the research by Mohassel et al. [46] and Keller et al. [31]. By removing the intricate power calculations in the MPC domain and leveraging the previously established low-interactive OblivReLU, we substitute the softmax function with the following function:

$$[\![\hat{\mathbf{z}}[i]]\!]_b := \begin{cases} \frac{\mathsf{OblivReLU}([\![z]\!]_b)}{\sum_i \mathsf{OblivReLU}([\![z[i]]\!]_b)}, & \text{if } \sum_i \mathsf{OblivReLU}([\![z[i]]\!]_b)>0 \\ 1/L, & \text{otherwise} \end{cases} \tag{2}$$

To ensure the obliviousness when evaluating the above equation, we rely on the implementation in MP-SPDZ [30] to obliviously compute the division via garbled circuit. Moreover, the branching conditions are replaced with oblivious if, i.e., the equations for all conditions will be evaluated, but the condition will be used as an oblivious selector $((a\&condition)|(b\&\neg condition))$ to obtain the final result.

Oblivious Argmax. This function lets each party P_b find the location of the largest element in array z with L element.

We use DPF.Equa and DPF.Comp to formally construct the Oblivious Argmax (OblivArgmax) protocol below.

1) Each P_b: $[\![s[j]]\!]_b\leftarrow\sum_{i\neq j}\mathsf{DPF.Comp}([\![z[i]-z[j]]\!]_b)$
2) Each P_b: $[\![z'[j]]\!]_b\leftarrow\mathsf{DPF.Equa}([\![s[j]-(L-1)]\!]_b)$

The protocol checks whether $z[i]\leq z[j]$ for each $i\neq j$ and sums the check results into $s[j]$. Then, we can use a DPF.Equa key to check if $s[j]=(L-1)$, which means DPF.Comp for $z[i]\leq z[j]$ returns 1 for all $L-1$ elements, and thus be the largest element. The OblivArgmax protocol returns an array z', which only has shared '1' at the position with the largest element and '0' in other places.

Algorithm 7: Full Protocol of *transductive* OblivGNN

```
Model Owner:
1) Model Owner secret-shares the matrices
   $\hat{\mathbf{A}}, \hat{\mathbf{F}}, \mathbf{W_0}, \mathbf{W_1}$ to party $P_b$.
2) Model Owner create FSS pools for:
   OblivAgg($[\![\hat{\mathbf{A}}]\!]_b$, $[\![\hat{\mathbf{F}}]\!]_b$),
   OblivAgg($[\![\hat{\mathbf{A}}\hat{\mathbf{F}}]\!]_b$,$[\![\mathbf{W_0}]\!]_b$),
   OblivAgg($[\![\hat{\mathbf{A}}]\!]_b$,$[\![\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})]\!]_b$),
   OblivAgg($[\![\hat{\mathbf{A}}\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})]\!]_b$,$[\![\mathbf{W_1}]\!]_b$).
Party $P_b$:
1) 1st Layer: $P_b$ runs:
   $[\![\hat{\mathbf{A}}\hat{\mathbf{F}}]\!]_b \leftarrow$OblivAgg($[\![\hat{\mathbf{A}}]\!]_b$, $[\![\hat{\mathbf{F}}]\!]_b$),
   $[\![\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0}]\!]_b \leftarrow$OblivAgg($[\![\hat{\mathbf{A}}\hat{\mathbf{F}}]\!]_b$,$[\![\mathbf{W_0}]\!]_b$),
   $[\![\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})]\!]_b \leftarrow$OblivReLU($[\![\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0}]\!]_b$)
2) 2nd Layer: $P_b$ runs:
   $[\![\hat{\mathbf{A}}\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})]\!]_b \leftarrow$
         OblivAgg($[\![\hat{\mathbf{A}}]\!]_b$,$[\![\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})]\!]_b$),
   $[\![\hat{\mathbf{A}}\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})\mathbf{W_1}]\!]_b \leftarrow$
         OblivAgg($[\![\hat{\mathbf{A}}\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})]\!]_b$,$[\![\mathbf{W_1}]\!]_b$).
3) For $i \in [0, n-1]$,
    $[\![\mathbf{Z}[\mathbf{i}]]\!]_b \leftarrow$
          OblivSoftmax($[\![\hat{\mathbf{A}}\mathrm{ReLU}(\hat{\mathbf{A}}\hat{\mathbf{F}}\mathbf{W_0})\mathbf{W_1}[\mathbf{i}]]\!]_b$).
    $[\![\mathbf{Z}'[\mathbf{i}]]\!]_b \leftarrow$OblivArgmax($[\![\mathbf{Z}[\mathbf{i}]]\!]_b$).
4) Recover masked $\mathbf{Z}'[\mathbf{i}]$.
Client:
1) For a targeted node $v_{i^*}$, generate the query DPF key
   $k_b^{\mathrm{Q}} \leftarrow$KeyGen($v_{i^*}, 1$) and send to $P_b$.
Party $P_b$:
1) For $i \in [0, h_1 - 1]$,
2)    For $j \in [0, n-1]$,
3)       $[\![\hat{\mathbf{r}}[j]]\!]_b$ += Eval($k_b^{\mathrm{Q}}, i$) $\times Z'[j][i]$.
4) Return $[\![\hat{\mathbf{r}}]\!]_b$ to Client
Client:
1) Recover $\hat{\mathbf{r}}$, de-mask it and get the inference result.
```

Remark. The DPF keys used in the activation function can also be pooled in advance. Particularly, both the OblivReLU and OblivArgmax require the DPF keys to check whether the input is < 0, while OblivArgmax needs additional keys to check whether the input is $= L-1$. Since the above-mentioned values are input-independent, we can create a DPF pool to keep those keys the same as in OblivAgg.

4.2.4 Putting It Together

Having examined all essential components of the proposed Oblivious Graph Neural Network (OblivGNN), we are now prepared to integrate these elements into cohesive protocols. The complete protocols are presented for both *transductive* and *inductive* settings in Algorithm 7 and 8, respectively.

At the end of the protocol execution, two servers obtain a masked $n \times h_1$ matrix (the full graph classification results)

Algorithm 8: Full Protocol of *inductive* OblivGNN

```
1) Model Owner/Client sends newly added node as
   additive shares to $P_b$.
2) Model Owner/Client runs PoolExpand to update
   FSS key pool
3) Model Owner/Client runs UpdateKeyGen($\mathbf{A}^*, \mathbf{F}^*$)
   and send $k_b^{\mathrm{A}}$, $k_b^{\mathrm{F}}$ to $P_b$.
4) $P_b$ attaches new node into original $\mathbf{A}$ and $\mathbf{F}$.
5) $P_b$ runs OblivUpdate($k_b^{\mathrm{A}}, k_b^{\mathrm{F}}$) to update the graph
6) Run the transductive protocol in Algorithm 7 to get
   the inference result.
```

from the OblivArgmax. The client uses a DPF key generated with the targeted node ID to retrieve the inference result.

Security. It is intuitive to see that OblivGNN achieves the required security level. First, the aggregation process is a secure version of matrix multiplication, which will access all elements in the matrix, and thus oblivious. Moreover, non-linear functions, including ReLU, Softmax, Argmax and graph update, are implemented with DPF. This ensures that even if the same input is given for those functions, the DPF scheme will randomise it, so the adversary who controls one server cannot know the actual input. On the other hand, those DPF-based functions will perform a linear scan over the entire input domain (e.g., all elements in the feature matrix for node update), hiding the access pattern against the adversary on one server. In Appendix A, we formally define the real/ideal game for each protocol and provide a detailed security analysis.

Efficiency. From the Table 2, it is clear that the OblivGNN achieved a remarkable imporvement in efficiency. First, by adopting the FSS multiplication and addition gate, these cryptographic techniques have effectively addressed the challenges of communication overhead and computational efficiency while providing inherent protection to access pattern. The adaptation of arithmetic FSS gates also limited the graph information (e.g. maximum neighbouring n) leakage compared to [14, 50, 59]. Therefore, making OblivGNN feasible and efficient for real-world applications involving large-scale graphs and sensitive data.

5 Implementation and Evaluation

In this section, we have implemented OblivGNN, based on the proposed protocols. We now present the experimental settings and results in this section.

5.1 Setup

Platform. Our system is developed in both Python and C++. We have deployed the OblivGNN using the MP-SPDZ secure framework [30]. The computational environment is a server with 3.70GHz Intel(R) Xeon(R) E-2288G CPU, 64GB RAM and 128GB external storage running Ubuntu 20.04.5 LTS.

	Communication Cost	Computational Cost
Baseline	$16 \cdot O(n^2 \cdot m)$	$(2 \times T_{\text{Sub}} + 4 \times T_{\text{Mul}} + 3 \times T_{\text{Add}}) \cdot O(n^2 \cdot m)$
OblivGNN	$8 \cdot O(n^2 \cdot m)$	$(2 \times T_{\text{Sub}} + 4 \times T_{\text{Mul}} + 3 \times T_{\text{Add}}) \cdot O(n^2 \cdot m)$

Table 2: Theoretical Complexity: communication and computational cost between baseline and OblivGNN in online phase. v is a full-domain evaluation parameter (see Section 5.3). T_{Mul} is the time cost for single multiplication, T_{Add} is the time cost for single addition, T_{Sub} is the time cost for single subtraction.

Dataset	Nodes	Feature	Edge	Classes
Cora	2708	1433	5429	7
Citeseer	3327	3703	4732	6
Pubmed	19717	500	44338	3

Table 3: Dataset Statistics

The experiment was conducted in two phases: the offline phase and the online phase. Each phase is conducted within two different settings, i.e., transductive and inductive settings.
Dataset. The OblivGNN system is designed to be compatible with a variety of graph datasets, provided they adhere to the correct data structure. For our experiments, we selected three widely recognized and trusted graph datasets: Cora, Citeseer and Pubmed. A summary of the statistical characteristics of these graph datasets is presented in Table 3 below.
Baseline. In our experiments, we employed our strawman approach, using ASS for aggregation and approximation for activation, as the baseline approach.

Our evaluation results are presented in the following sections. We will begin by providing the microbenchmark statistics of our protocol in Section 5.2, followed by detailed comparisons of runtime, communication, latency, graph update cost and accuracy in Section 5.3.

5.2 Microbenchmark

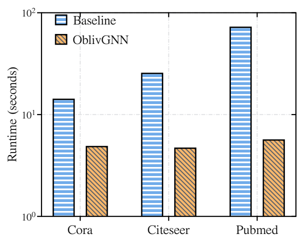
Runtime. In this subsection, we perform a comprehensive runtime analysis, comparing the detailed runtime cost between the OblivGNN and the baseline approach. The detailed figures are shown in Figure 4. The microbenchmark evaluation is segmented into four distinct sections, focusing on the runtime comparison between the linear layer and non-linear layer across the two aforementioned layers.

The first and second layer linear comparison centres around the dimensions between the normalised adjacency matrix $\hat{\mathbf{A}}$ and normalised feature matrix $\hat{\mathbf{F}}$. As evident from the data presented in 4a and 4c, our optimised oblivious matrix multiplication algorithm, denoted as OblivAgg, yields a notable decrease in the runtime compared to the baseline approach. In the baseline, the initial layer of Cora, Citeseer, and Pubmed exhibits runtimes of 14.34, 25.8, and 73.18 seconds, respectively. In contrast, in OblivGNN we proposed, these runtimes have been notably reduced to a mere 4.92, 4.75, and 5.72 seconds for three datasets, respectively. Moreover, the second layer also faces a significant reduction in runtime. In the baseline setup, the runtime for the second layer of three datasets is 496.03, 643.824 and 11252.50 seconds respectively, while it is merely 2.53, 2.90 and 1.64 seconds in the OblivGNN setup. Given the distinct sizes of these matrices across various datasets, the associated runtimes naturally vary.

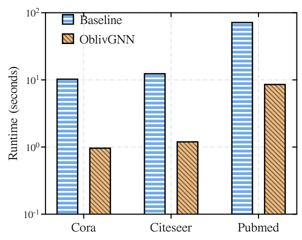
The runtime comparison between activation layers is notably evident in 4b, the OblivReLU function exhibits a significant reduction in computation time compared to the baseline approach. In the baseline approach, the runtime for ReLU for three datasets is 10.1072, 12.612 and 75.1263 seconds. In contrast to the baseline approach, the OblivGNN setup achieves a mere 0.99, 1.22 and 8.73 seconds respectively. Furthermore, the runtime (Figure 4d) significantly benefited from the improved MPC-friendly OblivSoftmax (Section 4.2.3). The strawman approach achieves 149.01, 159.52 and 509.45 seconds respectively in runtime, while the OblivGNN achieves 126.00, 132.40 and 393.83 seconds respectively.
Communication. In this subsection, we conduct a breakdown analysis of the communication time between OblivGNN and the baseline. Similar to the previous runtime microbenchmark, we analyse the communication cost in each layer.

The advantages observed in the first (Figure 5a) and second (Figure 5c) linear layers benefit from the integration of the low-communicative FSS addition and multiplication in the OblivAgg. Regarding the first linear layer, the highest communication reduction appears in the Pubmed dataset where the first linear layer reduced from 102.95MB to 31.30MB. The first linear layer for the other two datasets reduced from 209.35MB and 469.71MB to 98.88MB and 80.76MB. This trend of notable reductions is also observed in the second linear layer. The FSS-based matrix multiplication incurs noteworthy reductions in communication costs between servers and therefore, benefits both runtime and communication cost.

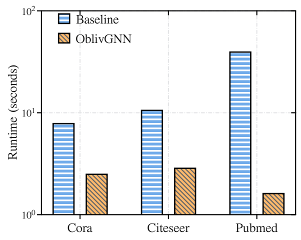
As evidenced in the first non-linear layer (Figure 5b), ReLU benefits from integrating DPF in the OblivReLU. The employment of DPF requires minimal inter-server communication in oblivious ReLU, OblivReLU. As evidenced in Figure 5b, the communication reduction in three datasets is 2.08MB to 0.26MB, 2.56MB to 0.36MB and 15.17MB to 1.89MB. The reduction of communication overheads is achieved while fortifying the protection of the access pattern inherent to the ReLU activation function. As shown in Figure 5d, the oblivious softmax OblivSoftmax is also advantaged by these adaptations. The communication cost for OblivSoftmax are reduced from 90.26, 95.47 and 295.03MB to 84.47, 88.97 and 263.58MB respectively among three datasets.



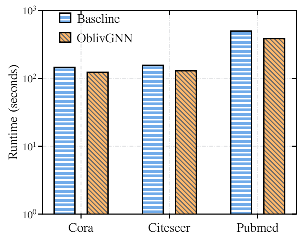
(a) First linear layer



(b) ReLU layer

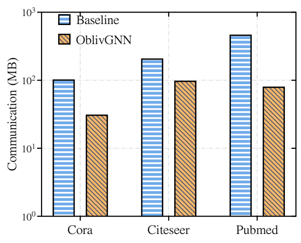


(c) Second linear layer

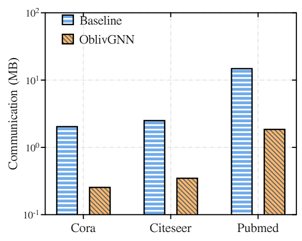


(d) Softmax layer

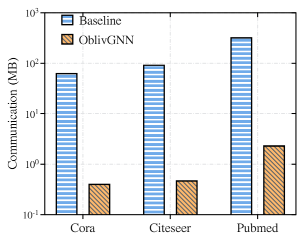
Figure 4: Microbenchmark Runtime: the layer-by-layer comparison between baseline and OblivGNN using three graph datasets in the log scale, in the unit of seconds(s).



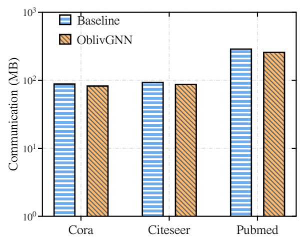
(a) First linear layer



(b) ReLU layer

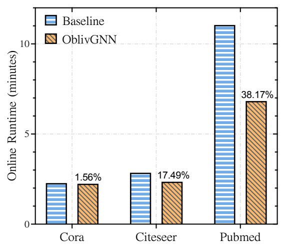


(c) Second linear layer

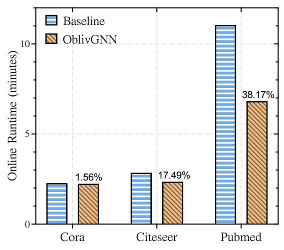


(d) Softmax layer

Figure 5: Microbenchmark Communication: the layer-by-layer inter-server communication comparison between baseline and OblivGNN using three graph datasets in the log scale, in the unit of MegaBytes (MB).



(a) Transductive



(b) Inductive

Figure 6: Runtime Comparison: the comparison between OblivGNN and baseline in three datasets, under transductive and inductive settings in the unit of minutes.

5.3 System

Runtime. This subsection discusses the runtime cost under both transductive and inductive settings. The online runtime comparisons are shown in Figure 6a and 6b.

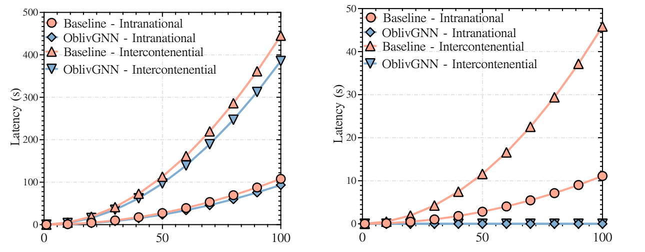
In Cora, Citeseer and Pubmed datasets, we observed substantial improvements in inference time, amounting to 1.56%, 17.49% and 38.17%, respectively. This notable enhancement in the performance of OblivGNN can be largely attributed to our innovative OblivAgg algorithm. We identified that the most time-consuming operation during GNN inference, particularly between the adjacency matrix and the feature matrix, involves matrix multiplication. Consequently, by employing FSS multiplication and addition gates, inspired by Boyle et al. [13], as opposed to additive secret shares, we achieved substantial time savings compared to the multiplication operation employed in additive secret sharing, which relies on Beaver's triple [5]. The overall cost (offline+online) varies between datasets, using Cora as an example, the total time of the baseline is 136.57 seconds, while for OblivGNN is 200.09 seconds. In Pubmed dataset, the total time for baseline is 662.97 seconds, while the OblivGNN achieves 410.06 seconds.

Communication. The analysis of communication costs is categorized into two scenarios: communication between the model owner and servers, and communication between servers. We will discuss each of the techniques below and the influences on communication cost.

The communication between the model owner and the servers involves transmitting two crucial information: graph information and key pools. To elaborate further, the graph information includes the adjacency matrix, feature matrix and weight matrix. Additionally, the key pools include the FSS key pool and the DPF key pool for oblivious secure computation. It is noteworthy that an equal volume of graph information is transmitted, thereby we note that the communication cost of graph information remains constant compared with the baseline and plaintext. Furthermore, it is important to note that the FSS and DPF key pool generated in the offline phase vary according to datasets. As demonstrated in Figure 3, the size of the FSS keys depends on the overall size of the graph (using Cora dataset as an example, the DPF key size is 32.23 MB and the FSS key size is 459.89 TB). Similarly, the size of DPF keys is directly affected by two primary factors: the size of the subgraph and the total number of inquiries.

	Baseline	**OblivGNN**
Cora	34.21	**0.29**
Citeseer	61.81	**0.41**
Pubmed	16.33	**1.65**

Table 4: Communication Cost Comparison between servers: the comparison between three baselines with three different graph datasets, in the unit of GB.



(a) Overall Intranational and Intercontinental Latency (b) Intranational and Intercontinental Latency in Linear Layers

Figure 7: Latency: the latency comparison between OblivGNN and baseline under intranational (65 ms) and intercontinental (268 ms) server geographic settings.

Table 4 provides a breakdown of the communication cost between servers.[1] Notably, the communication cost between servers is reduced when compared to the baseline, with a reduction range from 10× to 151×. This reduction benefits from the substitution of Beaver's triples with the non-interactive arithmetic FSS gates. The primary communication cost incurred in this process involves the revelation of the masked expressions $\hat{\mathbf{A}} + r_1^{in}$ and $\hat{\mathbf{F}} + r_2^{in}$. The adoption of non-interactive arithmetic FSS gates notably reduces inter-server communication expenses, thereby directly influencing the inference runtime, as shown in Figures 6a and 6b.

Latency. Here, we delve into two geographic scenarios concerning the spatial distribution of servers. The first scenario is an intra-national setup, where we presume the servers to be geographically situated within a country, such as the east and west coasts of America. The second scenario is an intercontinental setup, where we consider the servers as geographically distributed across separated continents, such as West America and East Asia. According to Google Cloud Inter-Region Latency and Throughput [42], the current intra-national latency is 65 ms and the intercontinental latency is 268 ms. We show our results in the Figure 7a and 7b.

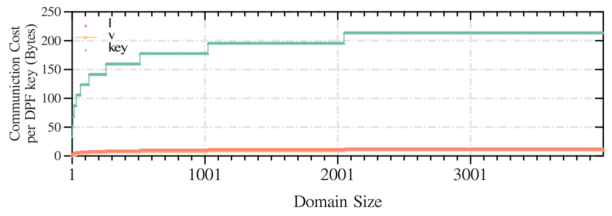
As shown in Figure 7b, the linear layer latency in OblivGNN is consistent, contrasting with the exponential growth shown in the baseline approach. However, the overall latency shown in Figure 7a suggests that the OblivGNN has a similar trend as the baseline approach. The reason can be found in Figure 5d, where the OblivSoftmax layer in OblivGNN demonstrates lower but similar communication cost as the baseline approach. It is caused by the oblivious if (if_else) in the MP-SPDZ framework [30], which uses multiplexer (MUX) on bit in arithmetic circuits to determine the conditional selection on numbers. The use of garbled circuit to implement the oblivious selection (if $\sum_i \mathsf{OblivReLU}([\![z[i]]\!]_b) > 0$ or otherwise) dominates the cost of OblivSoftmax and leads to a similar trend in overall latency. However, we also notice that due to the consistent growth in OblivGNN linear layer, OblivGNN is more scalable towards larger graphs compared to the baseline approach. We leave the optimisation of activation functions as future work.

Graph Update Cost. In this subsection, we will concentrate on the discussion of graph update costs. For the node insertion without modifying the current node connections and feature attributes, the communication cost will be the appended adjacency matrix plus the appended feature matrix. For instance, if the client or the model owner wants to add $n^\#$ (where $n^\# = n^* - n$) number of new nodes to the graph, the adjacency matrix will incur a communication cost of $2 \times n^\#$ and the feature matrix will incur $n^\# + c$. However, access pattern leaks if the model owner or the client have the opportunity to use their own proprietary graphs to update the original graph under the inductive setting. As described in Algorithm 5, DPF keys are employed to protect the access pattern from leakage. DPF keys $k^{\mathrm{A}}_{b\in\{0,1\}}$ and $k^{\mathrm{F}}_{b\in\{0,1\}}$ are generated and evaluated to update the adjacency matrix and feature matrix during the inference stage. Consequently, the time and communication costs will be directly related to the number of nodes that are inserted. We provide a summary of the full-domain DPF key generation evaluation [6] as follows.

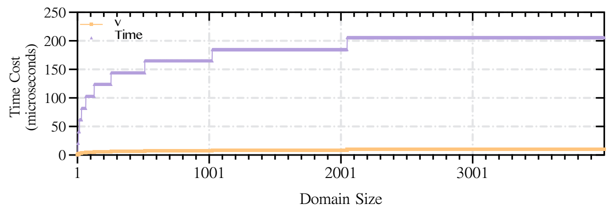
The size of the DPF keys varies depending on the dataset. We assume that the DPF key generation and evaluations are carried out within the domain of $\mathbb{Z}_{2^a}$, where a is calculated by $\lceil \log_2 n \rceil$ and n is the number of nodes in the dataset. For Cora and Citeseer dataset, there are 2708 and 3327 nodes respectively, therefore $a = \lceil \log_2 2708 \rceil = \lceil \log_2 3327 \rceil = 12$, while Pubmed has 19717 nodes, thus $a = \lceil \log_2 19717 \rceil = 15$. The a will be used in the key length calculation.

To expedite the evaluation process, we employ full-domain DPF evaluation, a technique introduced in [6]. Full-domain evaluation incorporates an optimisation called early termination. Such optimisation allows the evaluation process to halt at 2^v rather than continuing until 2^a, resulting in a speedup of 2^{a-v}. The DPF keys have a key length as follows: $18 \times (v+1) + 16$, where $v = \lceil a - \log_2(\frac{\lambda}{\log_2 |\mathbb{Z}_{2^\ell}|}) \rceil$, in the unit of bytes. λ (security parameter) is 128, and $\mathbb{Z}_{2^\ell}$ (output group) is 2^{32}. The value of v for each of the three datasets can be calculated as follows. As demonstrated in Figure 8a, both the Cora and Citeseer datasets share the same value of $a = 12$. Consequently, the values of v for these datasets are determined to be 10, resulting in a single DPF key size of

[1]The communication cost for baseline Pubmed is too big and leads to counter overflow. This blocks us from getting the precise result, which should be the highest among all three datasets due to its size. Nonetheless, the current result has shown a huge communication cost save with OblivGNN.



(a) Node/Feature Update Key Size



(b) Node/Feature Update Time Cost

Figure 8: Node/Feature Update cost: Comparison of communication cost and time cost according to the domain size.

0.214 KB. On the other hand, the Pubmed dataset has $v = 13$, which translates to a key size of 0.268 KB. As depicted in Figure 8b, the average time required for a full-domain DPF evaluation is 206 microseconds for Cora and Citeseer, and 249 microseconds for Pubmed.

Accuracy. We now discuss the accuracy comparison across three datasets between OblivGNN and baseline model, considering both transductive and inductive settings.

Table 5 provides a comprehensive comparison across three widely-accepted datasets: Cora, Citeseer, and Pubmed (refer to Table 3 for dataset details). Each dataset is evaluated in both transductive and inductive settings. The unsecured plaintext test serves as the benchmark for comparison. The plaintext achieves the highest accuracy results, with the highest in the transductive Cora (87.82%) and lowest in inductive Citeseer (74.77%). The baseline approach and OblivGNN achieve similar levels of accuracy. The baseline achieved the highest accuracy in transductive Cora (81.92%) and the lowest in inductive Citeseer (72.96%). While the OblivGNN achieved 83.76% in transductive Cora and 72.97% in inductive Citeseer. Considering the significant reduction in runtime cost, communication cost and latency, OblivGNN demonstrates a noticeable advantage compared with the baseline.

6 Related Works

Privacy-Preserving Machine Learning. Previous works in PPML [9, 16, 19, 28, 38, 44–46, 49, 51, 52, 56, 58] have primarily focused on Convolutional Neural Networks [48] and Recurrent Neural Networks [55]. Their primary objective is protecting users' data while enabling machine learning tasks (i.e., classification, prediction) on sensitive data. The existing solutions face limitations in processing and preserving the privacy of graph-structured data for GNNs tasks.

Despite the presence of previous works in the field of PPML, there are attempts to design secure GNN services. Notably, SecGNN [59] introduced the first system designed to support secure GNN training and inference services in a cloud environment. Moreover, CryptoGCN [50], introduced a secure GNN model that optimised matrix operations to substantially reduce computational overheads.

However, existing secure GNN solutions [14, 50, 59, 65] still exhibit limitations, such as the lack of support for *inductive* setting (dynamic graphs) and high computational/communication overheads. While LinGCN [14] minimises the computational/communication overheads, expensive cryptographic tools still pose challenges in cost.

Graph Neural Networks. GNNs have attracted significant attention in recent years due to their potential for handling complex graph-structured data in machine learning. Gori et al. [24] first introduced the concept of GNNs, and subsequent studies [10, 22, 36, 54] contributed to the development of Recurrent Graph Neural Networks (RecGNNs). Spectral-based Convolutional GNNs emerged by the introduction of spectral convolution methods for graphs [8], inspiring subsequent studies [11, 33]. NN4G by Micheli et al. [43] led to the emergence of Spatial-based Convolutional GNNs [20, 25, 57, 64].

Inference Attacks against GNNs. Inference Attacks, such as stealing and membership inference attacks, pose significant dangers to data privacy and model security in GNNs. Model stealing attacks (MSA) [61] reverse-engineer the GNN model by exploiting query access or its predictions. Membership inference attacks (MIA) [60] in GNN target to identify whether specific nodes [12] or links [26] were part of the training graph based on the model's behaviour [62]. Note that such attacks are marked orthogonal to typical PPML works [19, 38, 44, 52, 56] that aim to protect models/inputs/results during computation, and consider inference from computation results out of scope.

Some works [15, 47, 53] have shown that DP naturally provides defence against MIA. If DP noises(e.g., perturbation on graphs and/or gradients) are added during the training phase, the methods are complementary to OblivGNN. For customised DP-based methods such as Kolluri et al. [34], it requires careful adaption in OblivGNN, as those methods also modify the GNN model architecture. For MSA, watermarking techniques [63] were proposed to detect the stolen GNN models. As watermarks are injected during the training phase, OblivGNN can also readily support those defences. However, watermarking is limited to the detection of stealing without proactive defence.

7 Conclusion

We present OblivGNN, a lightweight and low-communicative/computational oblivious system designed for securing transductive and inductive inference on GNNs, in the semi-honest two-server setting. We emphasise the

	Plaintext		**Baseline**		**OblivGNN**	
	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive
Cora	87.82%	83.95%	81.92%	80.44%	83.76%	81.92%
Citeseer	76.13%	74.77%	74.93%	72.96%	75.53%	72.97%
Pubmed	85.12%	84.71%	82.60%	83.49%	83.24%	83.65%

Table 5: Accuracy Comparison: the comparison between three baselines with three different graph datasets, under transductive and inductive settings.

significance of this work and highlight the existing research gap between current PPML models and plaintext GNN models. OblivGNN contributes a set of comprehensive algorithms that enable OblivGNN to perform oblivious and low-interactive inference on both *transductive* (static graphs) and *inductive* (dynamic graphs) settings. Finally, we proceed to implement OblivGNN using the MP-SPDZ framework and evaluate its performance against the baseline model on three prominent datasets: Cora, Citeseer, and Pubmed. Through extensive experiments, we showcase promising results achieved by OblivGNN.

Acknowledgments

We thank all the anonymous reviewers and the shepherd for their invaluable comments. Zhibo Xu is supported by CSIRO's Data61 full scholarship. This work was supported in part by a Monash-Data61 CRP project, ARC Linkage Project (LP180101062), and a project of CSIRO-NSF AI Research Collaboration Program.

References

[1] S. Adeshina. Detecting Fraud in Heterogeneous Networks using Amazon SageMaker and Deep Graph Library. https://aws.amazon.com/blogs/machine-learning/detecting-fraud-in-heterogeneous-networks-using-amazon-sagemaker-and-deep-graph-library/, 2020.

[2] L. A. Alves et al. Graph Neural Networks as a Potential Tool in Improving Virtual Screening Programs. *Frontiers in Chemistry*, 9:787194, 2021.

[3] M. J. Atallah, M. Bykova, J. Li, K. B. Frikken, and M. Topkara. Private Collaborative Forecasting and Benchmarking. In *ACM WPES*, 2004.

[4] B. Balakreshnan. Graph Neural Network in Azure Machine Learning — Classification. https://balabala76.medium.com/graph-neural-network-in-azure-machine-learning-classification-df7978d7bd45, 2021.

[5] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, 1991.

[6] E. Boyle, N. Gilboa, and Y. Ishai. Function Secret Sharing: Improvements and Extensions. In *ACM CCS*, 2016.

[7] E. Boyle, N. Gilboa, and Y. Ishai. Secure Computation with Preprocessing via Function Secret Sharing. In *TCC*, 2019.

[8] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral Networks and Locally Connected Networks on Graphs. In *ICLR*, 2014.

[9] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*, 2020.

[10] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, and L. Song. Learning Steady-States of Iterative Algorithms over Graphs. In *ICML*, 2018.

[11] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*, 2016.

[12] V. Duddu, A. Boutet, and V. Shejwalkar. Quantifying Privacy Leakage in Graph Embedding. In *ACM MobiQuitous*, 2020.

[13] E. Boyle et al. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *EuroCrypt*, 2021.

[14] H. Peng et al. LinGCN: Structural Linearized Graph Convolutional Network for Homomorphically Encrypted Inference. In *NeurIPS*, 2023.

[15] Kai et al. Defense Sgainst Membership Inference Attack in Graph Neural Networks Through Graph Perturbation. *International Journal of Information Security*, 22(2):497–509, 2023.

[16] M. S. Riazi et al. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ACM AsiaCCS*, 2018.

[17] R. Ying et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *ACM KDD*, 2018.

[18] R. Zhu et al. AliGraph: A Comprehensive Graph Neural Network Platform. In *VLDB*, 2019.

[19] S. Wagh et al. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. In *PoPETs*, 2021.

[20] W. Chiang et al. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *ACM KDD*, 2019.

[21] F. Falzon and K. G. Paterson. An Efficient Query Recovery Attack Against a Graph Encryption Scheme. In *ESORICS*, 2022.

[22] C. Gallicchio and A. Micheli. Graph Echo State Networks. In *IEEE IJCNN*, 2010.

[23] C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *ACM STOC*, 2009.

[24] M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. In *IEEE IJCNN*, 2005.

[25] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, 2017.

[26] X. He, J. Jia, M. Backes, N. Zhenqiang Gong, and Y. Zhang. Stealing Links from Graph Neural Networks. In *USENIX Security*, 2021.

[27] A. Jain, I. Liu, A. Sarda, and P. Molino. Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations. https://www.uber.com/en-AU/blog/uber-eats-graph-learning/, 2019.

[28] C. Juvekar, V. Vaikuntanathan, and A. P. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*, 2018.

[29] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing Multi-Party Computation. Cryptology ePrint Archive, Paper 2011/272, 2011.

[30] M. Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS*, 2020.

[31] M. Keller and K. Sun. Effectiveness of MPC-friendly Softmax Replacement. In *NeurIPS PPML Workshop*, 2020.

[32] M. Keller and K. Sun. Secure Quantized Training for Deep Learning. In *ICML*, 2022.

[33] T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR (Poster)*, 2017.

[34] A. Kolluri, T. Baluta, B. Hooi, and P. Saxena. LPGNet: Link Private Graph Networks for Node Classification. In *ACM CCS*, 2022.

[35] B. Lackey. Use Graphs for Smarter AI with Neo4j and Google Cloud Vertex AI. https://cloud.google.com/blog/products/ai-machine-learning/analyze-graph-data-on-google-cloud-with-neo4j-and-vertex-ai, 2022.

[36] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated Graph Sequence Neural Networks. In *ICLR (Poster)*, 2016.

[37] Y. Lindell. How To Simulate It - A Tutorial on the Simulation Proof Technique. Cryptology ePrint Archive, Paper 2016/046, 2016.

[38] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM CCS*, 2017.

[39] 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*, 17:237–252, 2021.

[40] X. Liu, Y. Zheng, X. Yuan, and X. Yi. MediSC: Towards Secure and Lightweight Deep Learning as a Medical Diagnostic Service. In *ESORICS*, 2021.

[41] X. Liu, Y. Zheng, X. Yuan, and X. Yi. Securely Outsourcing Neural Network Inference to the Cloud With Lightweight Techniques. *IEEE Transactions on Dependable and Secure Computing*, 20(1):620–636, 2022.

[42] Google LLC. Google Cloud Inter-Region Latency and Throughput. http://lookerstudio.google.com/u/0/reporting/fc733b10-9744-4a72-a502-92290f608571/page/70YCB, 2023.

[43] A. Micheli. Neural Network for Graphs: A Contextual Constructive Approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[44] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A Cryptographic Inference Service for Neural Networks. In *USENIX Security*, 2020.

[45] P. Mohassel and P. Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*, 2018.

[46] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

[47] I. Olatunji, T. Funke, and M. Khosla. Releasing Graph Neural Networks with Differential Privacy Guarantees. *Transactions on Machine Learning Research*, 2023.

[48] K. O'Shea and R. Nash. An Introduction to Convolutional Neural Networks. *CoRR*, abs/1511.08458, 2015.

[49] A. Patra and A. Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*, 2020.

[50] R. Ran et al. CryptoGCN: Fast and Scalable Homomorphically Encrypted Graph Convolutional Network Inference. In *NeurIPS*, 2022.

[51] M. S. Riazi et al. XONN: XNOR-based Oblivious Deep Neural Network Inference. In *USENIX Security*, 2019.

[52] T. Ryffel, P. Tholoniat, D. Pointcheval, and F. R. Bach. AriaNN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. In *PoPETs*, 2022.

[53] S. Sajadmanesh, A. Shamsabadi, A. Bellet, and D. Gatica-Perez. GAP: Differentially Private Graph Neural Networks with Aggregation Perturbation. In *USENIX Security*, 2023.

[54] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[55] A. Sherstinsky. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[56] S. Tan, B. Knott, Y. Tian, and D. J. Wu. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *IEEE S&P*, 2021.

[57] P. Velickovic et al. Graph Attention Networks. In *ICLR (Poster)*, 2018.

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

[59] 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*, 16(4):2923–2938, 2023.

[60] B. Wu, X Yang., S. Pan, and X. Yuan. Adapting Membership Inference Attacks to GNN for Graph Classification: Approaches and Implications. In *IEEE ICDM*, 2021.

[61] B. Wu, X. Yang, S. Pan, and X. Yuan. Model Extraction Attacks on Graph Neural Networks: Taxonomy and Realisation. In *ACM AsiaCCS*, 2022.

[62] F. Wu, Y. Long, C. Zhang, and B. Li. LINKTELLER: Recovering Private Edges from Graph Neural Networks via Influence Analysis. In *IEEE S&P*, 2022.

[63] J. Xu, S. Koffas, O. Ersoy, and S. Picek. Watermarking Graph Neural Networks based on Backdoor Attacks. In *IEEE EuroS&P*, 2023.

[64] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks? In *ICLR*, 2019.

[65] W. Xu et al. MDP: Privacy-Preserving GNN Based on Matrix Decomposition and Differential Privacy. In *IEEE JCC*, 2023.

[66] H. Zhang et al. Trustworthy Graph Neural Networks: Aspects, Methods and Trends. *Proceedings of the IEEE*, pages 1–43, 2024.

[67] J. Zhang, H. Wang, and M. Zhu. Build a GNN-based Real-Time Fraud Detection Solution using Amazon SageMaker, Amazon Neptune, and the Deep Graph Library. https://aws.amazon.com/blogs/machine-learning/build-a-gnn-based-real-time-fraud-detection-solution-using-amazon-sagemaker-amazon-neptune-and-the-deep-graph-library/, 2022.

[68] X. Zhang, L. Liang, L. Liu, and M. Tang. Graph Neural Networks and Their Current Applications in Bioinformatics. *Frontiers in Genetics*, 12:690049, 2021.

A Security Analysis

A.1 Security Definition

Definition A.1 (FSS Security). *We say that the (*$\mathsf{KeyGen}^{=}$, $\mathsf{KeyGen}^{<}$, $\mathsf{Eval}^{=}$, $\mathsf{Eval}^{<}$*) as defines above, is a FSS scheme if it satisfies the following requirements:*

Security: *For each $b \in \{0,1\}$, there is a Probabilistic Polynomial Time (PPT) algorithm Simulator $\mathcal{S}_b$, such that for polynomial-size all-prefix function with sequence $((\alpha,\bar{\beta})_\lambda)_{\lambda\in\mathbb{N}}$ and polynomial size input sequence α_λ, the outputs of the following experiments* Real *and* Ideal *are indistinguishable:*

- $\mathrm{Real}_\lambda : (k_0,k_1) \leftarrow \mathsf{KeyGen}(1^\lambda,(\alpha,\bar{\beta})_\lambda)$*; Output k_b.*
- $\mathrm{Ideal}_\lambda :$ *Output $\mathcal{S}_b$.*

A.2 Security Proof

We employ a simulation framework [37] based on multiparty computation (MPC) to establish the security assurances of OblivGNN. Within this framework, we consider a probabilistic polynomial-time (PPT) adversary denoted as $\mathcal{A}$, who possesses the ability to statically compromise one of the two servers. In the influence from $\mathcal{A}$, the corrupted server is permitted to act as a semi-honest entity, implying that it may acquire knowledge while adhering to the protocol's specifications. To streamline the clarity of our proof, we assume that the client behaves honestly.

To prove security under the simulation paradigm entails the definition of two distinct worlds: the real world, where the protocol is executed by trustworthy parties, and an ideal world, where an ideal functionality denoted as $\mathcal{F}$ accepts inputs from the involved parties and produces the desired output for the relevant party. The adversary $\mathcal{A}$ monitors the interactions on compromised servers through the protocol execution. To ensure the adversary $\mathcal{A}$ cannot differentiate between the real world and the ideal world, we introduce a simulator, denoted as $\mathcal{S}$. The simulator $\mathcal{S}$ imitates messages between the model owner, honest server, client and the corrupt server in the real world. It is important to note that the simulator $\mathcal{S}$ only interacts with the adversary $\mathcal{A}$ and ideal functionality $\mathcal{F}$. If the adversary $\mathcal{A}$ cannot distinguish between the ideal world and the real world with the messages provided by ideal functionality $\mathcal{F}$ using the defined leakage, we consider our protocol to be secure.

Ideal Functionality $\mathcal{F}$. We first define our ideal functionality $\mathcal{F}$ to be a graph neural network and provide the graph data inference service as follows:

1) $\mathsf{OblivAgg}(k^{\times}, k^{+}, \hat{\mathbf{F}}, \hat{\mathbf{A}})$: $\mathcal{F}$ performs aggregation between $\hat{\mathbf{A}}$ and $\hat{\mathbf{F}}$ by employing the FSS keys, $k^{\times}, k^{+}$.
2) $\mathsf{OblivReLU}(k^{<})$ and $\mathsf{OblivSoftmax}(k^{<})$: $\mathcal{F}$ performs $\mathsf{OblivReLU}$ and $\mathsf{OblivSoftmax}$ on the aggregated graph data and repeats for the second layer.
3) $\mathsf{OblivUpdate}(k^{\mathrm{A}}$ and $k^{\mathrm{F}})$: $\mathcal{F}$ performs $\mathsf{OblivUpdate}$ using the DPF keys (k^{A}) to update the adjacency matrix and the DPF keys (k^{F}) to update the feature matrix.
4) $\mathsf{OblivArgmax}(k^{=}, k^{<})$: $\mathcal{F}$ performs $\mathsf{OblivArgmax}$ using the DPF keys $k^{=}$ and $k^{<}$ to calculate the most possible class and store the result.

We allow the ideal functionality $\mathcal{F}$ to leak the following information. 1) the size of the graph information (e.g., the total number of nodes, the feature size), 2) the model size (e.g., the weight sizes), 3) the number of FSS and DPF keys, 4) the FSS and DPF key size.

Definition A.2. *Let Π be a protocol for secure Graph Neural Network (GNN) inference service, which takes input Q from the model owner and client. The trustworthy party Π provides the functionality modelled by the above-defined ideal functionality $\mathcal{F}$. The adversary $\mathcal{A}$ observes the view from the corrupted server while the protocol runs. Let $\mathsf{View}^{\mathrm{Real}}_{\Pi(Q)}$ denote the view from $\mathcal{A}$ in real world. Let $\mathsf{View}^{\mathrm{Ideal}}_{\mathcal{S},\mathsf{Leak}(\mathcal{F}(Q))}$ denote the simulated view generated by Simulator $\mathcal{S}$ for $\mathcal{A}$. For all non-uniform PPT algorithms $\mathcal{A}$ with λ as security parameter, there exists a PPT algorithm $\mathcal{S}$ such that:*

$$\Pr[Q \leftarrow \mathcal{A}(1^{\lambda}); b \in \{0,1\}, \mathcal{A}(\mathsf{View}_b, Q) = b] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

where $\mathsf{View}_0 = \mathsf{View}^{\mathrm{Real}}_{\Pi(Q)}$, $\mathsf{View}_1 = \mathsf{View}^{\mathrm{Ideal}}_{\mathcal{S},\mathsf{Leak}(\mathcal{F}(Q))}$

Theorem A.1. *According to the Definition A.2,* OblivGNN *is secure when evaluating the inputs from the ideal function $\mathcal{F}$.*

Proof. We will initiate our proof by describing the construction of the simulator $\mathcal{S}$ in the context of the ideal world. Our approach adheres to the hybrid models described in [37], assuming the sub-protocols gradually replacing the protocols in the real world. As long as it is demonstrated that all the sub-protocols are proven to be indistinguishable from the real world, it is then conclusively proven that OblivGNN is secure.

Simulator Construction. Assuming that the P_0 is the corrupted party. Upon receiving inputs Q and known leakage $Leak(\mathcal{F}(Q))$, the simulator $\mathcal{S}$ performs as follows:

1) On receiving $(Leak(\mathcal{F}(Q)))$ from $\mathcal{F}$: $\mathcal{S}$ conducts DPF and FSS key generation, considering the leakage $\mathcal{L}$.
2) On receiving $(k^{\times}, k^{+}, \hat{\mathbf{F}}, \hat{\mathbf{A}})$ from $\mathcal{F}$: $\mathcal{S}$ conducts random number sampling for the matrix, considering the received matrix size. It maintains a local copy and simultaneously conveys a duplicate 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 matrix for P_0 to $\mathcal{A}$.
4) On receiving k^{A} and k^{F} from $\mathcal{F}$: $\mathcal{S}$ employs random number sampling to create updated graph matrices. The dimensions of these updated matrices are contingent on the size of the original matrices and the number of keys received. $\mathcal{S}$ then forwards the P_0's share of the updated sampled graph matrices to $\mathcal{A}$.
5) On receiving $k^{<}$ and $k^{=}$ from $\mathcal{F}$: $\mathcal{S}$ samples random numbers for matrix after Argmax. The matrix size depends on the input matrix size. $\mathcal{S}$ then sends share of P_0 to $\mathcal{A}$.

We now dive into the proof that the above-generated views from $\mathcal{S}$ are indistinguishable between the real world and ideal world with the security parameter λ through the following hybrids $\mathcal{H}0 \ldots \mathcal{H}3$.

Hybrid 0. We initiate the hybrid with the real world.

Hybrid 1. Simulator $\mathcal{S}$ replaces the FSS $k^{\times}$, k^{+}, DPF keys $k^{=}$ and DCF keys $k^{<}$ with the output from the above-defined FSS, DPF and DCF simulators $\mathcal{S}_{\mathrm{FSS}}$, $\mathcal{S}_{\mathrm{DPF}}$ and $\mathcal{S}_{\mathrm{DCF}}$ from model owner. Based on the security analysis of FSS, DPF and DCF schemes, it implies that the probability of $\mathcal{A}$ can distinguish between $\mathcal{H}0$ and $\mathcal{H}1$ is $\mathrm{negl}(\lambda)$.

Hybrid 2. Simulator $\mathcal{S}$ replaces the DPF keys k^{A}, k^{F} and k^{Q} with the generated DPF keys based on real graph matrices for OblivUpdate, client inquiry from client. Based on the security analysis of DPF schemes, the probability of $\mathcal{A}$ can distinguish between $\mathcal{H}1$ and $\mathcal{H}2$ is $\mathrm{negl}(\lambda)$.

Hybrid 3. Simulator $\mathcal{S}$ replaces adjacency matrix $\mathbf{A}$ and feature matrix $\mathbf{F}$ with the simulated graph matrices. The probability of $\mathcal{A}$ can distinguish between $\mathcal{H}2$ and $\mathcal{H}3$ is $\mathrm{negl}(\lambda)$.

This concludes the proof for Theorem A.1. It is proven that $\mathcal{A}$ with the views cannot distinguish between the real world and ideal world, except with the probability $\leq \frac{1}{2} + \mathrm{negl}(\lambda)$.