Master Thesis

University of Bergen

Department of Informatics

Gradient Inversion in Decentralized Differentially Private Learning

Author:
Jonathan Ski

Supervisors:
Eirik Rosnes
Hsuan-Yin Lin

June, 2026

Abstract

Distributed machine learning enables multiple participants to jointly train a shared model without revealing their training data. While this keeps the training data private, updates to the shared model can still expose sensitive information.

Differential privacy (DP) is often used to mitigate this type of data leakage, but it remains unclear how well it protects against modern gradient inversion attacks. Decentralized learning can also use different communication and training methods, such as gossip-based model averaging or sequential training in a logical ring, and these choices can affect privacy.

This thesis examines gradient leakage in decentralized learning with DP. It focuses on gossip-based averaging with local noise and sequential training that uses token passing in a logical ring. By applying gradient reconstruction attacks, we evaluate how privacy budgets, communication patterns, and node participation affect both privacy leakage and model accuracy. The results show how decentralization, noise, and network structure influence resistance to reconstruction attacks.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Eirik Rosnes and Hsuan-Yin Lin, for their invaluable guidance, and support throughout this thesis.

I also gratefully acknowledge the use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053. The computational resources provided through eX3 were valuable for the work presented in this thesis.

Finally, I would like to thank my family for their continued support and encouragement throughout my studies.

`Jonathan Ski`
June 1, 2026

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

DP differential privacy

LDP local differential privacy

NDP network differential privacy

RDP Rényi differential privacy

ResNet residual network

FC NN fully connected neural network

CNN convolutional neural network

GNN graph neural network

GCN graph convolutional network

GAT graph attention network

MLP multilayer perceptron

SGD stochastic gradient descent

FL federated learning

FedSGD Federated SGD

FedAvg Federated Averaging

TabLeak Tabular Data Leakage

DLG Deep Leakage from Gradients

iDLG Improved Deep Leakage from Gradients

GLG Graph Leakage from Gradients

GraphDLG Graph Deep Leakage from Gradients

GRAIN Graph Reconstruction Algorithm for Inversion of Gradients

DFS depth-first search

BoW bag-of-words

RNMSE root normalized mean squared error

MSE mean squared error

MMD maximum mean discrepancy

SVHT singular value hard thresholding

Nomenclature

f_θ	Neural network
θ	Model parameter
ℓ	Loss function
L	Number of layers in a neural network
η	Learning rate
$\mathcal{B}$	Minibatch
B	Minibatch size
C	Gradient clipping bound
ζ	Learning rate schedule parameter for Skip-Ring
$\mathcal{D}$	Dataset
x	Input feature vector
g	Gradient of the loss
F	Global objective
F_v	Local empirical objective of client or node v
T	Number of iterations or rounds, defined locally
$\hat{x}$	Reconstructed or dummy input
$\mathcal{M}$	Mechanism
ε	Privacy budget
δ	Failure probability in (ε, δ)-DP
σ	Gaussian noise scale in (ε, δ)-DP
p	Skipping probability
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Graph with node set $\mathcal{V}$ and edge set $\mathcal{E}$
$\mathcal{N}(v)$	Neighborhood of node v
A	Adjacency matrix

X	Feature matrix
$H^{(l)}$	Node embeddings at layer l
$W^{(l)}$	Weight matrix of neural network layer l
W_{mix}	Gossip matrix
n	Number of nodes in a graph
d	Feature, embedding, or model-parameter vector dimension; the specific meaning is stated locally

Chapter 1

Introduction

1.1 Motivation

Machine learning systems often train on sensitive user data. Distributed and decentralized learning frameworks help protect privacy by keeping training data on local devices instead of storing it centrally. Still, recent studies show that gradients and model updates shared during training can reveal private information, even if the data stays on the device. Because of this risk, many distributed learning systems now use DP [1] as a defense.

Gradient-based reconstruction attacks have advanced quickly. The initial Deep Leakage from Gradients (DLG) attack in [2] was followed by attacks on images, text, tabular data, and graph-structured inputs [3–7], which can recover individual training samples from shared model updates. While DP provides formal worst-case guarantees, how well it defends against these attacks under realistic training conditions has not been thoroughly measured.

Most studies on gradient leakage and DP assume centralized training, in which an attacker can observe the exact model updates made by a client. Fully decentralized learning differs because communication depends on the network structure. Updates can be modified locally and mixed with others, and an attacker sees only the updates it receives within the network. These differences may affect both privacy risk and the effectiveness of DP.

This thesis examines two decentralized training protocols that change which updates an attacker can observe. The first is gossip-based averaging with local noise, in which peers repeatedly mix noisy updates. This approach can enhance privacy depending on the network topology and the distance between nodes, even if an attacker observes only a subset of updates. The second is sequential token passing on a logical ring, where updates are revealed only to nodes the token encounters, making privacy dependent on update scheduling. Both methods alter the visibility and frequency of model updates that depend on individual training data, thereby influencing the risk of reconstruction attacks and the trade-off between privacy and accuracy. We also include a centralized training baseline for comparison with decentralized training under DP.

Empirical evidence on how decentralized design choices affect the risk of reconstruction attacks in differentially private training remains limited. Such an understanding is necessary before deploying these systems in sensitive areas such as healthcare or finance.

1.2 Objective

This thesis evaluates DP noise as a defense against gradient inversion attacks in decentralized machine learning. It studies how privacy-preserving training affects both reconstruction risk and model accuracy.

The evaluation covers DLG, Tabular Data Leakage (TabLeak), Graph Reconstruction Algorithm for Inversion of Gradients (GRAIN), Graph Leakage from Gradients (GLG), and Graph Deep Leakage from Gradients (GraphDLG) on tabular and graph-structured data. It compares these attacks across centralized and decentralized training to examine how topology changes the information available to an adversary.

The thesis also examines how reconstruction risk varies across different training and data representations. This includes the amount of privacy noise, batch size, number of training epochs, feature encoding, and the degree to which nodes participate in training.

This thesis focuses on the empirical evaluation of existing attacks and training methods in realistic scenarios. It does not introduce new attacks or new privacy techniques.

1.3 Thesis Organization

- **Chapter 2:** introduces neural network architectures, distributed learning settings, and DP variants used throughout the thesis.
- **Chapter 3:** covers the two decentralized training protocols and five gradient-reconstruction attacks (DLG, TabLeak, GRAIN, GLG, and GraphDLG).
- **Chapter 4:** describes the experimental setup, including datasets, model architectures, training topologies, privacy mechanisms, and the threat model.
- **Chapter 5:** presents the experimental results and findings.
- **Chapter 6:** provides a discussion of the results.
- **Chapter 7:** provides the conclusion.

Chapter 2

Background

This chapter introduces fundamental concepts in neural networks, federated learning (FL), and DP, which are relevant to this thesis.

2.1 Neural Network Basics

In supervised learning, the goal is to learn a function $f_\theta : \mathcal{X} \to \mathcal{Y}$, where θ denotes the model parameters (also called weights), $\mathcal{X}$ denotes the space of input values, and $\mathcal{Y}$ denotes the space of output values. A pair (x_i, y_i) is called a *training example*, and the *training set* used to learn the model is

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{|\mathcal{D}|}.$$

In practice, the available data is split into disjoint subsets: the training set $\mathcal{D}$ is used to learn the model parameters, a separate *validation set* is used for model selection and hyperparameter tuning, and a *test set* is used to evaluate the final performance on unseen data.

We want to choose parameters θ so that the loss function is minimized:

$$\hat{\theta} = \arg\min_\theta \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \ell\big(f_\theta(x_i), y_i\big),$$

where $\ell(\cdot, \cdot)$ is a loss function measuring the discrepancy between predictions and labels.

The process of finding $\hat{\theta}$ is called *training*, and the optimization algorithm is called *gradient descent*. The idea is to iteratively update the model parameters in the direction of the steepest decrease of the loss function until we reach a (local) minimum. Gradient descent consists of two main steps:

- Compute the gradients of the loss with respect to the parameters, $\nabla_\theta \ell$. The procedure for computing this gradient is called the *backpropagation algorithm*.
- Update the parameters according to

  $$\theta = \theta - \eta \cdot \nabla_\theta \ell,$$

  where η is the learning rate or the step size of each update.

Once the training process is completed, the learned parameters $\hat{\theta}$ are fixed. The model can then be used to make a *prediction* for a new, unseen input $x \in \mathcal{X}$ by computing

$$\hat{y} = f_{\hat{\theta}}(x).$$

This step is called *inference* or *prediction*. The quality of the model is determined by how well these predictions generalize to data that was not part of the training set [8].

2.2 Model Architectures

This section describes the model architectures used in the experiments.

2.2.1 Logistic Regression

Logistic regression is a supervised machine learning algorithm for binary classification, with input space $\mathcal{X} = \mathbb{R}^d$, where d denotes the number of input features. Given an input vector $x \in \mathcal{X}$ and a binary label $y \in \{0, 1\}$, the model estimates the conditional probability of y by applying the logistic sigmoid to a linear combination of the input features.

Formally, the model is defined as

$$\Pr[y = 1 \mid x] = \mathrm{sig}(\theta^\top x),$$

where $\theta \in \mathbb{R}^d$ is the parameter vector, and $\mathrm{sig}(\cdot)$ is the sigmoid function

$$\mathrm{sig}(z) = \frac{1}{1 + \mathrm{e}^{-z}}.$$

The model parameters θ are optimized by minimizing the negative log-likelihood over the dataset $\mathcal{D}$. Using the logistic loss, the empirical objective over $\mathcal{D}$ is defined as

$$F_{\mathcal{D}}(\theta) = -\frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \Big[y \log \mathrm{sig}(\theta^\top x) + (1 - y) \log\big(1 - \mathrm{sig}(\theta^\top x)\big) \Big],$$

where $y \in \{0, 1\}$ denotes the binary label.

2.2.2 Neural Network

Neural networks are a class of supervised machine learning models designed to approximate complex, nonlinear mappings between inputs and outputs.

A *neuron* receives an input vector $x \in \mathbb{R}^d$, computes a weighted linear combination of these inputs, and applies a nonlinear activation function. Formally, a single neuron is defined as

$$h = a(w^\top x + b),$$

where $w \in \mathbb{R}^d$ is the weight vector, $b \in \mathbb{R}$ is the bias term, and $a(\cdot)$ denotes the activation function. A common choice for an activation function is ReLU, defined as

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{otherwise.} \end{cases}$$

The activation function introduces nonlinearity, which allows the model to learn nonlinear relationships between inputs and outputs.

A neural network is formed by stacking many such neurons into layers. Within a layer, the weight vectors of the individual neurons form the rows of a weight matrix $W^{(l)}$, and the activation $a(\cdot)$ is applied componentwise. A feedforward network in which every layer is fully connected is called a fully connected neural network (FC NN), or equivalently an multilayer perceptron (MLP). In an FC NN consisting of L layers, each layer applies a transformation to the output of the previous one as follows:

$$\begin{aligned} h^{(1)} &= a\left(W^{(1)}x + b^{(1)}\right), \\ h^{(2)} &= a\left(W^{(2)}h^{(1)} + b^{(2)}\right), \\ &\dots \\ f_\theta(x) &= a\left(W^{(L)}h^{(L-1)} + b^{(L)}\right), \end{aligned}$$

where $\theta = \{W^{(1)}, \dots, W^{(L)}, b^{(1)}, \dots, b^{(L)}\}$ denotes the collection of all network parameters.

If every element in one layer connects to every element in the next, the network is *fully connected*, as shown in Figure 2.1. The layers between the input and output are called *hidden layers*, and increasing their number or width enables the network to represent increasingly complex functions. Networks with one hidden layer are called *shallow neural networks*, while networks with multiple hidden layers are called *deep neural networks*.

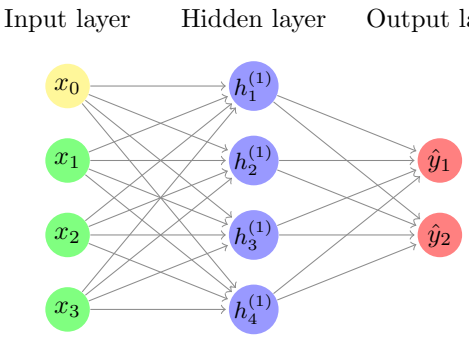

Figure 2.1: A shallow fully connected neural network.

Training the network consists of minimizing an empirical risk function, such as the *cross-entropy loss*, which is commonly used for classification tasks. For binary classification with labels $y \in \{0, 1\}$, the cross-entropy loss for a single sample is

$$\ell(f_\theta(x), y) = -\left(y \log f_\theta(x) + (1-y)\log(1 - f_\theta(x))\right),$$

where $f_\theta(x)$ is the network's predicted probability for class 1.

Optimizing θ is typically done using stochastic gradient descent (SGD), a variant of gradient descent introduced earlier. Instead of computing the gradient over the entire dataset, SGD estimates the gradient using a small minibatch $\mathcal{B}$ of size B. A *batch* is a subset of training examples drawn from the training set that the model processes in one update. One complete pass through the entire training dataset, typically processed in several batches, is called an *epoch*.

The parameter update becomes

$$\theta = \theta - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} \nabla_\theta \ell(f_\theta(x), y).$$

2.2.3 Convolutional Neural Network

Convolutional neural networks (CNNs) are a type of neural network designed for data with a grid-like structure, such as images. An image can be represented as a two-dimensional grid of values. A *convolutional layer* applies a small *kernel* to each local region of the image, and the same kernel weights are reused at every spatial position. This allows the network to detect the same pattern across the entire input while using far fewer parameters than a fully connected layer.

For illustration, consider a 3×3 kernel with weights W_{mn} and a bias b. The output at position (i, j) is computed as

$$h_{ij} = a\left(b + \sum_{m=1}^{3}\sum_{n=1}^{3} W_{mn}\, x_{i+m-2,j+n-2}\right),$$

where $a(\cdot)$ is the activation function.

Only the local 3×3 region around (i, j) contributes to h_{ij}. *Stride* controls how far the kernel moves between positions, and *padding* can be added to adjust the spatial size of the output.

Stacking multiple convolutional layers increases the region of the original image that influences each h_{ij}, known as the *receptive field*. Deeper layers can therefore combine simple local patterns into more abstract features. In many architectures, the final convolutional feature maps are *flattened* and passed to one or more fully connected layers, which perform the final classification or regression task.

2.2.4 Residual Network

Residual networks (ResNets) [9] are deep neural networks designed to facilitate the training of very deep models. When a network becomes very deep, optimization becomes difficult. The gradients can become very small, and the model may stop improving even when more layers are added. This problem is often referred to as the *vanishing gradient problem*.

ResNet solves this by introducing *residual connections*. Instead of each layer learning a full transformation of the input, it learns only a small change, called a residual. Formally, a residual block takes an input x and computes

$$h = a\Big(R(x) + x\Big),$$

where $R(x)$ is the output of a few stacked layers and x is added directly to this output. The shortcut from the input x to the output is called a *skip connection*. This allows the network to pass information forward without forcing every layer to transform it. As a result, very deep networks can be trained without losing important gradient information.

By stacking many residual blocks, ResNet models can become very deep while remaining stable during training. This is why ResNet is widely used in image classification tasks and often serves as a backbone in many modern computer vision architectures.

2.2.5 Graph Neural Networks

Graph neural networks (GNNs) are a class of neural network models designed for data that is naturally represented as graphs. In contrast to CNNs, which typically assume that input samples are independent, GNNs explicitly model dependencies between entities by exploiting the relational structure encoded in a graph.

Depending on the formulation, GNNs may be trained on a fixed, fully observed graph or on collections of graphs to generalize to unseen graphs. Most modern GNNs can be interpreted as message-passing architectures, where nodes iteratively exchange and aggregate information with their neighbors.

Graph Representation

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote an undirected graph, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. Here, $n = |\mathcal{V}|$ denotes the number of nodes in the graph. Two nodes are said to be *neighbors* if an edge connects them, and the neighborhood of a node v_i is denoted by $\mathcal{N}(v_i)$.

Each node v_i is associated with a feature vector $x_i \in \mathbb{R}^d$, where d is the dimensionality of the node features.

Stacking all node features yields the *feature matrix* $X \in \mathbb{R}^{n \times d}$. The structure of the graph is encoded by an *adjacency matrix* $A \in \{0,1\}^{n \times n}$ defined as

$$A_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \in \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases}$$

Intuitively, X describes the attributes of the nodes, while A encodes which pairs of nodes are connected. Together, X and A fully specify the input to a GNN.

A GNN processes the pair (X, A) through a sequence of L layers. At each layer, node representations are updated by applying a trainable graph layer to the current node representations and the adjacency matrix:

$$H^{(0)} = X, \qquad H^{(l+1)} = \mathrm{GNN}(\theta^{(l)}; H^{(l)}, A).$$

Here, $H^{(l)}$ denotes the node representations at depth l. The parameter $\theta^{(l)}$ denotes the parameters of layer l, and $\mathrm{GNN}(\cdot)$ is used as a generic notation for a graph neural network layer.

Initially, each node representation contains only local information. As the number of layers increases, node embeddings progressively incorporate information from larger neighborhoods in the graph. As a result, the final node embeddings encode both node features and structural context.

Tasks

Node-level tasks. In node-level tasks, the objective is to predict a label or a set of values for each node in the graph. The output of the final layer is used directly for node classification or regression.

Graph-level tasks. In graph-level tasks, the goal is to predict a label or value for an entire graph. This is achieved by aggregating the final node representations $H^{(L)}$ into a fixed-size graph representation $H_{\mathcal{G}}$ via a permutation-invariant readout, which is then passed to an output head.

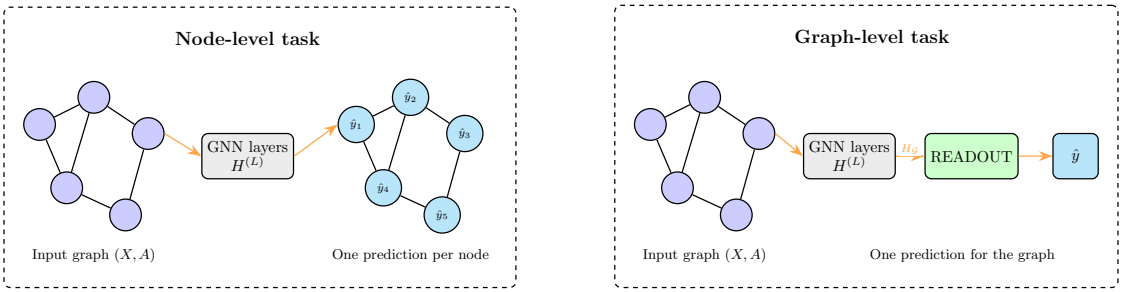

Figure 2.2: Node-level and graph-level GNN tasks. Both apply L GNN layers to the input graph (X, A) to obtain layer-L node embeddings $H^{(L)}$. Node-level tasks output one prediction $\hat{y}_i$ per node. Graph-level tasks, such as in GraphDLG, pool $H^{(L)}$ into a single graph embedding $H_{\mathcal{G}}$ via a permutation-invariant readout, which is then mapped to a graph-level prediction $\hat{y}$.

Graph Convolutional Networks

Graph convolutional networks (GCNs) are a specific class of GNNs that update node representations by applying a convolution-like operation over the graph [10]. The key idea is to combine node features with those of neighboring nodes using a normalized adjacency matrix.

To ensure that each node contributes its own features during aggregation, self-loops are added to the graph. This yields the augmented adjacency matrix:

$$\tilde{A} = A + I_n,$$

where I_n is the $n \times n$ identity matrix. The corresponding diagonal degree matrix $\tilde{D}$ has entries

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}.$$

A single GCN layer is then defined as

$$H^{(l+1)} = a\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}\right),$$

where $W^{(l)}$ is a learnable weight matrix, and $a(\cdot)$ is the activation function.

Graph Attention Networks

Graph attention networks (GATs) extend the GNN framework by allowing nodes to assign different importance to their neighbors during aggregation [11]. Instead of relying on a fixed normalization determined by the graph structure, GATs learn attention coefficients that govern how information flows between connected nodes.

At each layer, node features are first linearly transformed:

$$\tilde{h}_i^{(l)} = W^{(l)} h_i^{(l)},$$

where $h_i^{(l)}$ denotes the representation of node v_i at layer l, and $W^{(l)}$ is a learnable weight matrix shared across all nodes.

For each node v_i and its neighbor $v_j \in \mathcal{N}(v_i)$, including v_i itself, an unnormalized attention score is computed as

$$e_{ij}^{(l)} = \phi\left(\tilde{h}_i^{(l)}, \tilde{h}_j^{(l)}\right),$$

where $\phi(\cdot, \cdot)$ is a learnable attention mechanism, typically implemented as a small feedforward neural network followed by a nonlinear activation.

The attention scores are normalized over the neighborhood of node v_i using the softmax function:

$$\alpha_{ij}^{(l)} = \left(\text{softmax}\left(\{e_{ik}^{(l)}\}_{k \in \mathcal{N}(v_i)}\right)\right)_j = \frac{\exp\left(e_{ij}^{(l)}\right)}{\sum_{k \in \mathcal{N}(v_i)} \exp\left(e_{ik}^{(l)}\right)}.$$

The updated node representation is computed as a weighted sum of transformed neighbor features:

$$h_i^{(l+1)} = a\left(\sum_{j \in \mathcal{N}(v_i)} \alpha_{ij}^{(l)} \tilde{h}_j^{(l)}\right).$$

In practice, models often use multiple attention heads and combine their outputs by concatenation or averaging.

2.3 Encoding

The input vector x consists of a set of features, which may be continuous or categorical. For instance, a housing dataset may include continuous features such as floor area and construction year, and categorical features such as building type or location.

Continuous features take numerical values, while categorical features represent distinct groups without any inherent numerical order. Because neural networks require numerical inputs, categorical features must be encoded as numerical values.

The choice of encoding method influences both the size of the input space and how the model interprets categorical information. As will be shown in Section 5.2, the encoding strategy also affects the reconstruction accuracy of gradient inversion attacks. Here, we describe the encoding methods considered in this work.

Bag-of-words encoding: For text-derived inputs with a fixed vocabulary of size V, each sample is represented as a vector in $\mathbb{R}^V$ whose entries correspond to word occurrence counts. This representation ignores word order and yields a sparse, high-dimensional input vector.

One-hot encoding: A categorical feature with K distinct categories is represented as a binary vector of length K with exactly one entry equal to 1 and all others equal to 0. This representation assigns a distinct coordinate to each category, but increases dimensionality linearly with the number of categories.

Binary encoding: Each category is first assigned an integer index and then represented using its binary expansion. This reduces the dimensionality of the encoded vector from K (under one-hot encoding) to $\lceil \log_2 K \rceil$. However, it introduces an artificial ordinal structure through the binary representation. We also consider a randomized variant in which categories are assigned random binary codes, removing the ordinal structure while preserving logarithmic dimensionality.

Hash encoding: Hash encoding maps categories into a fixed number of dimensions using a hash function. In our implementation, we use `FeatureHasher` from scikit-learn [12], which employs the signed 32-bit MurmurHash3 variant. The resulting representation has constant width regardless of the number of unique categories. Since multiple categories may map to the same hash bucket, collisions can occur, making distinct categories indistinguishable in the encoded representation.

2.3.1 Encoding Example

To illustrate the different encoding methods, consider a categorical feature:

$$x \in \{\text{Red}, \text{Green}, \text{Blue}\}.$$

One-hot encoding. Table 2.1 shows the one-hot representation. Each category is assigned its own coordinate, so the vector must have one entry for every possible category. As a result, the representation length grows directly with the number of categories.

Category	b_1	b_2	b_3
Red	1	0	0
Green	0	1	0
Blue	0	0	1

Table 2.1: One-hot encoding.

Binary encoding. In binary encoding, each category is first assigned an integer index and then represented using its binary form. The resulting codes therefore follow the chosen index order. For example, if Red is assigned index 0, it is represented as $[0, 0]$, as shown in Table 2.2. Green and Blue receive the subsequent binary codes according to their assigned indices.

Category	b_1	b_2
Red	0	0
Green	0	1
Blue	1	0

Table 2.2: Binary encoding.

In random binary encoding, the representation length remains the same, but the binary codes are assigned arbitrarily rather than following the index order. This removes the fixed ordinal structure introduced by standard binary encoding. Table 2.3 shows one possible random assignment of codes.

Category	b_1	b_2
Red	1	1
Green	0	1
Blue	1	0

Table 2.3: Random binary encoding.

Hash encoding. Table 2.4 shows an example of hash encoding produced by the `FeatureHasher` implementation used in our experiments, rather than by a manually chosen mapping. To illustrate a collision, we use a hash width of 2 in this example. With a larger width, such as 3, the values Red, Green, and Blue would map to different positions, so no collision would occur. More generally, smaller hash widths make collisions more likely, whereas larger widths reduce the collision rate at the cost of higher dimensionality.

Category	h_1	h_2
Red	0	1
Green	0	1
Blue	1	0

Table 2.4: Hash encoding. Red and Green collide in this example.

Bag-of-words. Unlike the categorical encodings above, which assign a vector to one categorical feature value, bag-of-words (BoW) assigns a vector to a complete sample by counting how often each vocabulary word appears. This is why it is commonly used for text data, where each sample consists of multiple words rather than a single category. Table 2.5 shows example BoW representations for samples containing the words "Red", "Green", and "Blue".

Sample	Red	Green	Blue
Red, Blue, Red	2	0	1
Green, Blue	0	1	1
Blue, Blue	0	0	2

Table 2.5: BoW encoding.

2.4 Distributed Machine Learning

In a traditional centralized setup, all training data is collected and stored on a single machine or within a single data center. Training large models in this setting is demanding, as a single machine must handle both data storage and computation. Distributed training addresses this limitation by spreading computation across multiple devices or machines, allowing them to collaborate on a shared learning task.

2.4.1 Federated Learning

FL is a distributed learning approach in which training data remains on users' devices [13]. Instead of sending raw data to a central location, each device trains locally on its own dataset and communicates only model updates to a central server.

We consider a fixed set of clients $\mathcal{V}$ with $|\mathcal{V}| = n$. Each client $v \in \mathcal{V}$ holds a local dataset $\mathcal{D}_v$. In each training round t, the server selects a subset of clients $\mathcal{S}^{(t)} \subseteq \mathcal{V}$ and sends them the current global model parameters $\theta_{\mathrm{g}}^{(t)}$. Each selected client trains the model using its local data and returns an update. The server aggregates these updates to produce a new global model. This procedure is repeated until convergence.

The goal of FL is to learn model parameters $\hat{\theta}$ that minimize the average objective across all clients:

$$\hat{\theta} = \arg\min_{\theta} F(\theta), \quad F(\theta) = \frac{1}{n} \sum_{v \in \mathcal{V}} F_v(\theta).$$

Here, $F_v(\theta)$ denotes the local objective of client v. For a local dataset $\mathcal{D}_v$, this objective is defined as

$$F_v(\theta) = \frac{1}{|\mathcal{D}_v|} \sum_{(x,y) \in \mathcal{D}_v} \ell\big(f_\theta(x), y\big),$$

where $\ell(\cdot, \cdot)$ denotes the loss function and $f_\theta(x)$ is the model prediction for input x.

When a client update is computed on a minibatch $\mathcal{B}_v \subseteq \mathcal{D}_v$, we write

$$F_v(\theta; \mathcal{B}_v) = \frac{1}{|\mathcal{B}_v|} \sum_{(x,y) \in \mathcal{B}_v} \ell\big(f_\theta(x), y\big).$$

Thus, $\nabla F_v(\theta; \mathcal{B}_v)$ is the minibatch gradient used in the implementation. Throughout this thesis, F and F_v denote objective functions, while f_θ denotes the predictive model.

Two well-known training strategies for FL are Federated SGD (FedSGD) and Federated Averaging (FedAvg).

FedSGD

FedSGD extends SGD to the federated setting. Each selected client $v \in \mathcal{S}^{(t)}$ computes the gradient of its local objective using the current global parameters:

$$\nabla F_v\left(\theta_{\mathrm{g}}^{(t)}\right).$$

The server then forms a weighted average of the client gradients and updates the global model:

$$\theta_{\mathrm{g}}^{(t+1)} = \theta_{\mathrm{g}}^{(t)} - \eta \sum_{v \in \mathcal{S}^{(t)}} \frac{|\mathcal{D}_v|}{|\mathcal{D}_{\mathcal{S}^{(t)}}|} \nabla F_v\left(\theta_{\mathrm{g}}^{(t)}\right),$$

where

$$|\mathcal{D}_{\mathcal{S}^{(t)}}| = \sum_{v \in \mathcal{S}^{(t)}} |\mathcal{D}_v|.$$

This update is equivalent to computing a gradient using the combined data of all participating clients, but without transferring any individual data sample. A limitation of FedSGD is that communication is required after every gradient computation, which can become expensive when many training rounds are needed.

FedAvg

FedAvg reduces communication overhead by allowing clients to perform multiple local updates before communicating with the server. At the beginning of round t, each selected client receives the current global parameters $\theta_{\mathrm{g}}^{(t)}$ and performs several steps of SGD on its local dataset. This produces updated local parameters $\theta_v^{(t+1)}$ for each client $v \in \mathcal{S}^{(t)}$.

After completing its local training, each client sends its updated parameters back to the server. The server then computes the new global model by averaging the client models, weighted by the number of samples held by each client:

$$\theta_{\mathrm{g}}^{(t+1)} = \sum_{v \in \mathcal{S}^{(t)}} \frac{|\mathcal{D}_v|}{|\mathcal{D}_{\mathcal{S}^{(t)}}|} \theta_v^{(t+1)}.$$

By allowing clients to perform multiple local updates between communication rounds, FedAvg often achieves good performance with fewer communication exchanges. This is useful in federated systems where communication is a major bottleneck and client data distributions are often heterogeneous.

2.4.2 Decentralized Learning

Decentralized learning removes the need for a central server. As illustrated in Figure 2.3, clients communicate directly with one another according to a predefined communication graph. Each client maintains its own local model and dataset. Learning proceeds through repeated local training and communication with neighboring clients.

We consider a fixed set of n clients $\mathcal{V} = \{v_1, \ldots, v_n\}$ connected by an undirected communication graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ represents one client and holds its own local dataset and model. Each edge in $\mathcal{E}$ indicates that two clients can exchange model information. During each training round t, client v updates its local model $\theta_v^{(t)}$ using its own local data and then exchanges model information with its neighbors in $\mathcal{G}$.

The objective in decentralized learning is the same as in FL. All clients aim to converge to a shared model that minimizes:

$$F(\theta) = \frac{1}{n} \sum_{v \in \mathcal{V}} F_v(\theta).$$

The key difference is that no central server coordinates the aggregation process. Instead, agreement between clients emerges through repeated local communication steps.

This design avoids a single central aggregation point. Communication is distributed across the network rather than concentrated on one machine. However, decentralized learning often converges more slowly than centralized FL, since information must propagate through the network gradually.

Many decentralized learning algorithms build on gossip algorithms and decentralized optimization. Although specific update rules vary across methods, the general pattern is the same. Clients perform local training and periodically exchange model information with their neighbors, gradually driving the system toward consensus.

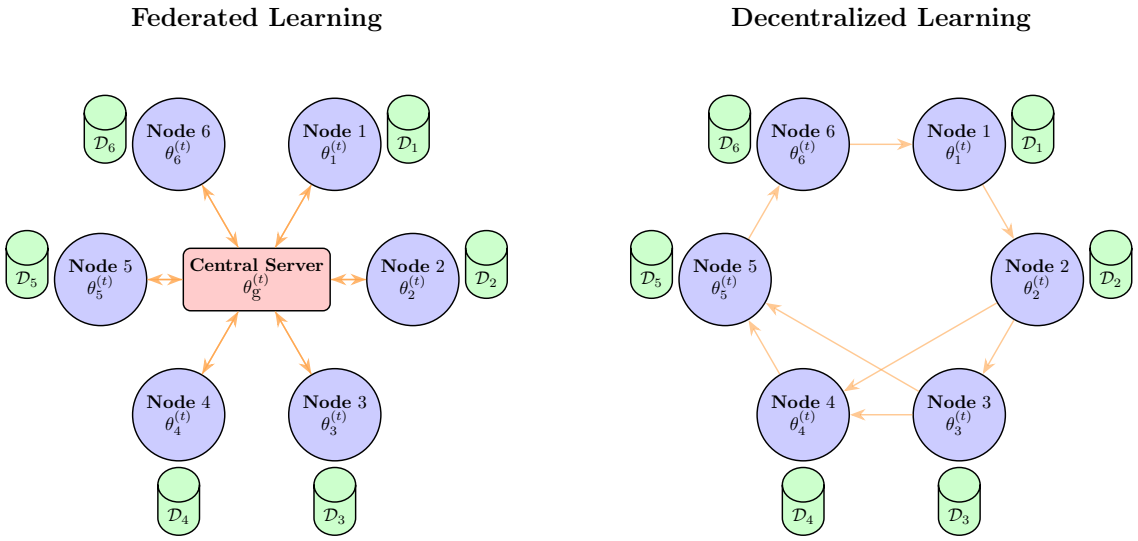

Figure 2.3: Federated and decentralized learning architectures. In FL, clients train locally on private datasets $\mathcal{D}_v$ and communicate with a central server, which maintains and updates the global model $\theta_{\mathrm{g}}^{(t)}$. In decentralized learning, no central server is used. Instead, each client maintains a local model $\theta_v^{(t)}$ and exchanges model information directly with neighboring clients according to a communication graph. Thus, both paradigms keep data local, but they differ in how model coordination and aggregation are performed.

2.5 Differential Privacy

Suppose we have a dataset containing sensitive information about individuals, such as medical records. When we analyze this data, we want the results to reveal patterns about the population without revealing whether any specific person was included. DP provides exactly this kind of guarantee. The output of a randomized algorithm should look nearly the same whether or not any one person's data is present in the dataset [1].

Formally, let $\mathcal{D}$ be a dataset and let $\mathcal{M}$ be a randomized algorithm, called a *mechanism*, taking $\mathcal{D}$ as input. Two datasets $\mathcal{D}$ and $\mathcal{D}'$ are said to be *neighboring*, written $\mathcal{D} \sim \mathcal{D}'$, if they differ in the unit of data protected by the privacy definition. The unit can be a single record, or a larger unit such as all data belonging to one user or node. The choice of neighboring relation determines what kind of change the privacy guarantee protects against.

Definition 2.5.1 (Differential privacy)**.** *A mechanism $\mathcal{M}$ guarantees (ε, δ)-DP if for any two neighboring datasets $\mathcal{D}$ and $\mathcal{D}'$, and for any $\mathcal{S} \subseteq \mathrm{Range}(\mathcal{M})$,*

$$\Pr[\mathcal{M}(\mathcal{D}) \in \mathcal{S}] \leq \mathrm{e}^{\varepsilon} \cdot \Pr[\mathcal{M}(\mathcal{D}') \in \mathcal{S}] + \delta,$$

where $\mathrm{Range}(\mathcal{M})$ is the set of all possible outcomes of mechanism $\mathcal{M}$.

The parameter ε is called the *privacy budget*. Smaller values of ε offer stronger protection, but typically reduce the accuracy of the released data. The parameter δ represents the probability of a worst-case privacy failure. When $\delta = 0$, we obtain *pure* ε-DP. In practice, δ is set to a negligible value, typically $\delta \ll 1/|\mathcal{D}|$.

Classical DP assumes a centralized curator with access to the raw data before any noise is applied. Local differential privacy (LDP) and network differential privacy (NDP) relax this trust assumption, extending privacy guarantees to settings where no such curator exists.

2.5.1 Local Differential Privacy

In several real-world settings, for example, when gathering data from phones or online services, users may be unwilling to trust a single organization with their unprocessed data. LDP avoids this problem by requiring each user to add randomness to their own data before transmitting it. This was first formalized in [14].

Definition 2.5.2 (Local differential privacy)**.** *A mechanism $\mathcal{M}$ guarantees ε-local DP if for any input x and x', and for any $\mathcal{S} \subseteq \mathrm{Range}(\mathcal{M})$,*

$$\frac{\Pr[\mathcal{M}(x) \in \mathcal{S}]}{\Pr[\mathcal{M}(x') \in \mathcal{S}]} \leq \mathrm{e}^{\varepsilon},$$

where $\mathrm{Range}(\mathcal{M})$ is the set of all possible outcomes of mechanism $\mathcal{M}$.

Here, each user independently applies $\mathcal{M}$ to their own data. This provides strong privacy guarantees but often introduces large amounts of noise because the protection must be applied before any aggregation occurs. This motivates models that use additional structure, such as a communication graph, to achieve stronger privacy.

2.5.2 Network Differential Privacy

NDP [15] is a relaxation of LDP designed for decentralized protocols. Instead of assuming that users send data directly to a central server, NDP is tailored to a system in which users exchange messages over a communication network.

Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be a set of users connected by a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where an edge (u, v) means user u can send messages to user v. A decentralized protocol produces a transcript consisting of all messages:

$$\mathcal{M}(\mathcal{D}) = \big\{(u, m, v) : \text{user } u \text{ sent message } m \text{ to user } v\big\}.$$

Unlike for LDP, here we exploit that users do not observe the full transcript. Each user u only sees the messages they send or receive. We denote this local view by

$$\mathcal{O}_u(\mathcal{M}(\mathcal{D})) = \big\{(v, m, v') \in \mathcal{M}(\mathcal{D}) : v = u \text{ or } v' = u\big\}.$$

For node level privacy, neighboring datasets are indexed by the node whose data may change. Writing $\mathcal{D} = \{\mathcal{D}_v\}_{v \in \mathcal{V}}$, two global datasets are neighboring with respect to node u, written $\mathcal{D} \sim_u \mathcal{D}'$, if $\mathcal{D}_v = \mathcal{D}'_v$ for all $v \neq u$, while $\mathcal{D}_u$ and $\mathcal{D}'_u$ may differ. This means that the protected unit is the local dataset of one node.

Definition 2.5.3 (Network differential privacy)**.** *A mechanism $\mathcal{M}$ satisfies (ε, δ)-NDP if, for all distinct users $u, v \in \mathcal{V}$, all neighboring datasets $\mathcal{D} \sim_u \mathcal{D}'$, and every $\mathcal{S} \subseteq \mathrm{Range}(\mathcal{O}_v(\mathcal{M}))$,*

$$\Pr[\mathcal{O}_v(\mathcal{M}(\mathcal{D})) \in \mathcal{S}] \leq \mathrm{e}^{\varepsilon} \Pr[\mathcal{O}_v(\mathcal{M}(\mathcal{D}')) \in \mathcal{S}] + \delta.$$

NDP sits between central DP and LDP. Users do not reveal their data directly, but the communication structure itself helps amplify privacy. This is particularly relevant in decentralized or peer-to-peer learning systems.

2.5.3 Rényi Differential Privacy

Rényi differential privacy (RDP) [16] is a relaxation of the (ε, δ)-DP definition that measures privacy loss in terms of Rényi divergence. RDP is particularly useful because it simplifies privacy accounting, especially for many sequential operations, and often yields tighter bounds.

Definition 2.5.4 (Rényi differential privacy)**.** *An algorithm $\mathcal{M}$ satisfies (α, ε)-RDP for $\alpha > 1$ and $\varepsilon > 0$ if for all pairs of neighboring datasets $\mathcal{D} \sim \mathcal{D}'$,*

$$D_\alpha\big(\mathcal{M}(\mathcal{D}) \,\|\, \mathcal{M}(\mathcal{D}')\big) \leq \varepsilon,$$

where for two distributions P and Q, $D_\alpha(P \,\|\, Q)$ is the Rényi divergence of P from Q:

$$D_\alpha(P \,\|\, Q) = \frac{1}{\alpha - 1} \ln \int \left(\frac{p(z)}{q(z)}\right)^{\alpha} q(z)\, dz,$$

with p and q being the respective densities of P and Q.

Rényi divergence is used because it composes more tightly than standard DP and allows privacy loss to be tracked across multiple rounds of communication.

Proposition 2.5.1 (RDP composition). *Let T be the number of composed mechanisms, and let $t \in \{1, \dots, T\}$ index one mechanism. Let $\mathcal{M}_1, \dots, \mathcal{M}_T$ be mechanisms run on the same dataset. If each $\mathcal{M}_t$ satisfies (α, ε_t)-RDP, then their adaptive composition satisfies $(\alpha, \sum_{t=1}^{T} \varepsilon_t)$-RDP.*

This result follows by repeated use of [16, Prop. 1]. It is useful for training because each update adds one privacy cost, so the total RDP cost is the sum over all updates.

Proposition 2.5.2 (Conversion from RDP to DP). *If a mechanism $\mathcal{M}$ satisfies (α, ε)-RDP for $\alpha > 1$, then for any $\delta \in (0, 1)$, it satisfies $\left(\varepsilon + \frac{\ln(1/\delta)}{\alpha - 1}, \delta\right)$-DP.*

This proposition follows from [16, Prop. 3], and is useful because the privacy accountant can add privacy costs in RDP form across training updates, and then convert the final bound to (ε, δ)-DP.

2.5.4 Gaussian Mechanism

The definitions above specify what privacy means, but not how to achieve it. The main tool used in this thesis is the *Gaussian mechanism*. It releases a function of the data, called a *query*, after adding Gaussian noise to its output [1].

The amount of noise depends on the *sensitivity* of the query, that is, how much its output can change between neighboring datasets. This change is measured with respect to the neighboring relation $\mathcal{D} \sim \mathcal{D}'$. For the decentralized protocols studied in this thesis, we use the node level neighboring relation $\mathcal{D} \sim_u \mathcal{D}'$ introduced in Section 2.5.2.

For a fixed target node u, and for a query f that maps a dataset to a vector in $\mathbb{R}^d$, the ℓ_2 *sensitivity* is

$$\Delta_2 f = \max_{\mathcal{D} \sim_u \mathcal{D}'} \left\| f(\mathcal{D}) - f(\mathcal{D}') \right\|_2,$$

the largest change in the output between two neighboring global datasets with respect to node u.

Let I_d be the $d \times d$ identity matrix. Given a query $f(\mathcal{D})$, the Gaussian mechanism outputs a noisy version of that query:

$$\mathcal{M}(\mathcal{D}) = f(\mathcal{D}) + z, \qquad z \sim \mathcal{N}(0, \sigma^2 I_d).$$

For a single release, the Gaussian mechanism theorem [1, Thm. 3.22] guarantees (ε, δ)-DP for $\varepsilon \in (0, 1)$ when

$$\sigma \geq \frac{\Delta_2 f \sqrt{2 \ln(1.25/\delta)}}{\varepsilon}.$$

Larger sensitivity and smaller values of ε require more noise. For fixed sensitivity and ε, a smaller δ also increases the required noise.

During training, the mechanism is applied over many updates. The accumulated privacy loss is therefore tracked with a privacy accountant, such as the RDP accounting above, and converted to an (ε, δ) guarantee when privacy levels are reported.

In gradient-based training, the released value is a gradient update. For a node v, let $\mathcal{B}_v$ be the minibatch sampled from its local dataset $\mathcal{D}_v$, with minibatch gradient $g_v = \nabla F_v(\theta_v; \mathcal{B}_v)$. Raw gradients have no fixed bound, so g_v is clipped to the norm bound C,

$$g_v \leftarrow g_v \cdot \min\left(1, \frac{C}{\|g_v\|_2}\right),$$

giving $\|g_v\|_2 \leq C$. The clipping bound therefore bounds the norm of the update released by a node before noise is added. For two clipped updates from the same node the distance between them is at most $2C$, so the gradient query has ℓ_2 sensitivity $\Delta_2 f = 2C$. Substituting into the bound above gives

$$\sigma \geq \frac{2C\sqrt{2\ln(1.25/\delta)}}{\varepsilon} = \frac{C\sqrt{8\ln(1.25/\delta)}}{\varepsilon},$$

so the Gaussian noise $z_v \sim \mathcal{N}(0, \sigma^2 I_d)$ is added to give the noisy update $g_v + z_v$. We set $C = 1$ in all experiments. With $k = C$, this is the noise scale used by Skip-Ring in Section 3.2.

Chapter 3

Related Work

This chapter reviews the two decentralized training protocols and the five gradient inversion attacks used in the experiments. We first describe decentralized learning protocols that define how models are trained and how updates are communicated, focusing on gossip-based averaging and straggler-resilient sequential training on a ring. We then review gradient-based reconstruction attacks, which exploit intermediate model updates to recover private data, and use these to motivate the experiments in this thesis.

3.1 Muffliato: Peer-to-Peer Privacy Amplification

Muffliato was introduced as a privacy mechanism showing how decentralization can amplify DP guarantees by combining local noise addition with gossip-based communication [17].

3.1.1 Gossip Algorithms

Muffliato runs on the decentralized communication network of Section 2.4.2, whose n users $\mathcal{V} = \{v_1, \dots, v_n\}$ exchange messages over the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

Muffliato builds on gossip communication, where users repeatedly exchange information with their neighbors and average their values [18]. The goal of gossip averaging is for all users to estimate the global average of their local vectors using only peer-to-peer communication.

Starting from an initial value $x_v^{(0)}$ at each user $v \in \mathcal{V}$, gossip proceeds in rounds. In round t, each active edge $\{u, v\} \in \mathcal{E}$ performs an averaging step, updating the values of the two users to $x_u^{(t+1)}$ and $x_v^{(t+1)}$.

To describe all updates in one matrix expression, we introduce the *gossip matrix* W_{mix}. This is a *symmetric stochastic* matrix such that:

$$(W_{\text{mix}})_{uv} > 0 \Rightarrow \{u, v\} \in \mathcal{E} \text{ or } u = v.$$

Intuitively, the gossip matrix W_{mix} encodes how information is mixed between neighboring users in a single communication round. Each row of W_{mix} specifies how a node averages its own value with the values received from its neighbors. Repeated multiplication by W_{mix} therefore corresponds to repeated rounds of local averaging across the graph.

One synchronous gossip round can be written compactly as

$$x^{(t+1)} = W_{\text{mix}} x^{(t)},$$

where $x^{(t)}$ is the vector of all user values at round t. After T rounds,

$$x^{(T)} = W_{\text{mix}}^{T} x^{(0)}.$$

Here W_{mix}^{T} denotes T repeated applications of the same gossip matrix. This expression shows how information from one user moves through the graph. The entry $(W_{\text{mix}}^{T})_{uv}$ tells us how much of user u's initial value can influence user v after T rounds. For distant users, it is very small, because information must pass through many rounds of mixing before reaching them.

This gradual mixing is the main source of privacy amplification. Because each user first adds noise to their value, and gossip repeatedly spreads and averages everything, users who are far apart only see a version of the value that has been heavily mixed with the values and noise of many others.

3.1.2 The Muffliato Mechanism

Muffliato combines local noise addition with gossip averaging to achieve privacy amplification across the communication graph. Each user $v \in \mathcal{V}$ begins by adding Gaussian noise to its initial value, yielding a randomized value that obscures the exact contribution of v. The users then run several rounds of gossip. Because gossip repeatedly mixes all values across the graph, the influence of a single user becomes weaker as it propagates through the network. Users who are far apart observe values that the noise and the contributions of many other users have strongly diluted.

This creates a privacy effect that depends on the distance between users. [17] formalizes this effect using pairwise NDP and shows that the privacy loss from user u to user v decays quickly with the graph distance between them.

While the theoretical description of Muffliato allows accelerated gossip, our experiments use plain gossip averaging. The corresponding implementation is given in Algorithm 1 below.

In the algorithm, $\mathcal{G}^{(t)} = (\mathcal{V}, \mathcal{E}^{(t)})$ is the communication graph used in training round t. It determines which pairs of nodes can exchange model parameters during the following gossip steps. We write $\mathcal{G}^{(t)} \sim G(n, q)$ for a random Erdős-Rényi graph on the n nodes, where each unordered pair of distinct nodes is included independently with probability q.

The matrix $W_{\text{mix}}^{(t)}$ is then built from $\mathcal{G}^{(t)}$. Its entry $(W_{\text{mix}}^{(t)})_{vw}$ is positive only when $v = w$ or when nodes v and w are connected by an edge in $\mathcal{G}^{(t)}$. The rows of $W_{\text{mix}}^{(t)}$ sum to one, so the update

$$\theta_v = \sum_{w \in \mathcal{V}} (W_{\text{mix}}^{(t)})_{vw} \theta_w$$

is a weighted average of node v's own model parameters and the parameters received from its neighbors.

Algorithm 1 Muffliato decentralized training

Input: Number of nodes n, local datasets $\{\mathcal{D}_v\}$, initial parameters $\theta^{(0)} \in \mathbb{R}^d$, learning rate η, clipping bound C, Gaussian noise scale σ, training iterations T, gossip steps T_{gossip}, edge probability q, batch size B

Output: Final averaged parameters $\bar{\theta}$

Initialize $\theta_v \leftarrow \theta^{(0)}$ for all $v \in \{v_1, \dots, v_n\}$
for $t \leftarrow 1$ to T **do**
  for all $v \in \{v_1, \dots, v_n\}$ **in parallel do**
    Sample minibatch $\mathcal{B}_v \subseteq \mathcal{D}_v$ of size B
    $g_v \leftarrow \nabla F_v(\theta_v; \mathcal{B}_v)$
    $g_v \leftarrow g_v \cdot \min\left(1, \frac{C}{\|g_v\|_2}\right)$
    Sample $z_v \sim \mathcal{N}(0, C^2\sigma^2 I_d)$
    $\theta_v \leftarrow \theta_v - \eta(g_v + z_v)$
  end for
  Sample communication graph $\mathcal{G}^{(t)} \sim G(n, q)$
  Construct gossip matrix $W_{\text{mix}}^{(t)}$ from $\mathcal{G}^{(t)}$
  for $j \leftarrow 1$ to T_{gossip} **do**
    for all $v \in \{v_1, \dots, v_n\}$ **in parallel do**
      $\theta_v^{\text{new}} \leftarrow \sum_{w \in \{v_1, \dots, v_n\}} (W_{\text{mix}}^{(t)})_{vw} \theta_w$
    end for
    for all $v \in \{v_1, \dots, v_n\}$ **in parallel do**
      $\theta_v \leftarrow \theta_v^{\text{new}}$
    end for
  end for
end for
$\bar{\theta} \leftarrow \frac{1}{n} \sum_{v \in \{v_1, \dots, v_n\}} \theta_v$
return $\bar{\theta}$

3.2 Straggler-Resilient Differentially Private Decentralized Learning

Distributed learning faces several practical challenges. Two common ones are protecting user privacy and handling stragglers, which are nodes whose computation times are much longer than those of other nodes. These challenges are examined through decentralized optimization under DP constraints and random computation delays [19], with a focus on how skipping slow nodes affects convergence, privacy amplification, and total training time.

The main idea is to model computation time as stochastic delays and to allow the algorithm to skip slow nodes in a given iteration. This reduces the wall-clock time per iteration, but it also changes which nodes contribute updates and how often they participate. Stragglers affect both the algorithm's privacy guarantees and its convergence behavior.

[19] therefore studies these aspects together:

- how skipping stragglers affects convergence,
- how it changes the DP guarantees,
- and how it affects the total training latency.

3.2.1 System Model

Skip-Ring uses the decentralized network of Section 2.4.2, with its n nodes now connected in a ring rather than a general graph. Each node holds its own local dataset and performs local computations.

The proposed algorithm Skip-Ring is organized around a single shared parameter vector $\tau_t \in \mathbb{R}^d$, called the *token*, that traverses the ring of nodes. The token represents the current global model, and a node contributes to training only while it holds the token. At each step, the node currently holding the token performs a local update and then forwards it to the next node along the ring. We index successive token visits by $t = 0, \ldots, T$, where T is the total number of token visits.

Each iteration has a random computation delay at each node. A node is treated as a straggler if it does not finish its computation before a fixed threshold t_{skip}. The probability that a node is skipped is denoted by $p = \Pr[\Lambda > t_{\text{skip}}]$, where Λ is the random per-node computation latency. Here, we study skipping probabilities $p \in \{10^{-4}, 1/2, 7/10\}$.

3.2.2 The Algorithm

The full algorithm is called *Skip-Ring* and is summarized in Algorithm 2. For each token visit $t \in \{1, \ldots, T\}$, a computation latency Λ_t is sampled from a latency model $\mathcal{P}$. Let Φ_Λ denote the cumulative distribution function of the per-node computation latency Λ. Although [19] studies several latency models, we only use the exponential distribution with mean 1, so that

$$\Phi_\Lambda(s) = 1 - \mathrm{e}^{-s}.$$

A node performs an update only if $\Lambda_t \leq t_{\text{skip}}$. In that case, the learning rate $\eta_t = \zeta/\sqrt{c_t}$ is used, where c_t is the number of successful updates counted before the current update. The active node v_t evaluates the minibatch empirical objective $F_{v_t}(\tau_{t-1}; \mathcal{B}_{v_t})$, and the minibatch gradient $\nabla F_{v_t}(\tau_{t-1}; \mathcal{B}_{v_t})$ is computed before clipping and noise addition. The token is then updated as

$$\tau_t = \Pi_{\mathcal{W}} \left(\tau_{t-1} - \eta_t \left(\nabla F_{v_t}(\tau_{t-1}; \mathcal{B}_{v_t}) + z_t \right) \right), \quad z_t \sim \mathcal{N}(0, \sigma^2 I_d),$$

where $\Pi_{\mathcal{W}}(\cdot)$ denotes Euclidean projection onto the convex constraint set $\mathcal{W}$ (with diameter $d_{\mathcal{W}}$), and z_t is the Gaussian noise added to the clipped gradient. The Gaussian noise standard deviation is

$$\sigma = \frac{k\sqrt{8\log(1.25/\delta)}}{\varepsilon}.$$

Here, k is the Lipschitz constant assumed for each local loss function F_v. In [19], this means that the local gradient norm is bounded by k. Since neighboring datasets differ in one record, the privacy analysis needs a bound on how much the released gradient can change when that record is changed. In our implementation, gradients are clipped to the norm bound $C = 1$, so clipping provides this bound directly. We therefore set $k = C = 1$.

3.2.3 Differential Privacy

Skip-Ring adopts NDP, introduced in Section 2.5.2, to quantify and limit the information that can be learned about a node's local dataset from the messages exchanged during training. NDP is particularly suitable in this setting because the learning process is fully decentralized and operates over a ring topology without a central server.

Proposition 3.2.1 (Skip-Ring privacy leakage [19, Thm. 2])**.** *After T token visits, Skip-Ring achieves $(\varepsilon_{\text{skip}}(T), \delta + \delta')$-NDP for all $\delta' \in (0, 1]$, where*

$$\varepsilon_{\text{skip}}(T) = \varepsilon \frac{\sqrt{\tilde{T}\log(1/\delta)}}{\sqrt{\log(1.25/\delta)}} + \frac{\varepsilon^2 \tilde{T}}{4\log(1.25/\delta)}, \tag{3.1}$$

with

$$\tilde{T} \triangleq \left\lceil T\frac{1-p}{n} + \sqrt{3T\frac{1-p}{n}\log(1/\delta')} \right\rceil.$$

After evaluating (3.1), we report the resulting privacy loss simply as ε. The subscript in $\varepsilon_{\text{skip}}$ only indicates that the expression comes from the skipping analysis. It does not denote a separate privacy parameter.

Algorithm 2 Skip-Ring decentralized training

Input: Datasets $\{\mathcal{D}_v\}$, constraint set $\mathcal{W} \subseteq \mathbb{R}^d$, initial token $\tau_0 \in \mathcal{W}$, learning rate schedule parameter ζ, Gaussian noise scale σ, node path $\{v_t\}_{t=1}^T$, timeout threshold t_{skip}, total number of update steps T, computation latency model $\mathcal{P}$, communication latency χ, clipping bound C, batch size B

Output: Final token $\tau_T \in \mathcal{W}$ and total execution latency t_{exec}

$t_{\text{exec}} \leftarrow 0$, $c \leftarrow 1$
for $t \leftarrow 1$ to T **do**
  Sample latency $\Lambda_t \sim \mathcal{P}$
  if $\Lambda_t \leq t_{\text{skip}}$ **then**
    $\eta_t \leftarrow \zeta/\sqrt{c}$
    Sample minibatch $\mathcal{B}_{v_t} \subseteq \mathcal{D}_{v_t}$ of size B
    $g_{v_t} \leftarrow \nabla F_{v_t}(\tau_{t-1}; \mathcal{B}_{v_t})$
    $g_{v_t} \leftarrow g_{v_t} \cdot \min\left(1, \frac{C}{\|g_{v_t}\|_2}\right)$
    $z_t \sim \mathcal{N}(0, \sigma^2 I_d)$
    $\tau_t \leftarrow \Pi_{\mathcal{W}}(\tau_{t-1} - \eta_t(g_{v_t} + z_t))$
    $t_{\text{exec}} \leftarrow t_{\text{exec}} + \chi + \Lambda_t$
    $c \leftarrow c + 1$
  else
    $\tau_t \leftarrow \tau_{t-1}$
    $t_{\text{exec}} \leftarrow t_{\text{exec}} + \chi + t_{\text{skip}}$
  end if
end for
return $(\tau_T, t_{\text{exec}})$

The original paper also considers a randomized variant (*Skip-Rand-Ring*), but in this work, we use the fixed-order *Skip-Ring* protocol.

3.3 Deep Leakage from Gradients

Early work showed that sharing gradients during distributed training can unintentionally reveal private training examples [2]. The attack, called DLG, treats data reconstruction as an optimization problem. It tries to find an input whose gradient matches the gradient that was originally sent during training.

Consider a model f_θ with parameters θ, and let $g = \nabla_\theta \ell\big(f_\theta(x), y\big)$ denote the gradient computed from a single training example (x, y), where $x \in \mathbb{R}^d$ is the input feature vector and $y \in \mathcal{Y}$ its label. An attacker aims to find a synthetic input–label pair $(\hat{x}, \hat{y})$ that produces nearly the same gradient. The basic version of the attack works as follows:

1. Start with randomly generated dummy inputs and labels:

$$\hat{x} \sim \mathcal{N}(0, I_d), \qquad \hat{y} \sim \mathcal{N}(0, I_{|\mathcal{Y}|})$$

2. Compute the gradients of the loss with respect to the model parameters for the dummy data:

$$\hat{g} = \nabla_\theta \ell\big(f_\theta(\hat{x}), \hat{y}\big).$$

3. Adjust the dummy data so that its gradient becomes closer to the real one by solving:

$$\hat{x}, \hat{y} = \arg\min_{\hat{x}, \hat{y}} \|\hat{g} - g\|_2^2.$$

By minimizing this difference, the dummy input gradually evolves into one that produces a gradient almost identical to that of the original training example. When the optimization succeeds, $(\hat{x}, \hat{y})$ closely resembles the true data point that generated g. Although the method requires computing higher-order gradients, standard optimizers can handle this efficiently.

DLG showed that even a single shared gradient can leak both the input and its label with surprising accuracy. This result led to numerous follow-up studies that improved reconstruction quality and the stability of the optimization procedure. These more advanced attacks are discussed in the next section.

Algorithm 3 presents the DLG algorithm used in our experiments, while Figure 3.1 provides an overview.

3.3.1 iDLG

Improved Deep Leakage from Gradients (iDLG) extends DLG by showing that, for classification models trained with cross-entropy loss, the ground-truth label can be inferred directly from the shared gradients [3]. Specifically, the gradient sign pattern in the final layer reveals the true class. This is often called the *sign trick*.

By exploiting this property, iDLG first recovers the correct label from the gradients and then optimizes only the input data while keeping the label fixed. This greatly stabilizes the reconstruction process and improves convergence compared to DLG, which jointly optimizes inputs and labels. The sign trick is also used in later attacks, such as TabLeak.

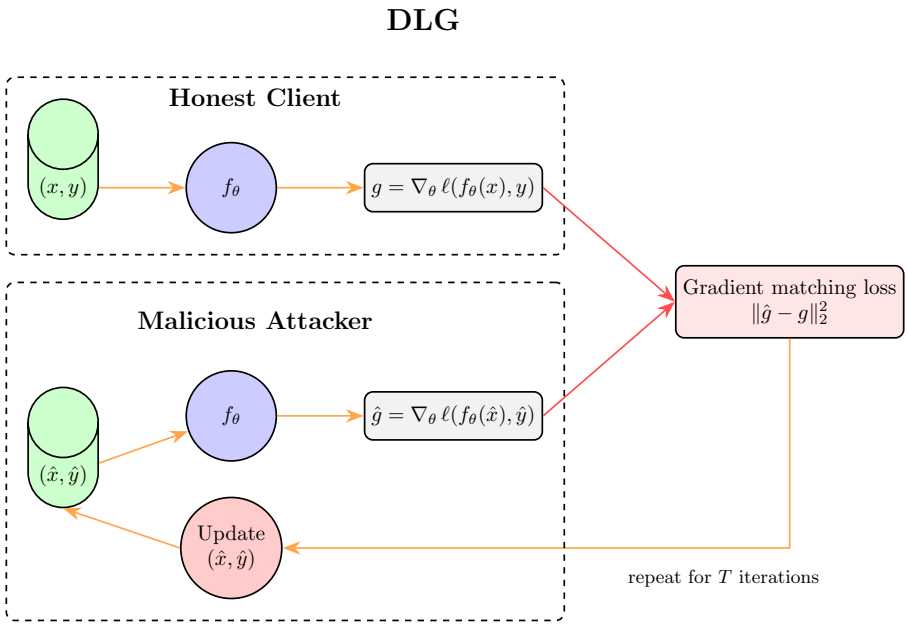

Figure 3.1: Overview of the DLG attack. An honest client computes a gradient $g = \nabla_\theta \ell(f_\theta(x), y)$ from private training data (x, y) and shares it during training. An attacker observes this gradient and initializes dummy variables $(\hat{x}, \hat{y})$. Using the same model f_θ, the attacker computes a dummy gradient $\hat{g}$ and iteratively updates the dummy variables to minimize the gradient distance $\|\hat{g} - g\|_2^2$. When the optimization converges, the reconstructed variables $(\hat{x}, \hat{y})$ approximate the original input–label pair.

Algorithm 3 Deep Leakage from Gradients

Input: Neural network f_θ, loss function $\ell(\cdot, \cdot)$, observed gradient $g = \nabla_\theta \ell(f_\theta(x), y)$, attack step size η, number of attack iterations T

Output: Reconstructed input–label pair $(\hat{x}, \hat{y})$

$\hat{x} \sim \mathcal{N}(0, I_d), \quad \hat{y} \sim \mathcal{N}(0, I_{|\mathcal{Y}|})$
for $i \leftarrow 1$ to T **do**
  $\hat{g} \leftarrow \nabla_\theta \ell\big(f_\theta(\hat{x}), \hat{y}\big)$ ▷ Compute dummy gradients
  $d_i \leftarrow \|\hat{g} - g\|_2^2$
  $\hat{x} \leftarrow \hat{x} - \eta \nabla_{\hat{x}} d_i, \quad \hat{y} \leftarrow \hat{y} - \eta \nabla_{\hat{y}} d_i$
end for
return $(\hat{x}, \hat{y})$

In our graph setting, f_θ is a GCN that takes (X, A) as input. The attacker reconstructs feature matrix $\hat{X}$ and dummy labels $\hat{y}$ corresponding to a target minibatch $\mathcal{B} \subseteq \mathcal{V}$ that produced the observed gradient. The adjacency matrix A is assumed to be known.

3.4 TabLeak

Gradient inversion attacks were originally successful in domains such as images and text, but they performed poorly on tabular data. This changed with the introduction of TabLeak [4]. Compared with settings such as DLG, tabular data introduces two challenges.

First, tabular datasets typically contain both discrete and continuous features, so the attacker must solve a mixed discrete–continuous optimization problem. Gradient-based methods work naturally in continuous spaces, but they do not directly handle discrete categories well. Second, unlike images, tabular reconstructions cannot be judged reliably by visual inspection. Even an incorrectly reconstructed row may still look plausible, so the attacker needs a reliable way to assess reconstruction quality.

3.4.1 Tabular Data

Tabular data consists of rows and columns. Many real-world datasets, especially in healthcare, finance, and the social sciences, contain both numerical (continuous) and categorical (discrete) features.

Following [4], let $\mathcal{X}$ denote the space of possible tabular rows, and let $x \in \mathcal{X}$ be one row. Assume that each row in $\mathcal{X}$ has K_{D} discrete features and K_{C} continuous features. The discrete features are one hot encoded, while the continuous features remain unchanged. The full encoded representation of a row is

$$c(x) = [c_1^{\mathrm{D}}(x), \ldots, c_{K_{\mathrm{D}}}^{\mathrm{D}}(x), x_1^{\mathrm{C}}, \ldots, x_{K_{\mathrm{C}}}^{\mathrm{C}}],$$

where $c_i^{\mathrm{D}}(x)$ is the one hot encoding of discrete feature i. The dimension of this encoded vector is denoted by d.

3.4.2 The Attack

Figure 3.2 provides an overview of the TabLeak attack. Implementation details are given in Algorithm 4.

Softmax Relaxation

TabLeak first converts the mixed discrete–continuous reconstruction problem into a fully continuous one. For each discrete feature, instead of optimizing directly over one-hot vectors, the attacker introduces a continuous relaxation and applies softmax to obtain a probability distribution over the possible categories. For a vector $z \in \mathbb{R}^m$, softmax is defined componentwise by

$$\big(\mathrm{softmax}(z)\big)_j = \frac{\mathrm{e}^{z_j}}{\sum_{k=1}^{m} \mathrm{e}^{z_k}}, \qquad j = 1, \ldots, m.$$

Here, the subscript j denotes the j-th component of the vector.

Let x be the true sample and let $c(x)$ denote its one hot encoded representation. TabLeak introduces a continuous variable $z \in \mathbb{R}^d$ and optimizes it so that its relaxed representation approximates $c(x)$. Following the notation in [4], let z_i^{D} denote the block of z corresponding to discrete feature i, and let z_i^{C} denote the entry corresponding to continuous feature i. We write

$$\tilde{c}(z) \coloneqq \big[\mathrm{softmax}(z_1^{\mathrm{D}}), \ldots, \mathrm{softmax}(z_{K_{\mathrm{D}}}^{\mathrm{D}}), z_1^{\mathrm{C}}, \ldots, z_{K_{\mathrm{C}}}^{\mathrm{C}}\big]$$

for the resulting blockwise relaxation, so that $c(x) \approx \tilde{c}(z)$, where the softmax is applied separately to each discrete feature block z_i^{D} and the continuous components are kept directly. This makes the reconstruction problem suitable for gradient based optimization.

Objective Function

TabLeak then optimizes the relaxed reconstruction by matching gradients. Following [4], we write $g(u, y)$ for the gradient $\nabla_\theta \ell(f_\theta(u), y)$ induced by input u and label y. With this notation, the cosine-similarity reconstruction loss from [20] is

$$\mathcal{E}_{\mathrm{CS}}\big(g(c(x), y),\, g(\tilde{c}(z), \hat{y})\big) := 1 - \frac{\langle g(c(x), y),\, g(\tilde{c}(z), \hat{y})\rangle}{\|g(c(x), y)\|_2 \, \|g(\tilde{c}(z), \hat{y})\|_2},$$

where $\hat{y}$ denotes the labels reconstructed before the feature reconstruction step.

This is the objective minimized during the attack. Lower values mean that the reconstructed features and label induce gradients that are more closely aligned with the observed gradient.

Pooled Ensembling

Because the reconstruction problem is highly non-convex and sensitive to initialization, TabLeak does not rely on a single optimization run. Instead, it repeats the attack N_{ens} times from different random initializations, yielding candidate reconstructions $\{\tilde{c}(z_j)\}_{j=1}^{N_{\mathrm{ens}}}$. Here, the index j refers to the j-th independent optimization run, not to a component of the vector.

When reconstructing batches, these candidates may recover the same rows in different orders. TabLeak therefore first selects the candidate with the lowest reconstruction loss as a reference, denoted z^{best}, and then aligns the remaining candidates to it. After this alignment step, the reconstructions are combined feature-wise using median pooling to produce a single pooled estimate. This reduces variance and yields a more stable reconstruction than using only one run.

Entropy-Based Assessment

Since tabular reconstructions cannot be evaluated visually, TabLeak also introduces a way to estimate how trustworthy a reconstruction is. The key idea is to measure agreement across the ensemble. If the candidate reconstructions consistently agree on a feature, that feature is more likely to have been recovered correctly. If they disagree substantially, the reconstruction is less reliable.

This agreement is quantified using entropy. For discrete features, entropy is computed from the empirical distribution of reconstructed categories across the ensemble. For continuous features, entropy is estimated by assuming Gaussian reconstruction errors and computing entropy from the sample variance. Lower entropy indicates greater agreement and therefore higher confidence in the reconstructed feature.

Reconstruction Accuracy

To evaluate reconstruction quality quantitatively, TabLeak defines the reconstruction accuracy of a sample $\hat{x}$ relative to the true sample x as the average over all features. Discrete features are counted as correct if they match exactly, while continuous features are counted as correct if the reconstructed value falls within a tolerance interval around the ground truth value. Let $\mathbb{I}\{\cdot\}$ be 1 when the statement in braces is true and 0 otherwise. Let ρ_i be the tolerance for continuous feature i.

$$\text{accuracy}(x, \hat{x}) = \frac{1}{K_\text{D} + K_\text{C}} \left(\sum_{i=1}^{K_\text{D}} \mathbb{I}\{x_i^\text{D} = \hat{x}_i^\text{D}\} + \sum_{i=1}^{K_\text{C}} \mathbb{I}\{\hat{x}_i^\text{C} \in [x_i^\text{C} - \rho_i,\ x_i^\text{C} + \rho_i]\} \right). \tag{3.2}$$

In TabLeak, this tolerance is set to $\rho_i = 0.319\,\sigma_i^\text{C}$, where σ_i^C is the standard deviation of continuous feature i.

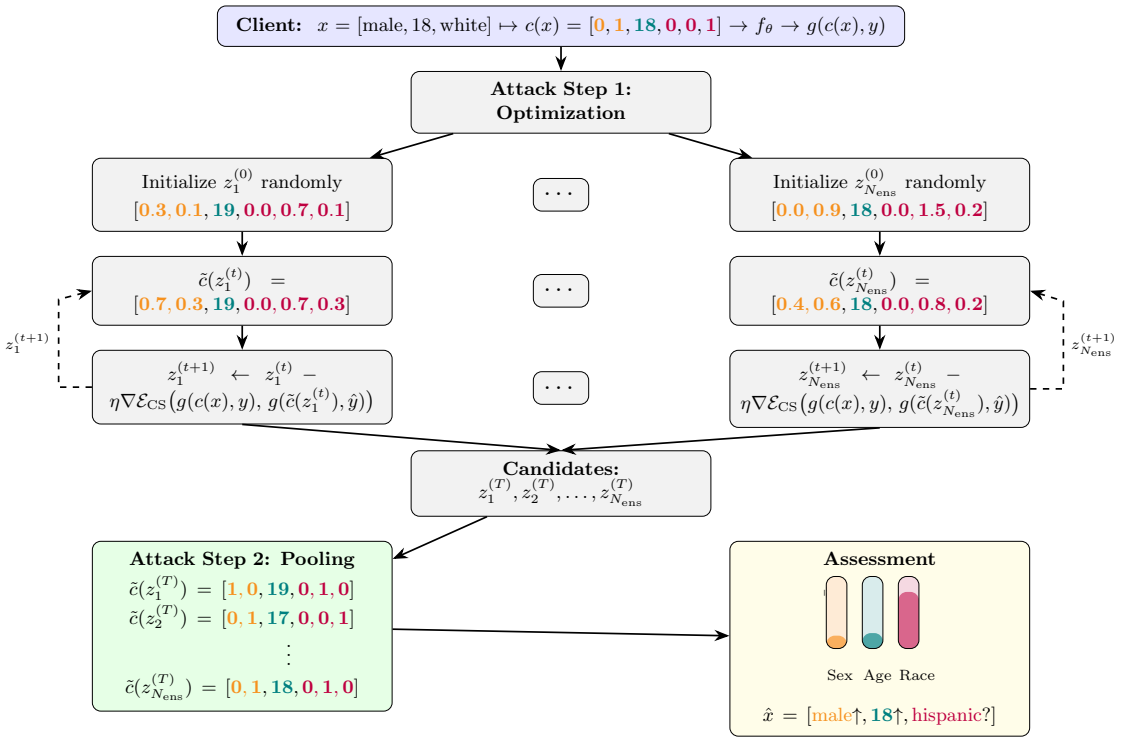

Figure 3.2: Overview of the TabLeak attack. **Attack Step 1:** The attacker initializes several random guesses $z_1^{(0)}, \ldots, z_{N_\text{ens}}^{(0)}$ and optimizes each one. Discrete features are relaxed with the softmax function $\text{softmax}(\cdot)$, allowing gradient-based updates to produce N_ens candidate reconstructions. **Attack Step 2:** The candidates are aligned with the best solution and then combined using feature-wise pooling to reduce variance. **Assessment:** The entropy of each feature across the ensemble is measured to estimate how reliable the reconstruction is.

Algorithm 4 TabLeak

Input: Neural network f_θ, observed gradients $g(c(x), y)$, reconstructed labels $\hat{y}$, ensemble size N_{ens}, iterations T, step size η

Output: Reconstructed input $\hat{x}$ and feature entropies $\bar{H}^{\text{D}}, H^{\text{C}}$

Initialize $z_j^{(0)} \sim \mathcal{U}([0,1]^d)$ for $j = 1, \ldots, N_{\text{ens}}$
for $j \leftarrow 1$ to N_{ens} **do**
    for $t \leftarrow 0$ to $T-1$ **do**
        Apply softmax to each discrete block of $z_j^{(t)}$ to form $\tilde{c}(z_j^{(t)})$
        $z_j^{(t+1)} \leftarrow z_j^{(t)} - \eta \nabla_z \mathcal{E}_{\text{CS}}\Big(g(c(x), y),\ g(\tilde{c}(z_j^{(t)}), \hat{y})\Big)$
    end for
end for
$\hat{z}^{\text{best}} \leftarrow \arg\min_{z_j^{(T)}} \mathcal{E}_{\text{CS}}\Big(g(c(x), y),\ g(\tilde{c}(z_j^{(T)}), \hat{y})\Big)$
$\tilde{c}(\hat{z}) \leftarrow \texttt{MatchAndPool}\Big(\{\tilde{c}(z_j^{(T)})\}_{j=1}^{N_{\text{ens}}},\ \hat{z}^{\text{best}}\Big)$
$\bar{H}^{\text{D}}, H^{\text{C}} \leftarrow \texttt{CalculateEntropy}\Big(\{\tilde{c}(z_j^{(T)})\}_{j=1}^{N_{\text{ens}}}\Big)$
$\hat{x} \leftarrow \texttt{Project}\big(\tilde{c}(\hat{z})\big)$
return $\hat{x}, \bar{H}^{\text{D}}, H^{\text{C}}$

3.5 GRAIN: Exact Graph Reconstruction from Gradients

GRAIN was introduced as the first gradient inversion attack able to reconstruct graph-structured data [5]. The attack targets FL with GNNs and shows that sharing gradients can reveal both the graph structure and the node features of a client.

Most prior gradient inversion attacks focus on images, text, or tabular data. In contrast, graph data consists of both node features and an adjacency matrix, and GNNs compute representations by aggregating node features along edges. As a result, standard optimization-based inversion attacks become ineffective. GRAIN addresses this challenge by exploiting structural properties of GNN gradients, enabling exact reconstruction without solving a continuous optimization problem.

3.5.1 Notation

We briefly introduce the notation used in the description of GRAIN. For a node $v \in \mathcal{V}$, its l-hop neighborhood is the subgraph induced by all nodes whose shortest-path distance from v is at most l. In a GNN, this neighborhood corresponds to the local structure that can influence the representation of v after l layers.

Each node has an m-dimensional feature vector. For each feature dimension i, let $\mathcal{F}_i$ denote the set of possible feature values. The set of all possible node feature vectors is then

$$\mathcal{F} = \mathcal{F}_1 \times \cdots \times \mathcal{F}_m.$$

GRAIN reconstructs graphs through local subgraphs called *building blocks*. A building block consists of a node together with its l-hop neighborhood. The candidate sets of building blocks are denoted by $\mathcal{T}_0, \mathcal{T}_1, \ldots$, where $\mathcal{T}_0$ is the initial proposal set of candidate node feature vectors, and $\mathcal{T}_l$ denotes the candidate set of l-hop building blocks. Filtered sets are denoted by $\mathcal{T}_0^*$ and $\mathcal{T}_l^*$, while $\mathcal{T}_\mathrm{B}^*$ denotes the final set of consistent building blocks used in the reconstruction phase. We write $\mathrm{dang}(\cdot)$ for the set of dangling nodes of a graph or building block, that is, nodes whose current number of attached neighbors is below their known degree.

3.5.2 Assumptions

GRAIN relies on a set of strong assumptions about the learning setup and the algebraic structure of the gradients. These assumptions are required for the attack to succeed, but they also limit its applicability to realistic systems.

Batch size $= 1$. The attack assumes that a single client update on a single graph instance produces the observed gradients. Equivalently, the effective batch size is one. This assumption is necessary because GRAIN performs exact consistency checks between candidate subgraphs and the observed gradients. If gradients from multiple samples were aggregated, the resulting gradients would mix information from different graphs, and the filtering procedure would no longer be exact.

Discrete features with known domains. GRAIN assumes that all node features are discrete and that the attacker knows the complete set of possible values

for each feature. This allows the attack to enumerate candidate node features and systematically eliminate inconsistent ones.

Low-rank structure induced by $n < d$. A central assumption of GRAIN is that the number of nodes n in the client graph is smaller than the embedding dimension d of the GNN layers. Together with a full-rank gradient at the layer output, this gives the span relation used by the attack. For linear layers, [5, Thm. 3.1] shows that the row space of the layer input equals the column space of the weight gradient under these conditions. GRAIN extends this span check to GNN layers in [5, Thm. 5.1].

GRAIN relies on this relation to test whether a candidate node feature or subgraph is consistent with the observed gradients. Intuitively, the low-rank structure restricts the space of possible inputs that could have generated the observed gradients, which allows GRAIN to eliminate inconsistent candidates.

Adjacency-dependent rank conditions. GRAIN relies on structural assumptions about the matrix that performs message passing within each GNN layer, which we denote by $A^{(l)}$. For GCNs this is the normalized adjacency $\bar{A}$, while for GATs it is the learned attention matrix. The corollary to Theorem 5.1 in [5, Cor. 5.2] assumes $n < d$ and a full-rank pre-activation gradient matrix. Under these conditions, all layer-l node features pass the span check when the normalized adjacency matrix has full rank. If this matrix is rank deficient, Theorem 5.1 can still certify individual node features.

In practice, [5] reports that the relevant gradient is almost always full rank for GAT architectures. For GCNs, the normalized adjacency matrix may be rank deficient depending on the graph topology. This can limit the exact reconstruction of all nodes.

Full knowledge of the model and data encoding. Finally, the attacker is assumed to know the exact model architecture, parameters, and data encoding, including feature semantics and normalization. This assumption is common in gradient inversion attacks, but it further limits the realism of the threat model.

3.5.3 The Attack

GRAIN proceeds in two main phases: subgraph filtering and graph building. In the first phase, the attacker reconstructs valid local subgraphs, called *building blocks*, from the observed gradients. In the second phase, these building blocks are combined into a full graph using a depth-first search (DFS). Figure 3.3 illustrates this pipeline.

Building blocks and filtering. GRAIN begins from the discrete feature domains known to the attacker. If the m node features take values in domains $\mathcal{F}_1, \ldots, \mathcal{F}_m$, then the attacker first forms the node proposal set

$$\mathcal{T}_0 = \mathcal{F}_1 \times \cdots \times \mathcal{F}_m,$$

which contains all candidate node feature vectors. This is the search space for the 0-hop neighborhoods, that is, individual nodes.

The first filtering stage uses the gradient of the first GNN layer, $\nabla_{W^{(0)}}\ell$, to determine which candidate vectors in $\mathcal{T}_0$ are compatible with the observed gradients. This follows the GNN span check in [5, Thm. 5.1]. For a candidate vector z, the distance to the column space of the layer-l gradient is measured as

$$d\left(z, \nabla_{W^{(l)}}\ell\right) := \left\| z - \mathrm{proj}(z, \mathrm{colspan}(\nabla_{W^{(l)}}\ell)) \right\|_2, \tag{3.3}$$

and the candidate is retained if this distance is smaller than a threshold τ. In the initial filtering step, this criterion is applied with $l = 0$, producing the reduced set

$$\mathcal{T}_0^* \subseteq \mathcal{T}_0,$$

which contains the recovered single-node feature vectors. GRAIN further performs partial span checks on subsets of feature dimensions and progressively extends them, controlling complexity for graph data where directly checking full vectors would be expensive.

After recovering $\mathcal{T}_0^*$, GRAIN incrementally constructs larger building blocks. At layer l, the attacker combines compatible $(l-1)$-hop building blocks from the previous stage to form a proposal set of candidate l-hop neighborhoods, denoted by $\mathcal{T}_l$. These candidates are then filtered using the gradient of the corresponding GNN layer, $\nabla_{W^{(l)}}\ell$, producing the reduced set of consistent l-hop building blocks

$$\mathcal{T}_l^* \subseteq \mathcal{T}_l.$$

This procedure is repeated layer by layer up to the depth L of the attacked GNN, which is also the maximum neighborhood radius. Intuitively, the reason this works is that, in a GNN, the representation of a node at layer l depends only on its original l-hop neighborhood, so a correctly reconstructed building block can be propagated through the first l layers to recover the corresponding layer-l embedding and enable another span check.

GRAIN then performs an additional structure-based consistency check on the final set $\mathcal{T}_L^*$. Building blocks that cannot be glued consistently to any other block at their dangling nodes are removed, unless they already correspond to the full input graph. The remaining set is denoted by $\mathcal{T}_{\mathrm{B}}^*$ and forms the final set of proposals for graph reconstruction.

Graph reconstruction. In the second phase, GRAIN reconstructs the full client graph from the filtered building blocks in $\mathcal{T}_{\mathrm{B}}^{*}$ using a DFS over partially reconstructed graphs. Each node in the DFS tree represents a partially reconstructed graph $\mathcal{G}_{\mathrm{curr}}$. At each step, the algorithm selects a dangling node

$$v \in \mathrm{dang}(\mathcal{G}_{\mathrm{curr}})$$

that still lacks neighbors, and tries to glue a building block from $\mathcal{T}_{\mathrm{B}}^{*}$ to that node. Every valid gluing operation creates a new branch in the search tree. In this way, DFS explores all graphs that can be assembled consistently from the filtered building blocks while respecting node features and degree constraints.

For each candidate reconstruction $\mathcal{G}'$, GRAIN computes the gradients induced by that graph and compares them to the observed client gradients. If the induced gradients match exactly, the candidate graph is accepted as the reconstruction. Otherwise, the search backtracks and explores a different branch. GRAIN formulates this DFS procedure as a search over partial graphs, returning immediately upon finding a branch with zero gradient discrepancy.

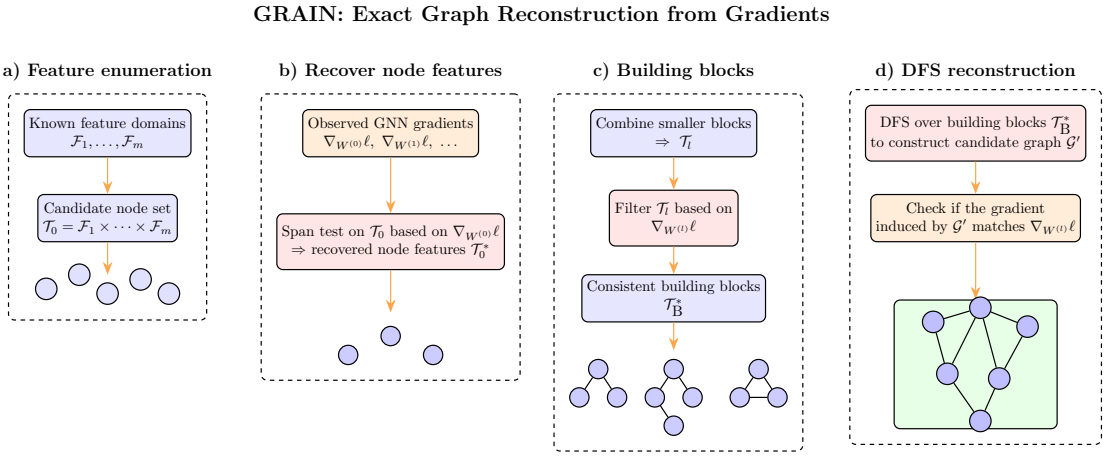

Figure 3.3: Overview of the GRAIN attack for reconstructing graph-structured training data from shared gradients. In stage a), the attacker forms the node proposal set $\mathcal{T}_0 = \mathcal{F}_1 \times \cdots \times \mathcal{F}_m$ from the known feature domains $\mathcal{F}_1, \ldots, \mathcal{F}_m$. In stage b), the first-layer weight gradient $\nabla_{W^{(0)}}\ell$ is used to recover the valid node feature vectors $\mathcal{T}_0^*$. In stage c), larger candidate l-hop building blocks $\mathcal{T}_l$ are formed by combining smaller blocks and iteratively filtered using the layer-l weight gradient $\nabla_{W^{(l)}}\ell$, yielding the final consistent building blocks $\mathcal{T}_{\mathrm{B}}^*$. In stage d), a DFS assembles candidate graphs $\mathcal{G}'$ from $\mathcal{T}_{\mathrm{B}}^*$, and reconstruction succeeds when the gradient induced by $\mathcal{G}'$ matches the observed gradient $\nabla_{W^{(l)}}\ell$ at every layer.

3.6 Graph Leakage from Gradients

GLG is a gradient inversion framework for GNNs in federated graph learning [6]. The attack considers reconstruction of node features and graph structure from shared gradients, and formulates the inversion problem as gradient matching over dummy graph inputs.

In contrast to exact graph-reconstruction methods such as GRAIN, GLG does not rely on exact structural recovery from closed-form properties of the gradients. Instead, it reconstructs graph inputs by optimizing dummy variables such that the resulting gradients match the observed gradients.

3.6.1 Attacker Variants

GLG distinguishes between several attacker models depending on the task and the attacker's prior knowledge:

- **Node Attacker 1:** observes gradients for a single target node in a node-classification task and reconstructs node features in the local neighborhood of that node.
- **Node Attacker 2:** observes gradients for all nodes in an egonet or subgraph and reconstructs node features, graph structure, or both.
- **Graph Attacker:** observes gradients in a graph-classification setting and reconstructs node features, graph structure, or both.

In this thesis, we consider **Node Attacker 1**, since it matches the node-level reconstruction setting used in our experiments.

3.6.2 Overview of the Attack

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a graph with adjacency matrix A and node feature matrix X, whose rows contain node feature vectors of dimension d. For a target node $v \in \mathcal{V}$, let x_v denote the corresponding row of X, and let $\mathcal{N}(v)$ denote its neighborhood. The attacker observes the gradient

$$g = \nabla_\theta \ell\big((f_\theta(X, A))_v, y_v\big),$$

where f_θ is a GCN with parameters θ, and $(f_\theta(X, A))_v$ denotes the model output for target node v.

GLG reconstructs private inputs by introducing dummy variables and matching the resulting dummy gradients to the observed gradients. For this node-classification setting, the attacker initializes a dummy local feature matrix $\hat{X}$ and fixes a dummy adjacency matrix $\hat{A}$, and computes

$$\hat{g} = \nabla_\theta \ell\big((f_\theta(\hat{X}, \hat{A}))_v, y_v\big).$$

The reconstruction is then obtained by minimizing a gradient-matching objective. The full algorithm is presented in Algorithm 5, and an overview is illustrated in Figure 3.4.

Node Attacker 1. In the Node Attacker 1 setting, the attack is restricted to a single target node. For GCN, the first layer depends on the degree-normalized aggregated representation of the target node,

$$x_v^{\text{agg}} = \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{x_u}{\sqrt{\deg(v)\ \deg(u)}},$$

where $\deg(v)$ counts the self-loop, so $\deg(v) = |\mathcal{N}(v)| + 1$. Hence, the observed gradients depend not only on x_v, but on the local neighborhood around v.

Accordingly, Node Attacker 1 optimizes a dummy local feature matrix $\hat{X}$ defined on a dummy tree adjacency $\hat{A}$. The rows of $\hat{X}$ contain the dummy target feature vector $\hat{x}_v$ and the dummy feature vectors for neighboring nodes. These neighbor variables are part of the optimization because they determine the aggregated input seen by the GCN, even though the primary reconstruction target is the feature vector of the target node.

Since the attacker does not know the true local graph structure, GLG initializes a dummy graph as a 2-layer tree rooted at the target node. The degree parameter d_{tree} specifies the branching factor used in this dummy tree. It is therefore a hyperparameter of the attack, not the true degree of the target node. The paper uses $d_{\text{tree}} = 10$ in the Node Attacker 1 experiments.

Gradient-matching objective. GLG uses a cosine-similarity loss from [20] between the observed and dummy gradients:

$$\mathcal{E}_{\text{CS}}(g, \hat{g}) = 1 - \frac{\langle g, \hat{g} \rangle}{\|g\|_2 \|\hat{g}\|_2}. \tag{3.4}$$

For Node Attacker 1, GLG optimizes only dummy node features and does not use the graph regularizers introduced for the stronger attacker settings.

Regularization in stronger settings. For settings in which node features and graph structure are reconstructed jointly, GLG augments the cosine loss with a feature-smoothness regularizer and an adjacency norm penalty:

$$R_{\text{s}}(X, A) = \text{tr}(X L_A X^\top), \qquad R_{\text{A}}(A) = \|A\|_{\text{F}}^2,$$

where L_A is the graph Laplacian of A and $\|\cdot\|_{\text{F}}$ denotes the Frobenius norm.

This gives the objective

$$\hat{\mathcal{E}}_{\text{CS}} = \mathcal{E}_{\text{CS}}(g, \hat{g}) + \alpha R_{\text{s}} + \beta R_{\text{A}},$$

where α and β are hyperparameters. These terms are used for Node Attacker 2 and Graph Attacker, but not for Node Attacker 1.

Optimization and evaluation. The dummy local feature matrix $\hat{X}$ is optimized by minimizing the gradient-matching objective (3.4). The reconstructed target feature $\hat{x}_v$ is the row of $\hat{X}$ corresponding to the root node v.

For node-feature reconstruction, GLG reports the root normalized mean squared error (RNMSE),

$$\text{RNMSE}(x_v, \hat{x}_v) = \frac{\|x_v - \hat{x}_v\|_2}{\|x_v\|_2}. \tag{3.5}$$

For graph reconstruction, [6] reports metrics such as accuracy, area under the curve, and average precision.

3.6.3 Comparison with GRAIN

GLG relies on fewer and weaker assumptions than GRAIN, making it more realistic in practical threat models.

GLG does not assume access to node degree information. GRAIN relies on degree-related leakage, but node degree is not generally available to an attacker unless it is explicitly included as an input feature.

GLG also does not require node features to be discrete or enumerable. GRAIN assumes discrete feature domains and uses enumeration-based filtering, which does not scale to continuous or high-dimensional features.

Finally, GLG does not rely on restrictive rank conditions such as requiring the number of nodes to be smaller than the embedding dimension. Instead, it formulates reconstruction as a continuous optimization problem augmented with weak graph priors.

Overall, GLG trades exact reconstruction guarantees for broader applicability. By avoiding strong structural assumptions and instead exploiting common statistical properties of graphs, it provides a more realistic assessment of the risks of gradient leakage in GNNs.

Algorithm 5 GLG – Node Attacker 1

Input: 2-layer GCN f_θ, observed gradient $g = \nabla_\theta \ell\big((f_\theta(X, A))_v, y_v\big)$ for target node v, label y_v, dummy degree d_{tree}, iterations T, step size η

Output: reconstructed local feature matrix $\hat{X}$

Build dummy adjacency $\hat{A}$ as a 2-layer tree rooted at v with branching factor d_{tree}
Initialize $\hat{X}^{(1)}$ with rows drawn i.i.d. from $\mathcal{N}(0, I_d)$
for $t \leftarrow 1$ to T **do**
    $\hat{g}^{(t)} \leftarrow \nabla_\theta \ell\big((f_\theta(\hat{X}^{(t)}, \hat{A}))_v, y_v\big)$
    $\mathcal{E}_{\text{CS}}^{(t)} \leftarrow 1 - \dfrac{\langle g, \hat{g}^{(t)} \rangle}{\|g\|_2 \, \|\hat{g}^{(t)}\|_2}$
    $\hat{X}^{(t+1)} \leftarrow \hat{X}^{(t)} - \eta \nabla_{\hat{X}} \mathcal{E}_{\text{CS}}^{(t)}$
end for
return $\hat{X}^{(T+1)}$

In Algorithm 5, the dummy adjacency matrix $\hat{A}$ is fixed after initialization. The optimization updates only the dummy feature matrix $\hat{X}$.

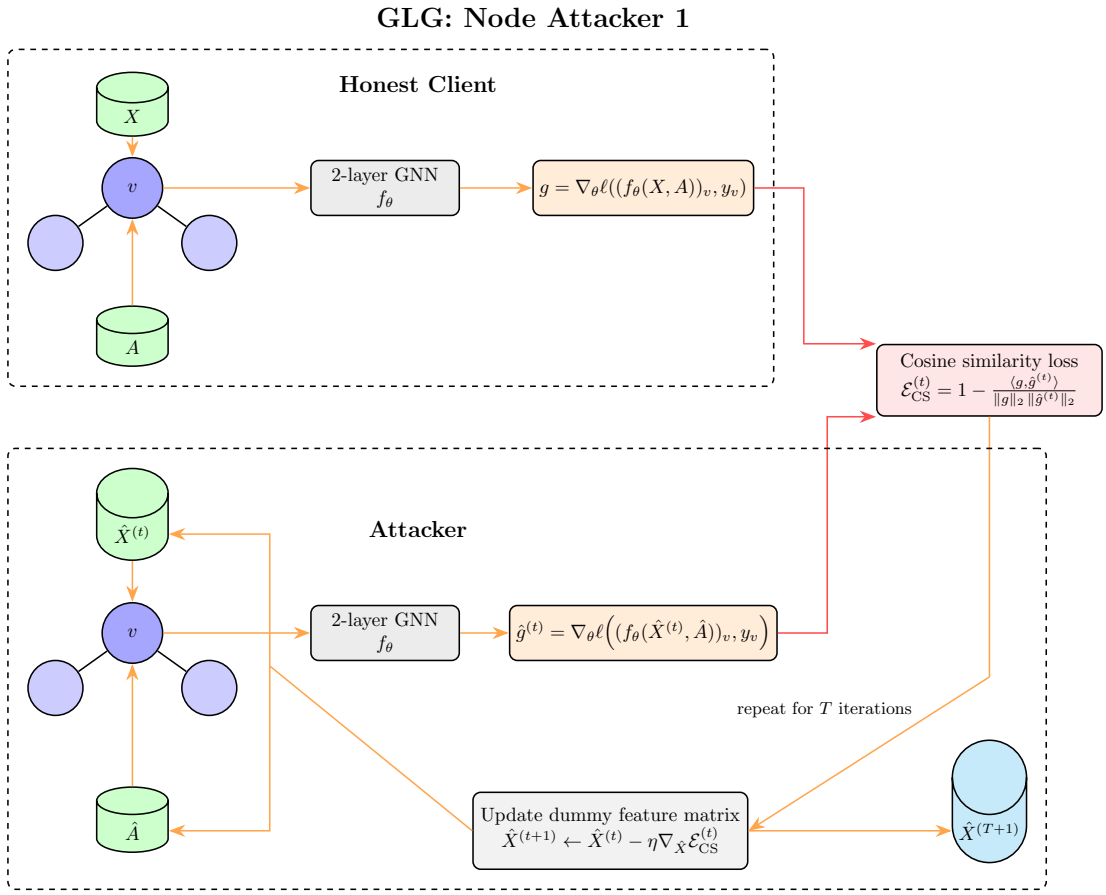

Figure 3.4: Overview of GLG for the Node Attacker 1 setting. An honest client computes the target-node gradient $g = \nabla_\theta \ell((f_\theta(X, A))_v, y_v)$ using a 2-layer GNN f_θ. The upper green cylinders denote the private feature matrix X and adjacency matrix A. The attacker fixes a dummy local tree adjacency $\hat{A}$ and initializes a dummy feature matrix $\hat{X}^{(t)}$ on that tree. At each iteration, the attacker computes the dummy gradient $\hat{g}^{(t)} = \nabla_\theta \ell((f_\theta(\hat{X}^{(t)}, \hat{A}))_v, y_v)$ and updates $\hat{X}^{(t)}$ by minimizing the cosine gradient-matching loss. The lower output cylinder denotes the final reconstructed local feature matrix $\hat{X}^{(T+1)}$. Orange arrows denote forward computation and feature updates, while red arrows denote the comparison between the observed and dummy gradients used to form the loss.

3.7 GraphDLG

GraphDLG was introduced as a gradient leakage attack on GNNs in a federated setting [7]. The attack is designed for graph classification, where each client owns private graphs and shares gradients during training. Its reconstruction target is the full input graph, consisting of both the adjacency matrix and the node feature matrix.

GraphDLG starts from the observation that graph data is harder to invert than images or tabular rows. In a GNN, message passing mixes node features with graph structure. Directly optimizing dummy node features and dummy edges can therefore be unstable. GraphDLG avoids this joint optimization by separating the attack into two stages. It first reconstructs the graph structure from a leaked graph embedding. It then uses the recovered structure to recover node features through a closed-form relation in the GCN gradients.

We first introduce the notation used in this section. Then we derive how the pooled graph embedding $H_{\mathcal{G}}$ can be read directly from the classifier gradients. Next, we describe how a decoder f_{dec} trained on an auxiliary graph set maps $H_{\mathcal{G}}$ to a recovered adjacency matrix $\hat{A}$. Finally, we show how $\hat{A}$ together with the GCN gradients yield the node feature matrix $\hat{X}$.

3.7.1 Notation

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a private graph with node feature matrix X and adjacency matrix A. Within this section, we write $\mathcal{G} = (X, A)$ as a shorthand for the same graph. The victim model f_θ consists of L GCN layers followed by a pooling layer and a linear classifier. We write $H^{(0)} = X$. For $l = 0, \ldots, L-1$, the l-th GCN layer maps $H^{(l)}$ to $H^{(l+1)}$ using weights $W^{(l)}$. Thus, $H^{(L)}$ is the output of the last GCN layer. Pooling produces the pooled graph embedding $H_{\mathcal{G}}$, the graph-level representation obtained after the GCN layers and pooling. The classifier then acts on $H_{\mathcal{G}}$ and has weight matrix $W^{(L)}$ and bias vector $b^{(L)}$. We write g for the released local update observed from the victim client, as defined in Section 4.2.5. In the clean single graph form used in the GraphDLG derivation, this update is

$$g = \nabla_\theta \ell(f_\theta(X, A), y).$$

Its components with respect to $W^{(l)}$ and $b^{(L)}$ are written as $g_{W^{(l)}}$ and $g_{b^{(L)}}$. The loss is graph level, so every node in the graph can affect the observed update through message passing and pooling.

GraphDLG also takes as input an auxiliary set of graphs

$$\mathcal{D}_{\text{aux}} = \{(X_k, A_k)\}_{k=1}^{|\mathcal{D}_{\text{aux}}|}.$$

This set is used to train the structure decoder in Section 3.7.3. The graphs in $\mathcal{D}_{\text{aux}}$ may be the attacker's own client graphs, public graphs, or randomly generated graphs. This auxiliary graph set is part of the adversary knowledge assumed by the GraphDLG threat model. We write $\mathcal{H}_{\text{victim}}$ for the set of pooled graph embeddings recovered from observed victim gradient releases.

The recovered graph is denoted by $(\hat{X}, \hat{A})$. For intermediate GCN layers, $\hat{H}^{(l)}$ denotes the recovered estimate of the layer-l embedding $H^{(l)}$.

3.7.2 Leakage from the MLP Layers

The classifier gradients reveal the pooled graph embedding because the final weight gradient factors into two terms. One term is the loss gradient at the classifier output. The other term is the graph embedding produced by pooling. The bias gradient gives the loss gradient directly.

Let $H^{(L)} \in \mathbb{R}^{n \times d^{(L)}}$ denote the output of the last GCN layer, where n is the number of nodes and $d^{(L)}$ is the feature dimension after L GCN layers. Let $M_{\mathrm{p}} \in \mathbb{R}^{1 \times n}$ denote the pooling row vector. The pooled graph embedding is

$$H_{\mathcal{G}} = M_{\mathrm{p}} H^{(L)} \in \mathbb{R}^{1 \times d^{(L)}}.$$

The final classifier maps this embedding to K_{out} output coordinates. It has weight matrix $W^{(L)} \in \mathbb{R}^{K_{\text{out}} \times d^{(L)}}$ and bias vector $b^{(L)} \in \mathbb{R}^{K_{\text{out}}}$. We write the loss gradient $\nabla_{\hat{y}} \ell$ as a row vector in $\mathbb{R}^{1 \times K_{\text{out}}}$. The gradients of the final classifier are

$$g_{W^{(L)}} = (\nabla_{\hat{y}} \ell)^{\top} H_{\mathcal{G}}, \qquad g_{b^{(L)}} = (\nabla_{\hat{y}} \ell)^{\top}.$$

Hence, $\nabla_{\hat{y}} \ell = g_{b^{(L)}}^{\top}$.

Each row of $g_{W^{(L)}}$ is the pooled graph embedding scaled by the matching entry of $g_{b^{(L)}}$. For any output coordinate $j \in \{1, \ldots, K_{\text{out}}\}$ with $(g_{b^{(L)}})_j \neq 0$,

$$H_{\mathcal{G}} = \frac{(g_{W^{(L)}})_j}{(g_{b^{(L)}})_j}.$$

Here, $(g_{W^{(L)}})_j$ denotes row j of $g_{W^{(L)}}$, and $(g_{b^{(L)}})_j$ denotes entry j of $g_{b^{(L)}}$. The recovered embedding is then used as input to the structure decoder in Section 3.7.3.

3.7.3 Structure Recovery

The adjacency matrix is recovered with a decoder trained on auxiliary graphs. The encoder f_{enc} is the GCN and pooling part of the federated model with the classifier removed. The decoder is denoted by f_{dec}, with trainable parameters ϕ_{dec}. For an auxiliary graph $(X, A) \in \mathcal{D}_{\text{aux}}$, the encoder and decoder compute

$$H_{\mathcal{G}} = f_{\text{enc}}(X, A), \qquad \hat{A} = f_{\text{dec}}(H_{\mathcal{G}}).$$

GraphDLG implements f_{dec} as a three layer MLP with a sigmoid output. It is trained to reconstruct adjacency matrices from graph embeddings by minimizing

$$\ell_{\text{recon}} = \frac{1}{|\mathcal{D}_{\text{aux}}|} \sum_{(X,A) \in \mathcal{D}_{\text{aux}}} \| f_{\text{dec}}(f_{\text{enc}}(X, A)) - A \|_{\mathrm{F}}^2 .$$

Here, $\| \cdot \|_{\mathrm{F}}$ denotes the Frobenius norm.

Auxiliary graphs may follow a different distribution from the victim graphs. GraphDLG handles this by aligning graph embeddings rather than raw graphs. For two sets of pooled graph embeddings $\mathcal{H}_i$ and $\mathcal{H}_j$, the maximum mean discrepancy (MMD) term is

$$\mathrm{MMD}(\mathcal{H}_i, \mathcal{H}_j) = \left\| \frac{1}{|\mathcal{H}_i|} \sum_{H \in \mathcal{H}_i} \varphi(H) - \frac{1}{|\mathcal{H}_j|} \sum_{H \in \mathcal{H}_j} \varphi(H) \right\|_{\mathrm{RKHS}} . \tag{3.6}$$

Here, $\|\cdot\|_{\mathrm{RKHS}}$ denotes the norm in the reproducing kernel Hilbert space associated with the feature map φ. The map φ sends pooled graph embeddings into that space. The auxiliary embedding set is $\mathcal{H}_{\mathrm{aux}} = \{f_{\mathrm{enc}}(X, A) : (X, A) \in \mathcal{D}_{\mathrm{aux}}\}$. The victim embedding set $\mathcal{H}_{\mathrm{victim}}$ is obtained from classifier gradients rather than from raw graphs. With weight λ, the decoder objective is

$$\ell_{\mathrm{total}} = \ell_{\mathrm{recon}} + \lambda\,\mathrm{MMD}(\mathcal{H}_{\mathrm{aux}}, \mathcal{H}_{\mathrm{victim}}).$$

The MMD term is used to reduce the mismatch between auxiliary and victim graph embeddings before the decoder is applied to the leaked victim embedding. After training, the decoder is applied to $H_{\mathcal{G}}$ to obtain $\hat{A}$.

3.7.4 Feature Recovery

Once $\hat{A}$ is available, GraphDLG recovers node features through the Graph Node Features Recovery procedure. The recovery proceeds from the last GCN layer back to the input. It reconstructs estimates $\hat{H}^{(L-1)}, \hat{H}^{(L-2)}, \ldots, \hat{H}^{(0)}$ in that order, where $\hat{H}^{(0)} = \hat{X}$. The order matters because each step uses quantities recovered from the layer above.

The recovered adjacency matrix is converted to the normalized adjacency used by the victim GCN. The self-loop lets each node retain its own embedding during aggregation, and the symmetric degree normalization keeps the scale of aggregated messages comparable across nodes with different degrees. Concretely,

$$\tilde{A} = \hat{A} + I_n, \qquad \bar{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}},$$

where $\tilde{D} \in \mathbb{R}^{n\times n}$ is the diagonal degree matrix of $\tilde{A}$, and $\tilde{A}, \bar{A} \in \mathbb{R}^{n\times n}$. If $\hat{A} = A$ and the activation derivatives match those in the victim forward pass, the recovery equations are exact. In the attack, they are applied using the recovered adjacency matrix and the available activation derivatives.

The key relation for a GCN layer l is

$$g_{W^{(l)}} = (H^{(l)})^{\top} r^{(l)}.$$

Here, $H^{(l)} \in \mathbb{R}^{n\times d^{(l)}}$ is the input embedding to layer l, and $r^{(l)} \in \mathbb{R}^{n\times d^{(l+1)}}$ is the coefficient matrix computed from the known model structure. Thus $g_{W^{(l)}} \in \mathbb{R}^{d^{(l)}\times d^{(l+1)}}$. Once $r^{(l)}$ has been fixed, the unknown in this equation is $H^{(l)}$.

Given the normalized adjacency, $r^{(l)}$ can be computed recursively as

$$\begin{aligned} r^{(L-1)} &= \bar{A}^{\top}\left(M_{\mathrm{p}}^{\top}\,\nabla_{\hat{y}}\ell\; W^{(L)} \odot a^{(L)\prime}\right), \\ r^{(l)} &= \bar{A}^{\top}\left(r^{(l+1)}\,(W^{(l+1)})^{\top} \odot a^{(l+1)\prime}\right), \qquad l = 0, \ldots, L-2. \end{aligned} \tag{3.7}$$

Here, $\odot$ denotes the Hadamard product, applied elementwise between matrices of the same dimensions. The term $a^{(l+1)\prime} \in \mathbb{R}^{n\times d^{(l+1)}}$ denotes the activation derivative at the output of layer l. During reverse recovery, GraphDLG derives this derivative from the recovered layer output when possible. For the last GCN layer, $a^{(L)\prime}$ is set to an all-ones matrix when pooling hides the node-level final activations. We write $J_{n\times d^{(L)}}$ for this all ones matrix, with the same shape as the last GCN layer output.

After computing $r^{(l)}$, the unknown layer input $H^{(l)}$ is recovered by solving a linear system. GraphDLG rewrites

$$g_{W^{(l)}}^{\top} = (r^{(l)})^{\top} H^{(l)}$$

in vectorized form. We write flatten for the vectorization that reads a matrix row by row and stacks the entries into a single column vector. We write unflatten for the inverse operation that reshapes the solution vector back to the matrix shape required by the current layer. With this convention,

$$\text{flatten}(UXV) = (U \otimes V^\top)\,\text{flatten}(X),$$

where $\otimes$ denotes the Kronecker product. Let

$$B = \text{flatten}(g_{W^{(l)}}^\top), \qquad R = (r^{(l)})^\top \otimes I_{d^{(l)}},$$

where $d^{(l)}$ is the feature dimension of $H^{(l)}$. Then $B \in \mathbb{R}^{d^{(l+1)}d^{(l)}}$ and $R \in \mathbb{R}^{d^{(l+1)}d^{(l)} \times nd^{(l)}}$. The identity above turns $g_{W^{(l)}}^\top = (r^{(l)})^\top H^{(l)}$ into the linear system $Rz = B$, with $z \in \mathbb{R}^{nd^{(l)}}$. Solving this system and applying unflatten recovers $\hat{H}^{(l)} \in \mathbb{R}^{n \times d^{(l)}}$.

Algorithm 6 gives the full attack procedure, while Figure 3.5 provides a visual overview of the attack.

3.7.5 Attack Metric

GraphDLG evaluates node feature recovery with mean squared error (MSE). For $X, \hat{X} \in \mathbb{R}^{n \times d}$, where n is the number of nodes and d is the feature dimension, we report

$$\text{MSE}(X, \hat{X}) = \frac{1}{nd} \sum_{u=1}^{n} \sum_{j=1}^{d} \left(X_{uj} - \hat{X}_{uj} \right)^2. \tag{3.8}$$

MSE measures the average squared difference between the true and recovered node feature values. Lower MSE means that the recovered node features are closer to the original features.

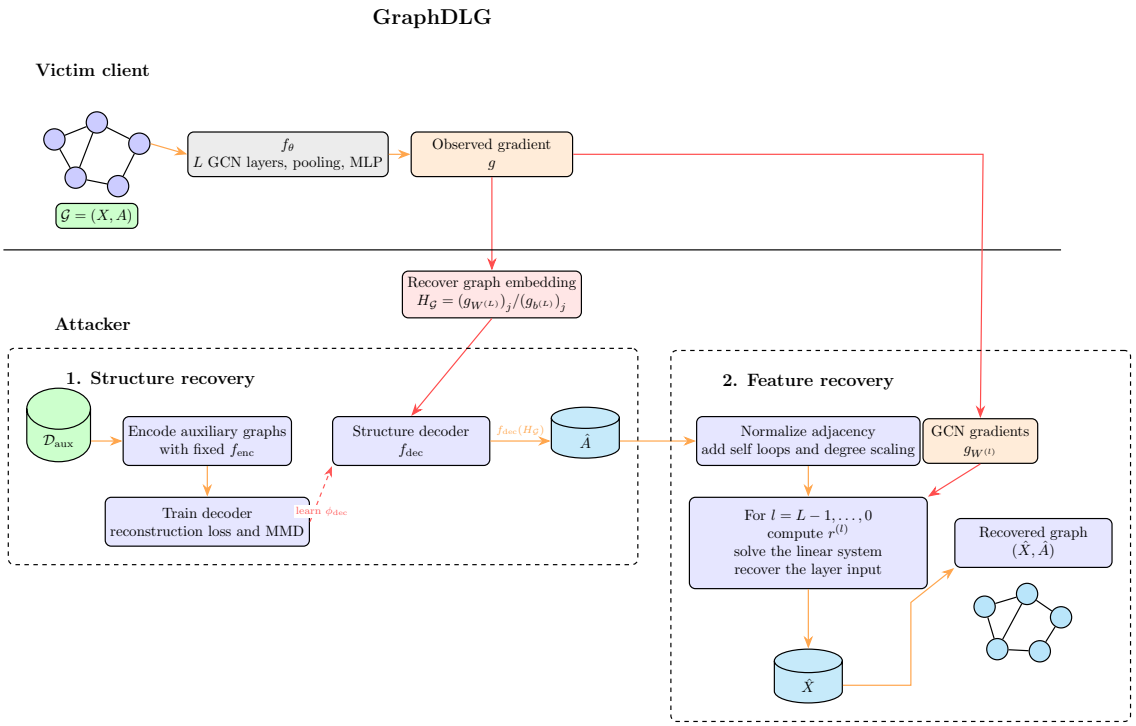

Figure 3.5: Overview of GraphDLG. The victim client computes the gradient g on a private graph $\mathcal{G} = (X, A)$. The attacker uses the classifier gradients to recover the pooled graph embedding $H_\mathcal{G}$. A decoder trained on auxiliary graphs maps this embedding to a recovered adjacency matrix $\hat{A}$. The recovered adjacency is normalized and then used with the GCN gradients to recover node features layer by layer. The output is the reconstructed graph $(\hat{X}, \hat{A})$.

Algorithm 6 GraphDLG

Input: model f_θ, victim gradient $g = \nabla_\theta \ell(f_\theta(X, A), y)$, auxiliary graph set $\mathcal{D}_{\text{aux}}$, victim embedding set $\mathcal{H}_{\text{victim}}$, number of GCN layers L, decoder training iterations T_{dec}, MMD weight λ.

Output: Recovered graph $(\hat{X}, \hat{A})$.

$\nabla_{\hat{y}} \ell \leftarrow g_{b^{(L)}}^\top$
Choose j such that $(g_{b^{(L)}})_j \neq 0$
$H_{\mathcal{G}} \leftarrow (g_{W^{(L)}})_j \,/\, (g_{b^{(L)}})_j$
Let f_{enc} be the GCN and pooling part of f_θ
Initialize decoder f_{dec} with parameters ϕ_{dec}
for $t \leftarrow 1$ to T_{dec} **do**
    $\mathcal{H}_{\text{aux}} \leftarrow \{f_{\text{enc}}(X, A) : (X, A) \in \mathcal{D}_{\text{aux}}\}$
    Update ϕ_{dec} by minimizing $\ell_{\text{recon}} + \lambda \, \text{MMD}(\mathcal{H}_{\text{aux}}, \mathcal{H}_{\text{victim}})$
end for
$\hat{A} \leftarrow f_{\text{dec}}(H_{\mathcal{G}})$
$\tilde{A} \leftarrow \hat{A} + I_n$
Let $\tilde{D}$ be the degree matrix of $\tilde{A}$
$\bar{A} \leftarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$
for $l \leftarrow L - 1$ down to 0 **do**
    if $l = L - 1$ **then**
        $a^{(L)\prime} \leftarrow J_{n \times d^{(L)}}$ ▷ no $\hat{H}^{(L)}$
        $r^{(L-1)} \leftarrow \bar{A}^\top \left(M_{\text{p}}^\top \, \nabla_{\hat{y}} \ell \, W^{(L)} \odot a^{(L)\prime} \right)$
    else
        Derive $a^{(l+1)\prime}$ from $\hat{H}^{(l+1)}$
        $r^{(l)} \leftarrow \bar{A}^\top \left(r^{(l+1)} \, (W^{(l+1)})^\top \odot a^{(l+1)\prime} \right)$
    end if
    $B \leftarrow \text{flatten}(g_{W^{(l)}}^\top)$
    $R \leftarrow (r^{(l)})^\top \otimes I_{d^{(l)}}$
    Solve $Rz = B$
    $\hat{H}^{(l)} \leftarrow \text{unflatten}(z)$
end for
$\hat{X} \leftarrow \hat{H}^{(0)}$
return $(\hat{X}, \hat{A})$

In Algorithm 6, f_{enc} denotes the encoder defined in Section 3.7.3. It shares its parameters with f_θ and outputs the pooled graph embedding $H_{\mathcal{G}}$.

The recursion starts at $l = L - 1$, where $a^{(L)\prime}$ is set to a matrix of all ones. As a result, $\hat{H}^{(L)}$ is never reconstructed. In each later step, the matrix $\hat{H}^{(l+1)}$ used to compute $a^{(l+1)\prime}$ was already recovered in the previous step.

For ReLU activations, GraphDLG computes $a^{(l+1)\prime}$ as the elementwise indicator of positive entries in the recovered layer output $\hat{H}^{(l+1)}$ when this derivative is not set to an all-ones matrix.

In our implementation, the recovered features $\hat{X}$ are clipped to the feature range observed in $\mathcal{D}_{\text{aux}}$ for numerical stability.

Chapter 4

Methodology and Model Architecture

This chapter describes the experimental setup used to evaluate leakage attacks under DP noise.

4.1 Adaptation of Attack Codebases

In this work, we evaluate multiple gradient-based leakage attacks, namely DLG [2], TabLeak [4], GRAIN [5], GLG [6], and GraphDLG [7]. For each attack, we base our implementation on the original codebase released by the respective authors.[1] For GraphDLG [7], no public implementation was available at the time of our experiments, so we implemented the attack from the description in the paper.

To align the implementations with our experimental objectives, we introduce targeted modifications to the original code. In particular, we adapt the attack pipelines to use gradients that are stored during privacy-preserving training, instead of relying on clean gradients computed directly via PyTorch's automatic differentiation during standard training. This modification enables evaluation of attack behavior under the NDP setting, in which gradients include noise added during training and are observed by an adversary.

The core optimization procedures and attack logic remain unchanged. Our modifications are limited to gradient retrieval, allowing the attacks to be evaluated under privacy-preserving training without altering their fundamental assumptions or objectives, even though the original attacks often do not involve model training.

4.2 Experimental Setup

This section describes the experimental environment used to evaluate the considered leakage attacks under privacy-preserving training settings. We outline the datasets, model architectures, training and communication protocols, and privacy mechanisms applied in our experiments to ensure reproducibility and clarity.

[1] DLG: https://github.com/mit-han-lab/dlg; TabLeak: https://github.com/eth-sri/tableak; GRAIN: https://github.com/insait-institute/GRAIN; GLG: https://openreview.net/forum?id=a0mLrqkWyx

4.2.1 Datasets

Experiments involving the TabLeak attack are conducted on the Adult dataset [21], following the experimental setup of the original TabLeak work. To assess robustness to dataset choice, we additionally evaluate TabLeak on the Health Heritage dataset [22], the Insurance dataset [23], and the Law School dataset [24].

GRAIN is evaluated on the CiteSeer dataset [25]. For graph-based attacks, we distinguish between feature encoding and privacy experiments. Feature encoding is evaluated on the WebKB dataset [26] using the Cornell subset, as it contains categorical node features. GLG privacy experiments and DLG experiments are conducted on the Facebook Page–Page network dataset [27], following the setup of [6]. GraphDLG is evaluated on the PROTEINS dataset [28], following the graph-level setting considered in the GraphDLG paper [7].

4.2.2 Model Architecture

For TabLeak, we adopt the model architectures used in the original evaluation, including linear models, fully connected networks, CNNs, and ResNet variants, with the same depth, layer dimensions, and activation functions. In the main experiments, results are reported for fully connected models, whereas linear, CNN, and ResNet architectures are evaluated in addition to examine how architectural choices influence leakage behavior. For the CNN, we remove batch normalization layers because this yields a more stable attack behavior.

For GRAIN experiments, we focus on GAT architectures, following the setting used in the original GRAIN evaluation. Although the attack can also be applied to a GCN, its theoretical guarantees depend on additional structural properties of the adjacency matrix, making the reconstruction behavior less predictable in practice.

For GLG experiments, we use a GCN model following [10]. The model consists of two graph convolutional layers with a nonlinear activation between them, as in the original formulation. The same architecture is used for both GLG and DLG.

For GraphDLG, we use the same model architecture as in the original paper [7], consisting of a two-layer GCN followed by mean pooling and a linear classifier.

For all attacks, the attack model is the target network instantiated with the architecture and parameters corresponding to the observed gradients. Gradient inversion is then performed using these gradients.

4.2.3 Training Topology

We evaluate leakage attacks under three distinct training topologies:

- **Centralized:** a centralized training setup, where nodes compute local gradients and a central server aggregates the released updates.
- **Muffliato:** a fully decentralized gossip-based protocol [17].
- **Skip-Ring:** a sequential decentralized protocol with straggler skipping [19].

The centralized topology serves as a baseline and reflects the standard FL setting. In contrast, the Muffliato and Skip-Ring topologies distribute gradient exchange across the network, limiting each participant's view to a subset of the communicated updates.

For Muffliato, we use the Erdős-Rényi communication graph model defined in Section 3.1. At each training round, we sample a new graph $\mathcal{G}^{(t)} \sim G(n, q)$, with edge probability $q = c\log(n)/n$. In the implementation used here, $c = 1 - p$, where p is the skipping probability used in the latency model. The sampled graph is then used for the T_{gossip} gossip steps in that training round. As in [19], we set $T_{\text{gossip}} = 2$ gossip steps per round in all experiments.

Following [19], we measure training progress in terms of total latency rather than epochs or iterations. Here, latency denotes cumulative computation and communication time. For Skip-Ring, this corresponds to the token-passing process with timeout-based skipping. For centralized and Muffliato training, latency is computed using the same framework described in [19, Appendix G], so that all three training topologies are compared under a common latency measure.

4.2.4 Privacy Mechanism

In all training configurations, local gradients are clipped to a norm bound $C = 1$, and Gaussian noise with scale σ is added before the released update is communicated, following the Gaussian mechanism of Section 2.5.4. At each step, every node forms its minibatch $\mathcal{B}_v$ by drawing B examples uniformly without replacement from its local dataset $\mathcal{D}_v$, independently across steps, so the sampling procedure is uniform subsampling.

For simplicity, we clip and add noise to the batch-average gradient rather than clipping each per-example gradient. The reported ε could be lower with per-example clipping, which would allow privacy amplification by subsampling [29], but the same scheme is used for all three topologies, so the comparison between them is unaffected.

We compute the privacy loss separately for each topology using its corresponding privacy accountant. For Skip-Ring, after T update steps, the privacy level $\varepsilon(T)$ is computed using the accounting method of [19, Thm. 2]. For Muffliato and centralized training, we use RDP accounting following [17, Thm. 4]. In that case, we optimize over the Rényi order α and apply the RDP-to-DP conversion of Proposition 2.5.2 to obtain an (ε, δ) guarantee.

When comparing topologies, the same reported ε should be interpreted as the same privacy loss level under each method's own accountant. Because each topology uses a different accountant, equal ε values do not correspond to identical noise scales or privacy mechanisms across topologies and are only meaningful within each method's own framework.

For each run, the noise scale σ is held fixed throughout training, and we set $\delta = 10^{-6}$. The reported privacy level is then obtained as a function of the total number of update steps T for the fixed pair (σ, δ), using the accountant associated with the corresponding topology. Thus, ε always reflects the privacy loss of that specific protocol after T updates.

4.2.5 Threat Model

We adopt an *honest-but-curious* threat model in all experiments. The adversary follows the training and communication procedure but tries to infer private information from what the protocol exposes at one target participant during training. We do not assume that the adversary has complete access to the participant's device. In particular, the adversary does not access the raw local training data and does not observe the sampled examples used in a local step.

As illustrated in Figure 4.1, at an attacked local step, the adversary observes the model state immediately before the participant performs its local update and the released local update immediately after that step, following clipping and noise addition. We use this released local update as the attack input in all topologies studied in this thesis. The model state observed before the attacked local step is the actual state reached by decentralized training up to that time and therefore already reflects the effect of gossip or averaging from earlier iterations.

To ensure that the attacks are evaluated under the actual privacy mechanism used during training, we use updates recorded during the real privacy-preserving training runs rather than recomputing clean gradients with automatic differentiation on a dummy model.

We do not impose a single label assumption across all attacks. For DLG, the label is optimized jointly with the input and is therefore not assumed to be known. For GLG, GRAIN, and TabLeak, we assume known labels during reconstruction, following the standard setting of each method. GraphDLG follows its original setting, where the attacker uses the observed classifier gradients together with an auxiliary graph pool. This keeps each attack in the setting actually used in our evaluation.

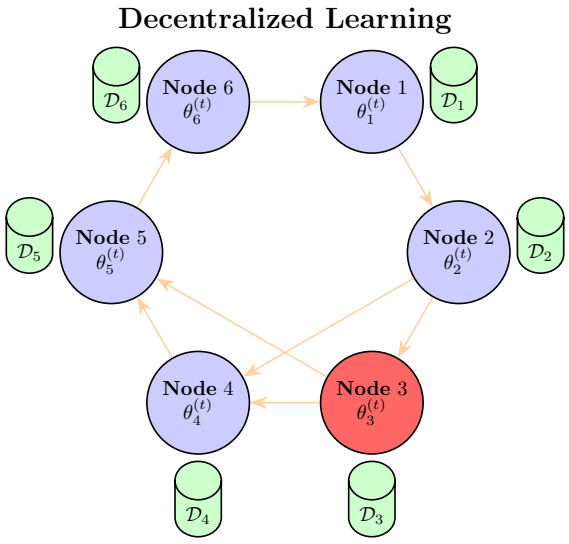

Figure 4.1: Decentralized learning network with a compromised node (red) performing gradient inversion attacks. Blue circles denote honest nodes, each storing local model parameters $\theta_v^{(t)}$; green cylinders denote local datasets $\mathcal{D}_v$; orange arrows indicate peer-to-peer communication along the communication graph.

Chapter 5

Results and Findings

This chapter reports the experimental results. We first describe the plotting conventions, baselines, and default settings, then present the privacy–reconstruction trade-off per attack, the effect of feature encoding, other influencing factors, and model accuracy.

Interpretation of plots and tables. Across all plots, centralized training is shown in blue, Skip-Ring in green, and Muffliato in orange, with the random baseline shown as a red dashed line. Reported values are means over 100 independent runs, using the reconstruction metric defined for each attack unless stated otherwise. In reconstruction tables, bold values mark the strongest defense among the compared settings. In model accuracy tables, bold values mark the highest mean predictive accuracy.

For all experiments except the batch size and epoch experiment, we scale the Gaussian noise scale σ across topologies to align the initial privacy level ε. Skip-Ring uses its chosen σ, while centralized and Muffliato use scaled values for visual comparability. This alignment only affects how easily the ε-axis can be compared across topologies, as the effective privacy depends on both σ and the communication topology. Thus, σ alone is not directly comparable across settings, and we do not report results as a function of σ.

During training, we store the adversary's per-step observations for the attacked node. To evaluate reconstruction at a chosen latency, we select the saved observation closest to that latency, instantiate the attack with the corresponding saved model parameters, and run the attack on the associated released noisy update. The privacy reconstruction plots are evaluated at several target latencies chosen to be comparable across training topologies. Each topology curve therefore has points from the same sequence of target latencies. The first point on each curve corresponds to an early training stage, the last point corresponds to the final training stage, and the middle points correspond to intermediate stages. The horizontal axis reports the resulting privacy loss ε at each selected observation. Latency is therefore represented through the ordering of the checkpoints rather than as the plotted axis. For both tables and values referenced in the text, we report the reconstruction based on the final observation available to the adversary.

Large ε values correspond to low noise and are included to illustrate the full privacy–reconstruction trade-off, rather than as practically meaningful privacy guarantees.

Random baseline. To compute a random baseline experiment involving TabLeak, we sample each feature independently according to its empirical distribution in the dataset. For each batch size, we generate multiple random batches as Monte Carlo samples to estimate the mean and standard deviation of the reconstruction error. Since the randomly sampled rows have no natural correspondence to the rows in the true batch, we align each random batch to the true batch using optimal bipartite matching. We then score the reconstruction using the same tolerance-based metric as in the main evaluation. We use the original implementation provided by [4].

For GLG and GraphDLG, the random baseline builds a guess $\hat{x}_v$ whose j-th coordinate is an independent uniform draw from the observed values of feature j across all nodes:

$$(\hat{x}_v)_j = X_{r_j,j}, \qquad r_j \sim U\{1,\dots,n\} \text{ i.i.d.}, \quad j = 1,\dots,d,$$

where $X \in \mathbb{R}^{n\times d}$ is the feature matrix and $U\{1,\dots,n\}$ denotes the uniform distribution over the node indices $\{1,\dots,n\}$. Over many independent trials, each guess is compared with the feature vector of a uniformly sampled node, and we report RNMSE for GLG and MSE for GraphDLG.

Default experimental settings. Unless stated otherwise, centralized and Muffliato training use a constant learning rate hyperparameter η. Skip-Ring uses the decreasing learning rate schedule defined in Section 3.2, parameterized by the hyperparameter ζ. Table 5.1 lists the main hyperparameter settings by result. The p column records the skipping probability.

For DLG, sigmoid activation is used because the attack requires the model to be twice differentiable [2]. For GRAIN, sampled subgraphs on non-chemical datasets have 10 to 20 nodes, with 40 sampled graphs. The GAT embedding dimension is $d = 300$.

Across topologies, we tune optimization hyperparameters separately because centralized, gossip-based, and sequential token-passing training exhibit different optimization dynamics. Our goal is to evaluate privacy leakage under well-tuned training conditions that reflect practical use, rather than enforcing identical optimization settings across fundamentally different protocols. While learning rate and schedule can influence the observed updates, we focus on comparing methods under their best-performing configurations.

Result	Arch.	Dataset	n	B	$d_{\mathcal{W}}$	Lat.	Enc.	p	η_{C}	η_{M}	ζ
Figure 5.1	FC NN	Adult	1k	16	20	20k	one-hot	10^{-4}	0.3	0.003	0.07
Figure 5.2, Figure 5.3, Table 5.2	GCN	Facebook	10	128	200	20k	BoW	10^{-4}	0.01	0.01	0.1
Table 5.3, Table 5.4	FC NN	Adult	1k	16	20	20k	all	10^{-4}	0.3	0.003	0.07
Table 5.3, Table 5.4	GCN	Cornell	10	128	200	20k	BoW	10^{-4}	0.0007	0.0007	0.0007
	GCN	Cornell	10	128	200	20k	(R.) binary, hashing	10^{-4}	0.0007	0.0007	0.007
	GCN	Cornell	10	128	200	20k	one-hot	10^{-4}	0.0001	0.0001	0.007
Figure 5.4	GCN	PROTEINS	10	1	200	20k	one-hot	10^{-4}	0.07	0.07	0.7
Figure 5.5, Figure 5.6	GAT	CiteSeer	$[10, 20]$	1	200	100	BoW	10^{-4}	0.001	0.001	0.7
Figure 5.7	FC NN	Adult	100	varied	30	10k	one-hot	10^{-4}	0.03	0.003	0.03
Figure 5.8	FC NN	Adult	1k	16	20	20k	one-hot	10^{-4}	0.3	0.003	0.07
	FC NN	Adult	1k	16	20	20k	one-hot	$\{0.5, 0.7\}$	0.07	0.003	0.07
Figure 5.9	linear	Adult	100	16	6	10k	one-hot	10^{-4}	0.7	0.007	0.03
	FC NN	Adult	100	16	30	10k	one-hot	10^{-4}	0.03	0.003	0.03
	FC NN large	Adult	100	16	80	10k	one-hot	10^{-4}	0.03	0.003	0.03
	CNN	Adult	100	16	60	10k	one-hot	10^{-4}	0.03	0.003	0.03
	ResNet	Adult	100	16	10	10k	one-hot	10^{-4}	0.03	0.003	0.03
Figure 5.10	FC NN	Adult	1k	16	20	epoch based	one-hot	10^{-4}	0.3	0.003	0.07
Figure 5.11	FC NN	Adult, Health Heritage, Law School, Insurance	10	16	40	5k	one-hot	10^{-4}	0.007	0.007	0.07

Table 5.1: Default experimental settings by result. The column p denotes the skipping probability. The columns η_{C} and η_{M} denote the learning rates for centralized and Muffliato training, respectively. The column ζ denotes the Skip-Ring learning rate parameter. The abbreviation k denotes 10^3.

5.1 Privacy–Reconstruction Trade-Off

5.1.1 TabLeak

Figure 5.1 shows the privacy–reconstruction trade-off for TabLeak with one-hot encoding. Reconstruction accuracy usually increases with ε, but the size of this effect depends on the noise scale and the training topology.

Gradient leakage depends strongly on the noise scale σ. Each point in Figure 5.1 is evaluated from an observation saved at a comparable cumulative latency, up to the latency budget reported in Table 5.1. As latency increases, more updates are released, and the privacy accountant accumulates a larger ε. Thus, movement to the right in the plot corresponds to later points in training with larger cumulative privacy loss.

At $\sigma = 1$, all three topology curves remain below the random baseline across the evaluated latencies, even though reconstruction accuracy rises with ε. The attack therefore does not extract more information than random guessing in this high-noise setting. At $\sigma = 0.1$, the curves move closer to the baseline, but they remain mostly below it. At $\sigma = 0.01$, centralized and Skip-Ring training are clearly above the baseline for most evaluated points, while Muffliato remains below or close to the baseline. Lower noise therefore increases leakage, but the evidence is topology-dependent rather than uniform.

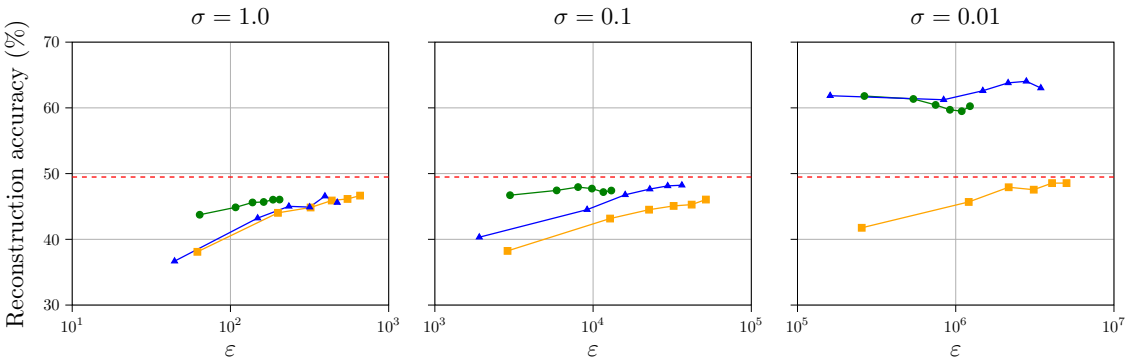

Figure 5.1: TabLeak reconstruction accuracy on the Adult dataset using one-hot encoding and the FC NN model, shown as a function of ε for noise scales $\sigma \in \{1, 0.1, 0.01\}$.

5.1.2 GLG

Figure 5.2 shows the corresponding results for GLG. The curves do not show a clean monotone dependence on ε. For each noise scale, RNMSE stays in a narrow band for much of the curve and falls mainly at the largest observed ε values.

Larger noise is associated with larger reconstruction error, but the separation between $\sigma = 1$ and $\sigma = 0.1$ is small. At both higher noise levels, most values lie between about 2.0 and 2.5, above the random baseline. At $\sigma = 0.01$, the values are lower, mostly between about 1.8 and 2.2, but they remain close to or above the baseline. These results suggest that GLG is only weakly responsive to privacy loss in this setting. The privacy–reconstruction relation is therefore much weaker than for TabLeak.

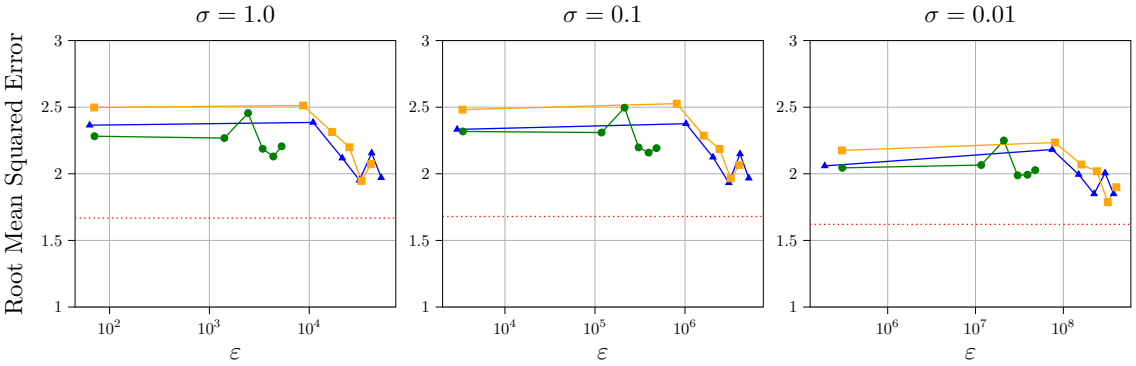

Figure 5.2: GLG reconstruction error (RNMSE) on the Facebook dataset using BoW encoding and the GCN model, shown as a function of ε for noise scales $\sigma \in \{1, 0.1, 0.01\}$.

5.1.3 DLG

Figure 5.3 shows the reconstruction error of DLG under different noise scales. The behavior differs substantially from the previous attacks. For $\sigma = 1$ and $\sigma = 0.1$, the RNMSE remains extremely large across all values of ε, ranging between approximately 10^{10} and 10^{16} and staying far above the random baseline. Such large values arise because the reconstructed input $\hat{x}$ becomes essentially unrelated to the true input x, causing $\|\hat{x} - x\|_2$ to dominate $\|x\|_2$ in the definition of RNMSE in Eq. (3.5). We therefore interpret these values as a complete reconstruction failure rather than a partial recovery.

For $\sigma = 0.01$, the reconstruction error is reduced by several orders of magnitude compared to larger noise scales. Nevertheless, the error remains high overall and does not reach successful recovery.

These results indicate that DLG is highly sensitive to gradient noise in this setting. The attack fails under moderate and high noise, and does not exhibit a meaningful privacy–reconstruction trade-off across ε. This behavior is consistent with [2, Sec. 5.1], which reports that DLG fails when the added gradient noise variance is larger than 0.01, as the recovered images become visually uninformative, even though no explicit reconstruction error metric is reported.

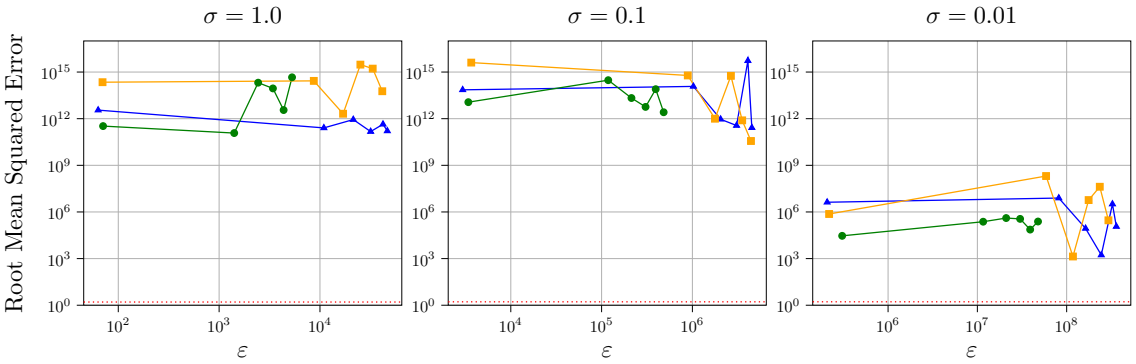

Figure 5.3: DLG reconstruction error (RNMSE) on the Facebook dataset using BoW encoding and the GCN model, shown as a function of ε for noise scales $\sigma \in \{1, 0.1, 0.01\}$.

5.1.4 GraphDLG

Figure 5.4 shows the GraphDLG results on the PROTEINS dataset. High noise moves node feature recovery close to the random baseline. At $\sigma = 1.0$, the final MSE is 0.33 for Central, 0.33 for Skip-Ring, and 0.32 for Muffliato. These values are close to the random baseline estimated for this experiment, which indicates that the attack recovers little beyond a draw from the empirical feature distribution. Lower noise gives lower reconstruction error. At $\sigma = 0.1$, the final MSEs are 0.31, 0.24, and 0.27 for Central, Skip-Ring, and Muffliato, respectively. At $\sigma = 0.01$, they are 0.28, 0.21, and 0.25. Skip-Ring has the lowest mean MSE at the two smaller noise scales, although the differences between topologies should be interpreted cautiously.

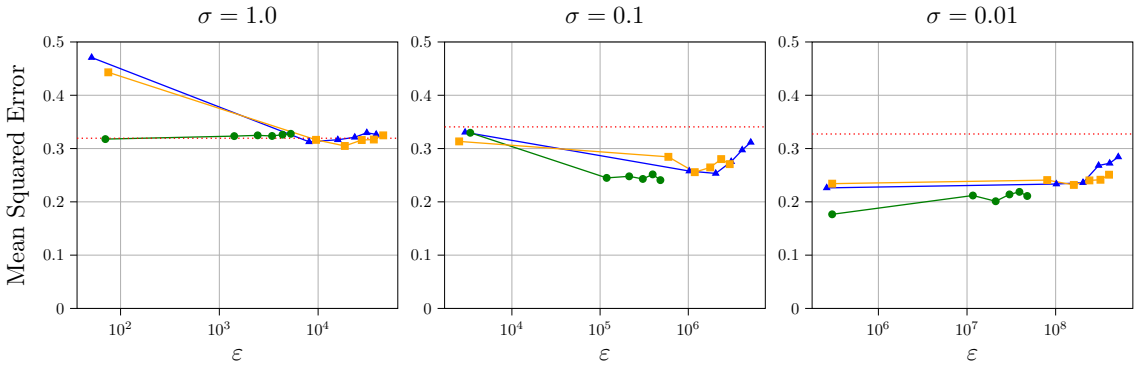

Figure 5.4: GraphDLG reconstruction error (MSE) on the PROTEINS dataset using one-hot encoding and the GCN model used by GraphDLG, shown as a function of ε for noise scales $\sigma \in \{1, 0.1, 0.01\}$.

5.1.5 GRAIN Fails under Noisy Gradients

As discussed earlier, GRAIN relies on gradients retaining a low-rank structure so that span-based filtering can narrow down valid subgraph candidates. Its exact reconstruction guarantee depends on the span check condition in [5, Thm. 5.1]. Figure 5.5 shows that even a very small Gaussian noise scale, around $\sigma = 10^{-7}$ in our experiments, already breaks this structure.

When this happens, the span of the gradient expands, and the filtering step can no longer separate valid subgraph candidates from invalid ones, and the attack fails. We verified that this behavior is stable across a wide range of tolerance values in the span check procedure. Using the original GRAIN tolerances ($\tau_1 = \tau_2 = 10^{-7}$, $\tau_3 = 10^{-8}$) as well as larger values up to 10^{-2} leads to the same outcome once noise is present. The exact numerical rank can vary with the tolerance, but this does not change the fact that the attack fails.

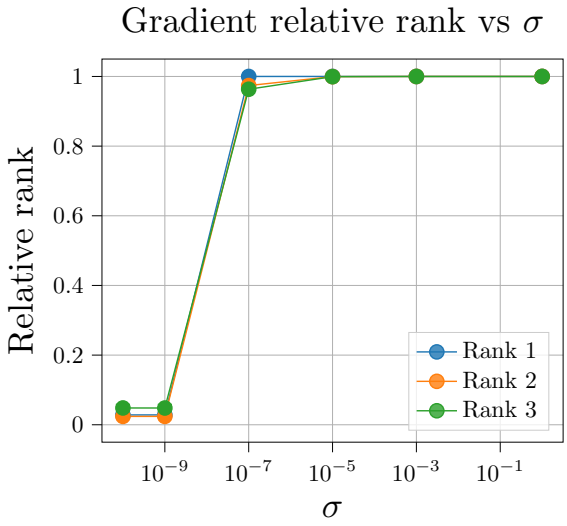

Figure 5.5: Relative gradient rank as a function of Gaussian noise scale σ. Each curve corresponds to one of the three GNN layers.

To address this, we applied singular value hard thresholding (SVHT) [30], a matrix denoising technique that zeroes out small singular values using an optimal threshold, to the gradient matrices prior to running the attack. However, the attack still failed. This suggests that denoising does not recover the structure required by GRAIN: even if a low-rank matrix is obtained, its row and column spaces no longer match those of the original gradient, which the span-based filtering relies on.

We verify this structural failure directly using GRAIN's own filter criterion in Eq. (3.3), measuring for each noise scale σ how far the true node features lie from the column space of the denoised weight gradient. This is the span condition used by GRAIN in [5, Thm. 5.1]. As a reference, we compare against the column space of the clean gradient recomputed at the same model state. Figure 5.6 shows two things. On clean gradients, the distance stays essentially flat around 10^{-4} across all σ, confirming that the span-check condition holds whenever the gradient is undisturbed. On SVHT-denoised gradients, the distance grows steadily with σ and crosses GRAIN's acceptance threshold around $\sigma \approx 10^{-7}$, the same noise level at which the rank in Figure 5.5 becomes full. The two figures capture the same underlying failure from different sides, where the rank plot shows that the necessary condition for the filter to discriminate is violated, and the span plot shows that even if we restore the rank via SVHT, the sufficient condition that the gradient's column space contains the true features is no longer met.

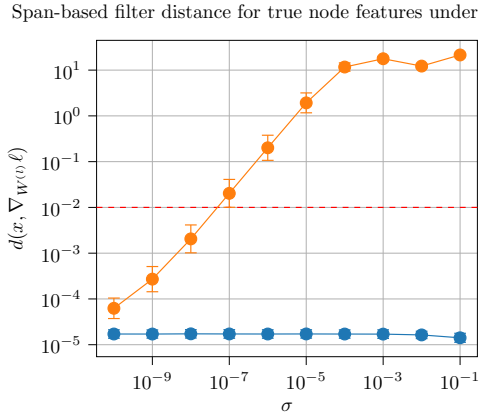

Figure 5.6: Span-check distance $d(x, \nabla_{W^{(l)}}\ell)$ of the true node features against the first-layer gradient column space, as a function of the noise scale σ. The blue line shows the column space of the clean gradient recomputed at the same model state. Orange shows the column space recovered from the SVHT-denoised noisy gradient. The red dashed line shows GRAIN's span filter acceptance threshold.

The observed threshold at which the gradient becomes numerically full rank should be interpreted as empirical and setting-dependent rather than universal. In particular, it may vary with factors such as batch size, the number of nodes, and the relation between the number of nodes and the embedding dimension. However, the key observation is that even very small noise is sufficient to destroy the low-rank structure in our setting, which in turn renders the span-based filtering ineffective.

5.1.6 Reconstruction Accuracy across Attacks

For completeness, Table 5.2 compares reconstruction error across attacks using the same GCN model and Facebook dataset with $\sigma = 1$. The comparison should still be read as an adapted evaluation rather than a direct comparison of the original attack papers, because the attacks use different prior knowledge and reconstruction procedures.

DLG and TabLeak were not originally designed for graph node features. Here, both are used as gradient-matching attacks on GCN gradients and are evaluated on the recovered target-node feature vectors using RNMSE. TabLeak keeps its tabular reconstruction objective, but is applied to graph model updates so that it can be compared with the graph attacks on the same metric.

GraphDLG was originally designed for graph classification, where each observed gradient corresponds to a whole input graph. To compare it on Facebook node classification, we adapt it by constructing a 2-hop egonet around the attacked node minibatch and treating this egonet as the private graph. Auxiliary egonets sampled from the training split are used to train the adjacency decoder and the graph embedding prior. We then evaluate recovery on the attacked target node features using RNMSE.

As shown in Section 5.1.5, GRAIN cannot handle noise and fails under this configuration. DLG gives extremely large and unstable errors. TabLeak, GLG, and the adapted GraphDLG produce finite errors across all three training topologies. Within this adapted comparison, GraphDLG achieves the lowest RNMSE across all three training topologies. This makes it the most effective attack in our adapted setup, but it should not be read as a general ranking, since TabLeak and DLG are not designed for graph node features and each attack relies on different prior knowledge and reconstruction procedures.

Attack	**Central**	**Skip-Ring**	**Muffliato**
DLG	1.6e11 $\pm$ 1.5e12	4.6e14 $\pm$ 4.5e15	5.8e13 $\pm$ 5.4e14
TabLeak	3.2 ± 1.6	3.6 ± 2.0	3.3 ± 1.5
GRAIN	failed	failed	failed
GLG	1.9 ± 0.7	2.2 ± 1.0	2.1 ± 0.7
GraphDLG	$\mathbf{1.3 \pm 0.03}$	$\mathbf{1.3 \pm 0.02}$	$\mathbf{1.3 \pm 0.05}$

Table 5.2: RNMSE reconstruction performance across attacks on the Facebook dataset using BoW encoding and the GCN model with $\sigma = 1$. All methods are evaluated under identical training settings. Bold marks the strongest attack, that is the lowest RNMSE in each column.

5.2 Effect of Feature Encoding on Gradient Leakage

Feature encoding changes reconstruction behavior, but not in the same direction for every attack, as shown in Table 5.3. For TabLeak on the Adult dataset, reconstruction accuracy is highest with one-hot encoding, which is also the default representation. All alternative encodings reduce attack success, and randomized binary encoding gives the strongest protection. Across all three topologies, it lowers reconstruction accuracy from about 46% with one-hot encoding to about 33%, corresponding to a reduction of roughly 13 percentage points.

For GLG on the Cornell dataset, the pattern is reversed. One-hot encoding yields a much higher RNMSE than the other encodings, substantially weakening the attack and effectively rendering reconstruction unstable. In contrast, the remaining encodings yield markedly lower reconstruction error and therefore leak more information under GLG. This advantage is tied to the feature dimension, since under one-hot encoding $\|x_v\|_2 = 1$ and RNMSE $= \|x_v - \hat{x}_v\|_2$ reduces to the raw Euclidean error in a d-dimensional ambient space, so datasets with many categorical features force the attacker to recover a sparse binary target from a high-dimensional continuous estimate. The other encodings compress the same information into a far lower-dimensional representation with $\|x_v\|_2 \gg 1$, which both shrinks the attacker's search space and rescales the error by a larger denominator.

Overall, randomized binary encoding provides the strongest defense against TabLeak, while one-hot encoding is far more effective against GLG. These results show that the privacy effect of feature encoding depends strongly on the choice of attack.

Encoding	**Central**	**Skip-Ring**	**Muffliato**
TabLeak (Accuracy %) ↑			
One-hot	46.3 ± 3.9	46.2 ± 4.0	46.4 ± 3.9
Binary	36.3 ± 3.5	36.3 ± 3.2	36.6 ± 3.3
Randomized binary	$\mathbf{33.0 \pm 3.4}$	$\mathbf{32.7 \pm 3.3}$	$\mathbf{32.6 \pm 3.2}$
Hashing	35.9 ± 3.5	35.9 ± 3.0	35.9 ± 2.9
GLG (RNMSE) ↓			
Bag-of-words	4.3 ± 1.2	4.2 ± 1.0	4.2 ± 1.2
Binary	3.3 ± 1.0	3.3 ± 1.1	3.1 ± 1.1
Randomized binary	2.6 ± 0.8	2.5 ± 0.7	2.4 ± 0.7
Hashing	8.7 ± 1.7	8.8 ± 1.7	8.2 ± 1.5
One-hot	$\mathbf{38.4 \pm 3.7}$	$\mathbf{37.4 \pm 3.8}$	$\mathbf{37.1 \pm 3.7}$

Table 5.3: Effect of feature encoding on reconstruction performance when $\sigma = 1$ for TabLeak on the Adult dataset using the FC NN model and for GLG on the Cornell dataset using the GCN model. TabLeak reports reconstruction accuracy using Eq. (3.2), whereas GLG reports RNMSE using Eq. (3.5). Arrows indicate whether higher or lower metric values correspond to stronger reconstruction by the attacker.

5.2.1 Effect of Feature Encoding at Different Noise Levels

Table 5.4 extends the analysis presented at $\sigma = 1$ by showing how feature encoding affects reconstruction performance at $\sigma = 0.1$ and $\sigma = 0.01$.

For GLG, one-hot encoding yields the highest RNMSE across all evaluated topologies, indicating the strongest resistance under this attack.

For TabLeak, the effect of encoding becomes clearer as the noise scale decreases. At $\sigma = 1$, the differences between encodings are moderate, whereas at $\sigma = 0.1$ and $\sigma = 0.01$, it becomes evident that the default one-hot encoding results in the highest reconstruction accuracy. In contrast, randomized binary encoding consistently achieves the lowest TabLeak accuracy across all noise levels and training topologies.

Across most settings, smaller noise scales increase reconstruction performance for both attacks, indicating that higher noise provides stronger protection against gradient leakage.

	$\sigma = 0.1$			$\sigma = 0.01$		
Encoding	**Central**	**Skip-Ring**	**Muffliato**	**Central**	**Skip-Ring**	**Muffliato**
TabLeak (Accuracy %) ↑						
One-hot	48.5 ± 4.0	47.1 ± 4.1	45.8 ± 4.2	63.0 ± 4.3	59.9 ± 4.4	48.6 ± 3.3
Binary	37.8 ± 3.1	37.9 ± 3.2	36.7 ± 3.4	40.6 ± 2.9	40.2 ± 3.2	38.2 ± 3.2
Randomized binary	$\mathbf{34.7 \pm 3.2}$	$\mathbf{33.7 \pm 3.2}$	$\mathbf{32.3 \pm 3.1}$	$\mathbf{38.5 \pm 3.4}$	$\mathbf{39.2 \pm 3.8}$	$\mathbf{34.0 \pm 3.1}$
Hashing	38.1 ± 3.4	37.0 ± 3.6	35.9 ± 3.3	48.9 ± 4.8	48.2 ± 5.2	38.5 ± 3.3
GLG (RNMSE) ↓						
Bag-of-words	4.3 ± 1.2	4.2 ± 1.0	4.2 ± 1.0	4.2 ± 1.1	4.0 ± 0.8	4.1 ± 1.0
Binary	3.4 ± 0.9	3.2 ± 1.1	3.1 ± 1.0	3.3 ± 1.2	3.1 ± 1.4	3.0 ± 1.1
Randomized binary	2.6 ± 0.8	2.4 ± 0.6	2.5 ± 0.7	2.6 ± 0.8	2.3 ± 0.8	2.6 ± 0.8
Hashing	8.8 ± 1.8	8.3 ± 1.7	8.2 ± 1.5	8.5 ± 1.8	7.2 ± 2.6	7.6 ± 2.5
One-hot	$\mathbf{37.2 \pm 3.4}$	$\mathbf{37.0 \pm 3.4}$	$\mathbf{37.5 \pm 3.8}$	$\mathbf{37.5 \pm 3.5}$	$\mathbf{35.8 \pm 3.3}$	$\mathbf{36.4 \pm 4.3}$

Table 5.4: Effect of feature encoding on reconstruction performance for $\sigma \in \{0.1, 0.01\}$ for TabLeak on the Adult dataset using the FC NN model and for GLG on the Cornell dataset using the GCN model. TabLeak reports reconstruction accuracy using Eq. (3.2), whereas GLG reports RNMSE using Eq. (3.5). Arrows indicate whether higher or lower metric values correspond to stronger reconstruction by the attacker.

5.2.2 GLG across Datasets and Encodings

Tables 5.5 and 5.6 report GLG reconstruction results for the DBLP [31] and Cora [32] datasets using $\sigma = 1$, with 50 trials per setting.

Across both datasets and all training topologies, one-hot encoding consistently yields substantially higher RNMSE than the other representations. In contrast, binary and randomized binary encodings yield the lowest RNMSE values. Hashing and BoW produce moderate reconstruction errors compared to the other encodings.

This pattern is consistent across both DBLP and Cora, as well as across Central, Skip-Ring, and Muffliato training topologies.

Encoding	**Central**	**Skip-Ring**	**Muffliato**
Bag-of-words	19.0 ± 4.8	18.3 ± 4.7	19.6 ± 5.7
Binary	4.1 ± 1.5	4.2 ± 1.4	4.4 ± 1.3
Randomized binary	3.6 ± 1.0	3.5 ± 0.8	3.8 ± 1.0
Hashing	12.0 ± 1.8	11.7 ± 1.8	12.7 ± 1.7
One-hot	$\mathbf{41.0 \pm 2.4}$	$\mathbf{41.7 \pm 2.6}$	$\mathbf{41.4 \pm 3.0}$

Table 5.5: Effect of feature encoding on GLG reconstruction performance on the DBLP dataset using the GCN model, with $\sigma = 1$.

Encoding	**Central**	**Skip-Ring**	**Muffliato**
Bag-of-words	10.0 ± 3.9	10.8 ± 4.8	9.9 ± 3.4
Binary	5.0 ± 1.7	4.8 ± 1.3	5.5 ± 1.7
Randomized binary	3.9 ± 0.9	3.8 ± 1.2	4.1 ± 1.0
Hashing	11.1 ± 1.5	11.6 ± 1.4	12.7 ± 2.2
One-hot	$\mathbf{37.8 \pm 2.4}$	$\mathbf{37.9 \pm 2.6}$	$\mathbf{37.6 \pm 1.7}$

Table 5.6: Effect of feature encoding on GLG reconstruction performance on the Cora dataset using the GCN model, with $\sigma = 1$.

5.3 Other Factors Affecting Reconstruction Performance

5.3.1 Effect of Batch Size

Figure 5.7 shows that reconstruction accuracy decreases with increasing batch size only at $\sigma = 0.01$. For $\sigma = 1.0$, reconstruction accuracy is already close to the random baseline, and increasing the batch size provides little additional protection. Since [4] evaluated the effect of batch size without noise, our results clarify that the reduction in leakage from larger batches primarily arises when noise levels are low or zero.

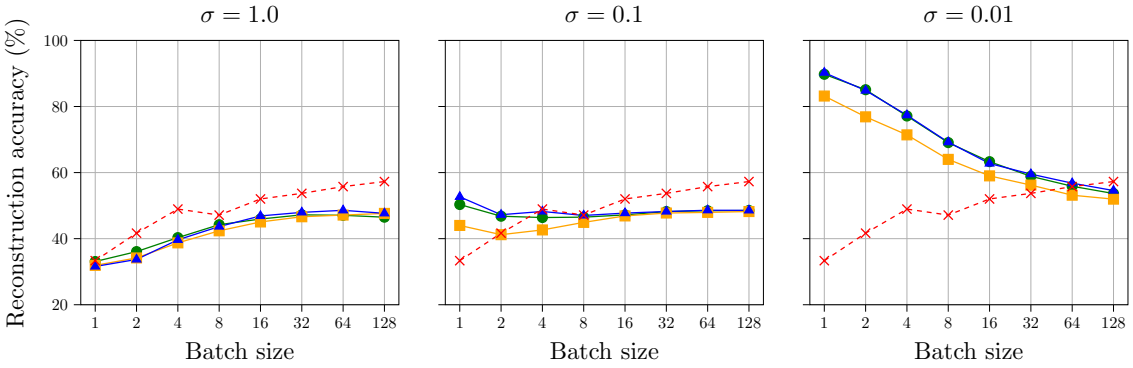

Figure 5.7: TabLeak reconstruction accuracy on the Adult dataset using one-hot encoding and the FC NN model, shown as a function of batch size for noise scales $\sigma \in \{1, 0.1, 0.01\}$.

5.3.2 Impact of Skipping Probability

For the larger skipping probabilities, the centralized learning rate is adjusted to maintain stable model accuracy, as shown in Table 5.1.

Figure 5.8 shows reconstruction accuracy as a function of ε for different skipping probabilities p. Overall, the curves largely overlap across $p \in \{10^{-4}, 1/2, 7/10\}$, indicating that the skipping probability has only a minor effect on reconstruction performance compared to the effective privacy level ε.

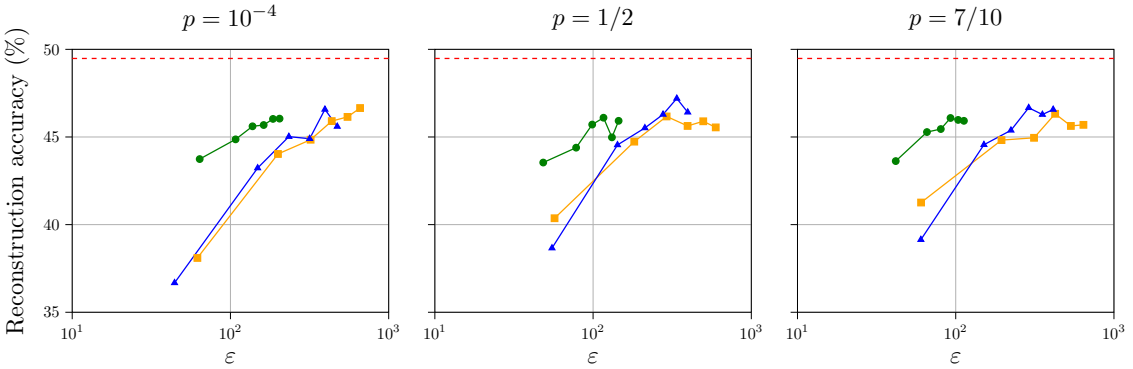

Figure 5.8: TabLeak reconstruction accuracy on the Adult dataset using one-hot encoding and the FC NN model, shown as a function of ε for skipping values $p \in \{10^{-4}, 1/2, 7/10\}$ and $\sigma = 1$.

5.3.3 Impact of Model Architecture

Figure 5.9 shows that reconstruction performance varies across model architectures. Consistent with the observations in [4], linear models are harder to attack, as reconstruction accuracy remains close to the random baseline across all values of ε.

Fully connected networks, both FC NN and FC NN large, achieve higher reconstruction accuracy, which increases as ε increases. The FC NN large is especially vulnerable, suggesting that greater model capacity and nonlinearity make gradient-based reconstruction easier. CNNs exhibit a similar trend, achieving higher reconstruction accuracy as privacy budgets increase.

ResNet, in contrast, remains relatively robust, with reconstruction accuracy staying near the baseline even at higher ε.

Overall, model architecture affects gradient leakage, with linear and residual models more robust and larger nonlinear networks more easily reconstructed.

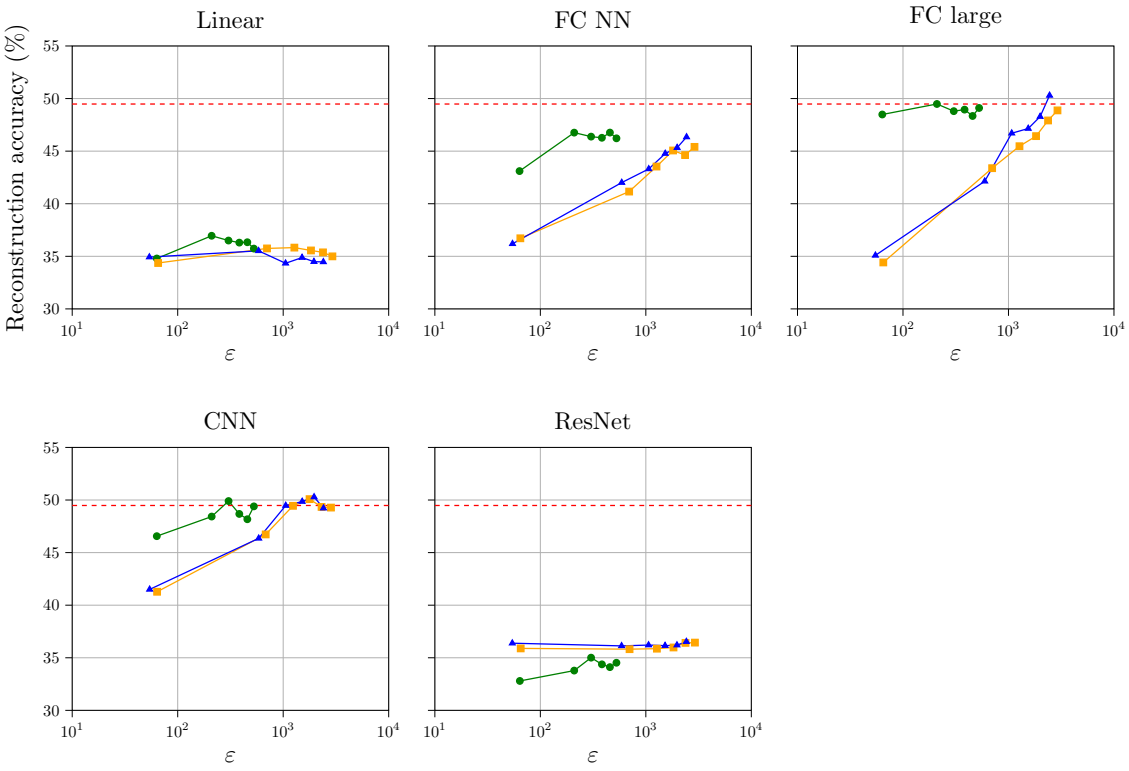

Figure 5.9: TabLeak reconstruction accuracy on the Adult dataset using one-hot encoding, shown as a function of ε for different model architectures with $\sigma = 1$.

5.3.4 Effect of Training Epoch

Figure 5.10 shows reconstruction accuracy as a function of the number of training epochs for different noise levels. We include more noise scales here than in the other experiments because the epoch effect only emerges at low noise, and the additional smaller values of σ are needed to capture the full trend.

The effect of training epochs depends strongly on the noise level. For $\sigma = 1$, reconstruction accuracy stays below the random baseline across training, with only small changes over epochs. For $\sigma = 0.1$, the curves remain close to the baseline and show no clear epoch effect. At $\sigma = 0.01$, reconstruction accuracy is above the baseline but remains nearly flat.

A clearer epoch effect appears only at lower noise levels. For $\sigma = 0.001$ and below, reconstruction accuracy starts high and decreases over training, with the largest drop occurring in the first few epochs. This is closest to the behavior observed by [4], who studied training epochs without privacy noise.

These results suggest that training epoch effects are visible when gradients retain enough signal for the attack to exploit. Once Gaussian noise is large enough to suppress reconstruction, additional training does not substantially change the attack outcome.

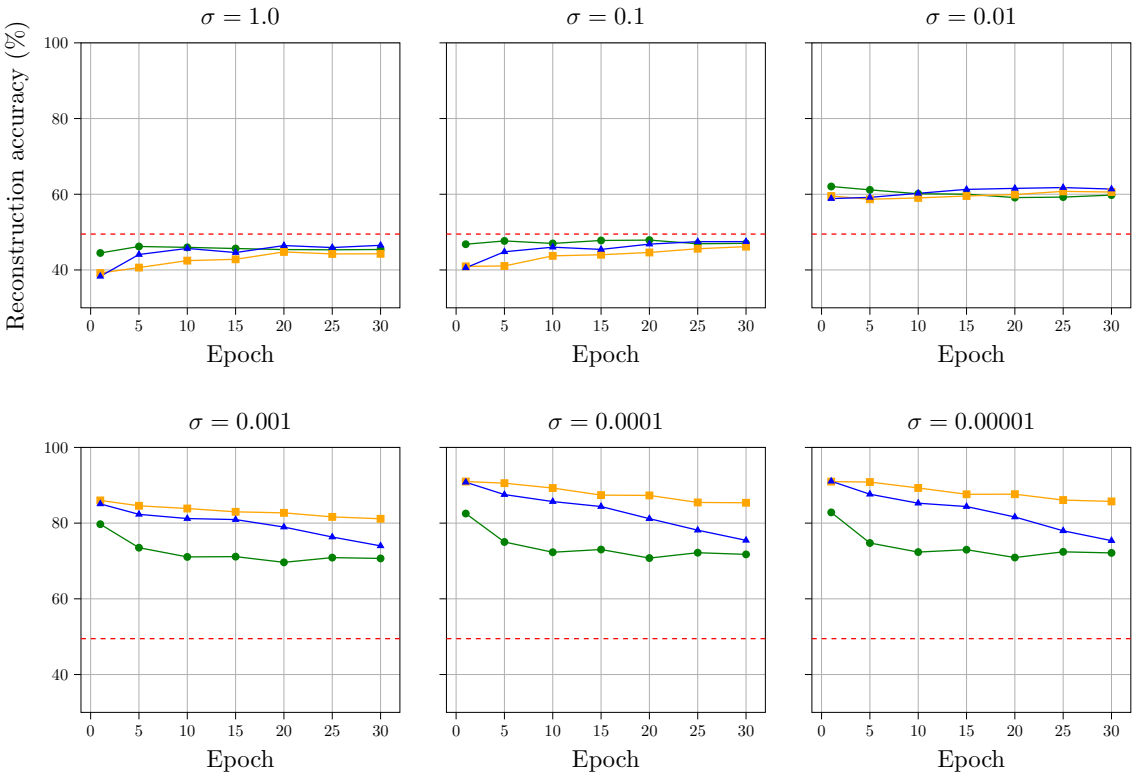

Figure 5.10: TabLeak reconstruction accuracy with one-hot encoding as a function of training epochs on the Adult dataset using the FC NN model, for noise scales $\sigma \in \{1, 0.1, 0.01, 0.001, 0.0001, 0.00001\}$.

5.3.5 TabLeak Reconstruction across Datasets

Figure 5.11 shows a privacy–reconstruction trade-off for most datasets. For Adult, Health Heritage, and Law School, reconstruction accuracy increases with ε, indicating stronger leakage as the effective privacy loss increases.

Insurance behaves differently. Its reconstruction accuracy remains nearly flat across ε, with only minor variation. Since all datasets are trained under the same configuration, this suggests that the weaker dependence on ε is due to dataset properties rather than the training setup. In particular, Insurance is by far the smallest dataset used here (Table A.1), so its gradients carry less information for the attack to exploit, which is a likely reason for the weaker dependence on ε.

Overall, the privacy–reconstruction trade-off appears across most datasets, but its strength varies, with Insurance showing a much weaker effect.

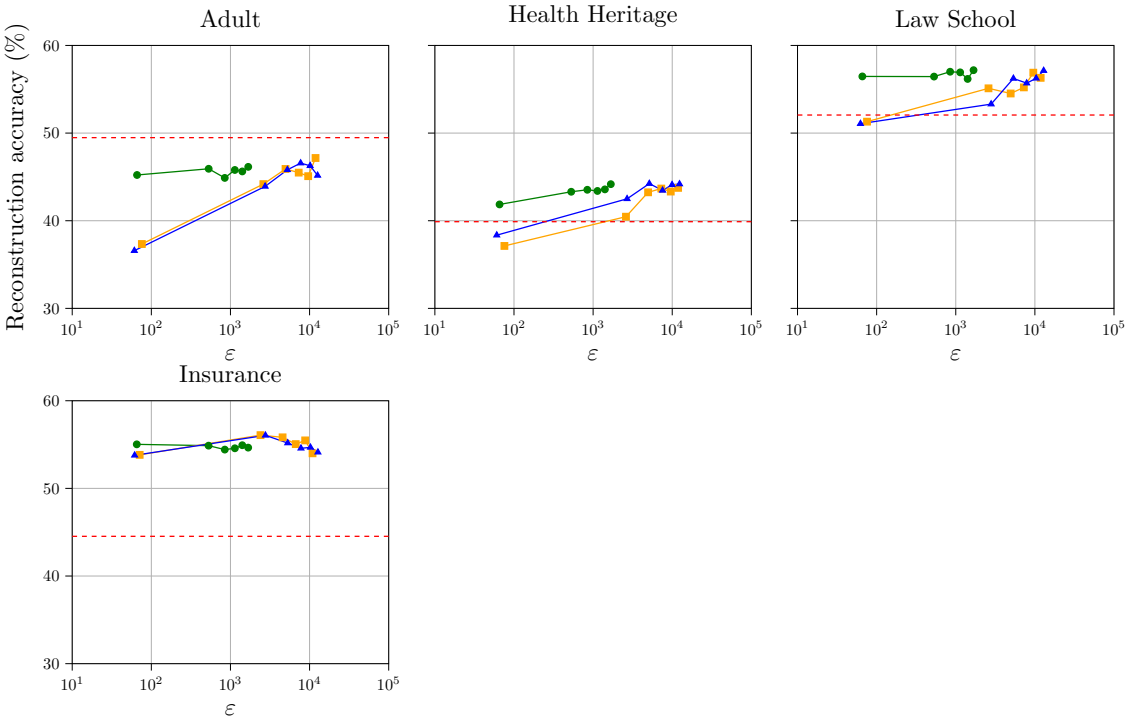

Figure 5.11: TabLeak reconstruction accuracy across datasets using one-hot encoding and the FC NN model, with $\sigma = 1$.

5.4 Model Accuracy

We evaluate model accuracy across different Gaussian noise scales σ and feature encoding schemes. This analysis examines how the noise scale and the choice of encoding affect predictive performance.

5.4.1 Impact of Noise on Model Accuracy

Tables 5.7, 5.8, and 5.9 report the model accuracy of the models used in the TabLeak, GLG, and GraphDLG experiments under different noise scales. For TabLeak, results are obtained using the FC NN model on the Adult dataset, following the original evaluation setup.

Overall, model accuracy remains largely stable across the evaluated noise levels. In the TabLeak setting, the centralized model shows essentially unchanged accuracy, while Skip-Ring and Muffliato exhibit only a small decrease at $\sigma = 1$ compared with $\sigma \in \{0.1, 0.01\}$.

In the GLG setting on the Facebook dataset with BoW encoding, model accuracy decreases as σ increases. Centralized training achieves the highest accuracy, Muffliato is slightly lower, and Skip-Ring is consistently lower still, indicating a higher utility cost in this GLG experiment. Some of the overall utility gap may also reflect the use of sigmoid activation, which was chosen to enable comparison with DLG.

For GraphDLG on PROTEINS, model accuracy is lower than in the TabLeak and GLG settings. The three topologies are close to each other, and the standard deviations overlap. Accuracy is slightly lower at $\sigma = 1$ than at the two smaller noise scales.

Noise	Central	Skip-Ring	Muffliato
$\sigma = 1$	82.7 ± 0.4	82.8 ± 0.2	$\mathbf{83.2 \pm 0.2}$
$\sigma = 0.1$	82.7 ± 0.4	$\mathbf{84.0 \pm 0.1}$	83.4 ± 0.2
$\sigma = 0.01$	82.7 ± 0.4	$\mathbf{84.0 \pm 0.1}$	83.4 ± 0.2

Table 5.7: Model accuracy (%) under different Gaussian noise scales σ on the Adult dataset using one-hot encoding and the FC NN model.

Noise	Central	Skip-Ring	Muffliato
$\sigma = 1$	$\mathbf{83.5 \pm 0.76}$	74.3 ± 1.78	81.0 ± 0.87
$\sigma = 0.1$	$\mathbf{86.4 \pm 0.33}$	79.5 ± 1.25	82.9 ± 0.67
$\sigma = 0.01$	$\mathbf{86.4 \pm 0.34}$	79.6 ± 1.20	82.9 ± 0.64

Table 5.8: Model accuracy (%) under different Gaussian noise scales σ on the Facebook dataset using BoW encoding and the GCN model.

Noise	Central	Skip-Ring	Muffliato
$\sigma = 1$	57.6 ± 6.64	57.4 ± 6.53	$\mathbf{58.6 \pm 5.58}$
$\sigma = 0.1$	61.3 ± 3.97	$\mathbf{61.4 \pm 3.76}$	61.0 ± 3.44
$\sigma = 0.01$	60.5 ± 4.35	$\mathbf{61.1 \pm 3.56}$	60.6 ± 3.47

Table 5.9: Model accuracy (%) under different Gaussian noise scales σ on the PROTEINS dataset using one-hot encoding and the GCN model used by GraphDLG.

5.4.2 TabLeak: Feature Encoding and Predictive Performance

Table 5.10 reports results for the main experimental setting with $\sigma = 1$. As shown, most feature encoding schemes maintain similar model accuracy. Binary and randomized binary encodings preserve predictive performance across all topologies, whereas hashing consistently results in a slight reduction in performance. This indicates that binary-based encodings improve privacy while maintaining comparable predictive accuracy.

Encoding	Central	Skip-Ring	Muffliato
One-hot	82.7 ± 0.4	82.8 ± 0.2	$\mathbf{83.2 \pm 0.2}$
Binary	82.7 ± 0.2	$\mathbf{83.1 \pm 0.2}$	$\mathbf{83.1 \pm 0.2}$
Randomized binary	82.7 ± 0.3	83.1 ± 0.2	$\mathbf{83.2 \pm 0.2}$
Hashing	82.5 ± 0.3	82.5 ± 0.3	$\mathbf{83.0 \pm 0.2}$

Table 5.10: Model accuracy (%) for the TabLeak feature encoding experiment on the Adult dataset using the FC NN model with $\sigma = 1$. Bold values indicate the highest mean accuracy for each encoding.

5.4.3 GLG: Feature Encoding and Predictive Performance

Table 5.11 reports results for the main experimental setting with $\sigma = 1$. Model accuracy under GLG varies across encoding schemes and training topologies. BoW achieves the highest average accuracy in the centralized setting but shows larger variability across runs. Binary-based encodings provide more consistent performance, though with slightly lower average accuracy. Hashing yields the most consistent accuracy across topologies. One-hot performs similarly in the centralized and Muffliato settings, but drops more under Skip-Ring. This suggests that the utility effect of feature encoding depends on the training topology.

Encoding	Central	Skip-Ring	Muffliato
Bag-of-words	$\mathbf{55.9 \pm 3.5}$	50.6 ± 8.2	53.2 ± 2.8
Binary	$\mathbf{53.6 \pm 1.1}$	$\mathbf{53.6 \pm 1.2}$	53.4 ± 1.3
Randomized binary	51.7 ± 2.0	$\mathbf{52.6 \pm 2.0}$	52.5 ± 2.1
Hashing	$\mathbf{54.0 \pm 0.3}$	$\mathbf{54.0 \pm 0.5}$	53.9 ± 0.7
One-hot	$\mathbf{54.1 \pm 0.3}$	50.3 ± 6.2	53.1 ± 2.7

Table 5.11: Model accuracy (%) for the GLG feature encoding experiment on the Cornell (WebKB) dataset with $\sigma = 1$, GCN model. Bold values denote the highest mean accuracy among topologies for each encoding.

Chapter 6

Discussion

6.1 Privacy Level Alone Does Not Fully Determine Reconstruction Risk

In general, higher noise reduces leakage, making TabLeak much less effective and causing DLG to fail at larger noise scales. GLG and GraphDLG follow the same direction, but with a weaker and more variable dependence. The same noise that limits reconstruction also affects model utility. Turning to model accuracy, we see a different pattern. In the setting used for the TabLeak experiments (Adult, FC NN), accuracy stays largely stable across the evaluated noise levels. In the GLG setting (Facebook, GCN with BoW encoding), accuracy decreases under higher noise, and the decrease is largest under Skip-Ring. In the GraphDLG setting (PROTEINS, GCN with one-hot encoding), accuracy is lowest at the highest noise level. The differences among topologies are small relative to the reported standard deviations (Section 5.4).

Reconstruction risk also depends on the attack, model architecture, dataset, and what the adversary observes. This is reflected in the contrast between TabLeak and GLG. TabLeak is strongly affected by the noise scale σ, whereas GLG shows a weaker and less consistent dependence, with smaller differences between σ levels and greater variability across runs. We also find that linear models are harder to reconstruct, and that the Insurance dataset shows only limited variation in ε, likely because its small size limits the information available in the gradients.

This interpretation is consistent with [33], who show that the same DP guarantee ε can correspond to different reconstruction success rates depending on the training configuration and the adversary's observations. This supports our view that while smaller ε generally improves privacy, reconstruction risk must still be evaluated in context.

6.2 Feature Encoding Influences Gradient Leakage

Our results show that feature encoding affects gradient leakage, but the effect varies by attack. In the tabular setting, every alternative to one-hot reduces TabLeak reconstruction accuracy, with randomized binary giving the strongest protection. In the graph setting, the effect is also large. For GLG, one-hot encoding consistently yields much higher RNMSE than any other encoding across noise levels and datasets. Feature encoding can therefore change the amount of information exposed through gradients by a wide margin.

These results do not imply that any single encoding is a universal defense against gradient leakage. The privacy benefit remains attack-dependent, and a defender cannot assume in advance which reconstruction method an adversary will use. An encoding that weakens one known attack may be less effective against another, including future attacks not considered here. Feature encoding should therefore be treated as one component of the overall privacy design rather than as a standalone protection mechanism.

Encoding is also practically important because it can improve privacy without necessarily causing large utility losses. In our tabular experiments, binary and randomized binary encodings maintain predictive performance close to that of one-hot encoding. In the graph setting, the privacy–utility trade-off is more mixed.

Feature encoding should therefore be included in privacy evaluations alongside the noise scale, and its effect read in the context of the specific attack.

6.3 Effects Observed without Noise Do Not Always Carry Over to Noisy Training

Another pattern in our results is that some effects often discussed in gradient leakage research become much weaker once training is performed with Gaussian noise. Several earlier observations in the leakage literature were established in settings without privacy noise. For example, DLG by [2] shows that larger batch sizes make reconstruction harder, and TabLeak by [4] studies both the effect of batch size and the effect of training over multiple epochs. However, our results suggest that these effects do not always carry over in the same way once gradients are perturbed throughout training.

This is visible in the TabLeak experiments. Figure 5.7 shows that increasing the batch size reduces reconstruction accuracy only when the noise level is low. At larger noise scales, such as $\sigma = 1$, reconstruction accuracy is already close to the random baseline, and increasing the batch size gives little additional protection. This suggests that the privacy benefit of larger batches depends on the gradients retaining sufficient useful structure for the attack to exploit. Once the gradients have been strongly distorted by noise, the additional averaging from a larger batch appears to matter much less.

A related pattern is observed for the number of training epochs. Figure 5.10 shows that epoch effects are weak when the noise level is high. At $\sigma = 1$ and $\sigma = 0.1$, reconstruction accuracy stays near or below the random baseline and changes little over training. At lower noise levels, especially $\sigma = 0.001$ and below, reconstruction accuracy decreases over epochs. This suggests that the epoch effect mainly appears

when the gradients still contain enough usable signal for the attack.

Taken together, these results suggest that noisy training changes how some commonly discussed factors influence leakage. When the noise level is high, the gradients may already contain too little usable information for longer training or larger batch sizes to have much additional effect. In lower noise regimes, these factors can still matter, but their role is no longer the same as in settings without privacy noise.

Observations made in standard training should not automatically be assumed to hold under privacy-preserving noisy training. Our results do not contradict earlier work, but they suggest that conclusions about leakage should be re-evaluated when noise is present throughout training.

6.4 Limitations

Our analysis assumes a single-node adversary that observes only the updates available to a single node in the network, consistent with an NDP setting. We do not consider adversaries that collude across multiple nodes or aggregate observations across the topology. Extending the analysis to such threat models is left for future work.

As noted in Section 4.2.4, we clip and add noise to the batch-average gradient rather than clipping each per-example gradient, but the comparison across the three topologies remains fair because the same scheme is used in all of them.

A further effect is privacy amplification by subsampling. Because each update uses a minibatch smaller than the full local dataset, random sampling amplifies privacy and would lower the reported ε. Our accountants do not include this, so the reported ε is an upper bound.

We also restrict our experiments to noise scales $\sigma \leq 1$ because higher values caused a marked drop in model accuracy.

6.5 Future Work

Future work could test additional defenses under the same decentralized training setup used in this thesis. The DLG paper evaluates Gaussian noise and Laplacian noise as well as low precision and gradient pruning [2]. Gradient pruning is especially relevant because it changes which parts of the update are visible rather than only changing their scale. Testing such defenses with TabLeak and graph-based attacks would show how far the results extend beyond Gaussian noise.

A second direction is to study cryptographic defenses. Secure aggregation [34] and additively homomorphic encryption [35] aim to prevent raw gradients from being exposed. This is different from perturbing or pruning gradients after they are computed. Future work could examine whether these methods fit decentralized protocols in which there is no single trusted server.

A final direction is to vary the communication graph more systematically. The experiments in this thesis already show that topology affects reconstruction risk. Larger networks and other gossip graphs could test the stability of this effect. For graph data, this should be paired with attacks that remain stable in the presence of privacy noise.

Chapter 7

Conclusion

This thesis studied gradient leakage in decentralized learning under DP. We compared centralized training, Muffliato, and Skip-Ring, and evaluated TabLeak, DLG, GRAIN, GLG, and GraphDLG on tabular and graph data. The results show a trade-off between privacy and reconstruction, but the relation is not uniform across attacks. In general, a higher ε leads to greater leakage. GRAIN breaks down under extremely small noise, whereas GLG remains much more stable at the noise levels we tested. GraphDLG moves close to the random baseline at $\sigma = 1$. DLG also fails to recover meaningful graph data under noisy training. For TabLeak, stronger noise reduces leakage, and larger batch sizes are most effective when the noise is low. We also found that the choice of feature encoding matters. Randomized binary encoding gives the best protection against TabLeak, while one-hot encoding is much more effective against GLG.

Overall, our results show that gradient leakage in decentralized learning depends on more than just the DP privacy level. Training topology, attack type, feature encoding, and batch size all influence the amount of information that can be recovered from gradients. The results indicate that the DP privacy level alone is insufficient to characterize reconstruction risk. The relevant training, data, and attack conditions should also be reported.

Appendix A

Datasets

This thesis evaluates gradient reconstruction attacks on both tabular and graph-structured datasets. Tabular datasets are used in the TabLeak experiments, whereas graph datasets are used to evaluate graph reconstruction attacks such as GLG, DLG, GRAIN, and GraphDLG.

Table A.1 summarizes the datasets used throughout the experiments.

Dataset	Type	Data points	Features	Task
Adult	Tabular	45,222	14	Binary classification
Health Heritage	Tabular	218,415	17	Binary classification
Law School	Tabular	96,584	7	Binary classification
Insurance	Tabular	1,338	6	Binary classification
WebKB (Cornell)	Graph	183	1,703	Node classification
CiteSeer	Graph	3,327	3,703	Node classification
Cora	Graph	2,708	1,433	Node classification
DBLP	Graph	17,716	1,639	Node classification
Facebook	Graph	22,470	128	Node classification
PROTEINS	Graph	1,113	3	Graph classification

Table A.1: Summary of datasets used in the experiments.

For tabular datasets, the number of features is the dimensionality of one row; for graph datasets, it is the dimensionality of each node-feature vector.

Appendix B

Statement on the Use of AI Tools

Several AI tools were used during the writing process, limited to language polishing, technical support, and feedback on the written work. GPT and Grammarly were used to polish grammar, including spelling, wording, and sentence clarity. Claude and GPT were also used as support tools when debugging Python code and LaTeX code. In addition, paperreview.ai was used to help identify possible weaknesses and missing elements in my written work. All AI-assisted suggestions were reviewed by me before being incorporated into the thesis.

I am aware that I am responsible for all content of this master's thesis.

Appendix C

Code Repository

The code used in these experiments is available at GitHub: https://github.com/Simula-UiB/Master-Thesis_Jonathan.

Bibliography

[1] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Found. Trends Theor. Comput. Sci.*, vol. 9, pp. 211–407, Aug. 2014.

[2] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, (Vancouver, BC, Canada), pp. 14747–14756, Dec. 2019.

[3] B. Zhao, K. R. Mopuri, and H. Bilen, "iDLG: Improved deep leakage from gradients," Jan. 2020. arXiv:2001.02610.

[4] M. Vero, M. Balunović, D. I. Dimitrov, and M. Vechev, "TabLeak: Tabular data leakage in federated learning," in *Proc. 40th Int. Conf. Mach. Learn. (ICML)*, (Honolulu, HI, USA), pp. 35051–35083, July 2023.

[5] M. Drencheva, I. Petrov, M. Baader, D. I. Dimitrov, and M. Vechev, "GRAIN: Exact graph reconstruction from gradients," in *Proc. 13th Int. Conf. Learn. Represent. (ICLR)*, (Singapore), Apr. 2025.

[6] D. A. Sinha, Y. Liu, R. Du, A. Markopoulou, and Y. Shen, "Gradient inversion attack on graph neural networks," *Trans. Mach. Learn. Res.*, vol. 2025, June 2025.

[7] S. Wei, W. Chen, T. Wei, C. Gong, Y. Tong, and L. Cui, "GraphDLG: Exploring deep leakage from gradients in federated graph learning," Jan. 2026. arXiv:2601.19745.

[8] S. J. D. Prince, *Understanding Deep Learning*. Cambridge, MA, USA: The MIT Press, 2023.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, (Las Vegas, NV, USA), pp. 770–778, June 2016.

[10] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, (Toulon, France), Apr. 2017.

[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, (Vancouver, BC, Canada), Apr. 2018.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[13] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Agüera y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Int. Conf. Artif. Intell. Stat. (AISTATS)*, (Fort Lauderdale, FL, USA), pp. 1273–1282, Apr. 2017.

[14] A. Evfimievski, J. Gehrke, and R. Srikant, "Limiting privacy breaches in privacy preserving data mining," in *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst. (PODS)*, (San Diego, CA, USA), pp. 211–222, June 2003.

[15] E. Cyffers and A. Bellet, "Privacy amplification by decentralization," in *Proc. 25th Int. Conf. Artif. Intell. Stat. (AISTATS)*, (Virtual Event), pp. 5334–5353, Mar. 2022.

[16] I. Mironov, "Rényi differential privacy," in *Proc. 30th IEEE Comput. Secur. Found. Symp. (CSF)*, (Santa Barbara, CA, USA), pp. 263–275, Aug. 2017.

[17] E. Cyffers, M. Even, A. Bellet, and L. Massoulié, "Muffliato: Peer-to-peer privacy amplification for decentralized optimization and averaging," in *Proc. 36th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, (New Orleans, LA, USA), pp. 15889–15902, Nov. 2022.

[18] A. G. Dimakis, S. Kar, J. M. F. Moura, M. G. Rabbat, and A. Scaglione, "Gossip algorithms for distributed signal processing," *Proc. IEEE*, vol. 98, pp. 1847–1864, Nov. 2010.

[19] Y. Yakimenka, C.-W. Weng, H.-Y. Lin, E. Rosnes, and J. Kliewer, "Straggler-resilient differentially-private decentralized learning," *IEEE J. Sel. Areas Inf. Theory*, vol. 5, pp. 407–423, May 2024.

[20] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients – how easy is it to break privacy in federated learning?," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, (Virtual Event), pp. 16937–16947, Dec. 2020.

[21] D. Dua and C. Graff, "UCI machine learning repository." University of California, Irvine, School of Information and Computer Sciences, 2017. Accessed 26 May 2026.

[22] A. Goldbloom and B. Hamner, "Heritage health prize." Kaggle competition, 2011. Accessed 26 May 2026.

[23] S. Jain, "Insurance premium data." Kaggle dataset, 2020. Version 1. CC0 Public Domain. Accessed 26 May 2026.

[24] L. F. Wightman, *LSAC National Longitudinal Bar Passage Study.* LSAC Research Report Series, Newtown, PA, USA: Law School Admission Council, 1998.

[25] C. L. Giles, K. D. Bollacker, and S. Lawrence, "CiteSeer: An automatic citation indexing system," in *Proc. 3rd ACM Conf. Digital Libraries (DL)*, (Pittsburgh, PA, USA), pp. 89–98, June 1998.

[26] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. M. Mitchell, K. Nigam, and S. Slattery, "Learning to extract symbolic knowledge from the World Wide Web," in *Proc. 15th Natl. Conf. Artif. Intell. (AAAI) and 10th Innovative Appl. Artif. Intell. Conf. (IAAI)*, (Madison, WI, USA), pp. 509–516, July 1998.

[27] B. Rozemberczki, C. Allen, and R. Sarkar, "Multi-scale attributed node embedding," *J. Complex Netw.*, vol. 9, pp. 1–22, May 2021.

[28] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann, "TUDataset: A collection of benchmark datasets for learning with graphs," in *Proc. ICML 2020 Workshop on Graph Represent. Learn. and Beyond*, (Virtual Event), July 2020. Preprint arXiv:2007.08663.

[29] B. Balle, G. Barthe, and M. Gaboardi, "Privacy amplification by subsampling: Tight analyses via couplings and divergences," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, (Montreal, QC, Canada), pp. 6280–6290, Dec. 2018.

[30] M. Gavish and D. L. Donoho, "The optimal hard threshold for singular values is $4/\sqrt{3}$," *IEEE Trans. Inf. Theory*, vol. 60, pp. 5040–5053, Aug. 2014.

[31] A. Bojchevski and S. Günnemann, "Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, (Vancouver, BC, Canada), Apr. 2018.

[32] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, (New York, NY, USA), pp. 40–48, June 2016.

[33] J. Hayes, B. Balle, and S. Mahloujifar, "Bounding training data reconstruction in DP-SGD," in *Proc. 37th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, (New Orleans, LA, USA), pp. 78696–78722, Dec. 2023.

[34] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for federated learning on user-held data," in *Proc. NIPS 2016 Workshop on Private Multi-Party Mach. Learn.*, (Barcelona, Spain), Dec. 2016. arXiv:1611.04482.

[35] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Trans. Inf. Forensics Secur.*, vol. 13, pp. 1333–1345, May 2018.