Entrada to Secure Graph Convolutional Networks [*]

Nishat Koti
Indian Institute of Science, Bangalore
kotis@iisc.ac.in

Varsha Bhat Kukkala
Indian Institute of Science, Bangalore
varshak@iisc.ac.in

Arpita Patra
Indian Institute of Science, Bangalore
arpita@iisc.ac.in

Bhavish Raj Gopal
Indian Institute of Science, Bangalore
bhavishraj@iisc.ac.in

Abstract

Graph convolutional networks (GCNs) are gaining popularity due to their powerful modelling capabilities. However, guaranteeing privacy is an issue when evaluating on inputs that contain users' sensitive information such as financial transactions, medical records, etc. To address such privacy concerns, we design Entrada, a framework for securely evaluating GCNs that relies on the technique of secure multiparty computation (MPC). For efficiency and accuracy reasons, Entrada builds over the MPC framework of Tetrad (NDSS'22) and enhances the same by providing the necessary primitives. Moreover, Entrada leverages the GraphSC paradigm of Araki et al. (CCS'21) to further enhance efficiency. This entails designing a secure and efficient shuffle protocol specifically in the 4-party setting, which to the best of our knowledge, is done for the first time and may be of independent interest. Through extensive experiments, we showcase that the accuracy of secure GCN evaluated via Entrada is on par with its cleartext counterpart. We also benchmark efficiency of Entrada with respect to the included primitives as well as the framework as a whole. Finally, we showcase Entrada's practicality by benchmarking GCN-based fraud detection application.

1 Introduction

Graph convolutional networks (GCNs) are gaining importance as a powerful machine learning technique for leveraging graph-structured data. It finds use in a diverse set of applications, including traffic prediction, rumour detection, targeted advertising and recommendation systems, to name a few. The massive size of the input graph and multiple layers of the neural network results in GCNs being computationally intensive. Hence, various platforms, such as Neptune ML by Amazon, offer GCN training and inference as a service, where a data owner (client) provides its input data in clear to the hired servers that carry out the computations on behalf of the client. However, the input may comprise private information such as the graph topology, node/edge features, etc., that the client may not wish to disclose to the servers. Such threats to privacy are evident when dealing with data with respect to health records [26], social networks [25], financial transactions [17], etc. This necessitates designing techniques that allow clients to outsource the computation such that their inputs remain private while enabling servers to operate on the private inputs. Although paradoxical, such privacy-preserving evaluation of GCNs can be achieved via MPC.

MPC is a technique that allows a set of n mutually distrusting parties to jointly evaluate a function on their private inputs. Even in the presence of an adversary that corrupts at most t out of the n parties, MPC guarantees that nothing beyond the function output is revealed. In the context of secure GCN evaluation, this translates to MPC enabling the following. The hired servers that offer computing facilities enact the role of the parties in the MPC protocol. The client secret-shares its data among the hired servers, i.e., data is distributed among the servers in a manner such that no single server or a subset of servers can learn the data on clear. The servers run MPC protocols designed to securely evaluate GCNs on the secret-shared data. The output of the computation continues to be in secret-shared form among the servers, which can then be reconstructed towards the intended entity. Since the client's sensitive data remains hidden throughout the process, the solution caters to the privacy requirement of the client. In fact, there arise scenarios where data required to evaluate the GCN could be distributed among multiple clients, each of which regards its data as private. We illustrate this with the example of fraudulent account detection in banks.

Consider the banking sector comprising many different banks. Each bank has several associated accounts, and there are financial transactions that are executed between the accounts within the same bank or across two different banks. Some accounts may be fraudulent, and it is important to detect them in time to prevent damages caused by the fraud. Banks would be interested in identifying such fraudulent accounts by using the transaction data and information regarding accounts that are already known to be fraudulent. Moreover,

[*]"Entrada" in Spanish means an expedition into unexplored territory.

since fraudulent accounts constitute only a small fraction of the accounts in a bank, it is desirable for the banks to pool their data together and facilitate more accurate identification of fraudulent accounts. This can be realized via GCNs, where bank accounts constitute the nodes in the graph, while transactions between the nodes correspond to an edge. Some of the nodes are labelled as fraudulent, while the rest are unlabelled. Each node will also be associated with additional information, say transaction history, date of account creation, etc., that act as the features associated with the node. The goal is to train a GCN model on this transaction graph using the labelled nodes and features, to facilitate labelling the unlabelled nodes of the graph. This is referred to as the task of node classification. Observe that GCN evaluation requires knowledge of the global transaction graph. In practice, the graph is held in a distributed fashion, where each bank is only aware of the transactions associated with accounts within the bank. Thus, each bank only possesses a partial view of the transaction graph that comprises a subset of nodes and the edges associated with these nodes. This necessitates cooperation from multiple banks. However, banks may not wish to reveal their view of the transaction graph since it contains highly sensitive data. Thus, there is a need for designing a privacy-preserving solution that allows securely evaluating GCN on the global transaction graph while ensuring the privacy of the banks' view of the transaction graph.

MPC protocols are known to have an efficiency overhead in comparison to cleartext computation. Hence, for compute-intensive GCNs, it is imperative to design efficient MPC protocols. We note that works in the literature look at securely computing neural networks (NN) via MPC [22, 23, 31]; however, GCNs have not been well explored. Prior works that explore GCNs only consider performing secure inference over relatively older GCN models [37]. Although one may consider extending these works to securely realize GCNs, they either lack the necessary primitives or the desired level of security, efficiency and accuracy. Elaborately, several works trade off accuracy for efficiency by relying on MPC-friendly alternatives for non-linear functions [10, 23, 31]. Instead, we strive to design *accurate* protocols for GCN evaluation while *not* compromising on *efficiency*. To aid in this, we make several design choices. It is well known that operating with a small number of parties (with an honest majority) allows for designing efficient protocols [11, 22, 23]. Further, protocols designed to operate on the ring algebraic structure have better efficiency than protocols designed for a field [12, 13]. Since many real-world applications, such as prediction for healthcare and traffic, demand a fast response time, our protocols are cast in the preprocessing paradigm. This enables offloading heavy input-independent computations to a preprocessing phase, paving the way for a fast input-dependent online phase. Since the 4-party computation framework of Tetrad [23] satisfies all of these properties and is known to outperform other frameworks in small-party (honest-majority) setting, we choose to build Entrada over Tetrad. Further discussion on the choice of Tetrad appears in §2.

1.1 Our contributions

Consequent to the above discussion, we design Entrada, a secure framework for efficiently evaluating GCNs. To the best of our knowledge, this is done for the first time. Entrada not only allows performing secure inference, but it also supports secure training of GCNs. Further, Entrada enables operating on *homogeneous* as well as *heterogeneous* GCN, where the former operates on a graph comprising a single type of edge, while the latter accounts for multiple types of edges.

To highlight the performance of Entrada, we carry out extensive experiments. Entrada provides an improved GCN accuracy of 79.3% in comparison to 74.1% of Tetrad, while cleartext computations provide an accuracy of 79.9%. We note that the reduced accuracy of Tetrad is due to its reliance on approximate variants for non-linear functions (e.g., Softmax), as opposed to Entrada which uses the accurate versions for the same. With respect to efficiency, Entrada witnesses gains of up to $4\times$ in online run time, and three orders of magnitude in preprocessing time over Tetrad for GCN training. We also showcase the practicality of our solution by benchmarking fraud detection algorithms of [41], [27]. These improvements are a result of two-fold contributions. First, Entrada enhances Tetrad by adding new primitives and improving existing primitives. Second, specific to GCN, we leverage GraphSC framework and bring in new contributions therein, including the shuffle primitive. We elaborate on these below.

Enhancing Tetrad Entrada enhances Tetrad by providing support for efficient realizations of prefix OR, double bit-injection, exponentiation, division, and inverse square root. While most of these are well-studied [7, 18, 21, 23], our optimizations aid in obtaining their efficient realizations. This makes Entrada a more accurate, comprehensive and efficient framework for PPML in comparison to Tetrad.

Tailoring GraphSC for GCN To further enhance efficiency, we leverage the GraphSC paradigm [5, 32]. GraphSC is a generic framework for efficiently realizing secure computations over graphs. It takes as input the underlying graph stored as a list G of nodes ($\mathcal{V}$) and edges ($\mathcal{E}$), where each entry $\mathsf{G}[i]$ in the list is associated with data or state of the correspoding node/edge. It considers graph algorithms that can be expressed as message-passing algorithms, where the latter involves updating data/state of the vertices and edges of the graph in an iterative manner. Thus, using GraphSC, the required updates in each iteration of the message-passing algorithm are defined in terms of the primitive operations of Scatter and Gather. These primitives facilitate sending (Scatter) and receiving (Gather) messages across edges when scanning through the entries in G, as required in the message-passing algorithm. In this way, GraphSC entails translating the graph algorithm as a sequence of Scatter-Gather primitives.

Further, to ensure that the invocations of Scatter-Gather do not leak information regarding the graph topology, G must be securely shuffled prior to each such invocation. Thus, securely realizing the graph algorithm reduces to securely shuffling G and invocations of Scatter-Gather primitives across multiple rounds by instantiating operations within them using desired MPC protocols. Apart from operating on an efficient graph representation, GraphSC framework also showcases how a multiprocessor setting can be leveraged to realise parallel variants of the Scatter-Gather primitives, and further improve the efficiency. However, GraphSC paradigm does not specify operations within Scatter-Gather as they are specific to the graph algorithm. Thus, our contribution entails–
(i) Cleartext to message-passing: Identifying the relevant cleartext computations in GCNs that can benefit from GraphSC and can be rendered as message-passing algorithms.
(ii) Message-passing to GraphSC: Redefining the graph algorithm in terms of the Scatter-Gather primitives as well as defining the GCN specific computations that are required within these Scatter-Gather primitives. The Scatter-Gather primitives are then securely realized via Entrada.
(iii) Secure shuffle: Designing a secure *shuffle* protocol as required for GraphSC, which is done for the first time in the 4-party setting. The protocol has an amortized communication of 3N ring elements (where N denotes the size of the vector to be shuffled) and 1 round of interaction in the online phase. Note that the secure shuffle protocol forms an integral part of various applications such as anonymous broadcast [15], secure sorting [5], etc. Hence, the inclusion of shuffle makes Entrada a versatile framework and opens up avenues for exploring its use in other shuffle-based application scenarios.
Concretely, while the contribution of 'Enhancing Tetrad' brings in an improvement of 1.7× in the online run time of GCN training over Tetrad (and 486× in preprocessing run time), additionally 'Tailoring GraphSC' further enhances the online efficiency up to 4× (5782× for preprocessing). In this way, Entrada is not only a more versatile framework than Tetrad, but is also more efficient.

Input sharing To enable multiple clients to efficiently secret-share their input to the servers, we design a secure protocol for input sharing. The input comprises the graph represented as an adjacency matrix and data (features) associated with the vertices of the graph. Note that a client's input only comprises a partial view of the entire graph (i.e., a subset of vertices together with the corresponding data, and their associated edges). Hence, ensuring that the input generated at the servers indeed corresponds to an adjacency matrix, and the associated data adheres to the structure as required for a GCN is challenging. Designed independent of the graph algorithm, our input-sharing protocol to generate secret-shares of the adjacency matrix and the associated data may find use in other graph-based applications too.

Note that GraphSC operates on a list representation of the graph. Hence, we additionally describe the method to translate the adjacency matrix-based representation of the graph, generated as part of the input-sharing phase in secret shares, into its list representation. Our shuffle protocol also finds use in performing this translation. Note that the adjacency matrix has $|\mathcal{V}|^2$ entries that account for every possible edge while the list representation only stores information regarding edges that are actually present. Hence, generating the $(|\mathcal{V}|+|\mathcal{E}|)$-sized list representation of the graph from the $|\mathcal{V}|^2$-sized matrix representation while hiding the graph topology is challenging.

2 Preliminaries

We use the following notation. Matrices are denoted by bold font capital letters, such as $\mathbf{M}$, where $\mathbf{M}_{ij}$ denotes the entry in matrix $\mathbf{M}$ present in i^{th} row and j^{th} column. Vectors are denoted by bold font small letters, such as $\mathbf{v}$.

2.1 Graph convolutional networks (GCN)

The celebrated result of Kipf and Welling [20] showcases how convolutions can be generalized to graphs. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a graph, where $\mathcal{V}$ is the set of n nodes, and $\mathcal{E}$ is the set of edges. Let $\mathbf{A}$ denote the adjacency matrix of $\mathcal{G}$ of dimension $n \times n$, and $\mathbf{X}$ denote the matrix of node features of dimension $n \times f$, where f denotes the number of features. The authors propose a simple GCN model which allows learning succinct representation of nodes in the graph. In the case of node classification via GCN, the task is to assign labels to the unlabelled nodes in $\mathcal{G}$ by learning from those nodes that are labelled. To achieve this, the GCN model is trained using the labelled nodes provided as input, which makes up the training phase. This entails generating labels for the (already labelled) nodes via the computations in the *forward pass*, and updating the model parameters by accounting for the difference in the generated labels and the true labels of the nodes via the computations in the *backward pass*. The trained model can then be used to perform inference (i.e., classification of unlabelled nodes). Each of these is described in detail next.
Forward pass In the forward pass, node representations $\mathbf{H}^{(i)}$, for the i^{th} layer are computed using the following equation[1].

$$\mathbf{H}^{(i)} = g^{(i)}(\hat{\mathbf{A}} \cdot \mathbf{H}^{(i-1)} \mathbf{W}^{(i-1)}) \tag{1}$$

Here $g^{(i)}$ denotes the activation function for layer i, and $\mathbf{H}^{(0)} = \mathbf{X}$, $\mathbf{W}^{(i)}$ is the weight matrix specific to layer i and $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$. Further, $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with self-loops, and $\tilde{\mathbf{D}}$ is the degree matrix for $\tilde{\mathbf{A}}$ (i.e., $\tilde{\mathbf{D}}$ stores the number of incident edges in $\tilde{\mathbf{A}}$ as the diagonal entry for each node). Matrices $\hat{\mathbf{A}}, \tilde{\mathbf{A}}, \tilde{\mathbf{D}}$ are all of dimension $n \times n$. Specifically, the work of [20] considers a two layer

[1]GCNs have a multilayered architecture. Informally $\mathbf{H}^{(i)}$ in the intermediate layers captures the feature maps (properties) of the nodes, and its dimensions may vary across the layers. For the final layer, it captures the likelihood of a node being mapped to a particular label and hence has the dimension of $n \times c$ where c denotes the number of class labels.

instantiation of Eq. (1), where the GCN model $\mathbf{Z}$ is given as a function of $\mathbf{X}, \mathbf{A}$ and is parameterised by the weight matrices $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ as given in Eq. (2).

$$\mathbf{Z} = \overbrace{\mathsf{Softmax}(\hat{\mathbf{A}} \cdot \underbrace{\mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})}_{\mathbf{H}^{(1)}} \mathbf{W}^{(1)})}^{\mathbf{H}^{(2)}} \tag{2}$$

Here, $\mathbf{W}^{(0)}$ has a dimension of $f \times h$, where h denotes the number of feature maps, whereas $\mathbf{W}^{(1)}$ is of dimension $h \times c$, where c is the number of labels to which the nodes of the graph will be mapped.

Backward pass Since the weight matrices are trainable parameters, these are updated in the backward pass by accounting for the error in the computed output representation $\mathbf{Z}$ and the true representation $\mathbf{Y}$, with respect to the labelled nodes in $\mathcal{G}$. This is captured using the cross-entropy error, denoted as $\mathcal{L}$, as given in Eq. (3). Here, both $\mathbf{Z}, \mathbf{Y}$ are of dimension $n \times c$, and L denotes the index set of labelled nodes in $\mathcal{G}$ and hence is a subset of $\{1, \ldots, n\}$.

$$\mathcal{L} = -\sum_{i \in L} \sum_{j=1}^{c} \mathbf{Y}_{ij} \ln \mathbf{Z}_{ij} \tag{3}$$

Given loss function $\mathcal{L}$, derivative of $\mathcal{L}$ with respect to weight matrix $\mathbf{W}^{(0)}$, and derivative of $\mathcal{L}$ with respect to $\mathbf{W}^{(1)}$ is computed. It is used to update weight matrices via stochastic gradient descent optimizer [20] and completes backward pass.

Traning and inference Computing the forward pass, followed by the backward pass, constitutes one epoch of the training phase. Computations over several such epochs minimize $\mathcal{L}$, thereby yielding the trained weight matrices. Having generated the trained model, performing GCN inference for node classification is now possible by computing the forward pass (Eq. (2)), using the obtained weight matrices[2].

Heteregeneous GCN While the GCN of [20] deals with a single type of edge, the work of [27] designs heterogeneous GCNs. The main difference lies in the fact that the underlying graph now consists of different types of edges, $\mathcal{D}$. Specifically, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can now be seen as a collection of $|\mathcal{D}|$ subgraphs $\{\mathcal{G}_d = (\mathcal{V}, \mathcal{E}_d)\}$, where each subgraph comprises all the vertices of $\mathcal{G}$, but edges of only type $d \in \mathcal{D}$. The heterogeneous graph representation $\{\mathcal{G}_d\}$ leads to $|\mathcal{D}|$ adjacency matrices $\{\mathbf{A}_d\}$, each of dimension $n \times n$. Thus, the heterogeneous GCN computation [27] is given as $\mathbf{H}^{(0)} = \mathbf{0}$ and for $i = 1, \ldots, T$:

$$\mathbf{H}^{(i)} = g^{(i)} \left(\mathbf{X} \cdot \mathbf{W}^{(i-1)} + \frac{1}{|\mathcal{D}|} \sum_{d=1}^{|\mathcal{D}|} \mathbf{A}_d \cdot \mathbf{H}^{(i-1)} \cdot \mathbf{W}_d^{(i-1)} \right) \tag{4}$$

Here, $\mathbf{0}$ denotes the matrix of all 0s of dimension $n \times c$ where c is the number of labels, T denotes the number of layers, and $g^{(i)}$ denotes the non-linear activation function for layer i. $\mathbf{W}^{(i)}, \mathbf{W}_d^{(i)}$ are the i^{th} layer weight matrices of dimension $c \times c$, where the latter additionally depends on the edge type d. To draw an analogy to the GCN of Kipf and Welling [20], note that the final output $\mathbf{Z}$ computed in [20] is equivalent to $\mathbf{H}^{(T)}$ which constitutes the final output for heterogeneous GCNs. Further, as seen from Eq. (4), heterogeneous GCNs additionally require summing the values over each of the subgraphs to account for the heterogeneity in the edges. The cross-entropy error for the backward pass is computed similarly to the GCN of [20], as given in Eq. (3). In our work, we take $T = 2$ and let the activation function $g^{(1)}, g^{(2)}$ be ReLU for the first layer and $\mathsf{Softmax}$ for the second layer, respectively.

[2]Since $\mathsf{Softmax}$ is used to normalize the final result between $[0, 1]$, note that inference can be performed without it.

2.2 Secure multiparty computation (MPC)

On the choice of Tetrad As described earlier, small-party MPC protocols with 2, 3, and 4 parties have been gaining importance lately due to the various customizations they allow resulting in highly efficient protocols. Honest majority among the parties (i.e., at most minority of the parties are corrupted by an adversary, leaving the majority of them being honest) is known to further enhance the efficiency since honest majority protocols do not require heavy public key primitives, which are otherwise required when there exists a dishonest majority among the parties. This rules out the 2-party computation (2PC) setting. Among 3PC and 4PC, we note that SWIFT [22] forms the state-of-the-art robust 3PC framework in the preprocessing paradigm that enables PPML inference, whereas Tetrad [23] is a robust 4PC framework that supports PPML inference as well as training. It is known that Tetrad outperforms SWIFT not only in terms of communication cost but also in terms of computation cost. Elaborately, the multiplication protocol of Tetrad has a communication cost of only 2 ring elements in the preprocessing phase, whereas SWIFT has a cost of 3 elements. Further, SWIFT relies on computationally heavy distributed zero-knowledge computation protocols, which are not required in Tetrad. This makes Tetrad a more efficient framework than SWIFT for computationally heavy applications such as GNNs. Moreover, since both the frameworks securely realize various primitives such as ReLU, $\mathsf{Softmax}$, etc. by relying on the same algorithms, both Tetrad and SWIFT are on par in terms of the accuracy of these primitives. Moreover, Tetrad outperforms other 4PC frameworks such as [10] not only in terms of efficiency but also in terms of security guarantees. Further, it also outperforms the 4PC framework of [11] in terms of efficiency. In this way, Tetrad forms the state-of-the-art framework in the 4PC setting. Hence, keeping in mind the goal of attaining an efficient protocol while not compromising on accuracy, we choose to build $\mathsf{Entrada}$ over the 4PC framework of Tetrad.

System and threat model We consider secure outsourced computation in the 4-party setting of Tetrad [23], where four hired servers enact the role of parties $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$. We let $\mathcal{A}$ denote a static malicious probabilistic polynomial time adversary which corrupts at most one party in $\mathcal{P}$. Each client (possibly malicious) secret-shares its input among servers, who evaluate the MPC protocol to obtain the secret-shared output. The output is then reconstructed towards the client.

Similar to Tetrad, our constructions are secure in the real-world/ideal-world simulation paradigm. Further, Entrada, similar to Tetrad, enables achieving two different levels of security: (i) fairness—depending on adversary's misbehaviour, either all parties obtain the output of the computation, or none do, and (ii) robustness—regardless of adversary's misbehaviour, all parties are guaranteed to obtain the output of the computation. In the remainder paper, we focus on describing robust protocols. Their fair variants can be attained easily and is described in A.

Overview of Tetrad [23] We begin with describing the secret-sharing semantics in Tetrad. A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be secret-shared (or $[\![\cdot]\!]$-shared) if there exists mask $\lambda_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ and masked value $\mathsf{m}_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$ is held by parties in $\{P_1, P_2, P_3\}$, and λ_v is (3, 1) replicated secret-shared (or $\langle\cdot\rangle$-shared) among parties in $\{P_1, P_2, P_3\}$, and all its shares are known on clear to P_0. Elaborately, there exist values $\lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v}, \in \mathbb{Z}_{2^\ell}$ such that $\lambda_\mathsf{v} = \lambda^1_\mathsf{v} + \lambda^2_\mathsf{v} + \lambda^3_\mathsf{v}$, which together with m_v is distributed among the parties as follows: P_0 holds $\lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v}$; P_1 holds $\mathsf{m}_\mathsf{v}, \lambda^1_\mathsf{v}, \lambda^3_\mathsf{v}$; P_2 holds $\mathsf{m}_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v}$; and P_3 holds $\mathsf{m}_\mathsf{v}, \lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}$. Linear operations such as $[\![\mathsf{c}_1\mathsf{x} + \mathsf{c}_2\mathsf{y}]\!]$ given $[\![\mathsf{x}]\!], [\![\mathsf{y}]\!]$ where $\mathsf{c}_1, \mathsf{c}_2 \in \mathbb{Z}_{2^\ell}$ are public constants can be performed non-interactively by computing $\mathsf{c}_1[\![\mathsf{x}]\!] + \mathsf{c}_2[\![\mathsf{y}]\!]$. Boolean secret-sharing over $\mathbb{Z}_2$, denoted as $[\![\cdot]\!]^{\mathbf{B}}$, is similar to $[\![\cdot]\!]$ except that addition operation is replaced with XOR. In general, the Boolean world is analogous to the arithmetic world, with arithmetic addition and multiplication operations being replaced with Boolean XOR and AND. Arithmetic equivalent of a bit b in ring $\mathbb{Z}_{2^\ell}$ is denoted as b^R.

We rely on several protocols provided in Tetrad such as joint message passing (Π_{jmp}), joint sharing (Π_{jsh}), multiplication, to name a few, which are described in Table 8 (§A). Tetrad enables non-interactive generation of common random values among parties via shared PRF keys generated during a one-time setup phase (abstracted via an ideal functionality $\mathcal{F}_{\mathsf{setup}}$ in Fig. 7). This facilitates non-interactive generation of $[\![\cdot]\!]$ and $\langle\cdot\rangle$-shares for random values.

2.3 GraphSC paradigm

Let $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathsf{Data})$ denote a data augmented graph, where $\mathcal{V}$ denotes the set of vertices (or nodes, used interchangeably), $\mathcal{E}$ is the set of edges and Data is a set of user-defined data values associated with each vertex and edge of the graph. GraphSC operates on a list based representation of the data augmented graph such that each entry in the list G stores information corresponding to the vertices and edges in the graph. The $O(|\mathcal{V}| + |\mathcal{E}|)$ space complexity of G makes it an efficient alternative to the $O(|\mathcal{V}|^2)$ representation via matrix. Given G, evaluating a graph algorithm entails multiple invocations of the primitive operations of Scatter and Gather. The operations in Scatter allow the vertices to propagate information through the edges, while that in Gather allow aggregating this information at the vertices. Efficiently performing Scatter requires ordering the entries in G such that each entry corresponding to a vertex is followed by entries corresponding to its outgoing edges, known as *source ordering*. This allows linearly scanning through G to perform a Scatter across all the vertices. Similarly, Gather requires ordering G such that all entries corresponding to the incoming edges appears before the corresponding vertex, known as *destination ordering*. A linear scan through G allows to aggregate data at all the vertices via Gather[3]. In this way, message-passing graph algorithms are expressed as a composition of Scatter-Gather.

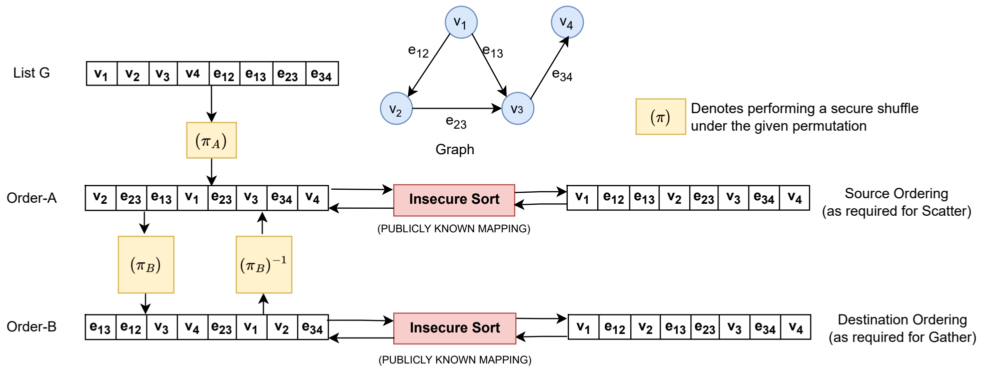
The GraphSC paradigm [5, 32] describes a way to securely evaluate graph algorithms by providing secure realizations of the Scatter-Gather primitives. To perform secure computation over the graph while hiding its topology, G is secret-shared entry-wise between the computing parties. Each entry in G corresponding to a vertex $\mathsf{v} \in \mathcal{V}$ is encoded as a tuple $(\mathsf{v}, \mathsf{v}, 1, \mathsf{data})$ and that of an edge $(\mathsf{u}, \mathsf{v}) \in \mathcal{E}$ is encoded as a tuple $(\mathsf{u}, \mathsf{v}, 0, \mathsf{data})$. The third entry in each tuple is a bit, isV, which equals 1 for a vertex and 0 otherwise. Once secret shared, the parties cannot distinguish between shares of a vertex entry and that of an edge. Rather than relying on secure sort to switch between source and destination ordering as required of Scatter and Gather, [5] observes that an efficient alternative is to rely on a secure shuffle followed by an insecure sort (where only the result of comparing two secret shared values is revealed). Moreover, the insecure sort is required to be performed only once in the beginning, subsequent to which, the public permutation (obtained as output from the insecure sort) can be used to sort G, non-interactively. This change brings in significant efficiency improvements to the GraphSC paradigm in [32]. An illustration of the operations involved in GraphSC paradigm of [5] is given in Fig. 1. The efficiency can be further enhanced via the parallel variants of Scatter-Gather described in [32] that leverage a multiprocessor setting. This reduces the $O(|\mathcal{V}| + |\mathcal{E}|)$ complexity required for the linear scan to a sub-linear solution in the multiprocessor setting. We refer interested readers to [32] for further details of the parallel variant and §A for summary of the GraphSC paradigm.

3 Secure GCN

Recall that while securely evaluating GCNs, the inputs comprise the matrices $\tilde{\mathbf{D}}$, $\tilde{\mathbf{A}}$, $\hat{\mathbf{A}}$, $\mathbf{X}$ and $\mathbf{Y}$, in secret shares. We begin by describing the input-sharing phase which comprises the steps to obtain these matrices (in secret-shares) from the clients. Following this, we discuss the steps to securely evaluate GCN. Since output reconstruction follows from Tetrad, we do not highlight the same.

Input sharing This phase involves generating $[\![\cdot]\!]$-shares of $\mathbf{A}, \mathbf{X}, \mathbf{Y}$. Given these matrices, the other inputs required for

[3]We assume data is propagated through outgoing edges and aggregated via incoming edges.



List G
v1 v2 v3 v4 e12 e13 e23 e34
(πA)
Order-A
v2 e23 e13 v1 e23 v3 e34 v4
Insecure Sort
(PUBLICLY KNOWN MAPPING)
v1 e12 e13 v2 e23 v3 e34 v4
Source Ordering (as required for Scatter)
(πB)
(πB)^{-1}
Order-B
e13 e12 v3 v4 e23 v1 v2 e34
Insecure Sort
(PUBLICLY KNOWN MAPPING)
v1 e12 v2 e13 e23 v3 e34 v4
Destination Ordering (as required for Gather)
v1 v2 v3 v4 e12 e13 e23 e34
Graph
(π) Denotes performing a secure shuffle under the given permutation

Figure 1: Computations via GraphSC. Note that each G entry v_i is a tuple $(\mathsf{v}_i, \mathsf{v}_i, 1, \mathsf{Data}_i)$ and entry e_{ij} is a tuple $(\mathsf{v}_i, \mathsf{v}_j, 0, \mathsf{Data}_{ij})$.

GCN evaluation, i.e., $\tilde{\mathbf{D}}$, $\tilde{\mathbf{A}}$, $\hat{\mathbf{A}}$, can be generated without the involvement of the client. Recall that in a distributed setting, the client may possess only a partial view of the input, i.e., information regarding some k nodes in the graph which correspond to k rows of $\mathbf{A}, \mathbf{X}$ and $\mathbf{Y}$ (see the use case in §1). Thus, each client secret-shares entries corresponding to the rows they possess towards the servers. Having received the entries of all rows from the clients, the servers generate the complete matrices by stacking up the rows (assuming the mapping between clients and rows of the matrices is known to servers). We give a high-level overview of the challenges in achieving this and their resolutions next.

For each of the inputs, servers must ensure that (a malicious) client C consistently $[\![\cdot]\!]$-shares each element x in the rows of the input matrix it possesses. This can be performed similar to [22], where– (i) servers non-interactively generate $\langle\cdot\rangle$-shares of the mask $\lambda_{\mathsf{x}} \in \mathbb{Z}_{2^\ell}$ and communicate the same to the client, (ii) communication complexity is reduced by ensuring one of the servers communicates only the hash corresponding to multiple shares, (iii) owing to the presence of at most one corrupt server, taking a majority over the received values enables C to obtain the correct value for each λ_{x}^i. C then computes $\lambda_{\mathsf{x}} = \lambda_{\mathsf{x}}^1 + \lambda_{\mathsf{x}}^2 + \lambda_{\mathsf{x}}^3$, $\mathsf{m}_{\mathsf{x}} = \mathsf{x} + \lambda_{\mathsf{x}}$, and sends m_{x} to servers P_1, P_2, P_3. To ensure that C has sent the consistent m_{x} to P_1, P_2, P_3, they exchange the value received from C among themselves. Since at most one among P_1, P_2, P_3 can be corrupt, there will exist a majority in the exchanged values, which is taken as the final value for m_{x}.

For structured inputs like $\mathbf{A}$ and $\mathbf{Y}$, additional checks are required. To verify if the resultant $[\![\mathbf{A}]\!]$ is symmetric, servers reconstruct $\mathbf{A} + \mathbf{R}$, and verify if this is symmetric. Here $\mathbf{R}$ is a random symmetric matrix used to mask $\mathbf{A}$ from the servers and can be generated non-interactively (relying on keys established via $\mathcal{F}_{\mathsf{setup}}$, §2.2). Next, to check if each entry in $\mathbf{A}$ is a 0/1, servers check if $\mathbf{A}_{ij}^2 - \mathbf{A}_{ij} = 0$. Similarly, since each row of $\mathbf{Y}$ has at most one position set as 1 (and all others as 0s), servers need to verify if the ith row $\{\mathbf{Y}_{i1}, \mathbf{Y}_{i2}, \ldots, \mathbf{Y}_{ic}\}$ satisfies this condition. For this, following along the idea of [28], the servers check if $\left(\sum_{j=1}^{c} \mathbf{Y}_{ij} \cdot \mathsf{r}_j\right)^2 - \left(\sum_{j=1}^{c} \mathbf{Y}_{ij} \cdot \mathsf{r}_j^2\right) = 0$. Here, $\mathsf{r}_j \in \mathbb{Z}_{2^\ell}$ are random public values. Having generated $[\![\mathbf{A}]\!]$, generation of $[\![\tilde{\mathbf{A}}]\!] = [\![\mathbf{A}]\!] + [\![\mathbf{I}]\!]$ can happen non-interactively. $[\![\cdot]\!]$-shares of diagonal entries of $\tilde{\mathbf{D}}$ are also generated non-interactively by summing entries in the corresponding rows of $[\![\tilde{\mathbf{A}}]\!]$. Finally, computing $[\![\cdot]\!]$-shares of $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$, involves computing element-wise inverse square-root of diagonal elements of $\tilde{\mathbf{D}}$ via secure inverse square-root protocol followed by matrix multiplications. Further details about input sharing are elaborated in §B.

Secure evaluation of GCN This entails servers securely computing secret-shares of $\mathbf{Z}$ via the forward pass followed by computing the derivative of $\mathcal{L}$ with respect to the weight matrices, as defined in the backward pass. Assuming the inputs are available in secret-shares, servers begin by initializing random weight matrices $\mathbf{W}^{(0)}$ and $\mathbf{W}^{(1)}$, which can be done non-interactively using keys established via $\mathcal{F}_{\mathsf{setup}}$ (see §2.2). Secure protocols are then required for matrix multiplication, ReLU and Softmax to compute shares of $\mathbf{Z}$. Additionally, a secure protocol for DReLU is required during back propagation to compute the derivative of loss with respect to the weights $\left(\frac{\delta\mathcal{L}}{\delta\mathbf{W}}\right)$. Next, using the secret-shares of $\mathbf{Z}$ and $\left(\frac{\delta\mathcal{L}}{\delta\mathbf{W}}\right)$, servers update weight matrices. Although [20] relies on gradient descent to update weights, due to drawbacks such as slow convergence and the possibility of converging to a local minimum, we rely on the optimized alternative of Adam [19] optimizer. The weight update computations within Adam, and the overall steps to evaluate a GCN are summarised in Fig. 2.

To securely evaluate GCN, each step in Fig. 2 is realized via its secure counterpart. Elaborately, for matrix multiplication and ReLU, we rely on the secure protocols from Tetrad [23]. Although Tetrad does not give an explicit protocol for DReLU, we note that it can be computed as $\mathsf{DReLU}(x) = \mathbf{1}(x > 0)$ using the comparison protocol from Tetrad. For Softmax, we observe that Tetrad relies on an approximate variant for it. Keeping GCN accuracy as needed for real-world applications in mind, we instead compute $\mathsf{Softmax}(\mathsf{x}) = e^{\mathsf{x}} / (\sum_i e^{\mathsf{x}_i})$, which is the more accurate definition. This computation requires secure protocols for exponentiation and division. While Tetrad does not support exponentiation, for division it relies on a garbled circuit (GC) based approach, which is known to be expensive in terms of communication cost [4, 16]. Hence, we provide efficient protocols for exponentiation as well as division. Moreover, Tetrad does not provide support for using Adam optimizer since it lacks square root primitive. Hence, we also design secure and efficient protocols for the same. We elaborate on our additions over Tetrad in §5. Although our approach provides efficiency and accuracy improvements in comparison to realizing GCNs via Tetrad (using SGD), we additionally incorporate the GraphSC paradigm in Entrada to further enhance efficiency.

```
Algorithm GCN evaluation

Training Phase
Input: $\hat{\mathbf{A}}$ (normalized adjacency matrix of dimension $n \times n$), $\mathbf{X}$ (feature matrix of dimension $n \times f$), $\mathbf{Y}$ (matrix of true output for labelled nodes of dimension $n \times c$), $\#epochs$ (number of epochs).
Output: $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ (trained weight matrices for layer 0,1, respectively).
  //Initialization
  – Randomly sample $\mathbf{W}^{(0)} \in \mathbb{R}^{f \times h}$, and $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times c}$, where $\mathbb{R}$ denotes the set of real numbers. Set $\beta_1 = 0.9, \beta_2 = 0.999, \mathbf{M}^{(0)} = \mathbf{M}^{(1)} = 0, \mathbf{V}^{(0)} = \mathbf{V}^{(1)} = 0$, $\varepsilon = 10^{-8}$.
For $t$ in range($\#epochs$) do:
  //Forward pass
  – $\mathbf{H}^{(0)} = \mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})$, $\mathbf{In} = \hat{\mathbf{A}}\mathbf{H}^{(0)}\mathbf{W}^{(1)}$, $\mathbf{Z} = \mathsf{Softmax}(\mathbf{In})$
  //Backward pass
  – $\left(\frac{\delta\mathcal{L}}{\delta\mathbf{W}^{(0)}}\right) = (\mathbf{H}^{(1)})^{\mathsf{T}}(\mathbf{Z} - \mathbf{Y})$
  – $\left(\frac{\delta\mathcal{L}}{\delta\mathbf{W}^{(1)}}\right) = (\hat{\mathbf{A}}\mathbf{X})^{\mathsf{T}}[\mathsf{DReLU}(\mathbf{In}) \odot \hat{\mathbf{A}}(\mathbf{Z} - \mathbf{Y})(\mathbf{W}^{(1)})^{\mathsf{T}}]$
  //Weights update
  – $\mathbf{M}^{(i)} = \beta_1\mathbf{M}^{(i)} + (1 - \beta_1)\left(\frac{\delta\mathcal{L}}{\delta\mathbf{W}^{(i)}}\right)$ for layer $i \in \{0, 1\}$
  – $\mathbf{V}^{(i)} = \beta_2\mathbf{V}^{(i)} + (1 - \beta_2)\left(\frac{\delta\mathcal{L}}{\delta\mathbf{W}^{(i)}}\right)^2$ for layer $i \in \{0, 1\}$
  – $\hat{\mathbf{M}}^{(i)} = \frac{\mathbf{M}^{(i)}}{1-(\beta_1)^t}$ and $\hat{\mathbf{V}}^{(i)} = \frac{\mathbf{V}}{1-(\beta_2)^t}$, for layer $i \in \{0, 1\}$
  – $\mathbf{W}^{(i)} = \mathbf{W}^{(i)} - \hat{\mathbf{M}}^{(i)}\left(\frac{\alpha}{\sqrt{\hat{\mathbf{V}}^{(i)}+\varepsilon}}\right)$, where $\alpha$ is learning rate.

Inference Phase
Input: $\hat{\mathbf{A}}$ (normalized adjacency matrix of dimension $n \times n$), $\mathbf{X}$ (feature matrix of dimension $n \times f$), $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ (trained weight matrices of dimensions $h \times f$ and $f \times c$, respectively).
Output: $\mathbf{Z}$ (matrix of dimension $n \times c$ that captures the likelihood of unlabelled nodes belonging to each class/label).
  – $\mathbf{Z} = \hat{\mathbf{A}} \cdot \mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})\mathbf{W}^{(1)}$
```

Figure 2: Steps involved in training and inference phase of GCN

4 GCN evaluation via GraphSC

Recall that computations via GraphSC require performing shuffles and invocations of Scatter-Gather operations across multiple rounds. Since we instantiate the MPC for GraphSC via Entrada, we first describe our shuffle protocol over the same. Following this, we discuss the components of GCN evaluation that can be cast in the message-passing paradigm, and subsequently define the Scatter-Gather primitives for the same. This entails defining GCN computations in a vertex-centric manner, unlike matrix operations, as described in §2.1. While the forward pass of GCN can be entirely computed within GraphSC, for the backward pass, the computation of derivatives of loss function benefits from GraphSC paradigm. Thus, other computations such as updating the weight matrices are performed outside GraphSC. We conclude by discussing the steps for generating the shares of the graph list G, as required for GraphSC, from the matrix representation.

4.1 Secure shuffle

Consider a $[\![\cdot]\!]$-shared N-sized vector $\mathbf{v}$, where each element $\mathsf{v}_i \in \mathbf{v}$ is $[\![\cdot]\!]$-shared. The goal of secure shuffle protocol is to generate a $[\![\cdot]\!]$-shared vector $\mathbf{u}$ such that it comprises the elements in $\mathbf{v}$ shuffled under a secret random permutation π. We denote this operation as $\mathbf{u} = \pi(\mathbf{v})$, where $\mathbf{u}$ is the vector of elements $\mathbf{v}_{\pi(1)}, \mathbf{v}_{\pi(2)}, \ldots, \mathbf{v}_{\pi(\mathsf{N})}$. To ensure that π is secret and hidden from all parties, following along the lines of [24] we define π to be a composition of four permutations $\pi = \pi_0 \circ \pi_3 \circ \pi_1 \circ \pi_2$ such that each party P_i misses the permutation π_i, and $\circ$ denotes the composition operation. The justification for the ordering among the permutations is made clear later and follows from the construction of our shuffle protocol. Since the protocol heavily relies on the sharing semantics of Tetrad and Π_{jmp} (enables $P_i, P_j \in \mathcal{P}$ to send v to P_k such that P_k receives the correct v, or a conflicting pair of parties among P_i, P_j, P_k is identified), Π_{jsh} (enables $P_i, P_j \in \mathcal{P}$ to generate $[\![\mathsf{v}]\!]$ where $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is held by P_i, P_j), we refer the readers to §A to familiarize themselves with the same.

As per $[\![\cdot]\!]$-sharing semantics, $\mathbf{u} = \mathbf{m}_{\mathsf{u}} - \lambda_{\mathsf{u}}$, where λ_{u} is $\langle\cdot\rangle$-shared. Thus, to generate $[\![\mathbf{u}]\!]$, our goal is to generate $\mathbf{m}_{\mathsf{u}}, \lambda_{\mathsf{u}} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ such that $\mathbf{u} = \pi(\mathbf{v}) = \pi(\mathbf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) = \pi(\mathbf{m}_{\mathsf{v}}) - \pi(\lambda_{\mathsf{v}}) = \mathbf{m}_{\mathsf{u}} - \lambda_{\mathsf{u}}$, and λ_{u} is $\langle\cdot\rangle$-shared. We explain the generation of $[\![\mathbf{u}]\!]$ into two parts– (1) assuming that parties have generated $[\![\cdot]\!]$-shares of $\mathbf{w} = \pi'(\mathbf{v})$ where $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$, we explain how $[\![\cdot]\!]$-shares of $\mathbf{u} = \pi_0(\mathbf{w})$ can be generated (observe here that $\mathbf{u} = \pi(\mathbf{v}) = \pi_0(\mathbf{w})$ holds true since $\pi = \pi_0 \circ \pi'$), (2) we then explain how $[\![\cdot]\!]$-shares of $\mathbf{w} = \pi'(\mathbf{v})$ can be generated where $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$.

We now describe how to realize (1) assuming that $[\![\cdot]\!]$-shares of $\mathbf{w}$ are available, i.e., $\mathbf{w} = \mathbf{m}_{\mathsf{w}} - \lambda_{\mathsf{w}}$ where λ_{w} is $\langle\cdot\rangle$-shared and can be written as $\lambda_{\mathsf{w}} = \lambda_{\mathsf{w}}^1 + \lambda_{\mathsf{w}}^2 + \lambda_{\mathsf{w}}^3$. Observe that $\mathbf{u} = \pi_0(\mathbf{w}) = \pi_0(\mathbf{m}_{\mathsf{w}} - \lambda_{\mathsf{w}}) = \pi_0(\mathbf{m}_{\mathsf{w}}) - \pi_0(\lambda_{\mathsf{w}}^1) - \pi_0(\lambda_{\mathsf{w}}^2) - \pi_0(\lambda_{\mathsf{w}}^3)$. Thus, $[\![\cdot]\!]$-shares of $\mathbf{u}$ can be generated by linearly combining the $[\![\cdot]\!]$-shares of $\pi_0(\mathbf{m}_{\mathsf{w}}), \pi_0(\lambda_{\mathsf{w}}^1), \pi_0(\lambda_{\mathsf{w}}^2), \pi_0(\lambda_{\mathsf{w}}^3)$. To generate the $[\![\cdot]\!]$-shares of $\pi_0(\mathbf{m}_{\mathsf{w}})$, observe that P_1, P_2, P_3 hold π_0 as well as $\mathbf{m}_{\mathsf{w}}$. Hence, parties can generate $[\![\cdot]\!]$-shares of $\pi_0(\mathbf{m}_{\mathsf{w}})$ non-interactively in the online phase by retaining the masked value as $\pi_0(\mathbf{m}_{\mathsf{w}})$ and setting the mask as 0 (i.e., $\langle\cdot\rangle$-shares of λ are set to be 0). On the other hand, each of the remainder terms $\pi_0(\lambda_w^i)$ for $i \in \{1,2,3\}$ is held by a distinct pair of parties in P_1, P_2, P_3. Hence, in the preprocessing phase, the corresponding pair of parties invoke Π_{jsh} to generate $[\![\cdot]\!]$-shares of $\pi_0(\lambda_w^i)$ among all the parties. This completes generation of $[\![\cdot]\!]$-shares of $\mathbf{u} = \pi_0(\mathbf{w})$.

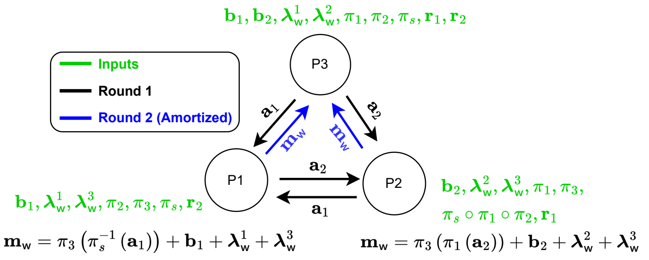
We now explain how $[\![\cdot]\!]$-shares of $\mathbf{w} = \pi'(\mathbf{v})$ are generated where $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$ and $\mathbf{v} = \mathbf{m}_v - \lambda_v$ such that λ_v is $\langle\cdot\rangle$-shared. One of our main goals in realizing this is to minimize the complexity of the online phase. Towards this, our high level approach to generate $[\![\cdot]\!]$-shares of $\mathbf{w}$ is to generate $\langle\lambda_w\rangle$ non-interactively, and define $\mathbf{m}_w$ as $\mathbf{m}_w = \pi'(\mathbf{m}_v) - \pi'(\lambda_v) + \lambda_w$. During the generation of the $[\![\cdot]\!]$-shares of $\mathbf{w}$, we maintain the invariant that every message to be communicated is held by two parties, which allows invoking the Π_{jmp} protocol. This guarantees correctness of the generated shares since invocation of Π_{jmp} ensures that any misbehaviour by a malicious party can be detected. In the following, we first explain how $\langle\cdot\rangle$-shares of λ_w can be generated followed by generation of $\mathbf{m}_w$ towards P_2, P_1, P_3. Since composition of permutations in $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$ is non-commutative, generation of $\mathbf{m}_w$ is handled differently towards each party.

Generation of $\langle\cdot\rangle$-shares of λ_w. Relying on keys generated via $\mathcal{F}_{\mathsf{setup}}$, parties non-interactively generate $\langle\cdot\rangle$-shares of λ_w in the preprocessing phase, i.e., P_0, P_1, P_3 sample $\lambda_w^1 \in \mathbb{Z}_{2^\ell}^N$, P_0, P_2, P_3 sample $\lambda_w^2 \in \mathbb{Z}_{2^\ell}^N$, and P_0, P_1, P_2 sample $\lambda_w^3 \in \mathbb{Z}_{2^\ell}^N$.

Generation of $\mathbf{m}_w$ *towards* P_2. We begin with the case of generating $\mathbf{m}_w = \pi'(\mathbf{m}_v) - \pi'(\lambda_v) + \lambda_w$ towards P_2. Consider the first summand $\pi'(\mathbf{m}_v) = \pi_3(\pi_1(\pi_2(\mathbf{m}_v)))$. To compute this term, P_2 only misses π_2. Thus, if $\pi_2(\mathbf{m}_v)$ can be made available to P_2, it can compute the first summand. Since $\pi_2(\mathbf{m}_v)$ is held by both P_1, P_3, they can invoke Π_{jmp} to send $\pi_2(\mathbf{m}_v)$ to P_2. However, it is required that P_2 does not learn anything about π_2. Hence, P_1, P_3 sample a random $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^N$, and instead send $\mathbf{a}_2 = \pi_2(\mathbf{m}_v + \mathbf{r}_2)$ to P_2, who can then compute $\pi_3(\pi_1(\mathbf{a}_2)) = \pi'(\mathbf{m}_v) + \pi'(\mathbf{r}_2)$. In the computation of $\mathbf{m}_w$, to remove $\pi'(\mathbf{r}_2)$ and account for the missing summands $\pi'(\lambda_v), \lambda_w^1$, P_2 must be provided with $\mathbf{b}_2 = \lambda_w^1 - \pi'(\lambda_v) - \pi'(\mathbf{r}_2)$. Obtaining $\mathbf{b}_2$ allows P_2 to compute $\mathbf{m}_w = \pi_3(\pi_1(\mathbf{a}_2)) + \mathbf{b}_2 + \lambda_w^2 + \lambda_w^3$. Observe that since $\mathbf{b}_2$ is independent of the input, it can be generated towards P_2 in the preprocessing phase (as discussed later). Thus, generating $\mathbf{m}_w$ towards P_2 requires communication of a single message $\mathbf{a}_2$ in one round in the online phase.

Generation of $\mathbf{m}_w$ *towards* P_1. Similar to above, a simple way of generating $\mathbf{m}_w$ towards P_1 is to now make P_2, P_3 send $\pi_1(\mathbf{a}_2 + \mathbf{r}_1)$ to P_1, where $\mathbf{r}_1 \in \mathbb{Z}_{2^\ell}^N$ is sampled randomly by P_2, P_3, and serves as a mask to hide π_1 from P_1. Party P_1 can apply π_3 on the received value and add it with analogous preprocessed terms (similar to $\mathbf{b}_2$) as described earlier to obtain $\mathbf{m}_w$. However, this approach requires an additional round of communication since P_1 requires to wait until P_2 receives $\mathbf{a}_2$ in the prior round. Instead, our approach is to enable P_1 to compute $\mathbf{m}_w$ in the same round as P_2. For this, in the preprocessing phase we make available towards P_2, P_3 the permutation $\pi_s \circ \pi_1 \circ \pi_2$ where π_s is a random permutation known to P_1, P_3. Observe that presence of π_s in $\pi_s \circ \pi_1 \circ \pi_2$, ensures that π_2 remains hidden from P_2. Given this permutation, generation of $\mathbf{m}_w$ towards P_1 proceeds as follows. P_2, P_3 compute and send $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\mathbf{m}_v + \mathbf{r}_1)))$ via Π_{jmp} to P_1. Given $\mathbf{b}_1 = \lambda_w^2 - \pi'(\lambda_v) - \pi'(\mathbf{r}_1)$ is generated towards P_1 in the preprocessing phase (as discussed later), P_1 can compute $\mathbf{m}_w = \pi_3(\pi_s^{-1}(\mathbf{a}_1)) + \mathbf{b}_1 + \lambda_w^1 + \lambda_w^3$. Note that P_1 can compute π_s^{-1} since it has π_s on clear. In this way, generating $\mathbf{m}_w$ towards P_1 requires communicating the message $\mathbf{a}_1$ and no additional rounds in the online phase.

Generation of $\mathbf{m}_w$ *towards* P_3. Having generated $\mathbf{m}_w$ towards P_1, P_2, they send it to P_3 via Π_{jmp}. Note that in scenarios such as GraphSC, which demand several shuffle invocations, sending of $\mathbf{m}_w$ towards P_3 with respect to all shuffle instances can be performed in a single round, thereby amortizing the cost of this round across several shuffle instances[4]. In such scenarios, our shuffle protocol requires only a single round of interaction per shuffle instance in the online phase. A pictorial representation appears in Fig. 3, where arrows capture the communication via Π_{jmp}.



$\mathbf{b}_1, \mathbf{b}_2, \boldsymbol{\lambda}_w^1, \boldsymbol{\lambda}_w^2, \pi_1, \pi_2, \pi_s, \mathbf{r}_1, \mathbf{r}_2$
Inputs
Round 1
Round 2 (Amortized)
P3
$\mathbf{a}_1$
$\mathbf{m}_w$
$\mathbf{m}_w$
$\mathbf{a}_2$
P1
$\mathbf{a}_2$
$\mathbf{a}_1$
P2
$\mathbf{b}_1, \boldsymbol{\lambda}_w^1, \boldsymbol{\lambda}_w^3, \pi_2, \pi_3, \pi_s, \mathbf{r}_2$
$\mathbf{b}_2, \boldsymbol{\lambda}_w^2, \boldsymbol{\lambda}_w^3, \pi_1, \pi_3, \pi_s \circ \pi_1 \circ \pi_2, \mathbf{r}_1$
$\mathbf{m}_w = \pi_3(\pi_s^{-1}(\mathbf{a}_1)) + \mathbf{b}_1 + \boldsymbol{\lambda}_w^1 + \boldsymbol{\lambda}_w^3$
$\mathbf{m}_w = \pi_3(\pi_1(\mathbf{a}_2)) + \mathbf{b}_2 + \boldsymbol{\lambda}_w^2 + \boldsymbol{\lambda}_w^3$

Figure 3: Online phase of shuffle protocol for generating $\mathbf{m}_w$ where $\mathbf{w} = \pi'(\mathbf{v})$ and $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$.

Generation of additional preprocessing data. To facilitate the one-round online phase, we now discuss how additional data such as $\pi_s \circ \pi_1 \circ \pi_2$ and the terms $\mathbf{b}_1, \mathbf{b}_2$ can be generated in the preprocessing phase. For P_2, P_3 to generate $\Pi = \pi_s \circ \pi_1 \circ \pi_2$, parties P_0, P_1, P_3 randomly sample a permutation π_s[5]. P_0, P_3 can then compute Π locally since they hold π_1, π_2, and invoke Π_{jmp} to send Π to P_2.

For generating $\mathbf{b}_1, \mathbf{b}_2$, we extend the 3-party semi-honest shuffle protocol of [5] to work in our 4-party malicious setting. The modified protocol continues to have 2 rounds as in the case of [5] which is possible due to the following observation. The protocol of [5] takes as input $\langle\cdot\rangle$-shares of a vector $\mathbf{v}$ and outputs $\langle\cdot\rangle$-shares of $\pi'(\mathbf{v})$. Here, we can view $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$ which is shared among P_1, P_2, P_3 such that parties P_2, P_3 hold π_1, parties P_1, P_3 hold π_2, and parties P_1, P_2 hold π_3. While extending the protocol of [5] in the preprocessing phase of our 4-party setting, we note that P_0 has all the inputs held by

[4]Note that with respect to the framework of Tetrad, any communication towards P_3 can be deferred until output reconstruction. We refer to [23] for further details.

[5]Steps for randomly sampling a permutation are elaborated in §C.1.

P_1, P_2, P_3. This allows P_0 to compute all protocol messages communicated in [5]. In this way, these messages can be communicated by two senders, one of which is P_0, by invoking Π_{jmp}. Use of Π_{jmp} facilitates attaining malicious security.

We next describe our concrete protocol steps for generating $\mathbf{b}_2 = \lambda^1_{\mathsf{w}} - \pi'(\lambda_{\mathsf{v}} - \mathbf{r}_2)$ towards P_2. Observe that using the protocol of [5] allows generating $\langle\cdot\rangle$-shares of $\pi'(\lambda_{\mathsf{v}} - \mathbf{r}_2)$. However, the requirement is to generate $\lambda^1_{\mathsf{w}} - \pi'(\lambda_{\mathsf{v}} - \mathbf{r}_2)$ on clear towards P_2. Hence, we slightly modify the protocol steps to instead generate this value on clear towards P_2 which entails providing P_2 its missing $\langle\cdot\rangle$-share of $\pi'(\lambda_{\mathsf{v}} - \mathbf{r}_2)$ which is masked with λ^1_{w}. Since the protocol takes as input $\langle\cdot\rangle$-shares of $\lambda_{\mathsf{v}} - \mathbf{r}_2$, we first discuss how this is generated. Let $\alpha_2 = \lambda_{\mathsf{v}} - \mathbf{r}_2$, where $\mathbf{r}_2 \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ is sampled randomly by P_0, P_1, P_3. Parties non-interactively generate $\langle\cdot\rangle$-shares of $\mathbf{r}_2$ by P_0, P_1, P_3 setting their common share as $\mathbf{r}_2$, and the other shares being set as 0. Parties then generate $\langle\cdot\rangle$-shares of α_2 using $\langle\cdot\rangle$-shares of λ_{v} and $\mathbf{r}_2$, and the linearity property of $\langle\cdot\rangle$-sharing. Given $\langle\cdot\rangle$-shares of α_2, the high-level overview of the protocol is as follows. Parties non-interactively sample $\mathbf{z}_i \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ for $i \in \{1,2,3\}$. Specifically, P_0, P_1, P_3 sample $\mathbf{z}_2$, P_0, P_2, P_3 sample $\mathbf{z}_1$, while P_0, P_1, P_2 sample $\mathbf{z}_3$. The $\mathbf{z}_i$'s serve as a random mask to ensure that parties only see random values throughout the protocol execution. Parties compute messages $\mathbf{x}_i, \mathbf{y}_i$ for $i \in \{1,2,3\}$ such that the following invariant is maintained: each $\mathbf{x}_i + \mathbf{y}_i$ is always a shuffle of α_2. The protocol proceeds in rounds as described in Fig. 10. Note that all communicated messages can be computed by P_0, and hence are sent via Π_{jmp} which ensures security against a malicious adversary. This communication from P_0 is omitted in the figure. Correctness of $\mathbf{b}_2$ follows by opening up the corresponding values of $\mathbf{x}_i, \mathbf{y}_i$ for $i \in \{1,2,3\}$. Observe that generating $\mathbf{b}_2$ towards P_2 requires communicating 3 messages in 2 rounds.

The protocol proceeds analogously for the generation of $\mathbf{b}_1$ towards P_1, except that the computation happens with $\alpha_1 = \lambda_{\mathsf{v}} - \mathbf{r}_1$, and in the second round P_2 sends $\mathbf{y}_3 + \lambda^2_{\mathsf{w}}$ to P_1 (see §C.1 for the details). Note that generation of $\mathbf{b}_1$ towards P_1 can be performed in parallel to generating $\mathbf{b}_2$ towards P_2, and does not incur any additional rounds.

A schematic representation of the various parts in the generation of $\mathbf{u}$ is given in Fig. 9. The complete protocol for secure shuffle appears in Fig. 13, Fig. 14. The communication and round complexity of the shuffle protocol is provided in §C.1.

Security. Formal security proof appears in §F.

4.2 Scatter and gather primitives

For better readability, we define Scatter-Gather primitives in cleartext, while their secure versions can be obtained using the secure protocols for the operations therein. Recall that computation in the l^{th} layer during the *forward pass* is given as $\mathbf{H}^{(l)} = g^{(l)}(\hat{\mathbf{A}}\mathbf{H}^{(l-1)}\mathbf{W}^{(l-1)})$, where $g^{(l)}(\cdot)$ denotes the activation function. $\mathbf{H}^{(0)}$ is initialized to $\mathbf{X}$, and the final output $\mathbf{Z} = \mathbf{H}^{(2)}$. Although our goal is to compute $\mathbf{H}^{(l)}$ via Scatter-Gather, we will define Scatter-Gather to compute $\mathbf{H}^{(1)} = g^{(1)}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})$. Computation of $\mathbf{H}^{(2)}$ proceeds analogously. Hence, we omit the superscript in $\mathbf{H}^{(1)}$ and $\mathbf{W}^{(0)}$.

We begin by describing the data components that are required to be stored at each entry $\mathsf{G}[i]$ in the DAG-list G (all in secret-shares) to facilitate the computation. For a vertex entry, $\mathsf{G}[i].\mathsf{deg}$ stores the degree of the vertex which also accounts for the self-loop (as defined in $\tilde{\mathbf{D}}$) whereas $\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}$ stores its inverse square-root. The i^{th} row of $\mathbf{X}$ (represented as x_i), which denotes the feature vector associated with the i^{th} vertex, is stored at $\mathsf{G}[i].x$. Note that these components are 0 if $\mathsf{G}[i]$ represents an edge. Additionally, vectors $\mathsf{G}[i].\mathbf{dt}, \mathsf{G}[i].\mathbf{agg}$ are used to store intermediate results. We assume that $\mathbf{W}$ is accessible to all nodes in the graph. Given this information, the goal of Scatter-Gather is to compute the i^{th} row h_i of matrix $\mathbf{H} = g\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}\right)$ and store it at vertex entry $\mathsf{G}[i].h$. The j^{th} component of h_i can be computed as:

$$h_{ij} = \mathbf{H}_{ij} = g((\hat{\mathbf{A}}\mathbf{X}\mathbf{W})_{ij}) = g(\sum_{k=1}^{n} \hat{\mathbf{A}}_{ik}\,(\mathbf{X}\mathbf{W})_{kj})$$
$$= g\left(\sum_{k=1}^{n} \frac{\tilde{\mathbf{A}}_{ik}}{\mathsf{deg}_i^{\frac{1}{2}} \cdot \mathsf{deg}_k^{\frac{1}{2}}}\,(\mathbf{X}\mathbf{W})_{kj}\right) \tag{5}$$

The matrix operations in Eq. (5) for computing h_{ij} when performed via GraphSC would involve aggregating (across the various ks) the j^{th} component of $(x_k \cdot \mathbf{W})$, scaling the aggregated value by the degree terms, followed by application of $g(\cdot)$ on the same. Observe that this aggregation accounts only for the neighbours of node i since $\tilde{\mathbf{A}}_{ik} = 0$ otherwise. Thus, each node k can compute $\dfrac{x_k \cdot \mathbf{W}}{\mathsf{deg}_k^{\frac{1}{2}}}$ and *scatter* it across its edges. The node i can then *gather* these vectors from its neighbors, scale it using $\mathsf{deg}_i^{-\frac{1}{2}}$, and apply $g(\cdot)$ on this vector to generate h_i. In this way, one invocation of Scatter and Gather results in populating $\mathsf{G}[i].h$ and accomplishes the computation of $\mathbf{H}^{(1)} = g^{(1)}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}\right)$ in a vertex-centric manner. Similarly, $\mathbf{Z} = \mathbf{H}^{(2)}$ can also be computed. The formal protocols for Scatter-Gather appear in Fig. 4, whose secure variant can be obtained as described in §C.2. We remark that although the definitions of Scatter-Gather have a linear complexity in $|\mathcal{V}| + |\mathcal{E}|$, their sub-linear variant can be obtained using the technique of [33].

```
Scatter(G)

for $i = 1$ to $|\mathcal{V}| + |\mathcal{E}|$ do:
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathbf{v} = (\mathsf{G}[i].x) \cdot \mathbf{W} \cdot (\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}})$
  else
    $\mathsf{G}[i].\mathbf{dt} = \mathbf{v}$
```

```
Gather(G)

for $i = 1$ to $|\mathcal{V}| + |\mathcal{E}|$ do:
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathsf{G}[i].h = g\left((\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}) \cdot \mathbf{agg}\right)$
    $\mathbf{agg} = 0$
  else
    $\mathbf{agg} = \mathbf{agg} + \mathsf{G}[i].\mathbf{dt}$
```

Figure 4: Scatter **and** Gather **to compute** $\mathbf{H}^{(1)} = g^{(1)}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}\right)$ **in forward pass.**

Recall that in the *backward pass*, the derivative of the cross-entropy loss with respect to the weight matrices is computed using the output of the forward pass, $\mathbf{Z}$, as well as the target result $\mathbf{Y}$ for the training data. This is then used to update the weight matrices $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ via the Adam optimizer. We will now showcase how the computation of the derivatives (Eq. (6), Eq. (7)) can be performed efficiently via GraphSC primitives of Scatter and Gather.

$$\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}} = {\mathbf{H}^{(1)}}^{\mathsf{T}} \hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y}) \tag{6}$$

$$\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(1)}} = (\hat{\mathbf{A}}\mathbf{X})^{\mathsf{T}} \left(\mathsf{DReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y}){\mathbf{W}^{(1)}}^{\mathsf{T}}\right) \tag{7}$$

Here, $\mathbf{In} = \hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}$, $\mathbf{M}^{\mathsf{T}}$ denotes the matrix transpose operation, and $\bigodot$ is the element-wise multiplication operator. As in the case of the forward pass, we begin by describing the data components associated with $\mathsf{G}[i]$. Recall that as part of the forward pass, the i^{th} row of $\mathbf{Z} = \mathbf{H}^{(2)}$, denoted as z_i is already computed. Let $\mathsf{G}[i].z$ denote this component. Similarly, the i^{th} row of $\mathbf{H}^{(1)}$ as well as $\mathbf{In}$ can be made available via the computations performed in the forward pass. In addition to the data components that were a part of the forward pass, the i^{th} row of $\mathbf{Y}$ that corresponds to the label for the i^{th} node, can be stored as a data component, $\mathsf{G}[i].y$.

```
Scatter(G)

for $i = 1$ to $|\mathcal{V}| + |\mathcal{E}|$ do:
  if $\mathsf{G}[i]$.isV then
    $v_1 = \mathsf{G}[i].z - \mathsf{G}[i].y$
    $v_2 = \mathsf{G}[i].x$
    $v_3 = (\mathsf{G}[i].z - \mathsf{G}[i].y)$
        $\cdot {\mathbf{W}^{(1)}}^{\mathsf{T}}$
  else
    $\mathsf{G}[i].v_1 = v_1$
    $\mathsf{G}[i].v_2 = v_2$
    $\mathsf{G}[i].v_3 = v_3$
```

```
Gather(G)

$\mathbf{agg} = 0$
for $i = 1$ to $|\mathcal{V}| + |\mathcal{E}|$ do:
  if $\mathsf{G}[i]$.isV then
    $\mathsf{G}[i].v_1 = \left(\mathsf{G}[i].\text{deg}^{-\frac{1}{2}}\right) \mathbf{agg}_1$
    $\mathsf{G}[i].v_2 = \left(\mathsf{G}[i].\text{deg}^{-\frac{1}{2}}\right) \mathbf{agg}_2$
    $\mathsf{G}[i].v_3 = \mathsf{DReLU}(\mathsf{G}[i].\mathbf{In})$
        $\bigodot \left(\mathsf{G}[i].\text{deg}^{-\frac{1}{2}}\right) \mathbf{agg}_3$
    for $j = 1$ to 3 do:
      $\mathbf{agg}_j = 0$
  else
    for $j = 1$ to 3 do:
      $\mathbf{agg}_j = \mathbf{agg}_j + \mathsf{G}[i].v_j$
```

Figure 5: Scatter **and** Gather **to compute** $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$**,** $\hat{\mathbf{A}}\mathbf{X}$**, and** $\mathsf{DReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$ **in backward pass.**

Computing $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}}$ Note that the computation of $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$ in Eq. (6) can be performed via GraphSC, similar to the forward pass computation. Elaborately, the i^{th} vertex computes and scatters $\mathbf{v} = (\mathsf{G}[i].z - \mathsf{G}[i].y)\,\mathsf{G}[i].\text{deg}^{-\frac{1}{2}}$ over its edges. All the $\mathbf{v}$ components scattered by the neighbours of a node j are then gathered in the node while also accounting for the scaling factor of $\mathsf{G}[j].\text{deg}^{-\frac{1}{2}}$ to generate the j^{th} row of $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$ stored in $\mathsf{G}[j].v_1$. The Scatter-Gather primitives for computing $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$ appear in Fig. 5. However, computation of ${\mathbf{H}^{(1)}}^{\mathsf{T}}\left(\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})\right)$ does not render itself well in the message-passing paradigm. This is because the multiplications performed while computing ${\mathbf{H}^{(1)}}^{\mathsf{T}}\left(\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})\right)$ are independent of the structure of the graph, and no longer require the neighbourhood information (which is otherwise leveraged while defining the Scatter-Gather primitives). Thus, to compute ${\mathbf{H}^{(1)}}^{\mathsf{T}}\left(\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})\right)$, we extract the matrices $\mathbf{H}^{(1)}$ and $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$ from the list representation, followed by performing matrix multiplication. For this, we proceed by: (i) sorting G such that vertex entries appear first followed by edge entries, and (ii) extract the first $|\mathcal{V}|$ entries of $\mathsf{G}[i].h, \mathsf{G}[i].v_1$ to generate $\mathbf{H}^{(1)}, \hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$, respectively. We then compute ${\mathbf{H}^{(1)}}^{\mathsf{T}}$ followed by multiplying with $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y})$ to generate $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}}$.

Computing $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(1)}}$ With respect to the computation of Eq. (7), we note that $\hat{\mathbf{A}}\mathbf{X}$ can be computed via GraphSC primitives. For this vertex i scatters $\mathsf{G}[i].x$ which is gathered in data component v_2 of G. Similarly, $\hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y}){\mathbf{W}^{(1)}}^{\mathsf{T}}$ can be computed via GraphSC by scattering $\mathbf{v} = (\mathsf{G}[i].z - \mathsf{G}[i].y) \cdot {\mathbf{W}^{(1)}}^{\mathsf{T}}$ followed by gathering it in data component v_3 of G. Moreover, since Eq. (7) requires computation of $\mathsf{DReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y}){\mathbf{W}^{(1)}}^{\mathsf{T}}$, after gathering $\mathbf{v}$ from the neighbors, its element-wise multiplication with $\mathsf{DReLU}(\mathsf{G}[i].\mathbf{In})$ can be performed because the i^{th} row of $\mathbf{In}$ is held with the i^{th} vertex in G. The formal details of Scatter-Gather appear in Fig. 5. Next, to multiply $\left(\hat{\mathbf{A}}\mathbf{X}\right)^{\mathsf{T}}$ with $\mathsf{DReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}}(\mathbf{Z}-\mathbf{Y}){\mathbf{W}^{(1)}}^{\mathsf{T}}$, we proceed as done in the previous case, where we extract the matrices from their list representations, followed by matrix multiplication. This computation is performed outside GraphSC due to similar reasons as provided for the case of $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}}$.

4.3 Generation of shares of G

Given $[\![\cdot]\!]$-shares of $\mathbf{A}, \mathbf{X}$ and $\mathbf{Y}$, generating G, entails generating $[\![\cdot]\!]$-shares of (i) $\mathsf{G}[i]$.isV to denote if the i^{th} tuple is a vertex or an edge, (ii) $\mathsf{G}[i].\text{deg}$ and $\mathsf{G}[i].\text{deg}^{-\frac{1}{2}}$ to store the degree and inverse degree, and (iii) $\mathsf{G}[i].\mathbf{dt}$ to store the data elements, some of which comprise a row of the feature matrix, row of $\mathbf{Y}$, intermediate results, etc. When the i^{th} entry is a vertex, we set $[\![\mathsf{G}[i].\text{isV}]\!] = [\![1]\!]$, $[\![\mathsf{G}[i].x]\!]$ as the i^{th} row of $\mathbf{X}$, $[\![\mathsf{G}[i].\text{deg}]\!]$ and $[\![\mathsf{G}[i].\text{deg}^{-\frac{1}{2}}]\!]$ as the $(i,i)^{\text{th}}$ entry of $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{D}}^{-\frac{1}{2}}$, respectively, while remaining $\mathsf{G}[i].\mathbf{dt}$ are initialized to 0 vectors. Since there may only be $|\mathcal{E}|$ edges but $[\![\mathbf{A}]\!]$ consists of $|\mathcal{V}|^2$ possibilities, the challenge arises in generating their corresponding entries in G while leaking no information. For this, we generate a list $[\![\mathsf{G}']\!]$ comprising of all possible edges (i.e. every element in $\mathbf{A}$). We set $[\![\mathsf{G}'[i].\text{isV}]\!] = [\![\mathbf{A}_{ij}]\!]$, and all other data components to $[\![0]\!]$. To extract the valid $|\mathcal{E}|$ edges from $|\mathcal{V}|^2$ entries in G', we sort $[\![\mathsf{G}']\!]$ based on the values in $[\![\text{isV}]\!]$, extract the first $|\mathcal{E}|$ entries, and append these to $[\![\mathsf{G}]\!]$.

Since, isV should be 1 only for vertices, values of $[\![\mathsf{isV}]\!]$ are set $1-[\![\mathsf{isV}]\!]$ for edges before appending them to $[\![\mathsf{G}]\!]$.

5 Improvements over Tetrad

Here, we provide a brief description of the building blocks that were either missing in Tetrad (double bit injection, prefix OR, exponentiation, inverse square root) or had an inefficient realization (division). Further details are provided in §D.

Double bit injection. Similar to *single* bit injection protocol of Tetrad [23], we design double bit injection or 2-bit-injection protocol ($\Pi_{\mathsf{2-bitInj}}$), which enables computing $[\![\mathsf{abv}]\!]$ given *two* Boolean shared bits $[\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}$ and an arithmetic shared value $[\![\mathsf{v}]\!]$. Combining terms and computing them together results in our protocol having same online cost as single bit-injection, and improves online rounds and communication by $2\times$. To achieve this, observe that,

$$\begin{aligned}\mathsf{y} = (\mathsf{abv})^{\mathsf{R}} = (\mathsf{m_a} \oplus \lambda_{\mathsf{a}})^{\mathsf{R}} (\mathsf{m_b} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}} (\mathsf{v}) = \mathsf{m_a^R m_b^R m_v} - \mathsf{m_a^R m_b^R} \lambda_{\mathsf{v}} + \\ + \mathsf{m_a^R}\left(1-2\mathsf{m_b^R}\right)\left(\lambda_{\mathsf{b}}^{\mathsf{R}} \mathsf{m_v} - \lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}\right) \mathsf{m_b^R}\left(1-2\mathsf{m_a^R}\right)\left(\lambda_{\mathsf{a}}^{\mathsf{R}}\mathsf{m_v} - \lambda_{\mathsf{a}}^{\mathsf{R}}\lambda_{\mathsf{v}}\right) \\ + \left(1-2\mathsf{m_a^R}\right)\left(1-2\mathsf{m_b^R}\right)\left(\lambda_{\mathsf{a}}^{\mathsf{R}}\lambda_{\mathsf{b}}^{\mathsf{R}}\mathsf{m_v} - \lambda_{\mathsf{a}}^{\mathsf{R}}\lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}\right)\end{aligned}$$

where, arithmetic equivalent of XOR, i.e., $\mathsf{x} \oplus \mathsf{y}$ is $\mathsf{x}^{\mathsf{R}} + \mathsf{y}^{\mathsf{R}}(1-2\mathsf{x}^{\mathsf{R}})$. Given that $[\![\cdot]\!]$-shares of λ terms (and their products) can be generated in preprocessing (as done in Tetrad via the protocol for converting bit to its arithmetic equivalent, Π_{bit2A}), the online phase involves generating $[\![\cdot]\!]$-shares of product terms comprising m, λ via Π_{jsh} followed by local addition to generate $[\![\mathsf{y}]\!]$. Formal details appear in Fig. 15.

Prefix OR. This protocol, denoted as Π_{preOR}, forms an important building block in division, square root, and inverse square root protocols. On input Boolean shared bits $\mathsf{x}_{\ell-1}, \ldots, \mathsf{x}_0$, it outputs Boolean shared bits $\mathsf{y}_{\ell-1}, \ldots, \mathsf{y}_0$ such that $\mathsf{y}_i = \vee_{j=i}^{\ell-1} \mathsf{x}_j$. We provide an efficient instantiation of prefix OR, which works over rings and leverages the presence of multi-input AND gates provided in Tetrad, to yield a protocol that requires 3 rounds for 64-bit inputs.

The below protocols for exponentiation, division and inverse square root follow from literature and are adapted to work over Tetrad while introducing optimizations where possible.

Exponentiation. Denoted as Π_{exp}, the protocol for exponentiation outputs $[\![e^{\mathsf{x}}]\!]$ on input $[\![\mathsf{x}]\!]$. Our protocol is similar to [18], but has better concrete efficiency due to avoiding the reliance on edabits, and a few intermediate conversions. Other optimizations such as reliance on double bit injection, avoiding the need for an explicit bit extraction circuit for computing the MSB of x, etc., further aid in reducing the round complexity of our exponentiation protocol.

Division. The garbled circuit based division in Tetrad is known to be expensive [3, 4]. We propose a division protocol, Π_{div}, that relies on Goldschmidt's approximation and follows a similar approach as in [7]. This protocol on input $[\![\mathsf{a}]\!]$ and $[\![\mathsf{b}]\!]$, outputs $[\![\mathsf{d}]\!]$ where $\mathsf{d} \approx \mathsf{a}/\mathsf{b}$ via an iterative approach.

Inverse square root. The secure protocol Π_{InvSqrt}, on input $[\![\mathsf{a}]\!]$, outputs $[\![\mathsf{y}]\!]$ where $\mathsf{y} \approx 1/\sqrt{\mathsf{a}}$. Our protocol follows on the lines of [18, 29] and uses polynomial approximation to compute the inverse square root of x.

Security. Since our constructions use primitives from Tetrad, their security follows directly from Tetrad.

6 Benchmarks

Benchmark environment and parameters Benchmarks are performed over LAN using Google Cloud instances with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors, 64vCPUs, 120GB of RAM memory and a bandwidth of 16Gbps. We implement all protocols in python. We use the Crypto library for AES and hashlib for generating SHA256 hash. We note that our code is developed for benchmarking, is not optimized for industry-grade use, and a C++ based implementation can give better performance. We consider run time for one epoch for efficiency comparison and report the online and preprocessing cost. However, when reporting accuracy, we consider a maximum of 200 epochs and stop the training earlier if the test loss does not change for 10 consecutive epochs. As in Tetrad, we rely on fixed-point arithmetic (FPA) to represent decimal values. The most significant bit (MSB) of the $\ell = 64$-bit integer represents the sign bit, $\mathsf{f} = 16$ least significant bits represent the fractional part and $\mathsf{k} = 32$ is input length.

6.1 Comparison of primitives

The overall performance of $\mathsf{Entrada}$ is heavily dependent on the underlying primitives. Hence, we first analyze the same to showcase the efficiency and accuracy improvements brought in by the new primitives in comparison to the ones in the literature. For a fair comparison, all algorithms are realized using the MPC of Tetrad. Since our improvements are in exponentiation and division which are also brought in by our improved double bit injection and prefix OR protocols in addition to other optimizations described in §5, we focus on comparing exponentiation with the protocol of [18], and division with GC-based division of Tetrad, in Table 1. We note that for both, exponentiation and division, our efficiency improvements come without compromising accuracy.

Operation	Reference	Communication(KB)		Run time(ms)		Relative Error (%)
		Preprocessing	online	Preprocessing	Online	
Exp	[18]	3.91	0.91	8.21	15.29	0.24
	Entrada	1.33	0.66	5.47	11.90	0.24
Div	Tetrad [23]	11028.61	125.44	1704.09	1665.66	3.72
	Entrada	2.94	1.99	48.57	136.45	3.72

Table 1: Comparison of primitives.

6.2 GCN

Since the secure computation framework relies on fixed-point arithmetic (FPA), which is known to have lesser accuracy than the floating-point counterpart, we first demonstrate the accuracy loss of the GCN in moving from cleartext floating-point to FPA. Moreover, due to operations such as truncation, and the approximations used within the secure protocols for exponentiation, division, and inverse square root, the accuracy of the secure GCN model may be affected. Hence, we analyze the accuracy of secure GCN (FPA) and demonstrate that it is on par with the cleartext (FPA) variant, and that it improves in comparison to Tetrad. We also showcase that Entrada outperforms Tetrad in terms of efficiency. Note that we do not compare against the protocol of [37] since it does not consider the GCN of [20], and only provides support for inference for a relatively old graph neural network. Moreover, [37] operates in a 3PC setting and provides semi-honest security (with only privacy against a malicious adversary), as opposed to our setting where we consider 4PC for efficiency reasons and attain stronger security notions of fairness/robustness. Finally, we showcase improvements brought in via GraphSC.

Dataset We use the Cora dataset to benchmark the performance. It contains 2708 documents that are treated as nodes, and 4732 citation links between documents, that are treated as undirected edges. Each document has a bag of words which is treated as the feature vector associated with the node. The documents are classified into seven classes (or labels).

Accuracy We report the accuracy of GCN using the Cora dataset in Table 2. A pictorial representation of variation in accuracy and test loss with number of epochs appears in Fig. 6. As evident from Table 2, moving from cleartext floating-point to cleartext FPA representation witnesses a slight drop in accuracy. Keeping the cleartext FPA accuracy as the benchmark for the secure variants, we observe that our protocol loses out on accuracy by only 0.4%. This is very small compared to the loss in accuracy witnessed by Tetrad, which is around 5.6%. On the contrary, realizing secure GCN via SGD in Terad results in an accuracy which is 76.6%. We note that the drop in accuracy in Tetrad while using Adam stems due to the use of the approximate softmax function (ASM), which degrades the accuracy of the overall model.

Model	Cleartext		Secure	
	Float	Fixed	Tetrad (fixed)	Entrada (fixed)
GCN [20]	80.2%	79.7%	74.1%	79.3%

Table 2: GCN accuracy on Cora using Adam–Tetrad is enhanced with inverse square root protocol to support Adam.

To showcase improvement brought in by Entrada over Tetrad, Table 3 reports impact on accuracy when sequentially replacing each of the following primitives–division, square root, exponentiation–in Tetrad with the newly designed ones. Elaborately, we begin with reporting the accuracy of GCN via Tetrad (version v1), which relies on GC-based division, ASM, and the SGD optimizer. In the next version, v2, we replace the GC-based division with our division protocol. As noted earlier in Table 1, our division protocol has the same relative error as the GC-based division of Tetrad, and hence does not impact the accuracy of GCN. This is followed by version v3, where ASM in v2 is replaced by the accurate computation of softmax, resulting in improved accuracy. Finally, in version v4, which constitutes our framework Entrada, SGD in v3 is replaced with Adam optimizer, which drastically improves accuracy by over 2%.

Version	Accuracy
v1:= Tetrad (GC division + ASM + SGD)	76.6%
v2:= v1 - GC division + our division	76.6%
v3:= v2 - ASM + accurate softmax	77.4%
v4:= Entrada (v3 - SGD + Adam optimizer)	79.3%

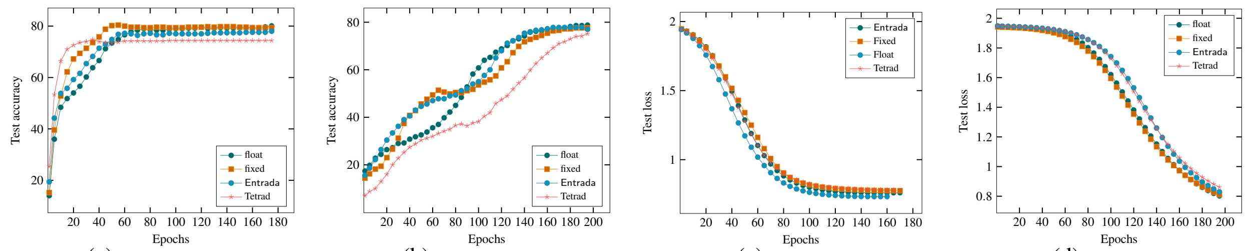
Table 3: GCN accuracy improvements (Cora dataset) when replacing primitives in Tetrad in a step-by-step manner.

Efficiency We compare the efficiency of evaluating GCN via Entrada and Tetrad. We first report the performance of Entrada for GCN inference. Note that the performance of Entrada is on par with Tetrad's, since inference does not require any of the newly designed primitives. Specifically, it has a run time of 2.4 seconds and 5.6 seconds in the preprocessing and online phase, respectively, for the Cora dataset.

With respect to GCN training, recall that Entrada is designed to support the Adam optimizer, and to also leverage the GraphSC paradigm to yield an efficient solution. However, Entrada can be modified to use SGD instead of the Adam optimizer. One can also avoid reliance on GraphSC, depending on the application scenario. Further, recall as discussed in §6.2, that Tetrad originally only supports SGD evaluation. Hence, to provide a fair comparison, we also report the performance Entrada when using only the SGD optimizer. In fact, we begin by analyzing Entrada's performance in comparison to Tetrad, while excluding GraphSC. As seen from Table 4, Entrada (SGD, w/o GraphSC) outperforms Tetrad in training. The overhead in Tetrad can be mainly attributed to the use of the GC. Entrada (SGD, w/o GraphSC) not only has better efficiency, but also outperforms Tetrad in terms of accuracy (see v3 in Table 3). To further improve the accuracy, we switch to the Adam optimizer, which results in Entrada having an increased online time in comparison to Tetrad. This is due to its reliance on additional operations required for supporting Adam. Interestingly, the overall efficiency of Entrada is still better than Tetrad's by a factor of around 30×. The use of GraphSC helps to further improve the efficiency of GCN training, as evident from Table 4.

Variant	Preprocessing	Online
Tetrad	7285.076	121.577
Entrada (SGD, w/o GraphSC)	14.988	71.578
Entrada (w/o GraphSC)	30.915	211.160
Entrada (SGD)	1.269	29.601
Entrada	19.572	189.885

Table 4: Comparison of GCN performance (training).



(a) Test accuracy (b) Test accuracy (c) Test loss (d) Test loss

Figure 6: Variation in GCN test accuracy and loss with number of epochs on Cora dataset. float and fixed denote cleartext variants. (a),(c) use Adam and (b),(d) use SGD.

6.3 Fraud detection

We use Entrada to securely realize the application of fraud detection via GCN from the work of [41] and [27], where the former performs fraud detection in online review platforms using the dataset from Tencent App Store, and the latter detects fraudulent accounts in online payment network of Alipay. Given the unavailability of the datasets considered in each of the works, we benchmark their performance on alternative datasets, i.e. [41] is evaluated on the Yelp while [27] is evaluated on the DBLP dataset. As done for the case of vanilla GCN, we first evaluate the accuracy loss, followed by analyzing the performance of the secure protocols. Our analysis of these alternative datasets is meant to establish the relative accuracy/performance of Entrada in comparison to the cleartext computation. We believe similar accuracy/performance trends will hold true when Entrada is evaluated on the actual fraud detection datasets. Further, a comparison with Tetrad is omitted since §6.2 establishes that we outperform it.

Dataset The work of [41] is evaluated on the Yelp dataset that contains 45,954 reviews, each of which is treated as a node. An edge between two nodes indicates that the corresponding reviews were posted by the same user, and there exist 3,846,979 edges. Each node/review is classified as fake or real. Since the work of [27] operates on a heterogeneous graph, we consider the DBLP dataset that is known to be similar to the original dataset of Alipay since both consider graphs having heterogeneous edges (i.e., multiple edges between the same two nodes may indicate different relations between these nodes). The DBLP dataset consists of 14,328 papers that are treated as nodes. There are three different types of edges, each of which relates two papers (nodes) if they– (i) appear in the same conference, (ii) have the same authors, and (iii) use common terms. There are 1,70,794 edges in total. Further, each node has an associated bag of words that are treated as its feature vector. Each paper/node is classified into 4 different classes (labels) that include database, data mining, machine learning, and information retrieval.

Accuracy and efficiency comparison The results appear in Table 5, Table 6 and Table 7. For accuracy, we observe similar trends as seen in the case of vanilla GCN, where the accuracy of secure variant is comparable to that of cleartext. Regarding efficiency, we observe up to 4× gain when adapting GCNs to work with GraphSC, thereby corroborating our claim of witnessing efficiency improvements when using GraphSC.

Algorithm	Metric	Cleartext		Secure variant
		Float	Fixed	Entrada (Fixed)
[41]	Recall	0.513	0.507	0.507
	Precision	0.681	0.676	0.669
	F1	0.585	0.579	0.576
[27]	Accuracy	68.5%	67.1%	66.3%

Table 5: Accuracy comparison of fraud detection algorithms.

Algorithm	Preprocessing	Online
[41]	34.574	73.678
[27]	21.350	55.511

Table 6: Fraud detection algorithms on Entrada (inference).

Algorithm	Preprocessing	Online
[41] (w/o GraphSC)	131.433	686.272
[27] (w/o GraphSC)	73.329	350.468
[41]	32.753	431.398
[27]	32.756	425.707

Table 7: Fraud detection algorithms on Entrada (training).

7 Conclusion

We present Entrada, a 4-party computation framework for securely and efficiently realizing GCNs, which is addressed for the first time. Our system Entrada provides the necessary primitives to securely realize both training as well as inference of GCN. Entrada additionally leverages the GraphSC paradigm to improve the efficiency of the training phase. We additionally design a secure shuffle protocol as required for the GraphSC paradigm. Extensive experiments establish the practicality of our solution with respect to efficiency and accuracy. Improving the protocols used in Entrada, and realizing other GNNs securely, is an interesting future work.

References

[1] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. 2019.

[2] Abdelrahaman Aly and Nigel P Smart. Benchmarking privacy preserving scientific operations. In *ACNS*, 2019.

[3] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE S&P*, 2017.

[4] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS*, 2016.

[5] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *CCS*, 2021.

[6] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, 2019.

[7] Octavian Catrina. Round-efficient protocols for secure multiparty fixed-point arithmetic. In *IEEE COMM*, 2018.

[8] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, 2010.

[9] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *FC*, 2010.

[10] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*, 2020.

[11] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security*, 2020.

[12] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, 2020.

[13] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE S&P*, 2019.

[14] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, 2006.

[15] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *NDSS*, 2022.

[16] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, 2017.

[17] Hiroki Kanezashi, Toyotaro Suzumura, Xin Liu, and Takahiro Hirofuchi. Ethereum fraud detection with heterogeneous graph neural networks. *arXiv preprint arXiv:2203.12363*, 2022.

[18] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *ICML*, 2022.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (poster)*, 2015.

[20] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[21] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *NeurIPS*, 2021.

[22] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In *USENIX Security*, 2021.

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

[24] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In *ISC*, 2011.

[25] Yujia Liu, Kang Zeng, Haiyang Wang, Xin Song, and Bin Zhou. Content matters: a gnn-based model combined with text semantics for social network cascade prediction. In *PAKDD*, 2021.

[26] Zheng Liu, Xiaohan Li, Hao Peng, Lifang He, and S Yu Philip. Heterogeneous similarity graph neural network on electronic health records. In *Big Data*, 2020.

[27] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. Heterogeneous graph neural networks for malicious account detection. In *CIKM*, 2018.

[28] Donghang Lu and Aniket Kate. Rpm: Robust anonymity at scale. *Cryptology ePrint Archive*, 2022.

[29] Wen-jie Lu, Yixuan Fang, Zhicong Huang, Cheng Hong, Chaochao Chen, Hunter Qu, Yajin Zhou, and Kui Ren. Faster secure multiparty computation of adaptive gradient descent. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, 2020.

[30] Peter Markstein. Software division and square root using goldschmidt's algorithms. In *RNC*, 2004.

[31] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS*, 2018.

[32] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[33] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[34] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Aby2. 0: Improved mixed-protocol secure two-party computation. In *USENIX Security*, 2021.

[35] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place, comparison-based sorting with cuda: A study with bitonic sort. *Concurrency and Computation: Practice and Experience*, 2011.

[36] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE*, 2004.

[37] Liyan Shen, Xiaojun Chen, Jinqiao Shi, Ye Dong, and Binxing Fang. An efficient 3-party framework for privacy-preserving neural network inference. In *ESORICS*, 2020.

[38] Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Find Thy Neighbourhood: Privacy-Preserving Local Clustering. *PETS*, 2023.

[39] Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-Party Shuffle Protocols. *PETS*, 2023.

[40] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETS*, 2021.

[41] Jianyu Wang, Rui Wen, Chunming Wu, Yu Huang, and Jian Xion. Fdgars: Fraudster detection via graph convolutional networks in online app review system. In *Companion@WWW*, 2019.

[42] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, 1982.

A Preliminaries

Shared key setup Parties can non-interactively generate common random values among themselves by relying on the common PRF keys established among themselves during a one-time setup phase. This is abstracted via the ideal functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 7). Elaborately, let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with $X = \mathbb{Z}_{2^\ell}$. The following keys are established between parties– (i) k_{ij} for every pair of parties P_i, P_j, (ii) k_{ijk} for every triple of parties P_i, P_j, P_k, and (iii) $k_{\mathcal{P}}$ known to all parties in $\mathcal{P}$. If P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively, they compute $F_{k_{01}}(id_{01})$ and obtain r. Here, id_{01} denotes a counter maintained by the parties, and is updated after every PRF invocation.

Functionality $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys k_{ij} and k_{ijk} for $i, j, k \in \{0,1,2,3\}$ and $k_{\mathcal{P}}$. Let y_s denote the keys corresponding to party P_s. Then

– $\mathsf{y}_s = (k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $k_{\mathcal{P}})$ when $P_s = P_0$.
– $\mathsf{y}_s = (k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_1$.
– $\mathsf{y}_s = (k_{02}, k_{12}, k_{23}, k_{012}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_2$.
– $\mathsf{y}_s = (k_{03}, k_{13}, k_{23}, k_{013}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ when $P_s = P_3$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 7: Ideal functionality for shared-key setup.

The list of protocols that we rely on from Tetrad [23] appears in Table 8. Since we heavily rely on joint message passing and joint sharing protocols, we elaborate on these next.

Joint message passing The joint message passing primitive Fig. 8 allows two senders P_i, P_j to relay a common message, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, to a recipient P_k, either by ensuring successful delivery of v, or by establishing a conflicting pair of parties, one among which is guaranteed to be corrupt. This implies the residual two parties are honest, one of which is then entrusted to take the computation to completion by enacting the role of a trusted third party (ttp). The instantiation of joint message passing can be viewed as consisting of two phases (send, verify), where the send phase consists of P_i sending v to P_k and the rest of the protocol steps go to verify phase (which ensures correct send or ttp identification). This requires 1 round of interaction and ℓ bits of communication. To leverage amortization, verify is executed only once, at the end of the

Building block	Notation	Description
Joint message passing	$\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{v})$	Enables $P_i, P_j \in \mathcal{P}$ to send v to P_k such that P_k receives the correct v, or a conflicting pair of parties among P_i, P_j, P_k is identified.
Joint sharing	$[\![\mathsf{v}]\!] = \Pi_{\mathsf{jsh}}(P_i, P_j, \mathsf{v})$	Enables $P_i, P_j \in \mathcal{P}$ to generate $[\![\mathsf{v}]\!]$ where $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is held by P_i, P_j
Multiplication	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{mult}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!], \mathsf{f})$	Multiplies x, y and outputs $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ truncated by f bits
Matrix Multiplication	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{MatMul}}([\![\mathsf{A}]\!], [\![\mathsf{B}]\!], \mathsf{f})$	Multiplies matrices A, B and outputs $\mathbf{C} = \mathsf{A} \cdot \mathsf{B}$
3-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{3\text{-}mult}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 3 inputs at once
4-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{4\text{-}mult}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}}, [\![\mathsf{d}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 4 inputs at once
MulR	$\langle \mathsf{z} \rangle = \Pi_{\mathsf{MulR}}(\langle \mathsf{x} \rangle, \langle \mathsf{y} \rangle)$	Multiplies x, y and outputs $\langle \cdot \rangle$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$
Arithmetic to Boolean	$[\![\mathsf{x}]\!]^{\mathbf{B}} = \Pi_{\mathsf{A2B}}([\![\mathsf{x}]\!])$	Converts arithmetic shares of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ to Boolean shares of each bit of x
Boolean to arithmetic	$[\![\mathsf{x}]\!] = \Pi_{\mathsf{B2A}}([\![\mathsf{x}]\!]^{\mathbf{B}})$	Converts Boolean shares of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ to arithmetic shares
Bit2A	$[\![\mathsf{b}]\!] = \Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Converts bit to its arithmetic equivalent
Bit extraction	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{bitext}}([\![\mathsf{x}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < 0$, else outputs $\mathsf{b} = 0$
Comparison	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{comp}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < \mathsf{y}$, else outputs $\mathsf{b} = 0$
Bit injection	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{bitInj}}([\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{x}]\!])$	Multiplies arithmetic equivalent $\mathsf{b}^{\mathsf{R}} \in \mathbb{Z}_{2^\ell}$ of $\mathsf{b} \in \mathbb{Z}_2$ with $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ and outputs $\mathsf{z} = \mathsf{b}^{\mathsf{R}} \cdot \mathsf{x}$
Oblivious select	$[\![\mathsf{x_b}]\!] = \Pi_{\mathsf{sel}}([\![\mathsf{x}_0]\!], [\![\mathsf{x}_1]\!], [\![\mathsf{b}]\!]^{\mathbf{B}})$	Obliviously selects $\mathsf{x_b}$ among $\mathsf{x}_0, \mathsf{x}_1$
Negation	$[\![\bar{\mathsf{x}}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}([\![\mathsf{x}]\!]^{\mathbf{B}})$	Outputs 1's complement of Boolean representation of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$

Table 8: Description of protocols from Tetrad [23].

computation, and requires 2 rounds. The protocol also relies on a collision resistant hash function $\mathsf{H}(\cdot)$.

Protocol $\Pi_{\mathsf{jmp}}(P_i, P_j, \mathsf{v}, P_k)$

$P_s \in \mathcal{P}$ initializes an inconsistency bit $\mathsf{b}_s = 0$. If P_s remains silent instead of sending b_s in any of the following rounds, the recipient sets b_s to 1.

– *Send:* P_i sends v to P_k.
– *Verify:* P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.
∘ P_k sets $\mathsf{b}_k = 1$ if the received values are inconsistent or if the value is not received.
∘ P_k sends b_k to all parties. P_s for $s \in \{i, j, l\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.
∘ P_s for $s \in \{i, j, l\}$ mutually exchange their bits. P_s resets $\mathsf{b}_s = \mathsf{b}'$ where b' denotes the bit which appears in majority among $\mathsf{b}_i, \mathsf{b}_j, \mathsf{b}_l$.
∘ All parties set $\mathsf{ttp} = P_l$ if $\mathsf{b}' = 1$, terminate otherwise.

Figure 8: Joint message passing [23]

Joint sharing Protocol Π_{jsh} enables parties P_i, P_j to generate $[\![\cdot]\!]$-share of value v. During the preprocessing phase, shares of λ_{v} are sampled such that both P_i, P_j will get the entire mask λ_{v}. During the online phase, P_i, P_j compute and send $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$ to parties P_1, P_2, P_3 via Π_{jmp}.

For joint-sharing a value v possessed by P_0 along with one other party in the preprocessing phase, the communication can be optimized further. The protocol steps based on the (P_i, P_j) pair are summarised below:

– $(P_0, P_1) : \mathcal{P} \setminus \{P_2\}$ sample $\lambda_{\mathsf{v}}^1 \in \mathbb{Z}_{2^\ell}$. Parties set $\lambda_{\mathsf{v}}^2 = \mathsf{m_v} = 0$. P_0, P_1 send $\lambda_{\mathsf{v}}^3 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_2 via Π_{jmp}.
– $(P_0, P_2) : \mathcal{P} \setminus \{P_3\}$ sample $\lambda_{\mathsf{v}}^3 \in \mathbb{Z}_{2^\ell}$. Parties set $\lambda_{\mathsf{v}}^1 = \mathsf{m_v} = 0$. P_0, P_2 send $\lambda_{\mathsf{v}}^2 = -\mathsf{v} - \lambda_{\mathsf{v}}^3$ to P_3 via Π_{jmp}.
– $(P_0, P_3) : \mathcal{P} \setminus \{P_1\}$ sample $\lambda_{\mathsf{v}}^2 \in \mathbb{Z}_{2^\ell}$. Parties set $\lambda_{\mathsf{v}}^3 = \mathsf{m_v} = 0$. P_0, P_3 send $\lambda_{\mathsf{v}}^1 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_1 via Π_{jmp}.

Achieving fairness Although the robust version of the protocols are described, we note that their fair version can be derived by making the following changes, as described in Tetrad [23]:

– Use of the fair version of Π_{jmp} instead of the robust version: for this the Π_{jmp} protocol is modified so that parties abort instead of identifying a TTP in case the jmp verification fails.
– Relying on a fair reconstruction protocol: to achieve fairness, it is necessary to ensure that all the honest parties are alive after the verification phase before proceeding with the output reconstruction. To accomplish this, the parties maintain an aliveness bit, b, which is initialized as "continue". If a party's verification phase is unsuccessful, it sets b to "abort". In the first round of reconstruction, the parties exchange their b bit and collectively agree on the majority value. Due to the presence of at most one corrupt party, it is guaranteed that all the honest parties will reach a consensus on the value of b. If b equals "continue", the parties proceed to exchange their missing shares and accept the majority decision. According to the sharing semantics, each missing share is owned by three parties, with at most one corrupt party among them.

Collision-resistant hash function [36] . A family of hash functions $\{\mathsf{H} : \mathcal{K} \times \mathsf{M} \rightarrow \mathcal{Y}\}$ is said to be collision resistant if for all PPT adversaries $\mathcal{A}$, given the hash function H_k for $k \in \mathcal{K}$, the following holds: $\mathsf{Pr}[(x,x') \leftarrow \mathcal{A}(k) : (x \neq x') \wedge \mathsf{H}_k(x) = \mathsf{H}_k(x')] = \mathsf{negl}(\kappa)$, where $x, x' \in \{0,1\}^m$ and $m = \mathsf{poly}(\kappa)$.

GraphSC paradigm This section details the GraphSC paradigm [5,32] which was also discussed in the work of [38].

Since real-world graphs are known to be sparse, naively using the adjacency matrix representation of the graph for computations would be expensive. Hence, designing an efficient solution to address the same involves- (i) designing an effective representation of the graph structure, (ii) ensuring the computation does not leak any private information, (iii) designing solutions that are highly parallelizable. The work by Nayak et al. [33] is the first to address the above problem and provides a framework for the same. The framework operates on a data augmented directed graph G(V, E, Data) which consists of a directed graph G(V, E) where V is the set of vertices (or nodes, used interchangeably), E is the set of edges and Data is a set of user-defined data values associated with each vertex and edge of the graph. The data augmented graph is expressed as a list of vertices and edges where every vertex $\mathsf{v} \in \mathsf{V}$ is encoded as a tuple $(\mathsf{v}, \mathsf{v}, 1, \mathsf{data})$ and every edge $(\mathsf{u}, \mathsf{v}) \in \mathsf{E}$ is encoded as a tuple $(\mathsf{u}, \mathsf{v}, 0, \mathsf{data})$. The third entry in each tuple is a bit, isV, which equals 1 for a vertex and 0 otherwise, while data refers to the state information stored at each vertex and edge. These tuples constitute the data augmented graph list (DAG-list). The DAG-list representation of the graph is used to effectively represent the graph. Note that an undirected graph can be converted into a directed graph by accounting for each edge twice (incoming and outgoing edge). To perform secure computation over the graph while hiding its topology, each tuple in the data augmented graph is secret-shared entry-wise between the computing parties. This ensures that parties cannot distinguish between shares of a tuple corresponding to a vertex from that of an edge.

In principle, the framework of [33] enables securely evaluating message-passing graph algorithms. The latter are graph algorithms that operate in multiple rounds, where in each round, the nodes in the graph- (i) use their state information to send messages over their outgoing edges; (ii) receive messages along their incoming edges and aggregate these messages; (iii) use these messages to update their state. These three operations are abstracted into three primitives–Scatter, Gather, and Apply, respectively. Assuming that the graph algorithm can be expressed as a composition of the above primitives, [33] enables its secure evaluation via MPC [6]. Since designing an MPC protocol naively may leak information regarding the graph topology, the framework first designs a data-oblivious algorithm for each of these primitives, followed by a translation of the same using the generic 2-party protocol of [42]. In general, an algorithm is said to be data-oblivious if the instructions executed and the memory accesses made during the run of the algorithm are independent of the input and hence leak no information about the input. To obtain a data-oblivious algorithm for the GraphSC primitives, it is important to ensure that each entry in the DAG-list is visited when realizing these primitives to ensure no information about the association between the entries (such as an edge being incident on a node) is leaked. Observe that Apply can be computed obliviously by scanning through the DAG-list representation and applying the user-defined function if the element is a vertex and performing a dummy operation otherwise. To compute Scatter and Gather obliviously, [33] relies on two different sorted orders of the DAG-list representation. The *source order* requires the DAG-list to be sorted such that every node in the graph is placed before all edges that originate from it. The *destination order* requires the DAG-list to be sorted such that all edges that end at a particular node are placed before that node. Consider an oblivious sort protocol that outputs a sorted list of elements in vector $\mathbf{x}$ based on the key *key*. Given a data-oblivious sort protocol such as Bitonic sort [35], switching between the source order and destination order each time a Scatter or Gather is applied, ensures obliviousness as follows. Scatter can be accomplished obliviously by linearly scanning through the DAG-list sorted in the source order. For this, if the current tuple in the list is a node, the data value at the node is picked up, and if the current tuple is an edge, then the value picked up at the most recent tuple is used to update the edges. Gather can also be done obliviously by a linear scan through the DAG-list sorted in the destination order. During the scan, if the current tuple is an edge, its value is stored in an aggregate variable by applying an aggregation operation, and if the current tuple is a node, then the aggregate variable is stored along with the node. This approach of performing Scatter and Gather by performing a linear scan over a sorted order is oblivious as every node and edge of the graph is operated on, without revealing the relationship between the nodes and edges. Given that the primitives can be performed obliviously, as described above, the graph computation can be performed securely using MPC protocols. To summarize, message-passing graph algorithms can be computed obliviously by using the following steps in every message-passing round– (i) sort based on source order, (ii) Scatter, (iii) sort based on destination order, (iv) Gather and (v) Apply. Further, the framework in [33] provides a parallel algorithm for each of the individual primitives– Scatter, Gather, and Apply. In a multiprocessor setting, the parallel variants allow the computations to be performed in sub-linear complexity rather than the linear complexity of $O(|\mathsf{V}| + |\mathsf{E}|)$ in the size of the graph. We remark that this technique of obtaining a sub-linear solution in the multiprocessor setting also extends to our protocols. We refer to [32] for details of

[6]The operations within Scatter and Gather will vary across different graph algorithms. Hence, [33] provides a generic GraphSC framework, where the operations to be performed within Scatter and Gather are assumed as a black-box.

the parallel variant.

The work of [5] improves on the work of [33]. First, it combines Gather and Apply primitives such that both operations can be achieved in a single pass through the DAG-list. Moreover, [5] observes that the approach of [33] has the drawback of requiring an oblivious sort each time a Scatter or Gather primitive is applied. This amounts to two calls to an oblivious sort in every message-passing round. Instead, [5] observes that a secret shuffle followed by an insecure sort (which reveals the result of the comparisons) can be used to realize an oblivious sort. Let $\Pi_{\mathsf{Shuffle}}([\![\mathbf{x}]\!])$ denote an oblivious shuffle protocol that outputs the elements of $\mathbf{x}$ in a randomly shuffled order. Observe that since the list is first shuffled, revealing the result of comparisons during sort does not break the obliviousness property. On the other hand, the insecure sort is required to be performed only once in the beginning, subsequent to which, the public permutation (obtained as output from the insecure sort) can be applied to sort the DAG-list, non-interactively. In summary, in the first message-passing round, a secret shuffle followed by an insecure sort (e.g., a comparison sort algorithm) is applied to get the required source or destination order. The permutations which map from the shuffled orders to the source and destination order are made public, as they do not reveal any information about the DAG-list. In the subsequent rounds, the secret shuffle, followed by the public permutation, is applied to get the required sorted order. Since shuffle can be performed much more efficiently than a sort, this change brings in significant efficiency improvements. An illustration of the operations involved in GraphSC paradigm of [5] is given in Fig. 1. Finally, to perform secure computation, [5] considers a 3PC setting to further enhance efficiency.

B Secure GCN - input sharing

Generating $[\![\cdot]\!]$-shares of X To ensure that a possibly malicious client, C, has not cheated while secret-sharing the rows of $\mathbf{X}$, it suffices to ensure that C consistently secret-shares each element in the respective rows of $\mathbf{X}$ that it possesses. This can be performed similar to as done in [22], albeit more efficiently without relying on commitments. Elaborately, say $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ is an input element to be shared by C. Servers non-interactively generate $\langle\cdot\rangle$-shares of the mask $\lambda_{\mathsf{x}} \in \mathbb{Z}_{2^\ell}$ using $\mathcal{F}_{\mathsf{setup}}$. To obtain $\mathsf{m}_{\mathsf{x}} = \mathsf{x} + \lambda_{\mathsf{x}}$ from the client, $\lambda_{\mathsf{x}} = \lambda_{\mathsf{x}}^1 + \lambda_{\mathsf{x}}^2 + \lambda_{\mathsf{x}}^3$ is sent to C as follows. Since each λ_{x}^i for $i \in \{1,2,3\}$ is held by three servers, two of them send λ_{x}^i to C, while the third sends $\mathsf{H}(\lambda_{\mathsf{x}}^i)$. Note that all servers are required to communicate λ_{x}^i to C to ensure that a corrupt server's attempt of cheating by sending an incorrect value is subverted. Moreover, since multiple elements are required to be shared by C, the use of hash allows computing and sending a single hash value on the concatenation of all these elements, thereby reducing the communication complexity. Further, note that among the three versions of the received λ_{x}^i, since at most one can be incorrect (owing to the presence of at most one corrupt server), taking the value which appears in majority enables C to obtain the correct value for each λ_{x}^i. C then computes $\lambda_{\mathsf{x}} = \lambda_{\mathsf{x}}^1 + \lambda_{\mathsf{x}}^2 + \lambda_{\mathsf{x}}^3$, $\mathsf{m}_{\mathsf{x}} = \mathsf{x} + \lambda_{\mathsf{x}}$, and sends m_{x} to servers P_1, P_2, P_3. Finally, to ensure that C has sent the consistent m_{x} to P_1, P_2, P_3, they exchange the value received from C among themselves. Since at most one among P_1, P_2, P_3 can be corrupt, there will exist a majority in the exchanged values, which is taken as the final value for m_{x}.

Generating $[\![\cdot]\!]$-shares of A In addition to performing the consistency checks as described above to ensure a consistent sharing, servers are also required to ensure that the client's inputs correspond to valid rows of $\mathbf{A}$, i.e. $\mathbf{A}^\mathsf{T} = \mathbf{A}$. For this, they non-interactively generate $[\![\cdot]\!]$-shares of a random symmetric matrix $\mathbf{R}$, compute $[\![\mathbf{S}]\!] = [\![\mathbf{A}]\!] + [\![\mathbf{R}]\!]$, and open the resultant matrix $\mathbf{S}$. Servers can then locally verify if $\mathbf{A}$ is symmetric by checking if $\mathbf{S}_{ij} = \mathbf{S}_{ji}$ for $i, j \in \{1,2,\ldots,n\}$. If so, they proceed to verify whether the elements of $\mathbf{A}$ as generated by the clients are 0 or a 1. For this, servers use the fact that $\mathsf{z}^2 - \mathsf{z} = 0$ only if the element $\mathsf{z} \in \{0,1\}$. In order to verify this equation with respect to all the elements shared by a client, servers compute a random linear combination with respect to each element z_i shared by the client and verify if the combination yields a 0. However, this check still allows a client to cheat with probability $1/2$ when working over the ring algebraic structure [1, 6]. Thus, to reduce the cheating probability the check is repeated κ times, which bounds the cheating probability by $1/2^\kappa$. If any of the checks fail, depending on the application scenario, one of the following can be done: (i) entries in i^{th} row and j^{th} column can be set to default, (ii) entries pertaining i^{th} and j^{th} nodes can be deleted from the graph (the same should be reflected in the other inputs, $\mathbf{X}, \mathbf{Y}$, as well), (iii) the computation is halted.

Generating $[\![\cdot]\!]$-shares of Y Recall that each row of $\mathbf{Y}$ has at most one position set as 1 (and all others as 0s). Thus, after verifying the consistency of the received shares, to verify if the i^{th} row $\{\mathbf{Y}_{i1}, \mathbf{Y}_{i2}, \ldots, \mathbf{Y}_{ic}\}$ satisfies this condition, we use the idea of [28]. The approach is to check if $\left(\sum_{j=1}^{c} \mathbf{Y}_{ij} \cdot \mathsf{r}_j\right)^2 - \left(\sum_{j=1}^{c} \mathbf{Y}_{ij} \cdot \mathsf{r}_j^2\right) = 0$. Here, $\mathsf{r}_j \in \mathbb{Z}_{2^\ell}$ are random public values and the check passes with high probability over a field if at most one $\mathbf{Y}_{ij}$ is a 1. However, similar to the case of $\mathbf{A}$, this check also has a failure probability of $1/2$ over rings. Hence, we repeat this check κ times to bound the failure probability by $1/2^\kappa$.

Generating $[\![\cdot]\!]$-shares of $\tilde{\mathbf{D}}$, $\tilde{\mathbf{A}}$ and $\hat{\mathbf{A}}$ Having generated $[\![\mathbf{A}]\!]$, generation of $[\![\tilde{\mathbf{A}}]\!] = [\![\mathbf{A}]\!] + [\![\mathbf{I}]\!]$ can happen non-interactively. The $[\![\cdot]\!]$-shares of diagonal entries of $\tilde{\mathbf{D}}$ can also be generated non-interactively by summing the entries in the corresponding rows of $[\![\tilde{\mathbf{A}}]\!]$. Finally, computing

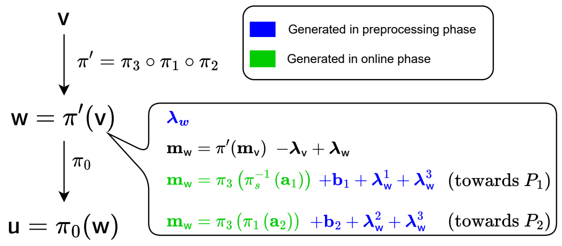
$\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$, involves computing element-wise inverse square-root of the diagonal element of $\tilde{\mathbf{D}}$ via the secure inverse square-root protocol.

C GCN evaluation via GraphSC

C.1 Secure shuffle

Non-interactively sampling a random permutation among a subset of parties Sampling a random permutation denotes choosing a random π which is a bijective function $\pi : \{1,\dots,\mathsf{N}\} \to \{1,\dots,\mathsf{N}\}$. We describe how parties P_i, P_j can do this non-interactively using the shared key established via $\mathcal{F}_{\mathsf{setup}}$. P_i, P_j non-interactively generate N common random values say $\mathsf{v}_1, \mathsf{v}_2, \dots, \mathsf{v}_\mathsf{N} \in \mathbb{Z}_{2^\ell}$ where $\ell >> \log_2 \mathsf{N}$. The parties tag each of the values v_i with its index to obtain a list $S = \{(\mathsf{v}_i, \mathsf{x}_i)\}_{i=1}^{\mathsf{N}}$, where $\mathsf{x}_i = i$. Each party then locally sorts this list of tuples based on the first entry v_i of each tuple to obtain a sorted list $S' = \{(\mathsf{v}'_j, \mathsf{x}'_j)\}_{j=1}^{\mathsf{N}}$. The second element in each tuple of S' defines a random permutation where index $\mathsf{x}_i \in \{1,\dots,\mathsf{N}\}$ gets mapped to a unique index $\mathsf{x}'_i \in \{1,\dots,\mathsf{N}\}$.

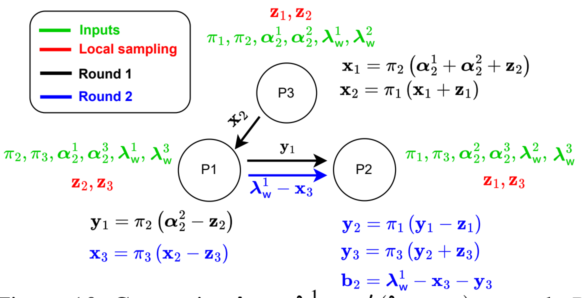
Overview of the steps performed in secure shuffle Fig. 9 gives an overview of the steps performed in secure shuffle.



v
$\pi' = \pi_3 \circ \pi_1 \circ \pi_2$
$\mathsf{w} = \pi'(\mathsf{v})$
π_0
$\mathsf{u} = \pi_0(\mathsf{w})$
🟦 Generated in preprocessing phase
🟩 Generated in online phase
$\boldsymbol{\lambda_w}$
$\mathbf{m}_\mathsf{w} = \pi'(\mathbf{m}_\mathsf{v}) \ -\boldsymbol{\lambda}_\mathsf{v} + \boldsymbol{\lambda}_\mathsf{w}$
$\mathbf{m}_\mathsf{w} = \pi_3\left(\pi_s^{-1}(\mathbf{a}_1)\right) + \mathbf{b}_1 + \boldsymbol{\lambda}_\mathsf{w}^1 + \boldsymbol{\lambda}_\mathsf{w}^3$ (towards P_1)
$\mathbf{m}_\mathsf{w} = \pi_3\left(\pi_1(\mathbf{a}_2)\right) + \mathbf{b}_2 + \boldsymbol{\lambda}_\mathsf{w}^2 + \boldsymbol{\lambda}_\mathsf{w}^3$ (towards P_2)

Figure 9: Overview of steps performed in secure shuffle.

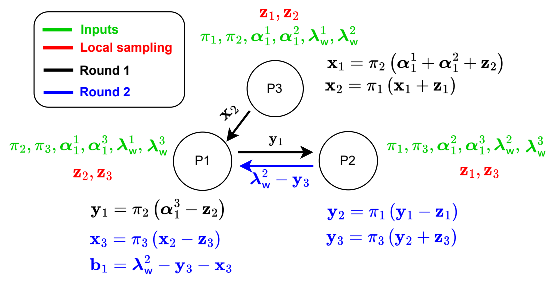
Generation of $\mathbf{b}_2$ towards P_2 Fig. 10 gives a schematic representation of the steps involved in generating $\mathbf{b}_2$ towards P_2.



Inputs
Local sampling
Round 1
Round 2
$\mathbf{z}_1, \mathbf{z}_2$
$\pi_1, \pi_2, \boldsymbol{\alpha}_2^1, \boldsymbol{\alpha}_2^2, \boldsymbol{\lambda}_\mathsf{w}^1, \boldsymbol{\lambda}_\mathsf{w}^2$
$\mathbf{x}_1 = \pi_2\left(\boldsymbol{\alpha}_2^1 + \boldsymbol{\alpha}_2^2 + \mathbf{z}_2\right)$
$\mathbf{x}_2 = \pi_1\left(\mathbf{x}_1 + \mathbf{z}_1\right)$
P3
$\mathbf{x}_2$
$\pi_2, \pi_3, \boldsymbol{\alpha}_2^1, \boldsymbol{\alpha}_2^3, \boldsymbol{\lambda}_\mathsf{w}^1, \boldsymbol{\lambda}_\mathsf{w}^3$
$\mathbf{z}_2, \mathbf{z}_3$
P1
$\mathbf{y}_1$
$\boldsymbol{\lambda}_\mathsf{w}^1 - \mathbf{x}_3$
P2
$\pi_1, \pi_3, \boldsymbol{\alpha}_2^2, \boldsymbol{\alpha}_2^3, \boldsymbol{\lambda}_\mathsf{w}^2, \boldsymbol{\lambda}_\mathsf{w}^3$
$\mathbf{z}_1, \mathbf{z}_3$
$\mathbf{y}_1 = \pi_2\left(\boldsymbol{\alpha}_2^2 - \mathbf{z}_2\right)$
$\mathbf{x}_3 = \pi_3\left(\mathbf{x}_2 - \mathbf{z}_3\right)$
$\mathbf{y}_2 = \pi_1\left(\mathbf{y}_1 - \mathbf{z}_1\right)$
$\mathbf{y}_3 = \pi_3\left(\mathbf{y}_2 + \mathbf{z}_3\right)$
$\mathbf{b}_2 = \boldsymbol{\lambda}_\mathsf{w}^1 - \mathbf{x}_3 - \mathbf{y}_3$

Figure 10: Generating $\mathbf{b}_2 = \lambda_\mathsf{w}^1 - \pi'\left(\lambda_\mathsf{v} - \mathbf{r}_2\right)$ towards P_2.

Generation of $\mathbf{b}_1$ towards P_1 The protocol appears in Fig. 11. Note that fresh random values $\mathbf{z}_i \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ for $i \in \{1,2,3\}$ are sampled to carry out this execution which are independent of the values used while generating $\mathbf{b}_2$ towards P_2.



Inputs
Local sampling
Round 1
Round 2
$\mathbf{z}_1, \mathbf{z}_2$
$\pi_1, \pi_2, \boldsymbol{\alpha}_1^1, \boldsymbol{\alpha}_1^2, \boldsymbol{\lambda}_\mathsf{w}^1, \boldsymbol{\lambda}_\mathsf{w}^2$
$\mathbf{x}_1 = \pi_2\left(\boldsymbol{\alpha}_1^1 + \boldsymbol{\alpha}_1^2 + \mathbf{z}_2\right)$
$\mathbf{x}_2 = \pi_1\left(\mathbf{x}_1 + \mathbf{z}_1\right)$
P3
$\mathbf{x}_2$
$\pi_2, \pi_3, \boldsymbol{\alpha}_1^1, \boldsymbol{\alpha}_1^3, \boldsymbol{\lambda}_\mathsf{w}^1, \boldsymbol{\lambda}_\mathsf{w}^3$
$\mathbf{z}_2, \mathbf{z}_3$
P1
$\mathbf{y}_1$
$\boldsymbol{\lambda}_\mathsf{w}^2 - \mathbf{y}_3$
P2
$\pi_1, \pi_3, \boldsymbol{\alpha}_1^2, \boldsymbol{\alpha}_1^3, \boldsymbol{\lambda}_\mathsf{w}^2, \boldsymbol{\lambda}_\mathsf{w}^3$
$\mathbf{z}_1, \mathbf{z}_3$
$\mathbf{y}_1 = \pi_2\left(\boldsymbol{\alpha}_1^3 - \mathbf{z}_2\right)$
$\mathbf{x}_3 = \pi_3\left(\mathbf{x}_2 - \mathbf{z}_3\right)$
$\mathbf{b}_1 = \boldsymbol{\lambda}_\mathsf{w}^2 - \mathbf{y}_3 - \mathbf{x}_3$
$\mathbf{y}_2 = \pi_1\left(\mathbf{y}_1 - \mathbf{z}_1\right)$
$\mathbf{y}_3 = \pi_3\left(\mathbf{y}_2 + \mathbf{z}_3\right)$

Figure 11: Generating $\mathbf{b}_1 = \lambda_\mathsf{w}^2 - \pi'\left(\lambda_\mathsf{v} - \mathbf{r}_1\right)$ towards P_1.

The complete shuffle protocol The ideal functionality for the secure shuffle appears in Fig. 12.

Functionality $\mathcal{F}_{\mathsf{Shuffle}}$

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Shuffle}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of the input $\mathbf{v}$ from all parties. Let $\mathbf{u}$ denote the randomly shuffled input. $\mathcal{F}_{\mathsf{Shuffle}}$ also receives from $\mathcal{S}$ its $[\![\cdot]\!]$-shares of $\mathbf{u}$.

$\mathcal{F}_{\mathsf{Shuffle}}$ proceeds as follows.

- Reconstruct input $\mathbf{v}$ using $[\![\cdot]\!]$-shares of the honest parties.
- Sample a random permutation π from the space of all permutations and generate $\mathbf{u} = \pi(\mathbf{v})$.
- Generate $[\![\cdot]\!]$-shares of $\mathbf{u}$ while accounting for shares received from $\mathcal{S}$. Let $[\![\mathbf{u}]\!]_x$ denotes the shares held by $P_x \in \mathcal{P}$.
- Send $(\mathsf{Output}, [\![\mathbf{u}]\!]_x)$ to P_x.

Figure 12: Ideal functionality for shuffle

The secure protocol for the preprocessing phase of shuffle appears in Fig. 13, while the protocol for the online phase appears in Fig. 14.

Protocol Π_{Shuffle}

Preprocessing

– P_0, P_1, P_3 sample π_2; P_0, P_2, P_3 sample π_2; P_0, P_1, P_2 sample π_3; and P_1, P_2, P_3 sample π_0, non-interactively.

– Define $\pi = \pi_0 \circ \pi_3 \circ \pi_1 \circ \pi_2$.

– P_0, P_1, P_3 sample $\lambda_\mathsf{w}^1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; P_0, P_2, P_3 sample $\lambda_\mathsf{w}^2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; and P_0, P_1, P_2 sample $\lambda_\mathsf{w}^3 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, non-interactively.

– P_i, P_3 for $i \in \{1,2\}$ invoke Π_{jsh} to generate $[\![\cdot]\!]$-shares of $\pi_0\left(\lambda_\mathsf{w}^i\right)$. Similarly, P_1, P_2 invoke Π_{jsh} to generate $[\![\cdot]\!]$-shares of $\pi_0\left(\lambda_\mathsf{w}^3\right)$.

//Generation of Π *towards* P_2, P_3

– P_0, P_1, P_3 randomly sample a permutation π_s.

– P_0, P_3 compute $\Pi = \pi_s \circ \pi_1 \circ \pi_2$ locally and invoke Π_{jmp} to send Π to P_2.

//Generation of $\mathbf{b}_1$ *towards* P_1

– P_0, P_2, P_3 non-interactively sample $\mathbf{r}_1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$.

– Parties non-interactively generate $\langle \mathbf{r}_1 \rangle$, and set $\langle \alpha_1 \rangle = \langle \lambda_{\mathsf{v}} \rangle - \langle \mathbf{r}_1 \rangle$.

– P_0, P_1, P_3 sample $\mathbf{z}_{21} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; P_0, P_2, P_3 sample $\mathbf{z}_{11} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; and P_0, P_1, P_2 sample $\mathbf{z}_{31} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, non-interactively.

– P_0, P_3 compute $\mathbf{x}_{11} = \pi_2(\alpha_1^1 + \alpha_1^2 + \mathbf{z}_{21})$, and $\mathbf{x}_{21} = \pi_1(\mathbf{x}_{11} + \mathbf{z}_{11})$, where α_1^1, α_1^2 denote two of the three $\langle \cdot \rangle$-shares of α_1. In parallel, P_0, P_1 compute $\mathbf{y}_{11} = \pi_2(\alpha_1^3 - \mathbf{z}_{21})$.

– P_0, P_3 invoke Π_{jmp} to send $\mathbf{x}_{21}$ to P_1, while P_0, P_1 invoke Π_{jmp} to send $\mathbf{y}_{11}$ to P_2.

– P_0, P_2 compute $\mathbf{y}_{21} = \pi_1(\mathbf{y}_{11} - \mathbf{z}_{11})$, and $\mathbf{y}_{31} = \pi_3(\mathbf{y}_{21} + \mathbf{z}_{31})$.

– P_0, P_2 invoke Π_{jmp} to send $\lambda_{\mathsf{w}}^2 - \mathbf{y}_{31}$ to P_1.

– P_1 computes $\mathbf{x}_{31} = \pi_2(\mathbf{x}_{21} - \mathbf{z}_{31})$, and sets $\mathbf{b}_1 = \lambda_{\mathsf{w}}^2 - \mathbf{y}_{31} - \mathbf{x}_{31}$.

//Generation of $\mathbf{b}_2$ *towards* P_2

– P_0, P_1, P_3 non-interactively sample $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$.

– Parties non-interactively generate $\langle \mathbf{r}_2 \rangle$, and set $\langle \alpha_2 \rangle = \langle \lambda_{\mathsf{v}} \rangle - \langle \mathbf{r}_2 \rangle$.

– P_0, P_1, P_3 sample $\mathbf{z}_{22} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; P_0, P_2, P_3 sample $\mathbf{z}_{12} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; and P_0, P_1, P_2 sample $\mathbf{z}_{32} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, non-interactively.

– P_0, P_3 compute $\mathbf{x}_{12} = \pi_2(\alpha_2^1 + \alpha_2^2 + \mathbf{z}_{22})$, and $\mathbf{x}_{22} = \pi_1(\mathbf{x}_{12} + \mathbf{z}_{12})$, where α_2^1, α_2^2 denote two of the three $\langle \cdot \rangle$-shares of α_2. In parallel, P_0, P_1 compute $\mathbf{y}_{12} = \pi_2(\alpha_2^3 - \mathbf{z}_{22})$.

– P_0, P_3 invoke Π_{jmp} to send $\mathbf{x}_{22}$ to P_1, while P_0, P_1 invoke Π_{jmp} to send $\mathbf{y}_{12}$ to P_2.

– P_0, P_1 compute $\mathbf{x}_{32} = \pi_3(\mathbf{x}_{22} - \mathbf{z}_{32})$, and invoke Π_{jmp} to send $\lambda_{\mathsf{w}}^1 - \mathbf{x}_{32}$ to P_2.

– P_2 computes $\mathbf{y}_{22} = \pi_1(\mathbf{y}_{12} - \mathbf{z}_{12})$, $\mathbf{y}_{32} = \pi_3(\mathbf{y}_{22} + \mathbf{z}_{32})$, and sets $\mathbf{b}_2 = \lambda_{\mathsf{w}}^1 - \mathbf{x}_{32} - \mathbf{y}_{32}$.

Figure 13: Preprocessing phase of secure shuffle protocol.

Protocol Π_{Shuffle}

Online

– P_1, P_3 compute and send $\mathbf{a}_2 = \pi_2(\mathbf{m}_{\mathsf{v}} + \mathbf{r}_2)$ to P_2 via Π_{jmp}. In parallel, P_2, P_3 compute and send $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\mathbf{m}_{\mathsf{v}} + \mathbf{r}_1)))$ via Π_{jmp} to P_1.

– P_2 computes $\mathbf{m}_{\mathsf{w}} = \pi_3(\pi_1(\mathbf{a}_2)) + \mathbf{b}_2 + \lambda_{\mathsf{w}}^2 + \lambda_{\mathsf{w}}^3$. P_1 computes $\mathbf{m}_{\mathsf{w}} = \pi_3(\pi_s^{-1}(\mathbf{a}_1)) + \mathbf{b}_1 + \lambda_{\mathsf{w}}^1 + \lambda_{\mathsf{w}}^3$.

– P_1, P_2 send $\mathbf{m}_{\mathsf{w}}$ to P_3 via Π_{jmp}.

– Parties non-interactively generate $[\![\cdot]\!]$-shares of $\mathbf{m}_{\mathsf{w}}$.

– Compute $[\![\mathbf{u}]\!] = [\![\mathbf{m}_{\mathsf{w}}]\!] - [\![\pi_0(\lambda_{\mathsf{w}}^1)]\!] - [\![\pi_0(\lambda_{\mathsf{w}}^2)]\!] - [\![\pi_0(\lambda_{\mathsf{w}}^3)]\!]$.

Figure 14: Online phase of secure shuffle protocol.

Communication and round complexity. Observe that the online phase involves sending a message of $\mathsf{N}\ell$ bits towards P_1, P_2 via Π_{jmp} in a single round of interaction. Since P_1, P_2 hold the required shares to evaluate the function under consideration, the Π_{jmp} towards P_3 can be deferred. Thus, in applications such as GraphSC that entail multiple shuffle invocations, the Π_{jmp} execution towards P_3 for the multiple shuffle instances can be performed in a single round, where each instance requires communication $\mathsf{N}\ell$ bits. In this way, the amortized online round complexity of our shuffle protocol is 1 round and has a communication of $3\mathsf{N}\ell$ bits. In the preprocessing phase, observe that generation of $[\![\cdot]\!]$-shares of $\pi_0(\lambda_{\mathsf{w}}^1), \pi_0(\lambda_{\mathsf{w}}^2), \pi_0(\lambda_{\mathsf{w}}^3)$ requires a total communication of $3\mathsf{N}\ell$ bits. Further, generating $\mathbf{b}_1, \mathbf{b}_2$ entails a total communication of $6\mathsf{N}\ell$ bits. Finally, sending $\pi_s \circ \pi_1 \circ \pi_2$ towards P_2 requires communicating $\mathsf{N}\ell$ bits. Thus, the total communication cost in the preprocessing phase is $10\mathsf{N}\ell$ bits.

C.2 Scatter-Gather primitives

The secure variants of the Scatter-Gather primitives can be obtained using the secure protocols for the operations therein. We note that the primitives are defined such that the number of iterations in each looping construct is dependent on a publicly known value ($|\mathcal{V}| + |\mathcal{E}|$ in this case). Further, all the private values such as the entries in G and other intermediate variables used in Scatter-Gather are operated on as secret-shares. The primitives additionally have branching statements such as the *if-else* construct. We rely on the Π_{sel} primitive (see Table 8) to obliviously evaluate only those steps within the correct branch of the construct. More specifically, every assignment operation within both the branches is evaluated via Π_{sel}. However, based on the branch condition provided as input to Π_{sel}, only those assignment statements where the condition is met is updated, while the others will remain unchanged.

D Building blocks

D.1 Formal protocols

Double bit injection The formal protocol appears in Fig. 15.

Protocol $\Pi_{2-\mathsf{bitInj}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{v}]\!])$

Preprocessing

– Invoke preprocessing phase of Π_{bit2A} to generate $\langle \mathsf{u_a} \rangle = \langle \lambda_{\mathsf{a}}^{\mathsf{R}} \rangle$ and $\langle \mathsf{u_b} \rangle = \langle \lambda_{\mathsf{b}}^{\mathsf{R}} \rangle$.

– Invoke Π_{MulR} to compute $\langle \mathsf{u_{ab}} \rangle = \langle \lambda_{\mathsf{a}}^{\mathsf{R}} \lambda_{\mathsf{b}}^{\mathsf{R}} \rangle$, $\langle \mathsf{u_{av}} \rangle = \langle \lambda_{\mathsf{a}}^{\mathsf{R}} \lambda_{\mathsf{v}} \rangle$, $\langle \mathsf{u_{bv}} \rangle = \langle \lambda_{\mathsf{b}}^{\mathsf{R}} \lambda_{\mathsf{v}} \rangle$ and $\langle \mathsf{u_{abv}} \rangle = \langle \mathsf{u_{ab}} \lambda_{\mathsf{v}}^{\mathsf{R}} \rangle$.

Online

Let $\mathsf{x_a} = \mathsf{m}_{\mathsf{a}}^{\mathsf{R}}, \mathsf{x_b} = \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}, \mathsf{x_{ab}} = (\mathsf{m_a m_b})^{\mathsf{R}}$, and $\mathsf{x_{abv}} = (\mathsf{m_a m_b})^{\mathsf{R}} \mathsf{m_v}$.

– P_1, P_2 compute $\mathsf{y}^1 = \mathsf{x}_{\mathsf{abv}} - \mathsf{x}_{\mathsf{ab}}\mathsf{u}^1_{\mathsf{v}} + \mathsf{x}_{\mathsf{a}}(1-2\mathsf{x}_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^1_{\mathsf{b}} - \mathsf{u}^1_{\mathsf{bv}}) + \mathsf{x}_{\mathsf{b}}(1-2\mathsf{x}_{\mathsf{a}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^1_{\mathsf{a}} - \mathsf{u}^1_{\mathsf{av}}) + (1-2\mathsf{x}_{\mathsf{a}})(1-2\mathsf{x}_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^1_{\mathsf{ab}} - \mathsf{u}^1_{\mathsf{abv}})$.
– P_2, P_3 compute $\mathsf{y}^2 = -\mathsf{x}_{\mathsf{ab}}\mathsf{u}^2_{\mathsf{v}} + \mathsf{x}_{\mathsf{a}}(1-2\mathsf{x}_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^2_{\mathsf{b}} - \mathsf{u}^2_{\mathsf{bv}}) + \mathsf{x}_{\mathsf{b}}(1-2\mathsf{x}_{\mathsf{a}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^2_{\mathsf{a}} - \mathsf{u}^2_{\mathsf{av}}) + (1-2\mathsf{x}_{\mathsf{a}})(1-2\mathsf{x}_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^2_{\mathsf{ab}} - \mathsf{u}^2_{\mathsf{abv}})$.
– P_3, P_1 compute $\mathsf{y}^3 = -\mathsf{x}_{\mathsf{ab}}\mathsf{u}^3_{\mathsf{v}} + \mathsf{x}_{\mathsf{a}}(1-2\mathsf{x}_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^3_{\mathsf{b}} - \mathsf{u}^3_{\mathsf{bv}}) + \mathsf{x}_{\mathsf{b}}(1-2\mathsf{x}_{\mathsf{a}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^3_{\mathsf{a}} - \mathsf{u}^3_{\mathsf{av}}) + (1-2\mathsf{x}_{\mathsf{a}})(1-2\mathsf{x}_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}}\mathsf{u}^3_{\mathsf{ab}} - \mathsf{u}^3_{\mathsf{abv}})$.
– $(P_1,P_2),(P_2,P_3)$ and (P_3,P_1) invoke Π_{jsh} on $\mathsf{y}^1, \mathsf{y}^2$ and y^3, respectively, to generate $[\![\mathsf{y}^1]\!], [\![\mathsf{y}^2]\!]$ and $[\![\mathsf{y}^3]\!]$.
– Output $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!] + [\![\mathsf{y}^3]\!]$.

Figure 15: Protocol for computing 2-bit-injection.

Prefix OR The protocol proceeds as follows. We define an operator Ψ which operates on a block of bits and outputs the prefix OR of the bits in the block. The Π_{preOR} protocol is designed to recursively call the Ψ operator such that in each round the size of the block increases exponentially, leading to a protocol with logarithmic rounds. Elaborately, our protocol proceeds in rounds such that after the j^{th} round, prefix OR of up to 4^j bits is computed. To achieve this, we define an operator Ψ which takes as input a block of bits t (where the number of bits in t is a power of 4), which is a concatenation of the bits in the four sub-blocks, b_3, b_2, b_1, b_0, i.e., $t = b_3||b_2||b_1||b_0$. Here, each sub-block, b_3, b_2, b_1, b_0, is such that it is already the prefix OR of some input sub-block, i.e. there exist input sub-blocks a_3, a_2, a_1, a_0 such that b_i is the prefix OR of a_i for $i \in \{0,1,2,3\}$. The operator Ψ is defined to output the prefix OR, t', of the bits in t. Computation of Ψ proceeds as follows. Let the last bit of sub-block b_i be denoted as z_i and let $t' = b'_3||b'_2||b'_1||b'_0$. Since each sub-block b_i already satisfies the prefix OR requirement, observe that we can set $b'_3 = b_3$. To compute the j^{th} bit in b'_2, it suffices to compute the OR of z_3 with the j^{th} bit in in b_2, because the latter is already the prefix OR of the bits in a_2. Similarly, j^{th} bit in b'_1 can be computed as the OR of $\mathsf{z}_3, \mathsf{z}_2$ with the j^{th} bit in in b_1. Finally, j^{th} bit in b'_0 can be computed as the OR of $\mathsf{z}_3, \mathsf{z}_2, \mathsf{z}_1$ with the j^{th} bit in in b_0. In this way, $t' = b'_3||b'_2||b'_1||b'_0$ can be generated by performing 2/3/4-input OR, which can be reduced to a combination of NOT and multi-input AND. Formal details of Ψ appear in Fig. 16. Having defined Ψ, we next describe how to compute prefix OR of $\{\mathsf{x}_i\}_{i=\ell-1}^{0}$ with the help of Ψ. The protocol proceeds in rounds, where in the j^{th} round, a processed version of $\{\mathsf{x}_i\}_{i=\ell-1}^{0}$ is split into $\ell/4^j$ blocks, t_i, each of which consists of 4^j bits. Observe that when each t_i comprises four bits (in the initial round), each sub-block b_i consists of a single bit and already satisfies the prefix OR requirement. Thus, applying Ψ on t_i ensures that the invariant of each input sub-block to Ψ being a prefix OR is satisfied. At the end of round 1, the application of Ψ on each of the four-bit block t_i, generates the prefix OR of the first four bits, as well as the prefix OR for each subsequent block of four bits. Thus, applying Ψ in the second round on each of the sixteen-bit blocks, generates the prefix OR for the first 16 bits, as well as the prefix OR for each subsequent block of 16 bits. In this way, at the end of j rounds, the prefix OR of 4^j bits can be generated. The protocol for computing prefix OR appears in Fig. 17. For our choice of $\mathsf{k} = 32$, we note that the prefix OR can be computed in 3 rounds.

Operator $\Psi\left([\![t]\!]^{\mathbf{B}}\right)$

Input: $[\![t]\!]^{\mathbf{B}} = \left([\![b_3]\!]^{\mathbf{B}}, [\![b_2]\!]^{\mathbf{B}}, [\![b_1]\!]^{\mathbf{B}}, [\![b_0]\!]^{\mathbf{B}}\right)$ where each b_i is block of d bits and constitutes the prefix OR of some sub-block of bits. Let the j^{th} bit in b_i be denoted as $b_{i,j}$ and its last bit be denoted as z_i.
Output: $[\![t']\!]^{\mathbf{B}}$ such that bits in t' comprise the prefix OR of the bits in t.

```
– For $i \in \{0,1,\ldots,\mathsf{d}-1\}$, set
  ◦ $[\![b'_{3,i}]\!]^{\mathbf{B}} = [\![b_{3,i}]\!]^{\mathbf{B}}$
  ◦ $[\![b'_{2,i}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{mult}}\left([\![\bar{\mathsf{z}_3}]\!]^{\mathbf{B}}, [\![\bar{b}_{2,i}]\!]^{\mathbf{B}}\right)\right)$
  ◦ $[\![b'_{1,i}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{3\text{-}mult}}\left([\![\bar{\mathsf{z}_3}]\!]^{\mathbf{B}}, [\![\bar{\mathsf{z}_2}]\!]^{\mathbf{B}}, [\![\bar{b}_{1,i}]\!]^{\mathbf{B}}\right)\right)$
  ◦ $[\![b'_{0,i}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{4\text{-}mult}}\left([\![\bar{\mathsf{z}_3}]\!]^{\mathbf{B}}, [\![\bar{\mathsf{z}_2}]\!]^{\mathbf{B}}, [\![\bar{\mathsf{z}_1}]\!]^{\mathbf{B}}, [\![\bar{b}_{0,i}]\!]^{\mathbf{B}}\right)\right)$
– Return $[\![t']\!]^{\mathbf{B}} = \left([\![b'_3]\!]^{\mathbf{B}}, [\![b'_2]\!]^{\mathbf{B}}, [\![b'_1]\!]^{\mathbf{B}}, [\![b'_0]\!]^{\mathbf{B}}\right)$
```

Figure 16: Operator Ψ.

Protocol $\Pi_{\mathsf{preOR}}\left(\{[\![\mathsf{x}_i]\!]\}_{i=\ell-1}^{0}\right)$

```
– $[\![b_i^0]\!]^{\mathbf{B}} = [\![\mathsf{x}_i]\!]^{\mathbf{B}}$ for $i \in \{0,\ldots,\ell-1\}$, and set $k = \ell$
– for $j = 0$ to $\lfloor \log_4(\ell) \rfloor$ do: ($j$ denotes round number)
◦ for $i = \left\lfloor \frac{k}{4} \right\rfloor$ to 1 do: ($i$ denotes block number)
  - $[\![t_i^j]\!]^{\mathbf{B}} = \left([\![b_{4i-1}^j]\!]^{\mathbf{B}}, [\![b_{4i-2}^j]\!]^{\mathbf{B}}, [\![b_{4i-3}^j]\!]^{\mathbf{B}}, [\![b_{4i-4}^j]\!]^{\mathbf{B}}\right)$
  - $[\![b_i^{j+1}]\!]^{\mathbf{B}} = \Psi\left([\![t_i^j]\!]^{\mathbf{B}}\right)$
◦ $k = \left\lfloor \frac{k}{4} \right\rfloor$
– Return $[\![b_3^{\lfloor \log_4(\ell) \rfloor + 1}]\!]^{\mathbf{B}}$
```

Figure 17: Protocol for computing prefix OR.

Exponentiation Our secure protocol for exponentiation, denoted as Π_{exp}, takes input $[\![\mathsf{x}]\!]$, and outputs $[\![e^{\mathsf{x}}]\!]$. Although our protocol is inspired from [2, 14, 18], it is much more efficient and also provides the desired accuracy as required for the considered applications. In the following, we first give an overview of the protocol of [2] followed by detailing our improvements over it.

To compute e^{x}, [2] proceeds as follows. It first computes the absolute value of x, denoted as $|\mathsf{x}|$, by obliviously selecting between $(\mathsf{x}, -\mathsf{x})$, depending on its sign. Then, $|\mathsf{x}|$ is split into its fractional (r) and integer (t) parts. Observe that $e^{|\mathsf{x}|} = e^{\mathsf{t}+\mathsf{r}} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$. Thus, the task reduces to computing e^{t} and e^{r}, where t, r are $[\![\cdot]\!]$-shared. e^{r} is computed using polynomial approximation. To compute e^{t}, t is decomposed into bits $\{\mathsf{t}_{\mathsf{k}-1}, \ldots, \mathsf{t}_0\}$, and bit-wise exponentiations are used as follows

$$e^{|\mathsf{t}|} = \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}} = \prod_{j=\mathsf{f}}^{\mathsf{k}-2} \left(\mathsf{t}_j \cdot \left(e^{2^{j-\mathsf{f}}} - 1\right) + 1\right)$$

where the value $e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}$ is computed by obliviously selecting between 1 and $e^{2^{j-\mathsf{f}}}$ with t_j as the selection bit.

Observe that if $\mathsf{x} > 0$, one can safely output $e^{|\mathsf{x}|} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$. However, if $\mathsf{x} < 0$, the value to be returned is $\frac{1}{e^{|\mathsf{x}|}}$. The latter additionally requires one call to secure division. Further, to ensure that no information about the sign of x is revealed, the work of [2] computes both the values $e^{|\mathsf{x}|}$ and $\frac{1}{e^{|\mathsf{x}|}}$, and then obliviously selects between them depending on the sign of x. The call to division increases the complexity of this protocol.

Unlike the approach of [2], we implicitly account for the sign of x while performing the bit-wise exponentiation. This allows us to explicitly avoid computing $\frac{1}{e^{|\mathsf{x}|}}$, and thereby the call to the secure division protocol. Elaborately, we rely on splitting x into its fractional (r) and integer (t) parts, rather than splitting $|\mathsf{x}|$, as done in [2]. The goal is to first compute $e^{\mathsf{t}}, e^{\mathsf{r}}$ and then $e^{\mathsf{x}} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$. We rely on Taylor series approximation to compute e^{r}, which implicitly accounts for the sign of x. To compute e^{t} while accounting for the sign, we make the following observation. Let $\{\mathsf{t}_i\}_{i=0}^{\mathsf{k}-1}$ denote the bits in t. Then,

$$e^{\mathsf{t}} = \begin{cases} \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{if } \mathsf{x} \geq 0 \\ \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{-\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{otherwise} \end{cases}$$

Note that selection between the two cases in the above equation are handled obliviously using Π_{sel} on inputs $e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}$, $e^{-\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}$ with the MSB of x as the selection bit. In this way, we avoid relying on a division protocol. Concretely, the value to be computed boils down to the following.

$$\begin{aligned} e^{\mathsf{t}} &= \prod_{j=\mathsf{f}}^{\mathsf{k}-2} \mathsf{t}_j \left(\mathsf{s}\left(\mathsf{e}^{-2^{j-\mathsf{f}}} - \mathsf{e}^{2^{j-\mathsf{f}}}\right) + \mathsf{e}^{2^{j-\mathsf{f}}} - 1\right) + 1 \\ &= \prod_{j=\mathsf{f}}^{\mathsf{k}-2} \mathsf{t}_j \left(\mathsf{s}\left(\mathsf{e}^{-2^{j-\mathsf{f}}} - \mathsf{e}^{2^{j-\mathsf{f}}}\right)\right) + \mathsf{t}_j \left(\mathsf{e}^{2^{j-\mathsf{f}}} - 1\right) + 1 \end{aligned}$$

where s denotes the sign bit of x and is a 1 if $\mathsf{x} < 0$. The formal protocol steps are provided in Fig. 18.

Protocol $\Pi_{\mathsf{exp}}(\llbracket \mathsf{x} \rrbracket)$

- $\llbracket \mathsf{t} \rrbracket = \Pi_{\mathsf{trunc}}(\llbracket \mathsf{x} \rrbracket, \mathsf{f}) \cdot 2^{\mathsf{f}}$
- $\llbracket \mathsf{r} \rrbracket = \llbracket \mathsf{x} \rrbracket - \llbracket \mathsf{t} \rrbracket$, $\llbracket \mathsf{t} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\llbracket \mathsf{t} \rrbracket)$ and $\llbracket \mathsf{s} \rrbracket^{\mathbf{B}} = \llbracket \mathsf{t}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}$
- for $i = 0$ to $\mathsf{k}-2$ do: $\llbracket \mathsf{t}_i \rrbracket^{\mathbf{B}} = \llbracket \mathsf{t}_i \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{s} \rrbracket^{\mathbf{B}}$
- for $j = \mathsf{f}$ to $\mathsf{k}-2$ do:
  - ◦ $\llbracket \mathsf{e}'_j \rrbracket = \Pi_{\mathsf{bitInj}}\left(\llbracket \mathsf{t}_j \rrbracket^{\mathbf{B}}, e^{2^{j-\mathsf{f}}} - 1\right)$
  - ◦ $\llbracket \mathsf{e}_j \rrbracket = \llbracket \mathsf{e}'_j \rrbracket + \Pi_{2-\mathsf{bitInj}}\left(\llbracket \mathsf{s} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{t}_j \rrbracket^{\mathbf{B}}, e^{-2^{j-\mathsf{f}}} - e^{2^{j-\mathsf{f}}}\right) + 1$
- set $\llbracket \mathsf{d} \rrbracket = \llbracket \mathsf{e}_f \rrbracket$
- for $j = \mathsf{f}+1$ to $\mathsf{k}-2$ do: $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{mult}}(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{e}_j \rrbracket, \mathsf{f})$
- $\llbracket \mathsf{z} \rrbracket = \Pi_{\mathsf{sel}}(1, 1/e, \llbracket \mathsf{s} \rrbracket^{\mathbf{B}})$ and $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{mult}}(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{z} \rrbracket, \mathsf{f})$
- set $\llbracket \mathsf{b}_0 \rrbracket = 1$, $\llbracket \mathsf{b}_1 \rrbracket = \llbracket \mathsf{r} \rrbracket$
- for $i = 2$ to θ do: $\llbracket \mathsf{b}_i \rrbracket = \Pi_{\mathsf{mult}}(\llbracket \mathsf{b}_{i-1} \rrbracket, \llbracket \mathsf{b}_1 \rrbracket, \mathsf{f})$
- $\llbracket \mathsf{b} \rrbracket = \sum_{i=0}^{\theta} \frac{\llbracket \mathsf{b}_i \rrbracket}{i!}$, $\llbracket \mathsf{g} \rrbracket = \Pi_{\mathsf{mult}}(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{b} \rrbracket, \mathsf{f})$

Figure 18: Protocol for exponentiation

While the high level approach of our protocol is similar to that of the one in [18], we note that our optimizations such as reliance on double bit injection, avoiding the need for an explicit bit extraction circuit for computing the MSB of x, etc., further aid in improving the efficiency of our exponentiation protocol. In our work, we take $\theta = 4$ since any higher value does not aid in improving the accuracy for GCNs.

Division When computing a/b where the divisor b is publicly known, and a is secret-shared, division can be performed easily by computing $1/\mathsf{b}$ on clear followed by secure multiplication with a. On the contrary, division when b is also secret-shared is non-trivial. Departing from the GC-based approach in Tetrad, we propose a division protocol, Π_{div}, that relies on Goldschmidt's approximation [30]. This protocol takes as input $\llbracket \mathsf{a} \rrbracket$ and $\llbracket \mathsf{b} \rrbracket$, and outputs $\llbracket \mathsf{d} \rrbracket$ where $\mathsf{d} \approx \mathsf{a}/\mathsf{b}$. Although one would expect the circuit-based approach via GC to provide higher accuracy, we note in Table 2 of the paper that we do not lose out on accuracy. Alternatively, [40] also provides a ring-based protocol for division. However, it suffers from requiring the *scaling factor* (described later) to be made public, whereas all intermediate values in our protocol are kept private. Further, [40] requires linear number of rounds as opposed to logarithmic in ours.

To design Π_{div}, we follow a similar approach as in [7], which involves computing an initial guess for $1/\mathsf{b}$, followed by iteratively computing the approximation of a/b. To ensure a fast convergence of this iterative method, the choice of the initial guess for $\mathsf{w} = 1/\mathsf{b}$ is critical. To compute w, we proceed along the lines of [7, 8]. We first compute the initial guess w' for the normalized input b', which is then used to obtain the initial guess w for the input b. Elaborately, if $\mathsf{b} > 0$, it is normalized to $\mathsf{b}' \in [0.5, 1)$ and $\mathsf{w}' = 1/\mathsf{b}'$ is approximated as $2.9142 - 2\mathsf{b}'$. Else, if $\mathsf{b} < 0$, it is normalized to $\mathsf{b}' \in (-1, -0.5]$ and $\mathsf{w}' = 1/\mathsf{b}'$ is approximated to $-2.9142 - 2\mathsf{b}'$. We refer readers to [8, 30] for the choice of the constant. Given w', the initial guess for $1/\mathsf{b}$ is computed as $\mathsf{w} = \mathsf{w}' \cdot \mathsf{v}$, where v is the scaling factor used to obtain the normalized $\mathsf{b}' = \mathsf{b}\mathsf{v}$. We let Π_{appRec} denote the protocol that computes the initial guess.

The following is the overview of Π_{appRec}. Observe that given v, computing w follows directly from the sequence of relations described earlier to compute b' and w'. Thus, the challenge lies in computing v, which is non-trivial to obtain when b is available only in secret-shared format. Observe that if $|\mathsf{b}| \in [2^{m-1}, 2^m - 1]$ ($|\mathsf{b}|$ denotes magnitude of b), then the scaling factor v is given by $2^{\mathsf{k}-m-1}$. Here $m-1$ denotes the index of the most significant non-zero bit of b if $\mathsf{b} > 0$, and the most significant zero bit of b if $\mathsf{b} < 0$. Consider the case when $\mathsf{b} > 0$. To determine m, we compute prefix OR of the bits of b to generate bits $\{\mathsf{c}_i\}_{i=0}^{\mathsf{k}-2}$ such that $\mathsf{c}_i = 0$ for $i \geq m$, and $\mathsf{c}_i = 1$ for $i < m$. Thus, the bits $\{\mathsf{c}_i\}_{i=0}^{\mathsf{k}-2}$, when composed, yield $\mathsf{c} = 2^m - 1$. Using this, $\mathsf{v} = 2^{\mathsf{k}-m-1}$ can be computed as follows. We XOR the consecutive bits in $\{\mathsf{c}_i\}_{i=0}^{\mathsf{k}-2}$ to generate $\{\mathsf{y}_i\}_{i=0}^{\mathsf{k}-2}$ which ensures that $\mathsf{y}_i = 1$ for $i = m$, and $\mathsf{y}_i = 0$ otherwise. Thus, composing the bits in

$\{y_i\}_{i=0}^{k-1}$ in the reverse order generates $v = 2^{k-m-1}$. The same steps can be used to compute v even when $b < 0$, provided we work on the flipped bits of b. Thus, we obliviously flip/retain the bits of b depending on its sign, before the prefix OR computation begins. Having computed v, we can compute b'. However, computing w' using b' additionally requires an oblivious selection between 2.1942 and -2.1942 depending on the sign of b. Then, computing w from w' follows easily. The formal protocol for computing w appears in Fig. 19. We use $(x)_f$ to denote that x has a precision of f bits. Note that this initial guess w of $1/b$ is known to have a relative error of $\varepsilon_0 = 1 - bw < 1$ [9].

Protocol $\Pi_{\mathsf{appRec}}(\llbracket b \rrbracket)$

```
– $\alpha = (2.9142)_{k-1}$
– $\llbracket b \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\llbracket b \rrbracket)$
– for $i = 0$ to $k-2$ do: $\llbracket b'_i \rrbracket^{\mathbf{B}} = \llbracket b_i \rrbracket^{\mathbf{B}} \oplus \llbracket b_{k-1} \rrbracket^{\mathbf{B}}$
– $\{\llbracket c_i \rrbracket\}_{i=k-2}^{0} = \Pi_{\mathsf{preOR}}(\{\llbracket b'_i \rrbracket^{\mathbf{B}}\}_{i=k-2}^{0})$ and $\llbracket c_{k-1} \rrbracket^{\mathbf{B}} = \llbracket 0 \rrbracket^{\mathbf{B}}$
– for $i = k-2$ to 1 do: $\llbracket c_i \rrbracket^{\mathbf{B}} = \llbracket c_i \rrbracket^{\mathbf{B}} \oplus \llbracket c_{i+1} \rrbracket^{\mathbf{B}}$
– $\llbracket v \rrbracket = \Pi_{\mathsf{B2A}}\left(\{\llbracket c_i \rrbracket^{\mathbf{B}}\}_{i=k-1}^{0}\right)$ (in reverse order)
– $\llbracket z \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}(\Pi_{\mathsf{mult}}(\llbracket b_{k-1} \rrbracket^{\mathbf{B}}, \llbracket c_0 \rrbracket^{\mathbf{B}}))$
– $\llbracket w' \rrbracket = \Pi_{\mathsf{sel}}(\alpha, -\alpha, \llbracket b_{k-1} \rrbracket^{\mathbf{B}}) - 2 \cdot \Pi_{\mathsf{mult}}(\llbracket b \rrbracket, \llbracket v \rrbracket, 0)$
– $\llbracket w \rrbracket = \Pi_{\mathsf{mult}}(\llbracket v \rrbracket, \llbracket w' \rrbracket, 2(k-f-1))$
– return $\llbracket w \rrbracket$, $\llbracket z \rrbracket^{\mathbf{B}}$
```

Figure 19: Protocol for computing initial approximation of $1/b$

Given the initial guess w, Π_{div} relies on Goldschmidt's method to output d which iteratively approximates a/b. The value of d in the θ^{th} iteration is computed as $d_\theta = d_{\theta-1} \cdot (1 - e_{\theta-1})$. Here, e_θ denotes the relative error in the θ^{th} iteration and can be obtained as $e_\theta = e_{\theta-1} \cdot e_{\theta-1}$. The algorithm begins with initializing $d_0 = a \cdot w$ and $e_0 = \varepsilon_0$. Observe that after θ iterations, d_θ has a relative error of $(\varepsilon_0)^{2^\theta}$, indicating that the error reduces exponentially. The formal protocol for division appears in Fig. 20. Since division by 0 is undefined, Π_{div} additionally outputs a flag bit z (computed as part of Π_{appRec}) which indicates if the divisor b is 0. For the applications considered, we note $\theta = 4$ suffices to obtain the desired level of accuracy. We remark that the round optimized ($\log_4(\ell)$ rounds) protocols for prefix OR and arithmetic to Boolean conversion aid in attaining improved round complexity.

Protocol $\Pi_{\mathsf{div}}(\llbracket a \rrbracket, \llbracket b \rrbracket)$

```
– $\llbracket w \rrbracket, \llbracket z \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{appRec}}(\llbracket b \rrbracket)$
– $\alpha = (1)_{2f}$ and $\llbracket e \rrbracket = \alpha - \Pi_{\mathsf{mult}}(\llbracket b \rrbracket, \llbracket w \rrbracket, 0)$
– $\llbracket d \rrbracket = \Pi_{\mathsf{mult}}(\llbracket a \rrbracket, \llbracket w \rrbracket, 0)$
– for $i = 1$ to $\theta$-1 do:
    ○ $\llbracket d \rrbracket = \Pi_{\mathsf{mult}}(\llbracket d \rrbracket, \alpha + \llbracket e \rrbracket, 2f)$, $\llbracket e \rrbracket = \Pi_{\mathsf{mult}}(\llbracket e \rrbracket, \llbracket e \rrbracket, 2f)$
– $\llbracket d \rrbracket = \Pi_{\mathsf{mult}}(\llbracket d \rrbracket, \alpha + \llbracket e \rrbracket, 2f)$
– return $\llbracket d \rrbracket, \llbracket z \rrbracket^{\mathbf{B}}$
```

Figure 20: Protocol for division

Inverse square root We follow along the lines of [29] which uses polynomial approximation to compute the inverse square root. Similar to division, to get a good approximation of y, the input a is first normalized to $a' \in (0.25, 0.5]$. Here, $a' = a \cdot v$, and $v = 2^{-(e+1)}$ is the scaling factor (where the most significant non-zero bit of a appears at index $e + f$ in the bit representation of a, assuming v has f bit precision). Then $1/\sqrt{a}$ is given by:

$$\frac{1}{\sqrt{a}} = \left(\frac{1}{\sqrt{a'}}\right) \cdot 2^{-(e+1)/2} \tag{8}$$

Here, $1/\sqrt{a'}$ is approximated using a low degree polynomial

$$g(a') = 4.63887a'^2 - 5.77789a' + 3.14736 \tag{9}$$

The protocol $\Pi_{\mathsf{PreInvSqrt}}$ computes the normalized input a'. Additionally, it computes $v' = 2^{-(e+1)/2}$, as required in Eq. (8). We now give an overview of how these two components are generated. To compute the normalized input $a' = a \cdot v$, we first compute $v = 2^{f-(e+1)}$ (accounting for f bit precision). For this, the input a is decomposed to obtain its bit-wise Boolean representation. These bits are manipulated using prefix OR and local operations to compute v. Similarly, $v' = 2^{f - \frac{(e+1)}{2}}$ is also computed with f bits of precision. Observe that v' can be computed by shifting the bit at index $e + f + 1$ to index $(e + f + 1)/2$. This is performed obliviously by computing the XOR of every consecutive pair of bits of v. Following this, the v' is multiplied by $2^{\frac{f}{2}}$ to compute $2^{f-(e+1)/2}$. Observe that when $(e + f + 1)$ is odd the value is computed correctly. However, when $(e + f + 1)$ is even, the value computed is offset by a factor of $\sqrt{2}$. Hence, v' should be instead multiplied by $2^{(\frac{f+1}{2})}$ to cancel out the offset. To handle both cases obliviously, we compute r which denotes the parity of $(e + f + 1)$. This is computed as the XOR of bits in v indexed with odd numbers. Then, invoking the $\Pi_{\mathsf{sel}}(2^{(\frac{f}{2})}, 2^{(\frac{f+1}{2})}, \llbracket r \rrbracket^{\mathbf{B}})$ allows computing the correct inverse square root of the scaling factor obliviously. Details of $\Pi_{\mathsf{PreInvSqrt}}$ appear in Fig. 21.

Protocol $\Pi_{\mathsf{PreInvSqrt}}(\llbracket a \rrbracket)$

```
– $\llbracket a \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\llbracket a \rrbracket)$
– $\{\llbracket c_i \rrbracket^{\mathbf{B}}\}_{i=k-2}^{0} = \Pi_{\mathsf{preOR}}\left(\{\llbracket a'_i \rrbracket^{\mathbf{B}}\}_{i=k-2}^{0}\right)$ and $\llbracket c_{k-1} \rrbracket^{\mathbf{B}} = \llbracket 0 \rrbracket^{\mathbf{B}}$
– $\llbracket v_{k-1} \rrbracket^{\mathbf{B}} = \llbracket c_{k-1} \rrbracket^{\mathbf{B}}$
– for $i = k-2$ to 1 do: $\llbracket v_i \rrbracket^{\mathbf{B}} = \llbracket c_i \rrbracket^{\mathbf{B}} \oplus \llbracket c_{i+1} \rrbracket^{\mathbf{B}}$
– $\llbracket v \rrbracket = \Pi_{\mathsf{B2A}}\left(\{\llbracket v_i \rrbracket\}_{i=2f-1}^{0}\right)$ (in reverse order)
– $\llbracket a' \rrbracket = \Pi_{\mathsf{mult}}(\llbracket a \rrbracket, \llbracket v \rrbracket, f)$
– $k' = \frac{k}{2}, f' = \frac{f}{2}, c_0 = 2^{\frac{f+1}{2}}, c_1 = 2^{\frac{f}{2}}$
– for $i = 0$ to $k'$ do: $\llbracket v'_i \rrbracket^{\mathbf{B}} = \llbracket v_{2i} \rrbracket^{\mathbf{B}} \oplus \llbracket v_{2i+1} \rrbracket^{\mathbf{B}}$
– $\llbracket r \rrbracket^{\mathbf{B}} = \llbracket v_1 \rrbracket^{\mathbf{B}} \oplus \llbracket v_3 \rrbracket^{\mathbf{B}} \oplus \llbracket v_5 \rrbracket^{\mathbf{B}} \oplus \ldots \llbracket v_{k-1} \rrbracket^{\mathbf{B}}$
– $\llbracket v' \rrbracket = \Pi_{\mathsf{B2A}}\left(\{\llbracket v'_i \rrbracket^{\mathbf{B}}\}_{i=2f'-1}^{0}\right)$
– $\llbracket v' \rrbracket = \Pi_{\mathsf{sel}}(c_0, c_1, \llbracket r \rrbracket) \cdot \llbracket v \rrbracket$ and return $\llbracket a' \rrbracket, \llbracket v' \rrbracket$
```

Figure 21: Sub-protocol for inverse square root

Given a', v', the protocol Π_{InvSqrt} computes the inverse square root of the input a as follows. The approximate inverse square root of the normalized input, denoted as y, is computed using the polynomial provided in Eq. (9). Following this, the inverse square root of the input a is given by $\frac{1}{\sqrt{\mathsf{a}}} = \mathsf{y} \cdot \mathsf{v}'$, which follows from Eq. (8). The formal protocol appears in Fig. 22.

Protocol $\Pi_{\mathsf{InvSqrt}}(\llbracket \mathsf{a} \rrbracket)$

- $\llbracket \mathsf{a}' \rrbracket, \llbracket \mathsf{v}' \rrbracket = \Pi_{\mathsf{PreInvSqrt}}(\llbracket \mathsf{a} \rrbracket)$
- $\llbracket \mathsf{y}_0 \rrbracket = \llbracket \mathsf{a}' \rrbracket, \llbracket \mathsf{y}_1 \rrbracket = \Pi_{\mathsf{mult}}(\llbracket \mathsf{a}' \rrbracket, \llbracket \mathsf{a}' \rrbracket, \mathsf{f})$
- $\llbracket \mathsf{y} \rrbracket = 4.63887 \llbracket \mathsf{y}_1 \rrbracket - 5.77789 \llbracket \mathsf{y}_0 \rrbracket + 3.14736$
- return $\Pi_{\mathsf{mult}}(\llbracket \mathsf{y} \rrbracket, \llbracket \mathsf{v}' \rrbracket, \mathsf{f})$

Figure 22: Protocol for computing inverse square root

D.2 Complexity

In this section, we briefly discuss the complexity of the proposed building blocks and primitives.

Double bit injection The protocol $\Pi_{2-\mathsf{bitInj}}$ requires communication of 3ℓ bits and 1 round in the online phase and $18\ell + 2$ bits in the preprocessing phase. This cost is explained as follows. In the online phase of $\Pi_{2-\mathsf{bitInj}}$ parties compute $\mathsf{y}_1, \mathsf{y}_2$ and y_3 locally. Following this the parties $(P_1, P_2), (P_2, P_3), (P_1, P_3)$ joint share $\mathsf{y}_1, \mathsf{y}_2$ and y_3 respectively in parallel. This requires 1 round and a communication of 3ℓ bits in the online phase. The communication cost for the preprocessing phase follows from Table 9.

Prefix OR The protocol Π_{preOR} requires communication of 432 bits and 3 rounds in the online phase and 1680 bits in the preprocessing phase. This cost is explained as follows. For our choice of $\mathsf{k} = 32$, the prefix OR can be computed in $\log_4(32) = 3$ rounds. Observe that each round has 16 invocations of Π_{mult}, $\Pi_{\mathsf{3\text{-}mult}}$, and $\Pi_{\mathsf{4\text{-}mult}}$ that are computed in parallel. Thus, the communication cost for the online phase is 432 bits and for the preprocessing phase is 1680 bits.

Exponentiation The protocol Π_{exp} requires communication of $156\ell + \mathsf{u}_1$ bits and 10 rounds in the online phase and $364\ell + \mathsf{u}'_1$ bits in the preprocessing phase. Here, $\mathsf{u}'_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_1 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [34] with $2, 3, 4$ inputs, respectively. This cost is explained as follows. Parties first truncate the input to compute the sharing of integer part ($\llbracket \mathsf{t} \rrbracket$) of the input, followed by a call to Π_{A2B} to get the bitwise sharing of the same. This requires 4 (i.e., 1+3) rounds and communication of $\mathsf{u}_1 + 3\ell$ bits (see Table 9) in the online phase. The parties then XOR the bits of $\llbracket \mathsf{t} \rrbracket$ with the MSB of $\llbracket \mathsf{t} \rrbracket$ which is computed non-interactively. Following this, the exponentiation for the integer part and the decimal part is computed in parallel. For the integer part, the bitwise exponentiation is computed in parallel. This requires one call to Π_{bitInj} and $\Pi_{2-\mathsf{bitInj}}$ with respect to each bit. For our choice of $\mathsf{k} = 32$ and $\mathsf{f} = 16$, there are 15 bits that correspond to the integer part. Hence, the total cost for this step is 1 round, and $15 \times (3\ell + 3\ell) = 90\ell$ bits of communication in the online phase. Following this, the parties multiply the computed bitwise exponentiation to obtain $\llbracket \mathsf{e}^{\mathsf{t}} \rrbracket$. This requires 15 multiplications which can be computed in 4 rounds, and communication of 45ℓ bits. To compute the exponentiation of the decimal part, the taylor series approximation is used. We use $\theta = 4$ that gives the desired accuracy. Each iteration has one multiplication. Thus, the communication cost for computing the exponentiation of the decimal part is 12ℓ bits in the online phase. Finally, the exponentiation of the integer part and the decimal part is multiplied to get the output which requires 1 round and 3ℓ bits of communication in the online phase. Thus, the protocol Π_{exp} requires 10 rounds and $364\ell + \mathsf{u}_1$ bits of communication in the online phase. The preprocessing cost follows from Table 9.

Division The protocol Π_{appRec} requires a communication of $15\ell + \mathsf{u}_1 + 432$ bits and 9 rounds in the online phase and $205\ell + \mathsf{u}'_1 + 1681$ bits in the preprocessing phase. Here, $\mathsf{u}'_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_1 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [34] with $2, 3, 4$ inputs, respectively. This cost is explained as follows. Parties run one instance of Π_{A2B} which requires a communication of u_1 bits (ref. Table 9) and 3 rounds in the online phase. This is followed by one instance of Π_{preOR}, which requires 3 rounds and 432 bits of communication. Next, the parties run one instance of Π_{B2A}, which requires 1 round and 3ℓ communication. Next, the parties run two instances of multiplication in parallel to compute $\llbracket \mathsf{z} \rrbracket^{\mathbf{B}}$ and $\llbracket \mathsf{w}' \rrbracket$ along with one instance of Π_{sel}. This requires 1 round and has a communication cost of 9ℓ bits. Finally, the parties run one instance of multiplication to compute the initial approximation $\llbracket \mathsf{w} \rrbracket$. This constitutes one round and 3ℓ bits of communication. Thus the total online cost for Π_{appRec} is 10 rounds and $15\ell + u_1 + 432$ bits in the online phase. The preprocessing cost follows from Table 9 and the cost of Π_{preOR}.

The protocol Π_{div} requires a communication of $42\ell + \mathsf{u}_1 + 432$ bits and 14 rounds in the online phase and $223\ell + \mathsf{u}'_1 + 1681$ bits in the preprocessing phase. This cost is explained as follows. Parties invoke 1 instance of Π_{appRec}, which has a communication cost of $15\ell + u_1 + 432$ and constitutes 9 rounds. This is followed by Goldschmidt's approximation. We observe that for our choice of $\mathsf{k} = 32$ and $\mathsf{f} = 16$, three iterations (i.e $\theta = 4$) of Goldschmidt's approximation is required to obtain good accuracy. In each iteration, the severs run 2 multiplications in parallel, which constitutes 1 round and communication of 6ℓ bits per round. Finally, the parties run another multiplication to get the final result. Thus the total number of rounds required is 14 and the communication

Building block	Online		Preprocessing
	Rounds	Comm. (in bits)	Comm. (in bits)
Joint sharing^{+}	1	2ℓ	-
Multiplication	1	3ℓ	2ℓ
3-input Multiplication	1	3ℓ	9ℓ
4-input Multiplication	1	3ℓ	24ℓ
MulR†	-	-	3ℓ
Multiplication with truncation	1	3ℓ	2ℓ
Bit to arithmetic	1	3ℓ	$3\ell+1$
Boolean to arithmetic	1	3ℓ	$192\ell+1$
Arithmetic to Boolean	$\log_4 \ell$	u_1	$\ell + u'_1$
Bit extraction	$\log_4 \ell$	u_2	$\ell + u'_2$
Bit injection	1	3ℓ	$6\ell+1$
Comparison	$\log_4 \ell$	u_2	$\ell + u'_2$
Oblivious select	1	3ℓ	$6\ell+1$

- ℓ: size of ring in bits, instantiated with $\ell = 64$ in our work
- $u'_1 = 2n_2 + 9n_3 + 24n_4$, $u_1 = 3(n_2+n_3+n_4)$ where $n_2 = 216$, $n_3 = 184$, $n_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [34] with $2,3,4$ inputs, respectively.
- $u'_2 = 2n_2 + 9n_3 + 24n_4$, $u_2 = 3(n_2+n_3+n_4)$ where $n_2 = 41, n_3 = 27, n_4 = 47$ denote the number of AND gates in the optimized bit extraction circuit of [34] with $2,3,4$ inputs, respectively.
$^{+}$ Joint-sharing a value v in the preprocessing phase where v is held by P_0 along with another party, has a communication cost of only ℓ bits. We refer readers to [23] for further details.
† The protocol MulR is only invoked in the preprocessing phase of $\Pi_{2-\mathsf{bitInj}}$

Table 9: Complexity of building blocks of Tetrad [23]

cost is $42\ell + u_1 + 432$. The preprocessing cost follows from Table 9 and the cost of Π_{appRec}.

Inverse square root The protocol $\Pi_{\mathsf{PreInvSqrt}}$ requires a communication of $9\ell + u_1 + 432$ bits and 8 rounds in the online phase and $387\ell + u'_1 + 1682$ bits in the preprocessing phase. Here, $u'_1 = 2n_2 + 9n_3 + 24n_4$, $u_1 = 3(n_2+n_3+n_4)$ where $n_2 = 216$, $n_3 = 184$, $n_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [34] with $2,3,4$ inputs, respectively. This cost is explained as follows. Parties invoke Π_{A2B} to decompose the input (a) into bits. This is followed by computing prefix OR of the bits and XORing the bits with the msb of the input, similar to the protocol Π_{appRec}. This requires 4 rounds and communication $u_1 + 432$ bits in the online phase. This is followed by an invocation of Π_{B2A} to compute the scaling factor v and a multiplication to compute the scaled input. This requires 2 rounds and communication of 6ℓ bits in the online phase. In parallel, the parties also compute the square root of the scaling factor(v'), which also requires one call to Π_{B2A}, which requires communication of 3ℓ bits. The bit r, denoting parity of the scaling factor, is computed non interactively. Thus the protocol $\Pi_{\mathsf{PreInvSqrt}}$ requires 8 rounds and $9\ell + 432 + u_1$ bits of communication. The preprocessing cost follows from Table 9 and the cost of Π_{preOR}.

The protocol Π_{InvSqrt} requires a communication of $15\ell + u_1 + 432$ bits and 10 rounds in the online phase and $391\ell + u'_1 + 1682$ bits in the preprocessing phase. This cost is explained as follows. Parties invoke $\Pi_{\mathsf{PreInvSqrt}}$ to compute the scaled input and the square root of scaling factor. Following this, parties compute the square of the scaled input which requires one multiplication. Following this the inverse square root of the scaled input is computed non interactively using the polynomial provided in (9). Finally, the parties invoke one call to multiplications to compute the output. Thus the protocol Π_{InvSqrt} requires 10 rounds and $15\ell + 432 + u_1$ bits of communication in the online phase. The preprocessing cost follows from Table 9 and the cost of $\Pi_{\mathsf{PreInvSqrt}}$.

E Discussion

GCN evaluation with 3PC Depending on the application scenario, one may wish to evaluate GCNs in the 3PC setting. Hence, for completeness, we also showcase the practicality of GCN evaluation in the 3PC setting by relying on the state-of-the-art robust framework of [22]. For this, we adapt the primitives designed in our work (exponentiation and inverse square root) as well as rely on the primitives from [38] (prefix

OR and division) and the shuffle protocol of [39] to enhance the 3PC of [22] to support GCN evaluation. The performance is reported in Table 10. As expected, observe that Entrada fares better than 3PC.

Variant	Preprocessing (s)	Online (s)
Inference	4.16	7.88
Training w/ Adam w/o GraphSC	57.97	277.04
Training w/ Adam w/ GraphSC	29.60	240.23

Table 10: GCN performance in 3PC.

GCN evaluation via GraphSC Observe that it is only the training of GCNs that leverages the GraphSC paradigm. One may be misled to believe that performing inference via GraphSC may also lead to improved efficiency. However, the justification for why this is not the case is as follows. Recall that inference is the forward pass sans the Softmax, which comprises matrix multiplications. Hence, relying on the matrix representation allows performing inference in 1 round of interaction and requires $O(n^2)$ communication and $O(n^3)$ computations. This is because there are $O(n^2)$ dot products that are required for matrix multiplication and all of these can be performed in parallel and hence require just 1 round of interaction. On the contrary, this round-efficient approach comes at the cost of prohibitively high memory, as required to store the underlying adjacency matrix. For instance, operating on the Yelp dataset (see §6.3) would require 50GB of memory at a processor when executing matrix multiplication via MPC protocol. This being the case, leveraging the multiprocessor setting to enhance the efficiency by performing the computations in parallel via m processors would require $m \times 50$GB memory which is prohibitively high. In fact, we observe that for the system configuration specified in the benchmarks, evaluating the GCN under the matrix representation for even two processors runs into insufficient memory issues. Thus, to capitalize on the multiprocessor setting, it is important to have a memory-efficient representation of the underlying graph. Moreover, the representation must also facilitate leveraging the multiprocessor setting. Since the GraphSC paradigm satisfies both the above requirements, we can rely on the same. That is, operating over the list representation of the graph as required by GraphSC only needs 47MB of memory at each processor when executing the MPC protocol. This amounts to a total of 3GB for the 64 vCPUs considered in the current setting. Further, GraphSC provides a way to translate matrix operations into Scatter-Gather operations on the list that are designed to leverage the multiprocessor setting. However, unlike the matrix representation that allows matrix multiplications to be performed in a single round, the GraphSC framework requires $O(\log(|\mathcal{V}|+|\mathcal{E}|))$ number of rounds. Further it requires $O(n^2)$ communication as well as computation, where $n = |\mathcal{V}|$. Thus, performing inference via GraphSC would incur additional overhead in comparison to directly operating on the matrices. On the contrary, training witnesses improvements via GraphSC (see Table 4, Table 7). The reason for the same is as follows. Recall that training comprises invocations of Softmax and the backward pass in addition to the steps of inference. The improvements brought in by the multiprocessor setting of GraphSC in the computation of Softmax significantly overpower the inefficiencies introduced during the inference phase. This is corroborated by the numbers reported in Table 4, Table 7.

Although one may be mislead to believe that performing inference part of forward pass via matrices and switching to GraphSC for the rest of the computations in training would be more efficient, we would like to note that this approach is computationally expensive. This is because, switching between the matrix and the list based representation would require expensive sort operations. For instance, updating entries from (GraphSC) list representation to the matrix representation in a data-oblivious manner would first require transforming the list of $O(|V|+|E|)$ entries to a list with $O(|V|^2)$ entries followed by sorting the latter list to obtain the matrix representation. Sorting the $O(|V|^2)$ list would thus require $O(|V|^2\log(|V|))$ computations. This would render the solution highly inefficient. Further, note that this operation of sort is required each time we switch between the representations and hence would be required in each epoch.

F Security proofs

The simulation-based security proofs for the designed primitives are presented in this section. At a high level, observe that the designed protocols rely on invoking protocols given in Tetrad [23] whose security was established therein in the standard real-world/ideal-world simulation paradigm. Hence, the security of the designed protocols follows directly from the security of the underlying protocols of Tetrad. We next elaborate on these.

We let the following denote the ideal functionalities for the protocols provided by Tetrad.

1. $\mathcal{F}_{\mathsf{mult\text{-}Tr}}$: This functionality takes as input $[\![\cdot]\!]$-shares of x, y and outputs $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ by truncated by f bits using probabilistic truncation.

2. $\mathcal{F}_{\mathsf{A2B}}$: This functionality takes as input $[\![\cdot]\!]$-shares of a value x, and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares for its equivalent Boolean representation.

3. $\mathcal{F}_{\mathsf{BitInj}}$: This functionality takes as input $[\![\cdot]\!]$-shares of x and $[\![\cdot]\!]^{\mathbf{B}}$-shares of a bit b and outputs $[\![\cdot]\!]$-shares of $\mathsf{b} \cdot \mathsf{x}$.

Prefix OR

The ideal functionality for prefix OR appears in Fig. 23.

Functionality $\mathcal{F}_{\mathsf{PreOr}}$

$\mathcal{F}_{\mathsf{PreOr}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]^{\mathbf{B}}$-shares of bits $\mathsf{x}_{\ell-1},\ldots,\mathsf{x}_0$ from all parties.

– Let $\mathsf{y}_{\ell-1},\ldots,\mathsf{y}_0$ denote the output bits. Receive from $\mathcal{S}$ its $[\![\cdot]\!]^{\mathbf{B}}$-shares of the output bits.

– Reconstruct $\mathsf{x}_{\ell-1},\ldots,\mathsf{x}_0$ using the shares of honest parties.

– Compute $\mathsf{y}_i = \vee_{j=i}^{\ell-1}\mathsf{x}_j$ for $i \in \{0,\ldots,\ell-1\}$.

– Generate $[\![\cdot]\!]^{\mathbf{B}}$-shares of y_i for $i \in \{0,\ldots,\ell-1\}$ accounting for the shares received from $\mathcal{S}$.

– Send $(\mathsf{Output}, [\![\mathsf{y}_i]\!]^{\mathbf{B}}_s)$ for $i \in \{0,\ldots,\ell-1\}$ to $P_s \in \mathcal{P}$.

Figure 23: Ideal functionality for prefix OR.

Lemma F.1 (Security)**.** *Protocol Π_{preOR} (Fig. 17) securely realizes $\mathcal{F}_{\mathsf{PreOr}}$ (Fig. 23) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{3\text{-}mult}}, \mathcal{F}_{\mathsf{4\text{-}mult}}, \mathcal{F}_{\mathsf{Not}})$-hybrid model.*

Proof. The simulator begins by defining the blocks and sub-blocks. The simulator then emulates $\mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{3\text{-}mult}}, \mathcal{F}_{\mathsf{4\text{-}mult}}, \mathcal{F}_{\mathsf{Not}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. □

Exponentiation

Ideal functionality for exponentiation appears in Fig. 24.

Functionality $\mathcal{F}_{\mathsf{Exp}}$

$\mathcal{F}_{\mathsf{Exp}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of x from all parties.

– Let g denote the output. Receive from $\mathcal{S}$ its $[\![\cdot]\!]$-shares of g.

– Reconstruct x using the shares of honest parties.

– Split x into its integer part t and the fractional part r such that $\mathsf{x} = \mathsf{t} + \mathsf{r}$.

– Compute e^{r} using Taylor series approximation up to θ terms.

– Let $\{\mathsf{t}_i\}_{i=0}^{\mathsf{k}-1}$ denote the bits in $|\mathsf{t}|$ (absolute value of t). Compute the following using probabilistic truncation after multiplication.

$$e^{\mathsf{t}} = \begin{cases} \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{if } \mathsf{x} \geq 0 \\ \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{-\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{otherwise} \end{cases}$$

– Compute $\mathsf{g} = e^{\mathsf{x}} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$.

– Generate $[\![\cdot]\!]$-shares of g while accounting for the shares received from $\mathcal{S}$.

– Send $(\mathsf{Output}, [\![\mathsf{g}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 24: Ideal functionality for exponentiation.

Lemma F.2 (Security)**.** *Protocol Π_{exp} (Fig. 18) securely realizes $\mathcal{F}_{\mathsf{Exp}}$ (Fig. 24) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{BitInj}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}})$-hybrid model.*

Proof. The simulator emulates $\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{BitInj}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. □

Division

The ideal functionalities for approximate reciprocal and division appear in Fig. 25, Fig. 26, respectively.

Functionality $\mathcal{F}_{\mathsf{AppRec}}$

$\mathcal{F}_{\mathsf{AppRec}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of b from all parties.

– Let w, z denote the outputs. Receive from $\mathcal{S}$ its $[\![\cdot]\!]^{\mathbf{B}}$-shares of the outputs.

– Reconstruct b using the shares of honest parties.

– Set $\mathsf{z} = 1$ if $\mathsf{b} = 0$, else set $\mathsf{z} = 0$. Compute $\mathsf{w} = 1/\mathsf{b}$ in fixed-point arithmetic representation as follows using probabilistic truncation when performing multiplication.

- If $\mathsf{b} \geq 0$, normalize it to $\mathsf{b}' \in [0.5, 1)$, else if $\mathsf{b} < 0$, normalize it to $\mathsf{b}' \in (-1, -0.5]$ by computing $\mathsf{b}' = \mathsf{b}\mathsf{v}$, where v is the scaling factor used to normalize b.
- If $\mathsf{b} \geq 0$, approximate $\mathsf{w}' = 1/\mathsf{b}'$ as $2.9142 - 2\mathsf{b}'$, else $\mathsf{w}' = 1/\mathsf{b}'$ is approximated to $-2.9142 - 2\mathsf{b}'$.
- Set $\mathsf{w} = \mathsf{w}' \cdot \mathsf{v}$, where v is the scaling factor used to obtain the normalized $\mathsf{b}' = \mathsf{b}\mathsf{v}$.

– Generate $[\![\cdot]\!]$-shares of w, z while accounting for the shares received from $\mathcal{S}$.

– Send $(\mathsf{Output}, [\![\mathsf{w}]\!]_s, [\![\mathsf{z}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 25: Ideal functionality for approximate reciprocal.

Lemma F.3 (Security)**.** *Protocol Π_{appRec} (Fig. 19) securely realizes $\mathcal{F}_{\mathsf{AppRec}}$ (Fig. 25) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Not}})$-hybrid model.*

Proof. The protocol does not involve any interaction apart from what is required while invoking the protocols for $\Pi_{\mathsf{A2B}}, \Pi_{\mathsf{B2A}}, \Pi_{\mathsf{preOR}}, \Pi_{\mathsf{mult}}, \Pi_{\mathsf{sel}}, \Pi_{\mathsf{NOT}}$. Hence, the simulator emulates $\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Not}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. □

Functionality $\mathcal{F}_{\mathsf{Div}}$

$\mathcal{F}_{\mathsf{Div}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of a, b from all parties.

– Let d, z denote the outputs. Receive from $\mathcal{S}$ its $[\![\cdot]\!]^{\mathbf{B}}$-shares of the outputs.

– Reconstruct a, b using the shares of honest parties.

– Set $\mathsf{z} = 1$ if $\mathsf{b} = 0$, else set $\mathsf{z} = 0$. Compute $\mathsf{d} = \mathsf{a}/\mathsf{b}$ using Goldschmidt's approximate division method in fixed-point arithmetic representation, where $1/\mathsf{b}$ is computed using the approximate reciprocal approach described in Fig. 25.

– Generate $[\![\cdot]\!]$-shares of d, z while accounting for the shares received from $\mathcal{S}$.

– Send $(\mathsf{Output}, [\![\mathsf{d}]\!]_s, [\![\mathsf{z}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 26: Ideal functionality for division.

Lemma F.4 (Security)**.** *Protocol Π_{div} (Fig. 20) securely realizes $\mathcal{F}_{\mathsf{Div}}$ (Fig. 26) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{AppRec}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}})$-hybrid model.*

Proof. The simulator begins by emulating $\mathcal{F}_{\mathsf{AppRec}}$. Following this, it emulates $\mathcal{F}_{\mathsf{mult\text{-}Tr}}$ as per its invocation in the real-world protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. □

Inverse square root

The ideal functionalities for $\Pi_{\mathsf{PreInvSqrt}}$ and inverse square root appear in Fig. 27, Fig. 28, respectively.

Lemma F.5 (Security)**.** *Protocol $\Pi_{\mathsf{PreInvSqrt}}$ (Fig. 21) securely realizes $\mathcal{F}_{\mathsf{PreInvSqrt}}$ (Fig. 27) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}})$-hybrid model.*

Proof. The simulator emulates $\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. □

Lemma F.6 (Security)**.** *Protocol Π_{InvSqrt} (Fig. 22) securely realizes $\mathcal{F}_{\mathsf{InvSqrt}}$ (Fig. 28) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{InvSqrt}}, \mathcal{F}_{\mathsf{mult\text{-}Tr}})$-hybrid model.*

Proof. The simulator begins by emulating $\mathcal{F}_{\mathsf{InvSqrt}}$. Following this, it emulates $\mathcal{F}_{\mathsf{mult\text{-}Tr}}$ as per its invocation in the real-world protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. □

Functionality $\mathcal{F}_{\mathsf{PreInvSqrt}}$

$\mathcal{F}_{\mathsf{PreInvSqrt}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of a from all parties.

– Let a', v denote the outputs. Receive from $\mathcal{S}$ its $[\![\cdot]\!]$-shares of the outputs.

– Reconstruct a using the shares of honest parties.

– Compute $\mathsf{a}' = \mathsf{a} \cdot \mathsf{v}$, which is a normalized to lie in $(0.25, -.5]$ in fixed-point arithmetic representation as follows using probabilistic truncation when performing multiplication.

∘ Compute the scaling factor $\mathsf{v} = 2^{-(e+1)}$, where the most significant non-zero bit of a appears at index $e + \mathsf{f}$ in the bit representation of a, (with v having f bit precision).

∘ Compute $\mathsf{a}' = \mathsf{a} \cdot \mathsf{v}$.

∘ Set $\mathsf{v}' = 2^{-(e+1)/2}$.

– Generate $[\![\cdot]\!]$-shares of a', v' while accounting for the shares received from $\mathcal{S}$.

– Send $(\mathsf{Output}, [\![\mathsf{a}']\!]_s, [\![\mathsf{v}']\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 27: Ideal functionality for $\Pi_{\mathsf{PreInvSqrt}}$.

Functionality $\mathcal{F}_{\mathsf{InvSqrt}}$

$\mathcal{F}_{\mathsf{InvSqrt}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of a from all parties.

– Let y denote the output. Receive from $\mathcal{S}$ its $[\![\cdot]\!]$-shares of the output.

– Reconstruct a using the shares of honest parties.

– Normalize a to $\mathsf{a}' \in (0.25, 0.5]$ and compute square root of the scaling factor v' as described in Fig. 27.

– Compute $\mathsf{y}' = 4.63887\mathsf{a}'^2 - 5.77789\mathsf{a}' + 3.14736$ and $\mathsf{y} = \mathsf{y}' \cdot \mathsf{v}'$ using probabilistic truncation after multiplication.

– Generate $[\![\cdot]\!]$-shares of y while accounting for the shares received from $\mathcal{S}$.

– Send $(\mathsf{Output}, [\![\mathsf{y}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 28: Ideal functionality for inverse square root.

Shuffle

Lemma F.7. *The shuffle protocol, Π_{Shuffle} (Fig. 13, Fig. 14) securely realizes the functionality $\mathcal{F}_{\mathsf{Shuffle}}$ (Fig. 12) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model.*

Proof. Let $\mathcal{S}$ represent the ideal-world adversary and $\mathcal{A}$ represent the adversary in the real world. $\mathcal{S}$ starts by simulating $\mathcal{F}_{\mathsf{setup}}$, where common keys are established with $\mathcal{A}$. These keys are used to sample the common randomness needed throughout the protocol. As a result, $\mathcal{S}$ is aware of all the randomness that $\mathcal{A}$ uses (more precisely, $\mathcal{S}$ is aware of shares of $\lambda_{\mathbf{v}}$, and $\mathsf{m}_{\mathbf{v}}$ held by $\mathcal{A}$) and can extract the input from $\mathcal{A}$ as well as to confirm the correctness of messages sent by $\mathcal{A}$. After this, it simulates the steps of the shuffle protocol. Since P_0 is active only in the preprocessing phase and it possesses all the input-independent data, security against P_0 follows easily. Simulation steps for a corrupt P_2 are provided in Fig. 29, where the corresponding simulator is denoted as $\mathcal{S}^{P_2}$. Simulation for a corrupt P_1 follows along the same lines as P_2.

Simulator $\mathcal{S}^{P_2}$

$\mathcal{S}^{P_2}$ proceeds as follows.

Preprocessing:

– Using the keys established via $\mathcal{F}_{\mathsf{setup}}$, sample the common randomness with $\mathcal{A}$.

– Simulate the steps of Π_{jsh} acting as the sender together with $\mathcal{A}$ to generate $[\![\cdot]\!]$-shares of $\pi_0\left(\lambda_{\mathsf{w}}^{i}\right)$ for $i \in \{2,3\}$.

– Simulate the steps of Π_{jsh} with $\mathcal{A}$ as the receiver to generate $[\![\cdot]\!]$-shares of $\pi_0\left(\lambda_{\mathsf{w}}^{1}\right)$.

//Simulation of generation of $\mathbf{b}_1$ *towards* P_1

– Sample a random π_s, π_2 and compute $\Pi = \pi_s \circ \pi_1 \circ \pi_2$, where π_1 is held by $\mathcal{A}$. Simulate the steps of Π_{jsh} acting as the sender to send Π to $\mathcal{A}$.

– Sample a random $\mathbf{z}_{21} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and compute $\mathbf{y}_{11} = \pi_2(\alpha_1^3 - \mathbf{z}_{21})$ as per the protocol. Simulate steps of Π_{jmp} acting as the sender to send $\mathbf{y}_{11}$ to $\mathcal{A}$.

– Compute $\mathbf{y}_{21} = \pi_1(\mathbf{y}_{11} - \mathbf{z}_{11})$ and $\mathbf{y}_{31} = \pi_3(\mathbf{y}_{21} + \mathbf{z}_{31})$ as per the protocol. Simulate steps of Π_{jmp} acting as the sender together with $\mathcal{A}$ to send $\lambda_{\mathsf{w}}^{2} - \mathbf{y}_{31}$ to P_1.

//Simulation of generation of $\mathbf{b}_2$ *towards* P_2

– Sample a random $\mathbf{z}_{22} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and compute $\mathbf{y}_{12} = \pi_2(\alpha_2^3 - \mathbf{z}_{22})$ as per the protocol. Simulate steps of Π_{jmp} acting as the sender to send $\mathbf{y}_{12}$ to $\mathcal{A}$.

– Compute $\mathbf{x}_{32} = \pi_3(\mathbf{x}_{22} - \mathbf{z}_{32})$ as per the protocol. Sample a random $\lambda_{\mathsf{w}}^{1} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and simulate steps of Π_{jmp} acting as the sender to send $\lambda_{\mathsf{w}}^{1} - \mathbf{x}_{32}$ to $\mathcal{A}$.

Online:

– Sample a random $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and compute $\mathbf{a}_2 = \pi_2(\mathbf{m}_{\mathsf{v}} + \mathbf{r}_2)$. Simulate the steps of Π_{jmp} acting as the sender to send $\mathbf{a}_2$ to $\mathcal{A}$.

– Compute $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\mathbf{m}_{\mathsf{v}} + \mathbf{r}_1)))$ as per the protocol, and simulate steps of Π_{jmp} acting as the sender together with $\mathcal{A}$ to send $\mathbf{a}_1$ to P_1.

– Compute $\mathbf{m}_{\mathsf{w}}$ as per the protocol, and simulate steps of Π_{jmp} acting as the sender together with $\mathcal{A}$ to send $\mathbf{m}_{\mathsf{w}}$ to P_3.

Figure 29: Simulator $\mathcal{S}^{P_2}$ for corrupt P_2 in Π_{Shuffle}

Observe that the messages received by $\mathcal{A}$ in the real-world comprise of the random $\mathbf{a}_2$ in the online phase. In the preprocessing phase, P_2 receives a random permutation Π, and $\mathbf{y}_{11}$, $\mathbf{y}_{12}$, $\lambda_{\mathsf{w}}^{1} - \mathbf{x}_{32}$ which come from a uniform random distribution. Observe that in the simulation in Fig. 29, these messages received by $\mathcal{A}$ continue to come from a uniform distribution. This is because each of these messages is generated by using randomness sampled from a uniform distribution by $\mathcal{S}^{P_2}$. Hence, the real-world and ideal world views for $\mathcal{A}$ are indistinguishable.

Simulator $\mathcal{S}^{P_3}$

$\mathcal{S}^{P_3}$ proceeds as follows.

Preprocessing:

– Using the keys established via $\mathcal{F}_{\mathsf{setup}}$, sample the common randomness with $\mathcal{A}$.

– Simulate the steps of Π_{jsh} acting as the sender together with $\mathcal{A}$ to generate $[\![\cdot]\!]$-shares of $\pi_0\left(\lambda_{\mathsf{w}}^{i}\right)$ for $i \in \{1,2\}$.

– Simulate the steps of Π_{jsh} with $\mathcal{A}$ as the receiver to generate $[\![\cdot]\!]$-shares of $\pi_0\left(\lambda_{\mathsf{w}}^{3}\right)$.

//Simulation of generation of Π *towards* P_2

– Compute $\Pi = \pi_s \circ \pi_1 \circ \pi_2$ as per the protocol. Simulate the steps of Π_{jmp} acting as the sender together wit $\mathcal{A}$ to send Π to P_1.

//Simulation of generation of $\mathbf{b}_1$ *towards* P_1

– Compute $\mathbf{x}_{21} = \pi_1(\mathbf{x}_{11} + \mathbf{z}_{11})$ as per the protocol. Simulate steps of Π_{jmp} acting as the sender along with $\mathcal{A}$ to send $\mathbf{x}_{21}$ to P_1.

//Simulation of generation of $\mathbf{b}_2$ *towards* P_2

– Compute $\mathbf{x}_{22} = \pi_1(\mathbf{x}_{12} + \mathbf{z}_{12})$ as per the protocol. Simulate steps of Π_{jmp} acting as the sender along with $\mathcal{A}$ to send $\mathbf{x}_{22}$ to P_1.

Online:

– Compute $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\mathbf{m}_{\mathsf{v}} + \mathbf{r}_1)))$ as per the protocol, and simulate steps of Π_{jmp} acting as the sender together with $\mathcal{A}$ to send $\mathbf{a}_1$ to P_1.

– Compute $\mathbf{a}_2 = \pi_2(\mathbf{m}_{\mathsf{v}} + \mathbf{r}_2)$ as per the protocol, and simulate steps of Π_{jmp} acting as the other sender together with $\mathcal{A}$ to send $\mathbf{a}_2$ to P_2.

– Compute $\mathbf{m}_{\mathsf{w}}$ as per the protocol, and simulate steps of Π_{jmp} acting as the sender to send $\mathbf{m}_{\mathsf{w}}$ to $\mathcal{A}$.

Figure 30: Simulator $\mathcal{S}^{P_3}$ for corrupt P_3 in Π_{Shuffle}

The simulation steps for a corrupt P_3 are provided in Fig. 30. Observe that the messages received by $\mathcal{A}$ in the real-world comprise of the random $\mathbf{m}_{\mathsf{w}}$ in the online phase. In the preprocessing phase, P_3 does not receive any values. Observe that in the simulation in Fig. 29, these messages received by $\mathcal{A}$ continue to come from a uniform distribution. This is because each of these messages is generated by using randomness sampled from a uniform distribution by $\mathcal{S}^{P_3}$. Hence, the real-world and ideal world views for $\mathcal{A}$ are indistinguishable.

□