Inference Attacks Against Graph Neural Networks

Zhikun Zhang[1]* Min Chen[1]* Michael Backes[1] Yun Shen[2] Yang Zhang[1]

[1]*CISPA Helmholtz Center for Information Security*
[2]*Norton Research Group*

Abstract

Graph is an important data representation ubiquitously existing in the real world. However, analyzing the graph data is computationally difficult due to its non-Euclidean nature. Graph embedding is a powerful tool to solve the graph analytics problem by transforming the graph data into low-dimensional vectors. These vectors could also be shared with third parties to gain additional insights of what is behind the data. While sharing graph embedding is intriguing, the associated privacy risks are unexplored. In this paper, we systematically investigate the information leakage of the graph embedding by mounting three inference attacks. First, we can successfully infer basic graph properties, such as the number of nodes, the number of edges, and graph density, of the target graph with up to 0.89 accuracy. Second, given a subgraph of interest and the graph embedding, we can determine with high confidence that whether the subgraph is contained in the target graph. For instance, we achieve 0.98 attack AUC on the DD dataset. Third, we propose a novel graph reconstruction attack that can reconstruct a graph that has similar graph structural statistics to the target graph. We further propose an effective defense mechanism based on graph embedding perturbation to mitigate the inference attacks without noticeable performance degradation for graph classification tasks.[1]

1 Introduction

Many real-world systems can be represented as graphs, such as social networks [41], financial networks [30], and chemical networks [27]. Because of their non-Euclidean nature, graphs do not present familiar features that are common to other systems, like a coordinate or vector space, making the analysis of graph data challenging. To address this issue, the graph embedding algorithms have been proposed to obtain effective graph data representation that represents graphs concisely in Euclidean space [19, 38, 52]. The core idea of those algorithms is to transform graphs from non-Euclidean space into low dimensional vectors, in which the graph information is implicitly preserved. After the transformation, a plethora of downstream tasks can be efficiently performed, such as node classification [10, 19] and graph classification [60].

Recently, a new family of deep learning models known as graph neural networks (GNNs) has been proposed to obtain the graph embedding and achieved state-of-the-art performance. The core idea of GNNs is to train a deep neural network that aggregates the feature information from neighborhood nodes to obtain *node embedding*. They can be further aggregated to obtain the *graph embedding* for graph classification. Such graph embedding is empirically considered sanitized since the whole graph is compressed to a single vector. In turn, it has been shared with third parties to conduct downstream graph analysis tasks. For example, the graph data owner can generate the graph embeddings locally and upload them to the Embedding Projector service[2] provided by Google to visually explore the properties of the graph embeddings. Despite that sharing graph embeddings for downstream graph analysis tasks is intriguing and practical, the associated security and privacy implications remain unanswered.

Our Contributions. In this paper, we initiate a systematic investigation of the privacy issue of graph embedding by exploring three inference attacks. The first attack is *property inference* attack, which aims to infer the basic properties of the target graph given the graph embedding, such as the number of nodes, the number of edges, the graph density, etc. We then investigate the *subgraph inference* attack. That is, given the graph embedding and a subgraph of interest, the adversary aims to determine whether the subgraph is contained in the target graph. For instance, an adversary can infer whether a specific chemical compound structure is contained in a molecular graph if gaining access to its graph embedding, posing as a direct threat to the intellectual property of the data owner. The challenge of subgraph inference attack is that the formats of the graph embedding (*i.e.* a vector) and the subgraph of interest (*i.e.* a graph) are different and not directly comparable. Finally, we aim to reconstruct a graph that shares similar structural properties (*e.g.* degree distribution, local clustering coefficient, etc.) with the target graph. We call this attack

*Zhikun and Min contributed equally to the paper.
[1]Our code is available at https://github.com/Zhangzhk0819/GNN-Embedding-Leaks.
[2]https://projector.tensorflow.org/

graph reconstruction attack. For instance, if the target graph is a social network, the reconstructed graph would then allow an adversary to gain direct knowledge of sensitive social relationships. In summary, we make the following contributions.

- To launch the property inference attack, we model the attack as a multi-task classification problem, where the attack model can predict all the graph properties of interest simultaneously. We conduct experiments on five real-world graph datasets and three state-of-the-art graph embedding models to validate the effectiveness of our proposed attack. The experimental results show that we can achieve up to 0.89 attack accuracy on the DD dataset.
- We design a novel graph embedding extractor, enabling the subgraph inference attack model to simultaneously learn from both the graph embedding and the subgraph of interest. The experimental results on five datasets and three graph embedding models validate the effectiveness of our attack. For instance, we achieve 0.98 attack AUC on the DD dataset. We further successfully launch two transfer attacks when the sampling method and embedding model architecture for training and testing attack model are different.
- We propose to use the graph auto-encoder paradigm to mount the graph reconstruction attack. Once the graph auto-encoder is trained, its decoder is employed as our attack model. Extensive experiments show that the proposed attack can achieve high similarity in terms of graph isomorphism and macro-level graph statistics such as degree distribution and local clustering coefficient distribution. For instance, the cosine similarity of local clustering coefficient distribution between the target graph and the reconstructed graph can achieve 0.99. The results exemplify the effectiveness of our graph reconstruction attack.
- To mitigate the inference attacks, we further propose a defense mechanism based on graph embedding perturbation. The main idea is to add well-calibrated Laplace noise to the graph embedding before sharing with third parties. We demonstrate through several experiments that our proposed defense can effectively mitigate all the three inference attacks without noticeable performance degradation for graph classification tasks.

2 Preliminaries

2.1 Notations

We denote an undirected, unweighted, and attributed graph by $\mathcal{G} = \langle \mathcal{V}, A, X \rangle$, where $\mathcal{V}$ represents the set of all nodes, A is the adjacency matrix, X is the attributes matrix. We denote the embedding of a node $u \in \mathcal{V}$ as H_u and the whole graph embedding as $H_{\mathcal{G}}$ (see Section 2.2 for details). We summarize the frequently used notations introduced here and in the following sections in Table 5 of Appendix A.

2.2 Graph Neural Network

Many important real-world datasets are in the form of graphs, e.g., social networks [41], financial networks [30], and chemical networks [27]. The classical machine learning architectures and algorithms oftentimes do not perform well with these kinds of data. Most of them were designed to learning from data that can naturally be represented individually (*i.e.* data points) but are less effective in dealing with relational data with more complex structure. To effectively extract useful information from the graph data, a new family of deep learning algorithms, i.e., graph neural networks (GNNs), has been proposed and achieved superior performance in various tasks [1, 10, 28, 53]. GNNs generalize the deep neural network models to graph-structured data and learn representations for graph-structured data by aggregating information from a node's neighbors using neural networks, i.e., learning a model $\mathcal{F} : \mathcal{G} \rightarrow H$. The learned embedding H can be used for different graph analytics tasks - node classification [21, 28] and graph classification [59, 60].

- **Node Classification.** The objective of node classification is to determine the label of nodes in the graph, such as the gender of a user in a social network. GNNs first generate node embeddings H_u, and feed them to a classifier to determine the node labels.
- **Graph Classification.** The objective of graph classification is to determine the label of the whole graph, such as a molecule's solubility or toxicity. In graph classification, one needs to further transform all the node embeddings $H_u, \forall u \in \mathcal{V}$ to a whole graph embedding $H_{\mathcal{G}}$ to determine the label of the whole graph.

2.2.1 Message Passing

Most of the existing GNNs use *message passing* to obtain the *node embedding* H_u. It starts by assigning the node attributes as the node embeddings. Then, every node receives a "message" from its neighbor nodes and aggregates the messages as its intermediate embedding. After K steps, the node embedding aggregates information from its K-hop neighbors. Formally, during each message passing iteration, the node embedding H_u^k of node $u \in \mathcal{V}$ is updated using "message" aggregated from u's graph neighborhood $\mathcal{N}_u$ using a pair of *aggregation operation* Φ and *updating operation* Ψ:

$$H_u^{k+1} = \Psi^k \left(H_u^k, \mathbf{m}_{\mathcal{N}_u}^k \right) = \Psi^k \left(H_u^k, \Phi^k \left(H_v^k, \forall v \in \mathcal{N}_u \right) \right)$$

where $H_u^k \in \mathbb{R}^{n \times d_H}$ is the node embedding of node u after k steps of message passing, $\mathbf{m}_{\mathcal{N}_u}^k$ is the message received from node u's neighborhood $\mathcal{N}_u$, which is calculated by Φ.

Aggregation Operation. Recently, researchers have proposed many practical implementations of Φ. Graph Isomorphism Networks (GIN) [59] uses *sum* operation to aggregate the embeddings of all node $u \in \mathcal{G}_{\mathcal{N}_u}$. Graph SAmple and aggreGatE (SAGE) [21] uses *mean* operation to aggregate all node embeddings of $\mathcal{G}_{\mathcal{N}_u}$ instead of summing them up. The Graph Convolution Networks (GCN) [28] method uses the symmetric normalization, and the Graph Attention Networks (GAT) [53] method uses the attention mechanism to

learn a weight matrix to aggregate the embeddings of all node $u \in \mathcal{G}_{\mathcal{N}_u}$.

Updating Operation. The updating operation Ψ combines the node embeddings from node u and the message from u's neighborhood. The most straightforward updating operation is to calculate the weighted combination [43]. Formally, we denote the basic updating operation as $\Psi_{base} = \sigma(W_{self}H_u + W_{neigh}\mathbf{m}_{\mathcal{N}_u})$, where W_{self} and W_{neigh} are learnable parameters, σ is a non-linear activation function. Another method is to treat the basic updating operation as a building block, and concatenate it with the current embedding [21]. We denote the concatenation-based updating operation as $\Psi_{concat} = \Psi_{base}||H_u$, where $||$ is the concatenation operation. An alternative is to use the weighted average of the basic updating method and the current embedding [39], which is referred as *interpolation-based* updating operation and is formally defined as $\Psi_{inter} = \alpha_1 \circ \Psi_{base} + \alpha_2 \circ H_u$.

2.2.2 Graph Pooling

The *graph pooling operation* Σ aggregates the embeddings of all nodes in the graph to form a whole graph embedding, i.e., $H_{\mathcal{G}} = \Sigma(H_u, \forall u \in \mathcal{G})$.

Global Pooling. The most straightforward approach for graph pooling is to directly aggregate all the node embeddings, which is called global pooling, such as *max pooling* and *mean pooling*. Although simple and efficient, the global pooling operation could lose the graph structural information, leading to unsatisfactory performance [3, 60].

Hierarchical Pooling. To better capture the graph structural information, researchers have proposed many hierarchical pooling methods [3, 60]. The general idea is to aggregate n node embeddings to one graph embedding hierarchically, instead of aggregating them in one step as global pooling. Concretely, we first obtain n node embeddings using message passing modules, and finds m clusters according to the node embeddings, where $1 < m < n$. Next, we treat each cluster as a node with features being the graph embedding of this cluster, then iteratively applying the message passing and clustering operations until there are only one graph embedding.

Formally, in the ℓ-th pooling step, we need to learn a cluster assignment matrix $S^\ell \in \mathbb{R}^{n_\ell \times n_{\ell+1}}$, which provides a soft assignment of each node at layer ℓ to a cluster in the next coarsened layer $\ell+1$. Suppose S^ℓ in layer ℓ has already been computed, we can use the following equations to compute the coarsened adjacency matrix $A^{\ell+1}$ and a new matrix of node embeddings $H^{\ell+1}$:

$$H^{\ell+1} = {S^\ell}^T H^\ell \in \mathbb{R}^{n_{\ell+1} \times d_H}$$
$$A^{\ell+1} = {S^\ell}^T A^\ell S^\ell \in \mathbb{R}^{n_{\ell+1} \times n_{\ell+1}}$$

The main challenge lies in how to learn the cluster assignment matrix S^ℓ. In the following, we introduce two state-of-the-art methods.

- **Differential Pooling [60].** The DiffPool method uses a message passing module to calculate the assignment matrix as $S^\ell = \text{softmax}\left(\text{GNN}(A^\ell, H^\ell)\right)$. In practice, it can be difficult to train the GNN models using only gradient signal from the output layer. To alleviate this issue, DiffPool introduces an auxiliary link prediction objective to each pooling layer, which encodes the intuition that nearby nodes should be pooled together. In addition, DiffPool introduces another objective to each pooling layer that minimizes the entropy of the cluster assignment.
- **MinCut Pooling [3].** The MinCutPool method uses an MLP (multi-layer perceptron) module to compute the assignment matrix as $S^\ell = \text{softmax}\left(\text{MLP}(A^\ell, H^\ell)\right)$. Different from DiffPool, MinCutPool introduces the minimum cut objective to each pooling layer that aims to remove the minimum volume of edges, which is in line with the objective of graph pooling aiming to assign the closely connected nodes into the same cluster.

Implementation of GNN Model. Typically, the *graph-level* GNN models consist of a graph embedding module, which encode the graph into the graph embedding, and a multi-class classifier, which predict the label of the graph using the graph embedding. To train the GNN model, we normally adopt the cross-entropy loss. For graph embedding modules containing hierarchical pooling operations, we need to incorporate additional loss such as minimum cut loss in MinCutPool. After the GNN model is trained, we use the graph embedding module as our embedding generation model in the following parts.

3 Threat Model and Attack Taxonomy

3.1 Motivation

In this paper, we focus on the whole graph embedding $H_{\mathcal{G}}$, which is oftentimes computed on a sensitive graph (*e.g.* biomedical molecular network and social network). Such graph embedding $H_{\mathcal{G}}$ is empirically considered sanitized since the whole graph is compressed to a single vector. In practice, it has been shared with third parties to conduct downstream graph analysis tasks. For example, the graph data owner can calculate the graph embeddings locally and upload them to the Embedding Projector service provided by Google to visually explore the properties of the graph embeddings. Another example is that some companies release their graph embedding systems, together with which they publish some pretrained graph embeddings to facilitate the downstream tasks. These systems including the PyTorch BigGraph[3] system developed by Facebook, DGL-KE[4] system developed by Amazon, and GROVER developed by Tencent[5]. Besides, the graph embeddings can also be shared in the well-known model partitioning paradigm [26, 29]. This paradigm can effectively improve the scalability of inference by allowing the graph data owner to

[3] https://github.com/facebookresearch/PyTorch-BigGraph
[4] https://github.com/awslabs/dgl-ke
[5] https://github.com/tencent-ailab/grover

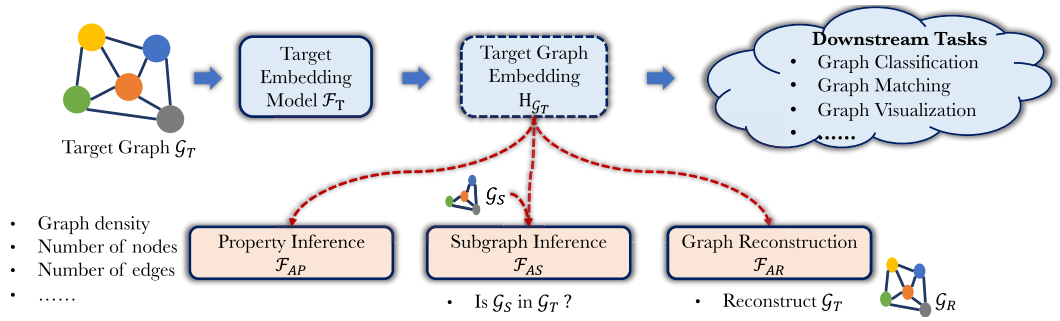

Figure 1: Attack taxonomy of the graph embedding. The adversary obtains the whole graph embedding $H_{\mathcal{G}_T}$ of a sensitive target graph $\mathcal{G}_T$, which is primarily shared to third parties for downstream tasks, and aims to infer sensitive information about $\mathcal{G}_T$: (1) Infer the basic properties of $\mathcal{G}_T$, such as the number of nodes, the number of edges, and graph density ($\mathcal{F}_{AP}$); (2) given a subgraph of interest $\mathcal{G}_S$, infer whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$ ($\mathcal{F}_{AS}$); (3) reconstruct a graph $\mathcal{G}_R$ that is similar with $\mathcal{G}_T$ ($\mathcal{F}_{AR}$).

calculate the graph embeddings locally, and upload them to the cloud for further inference or analysis.

Despite sharing graph embeddings for downstream graph analysis tasks is intriguing and promising, the associated security and privacy implications remain unanswered. For instance, Song *et al.* [47, 49] demonstrated that the embeddings can leak sensitive information about image and text data in the Euclidean space. Recall that the goal of graph embedding $H_{\mathcal{G}}$ is to preserve graph-level similarity, a natural question is: would the graph embedding $H_{\mathcal{G}}$ leak sensitive structural information of its corresponding graph $\mathcal{G}$?

3.2 Threat Model

We consider the scenario where the adversary obtains a whole graph embedding (which is referred to as *target graph embedding* $H_{\mathcal{G}_T}$) from the victim, either from Embedding Projector, pretrained graph embeddings, or model partitioning paradigm. The goal of the adversary is to infer the sensitive information of the graph that is used to generate this graph embedding. We call this graph *target graph* $\mathcal{G}_T$, and the GNN model that used to generate the target graph embedding *target embedding model* $\mathcal{F}_T$. Note that inferring the sensitive information of target graph with "graph embedding" is more challenging than that with "node embeddings" in previous study [11]. From the attacker's perspective, it represents the most difficult setting since the whole graph is compressed to a single vector by the aforementioned pooling methods in Section 2. To train the attack model $\mathcal{F}_A$, we assume the adversary has an auxiliary dataset $\mathcal{D}_{aux}$ that comes from the same distribution of the target graph. This is plausible in practice. For instance, if the target graph embedding is generated from a social network, the adversary can collect social network graphs by themselves through public data API.[6] For molecular networks, the adversary can use the public datasets online.[7] We also show that our attacks is still effective when $\mathcal{D}_{aux}$ comes from different distribution than the target graphs in Section 7. We further assume the adversary only has black-box access to the target embedding model [47, 49], which is the most difficult setting for the adversary [23, 34, 45, 48, 48]. This assumption is plausible when the target embedding model is accessible via public API or freely available online.[8]

3.3 Attack Taxonomy

We formalize three inference attacks that can reveal sensitive information of the target graph given the threat model. An overview of the attack taxonomy is shown in Figure 1.

Property Inference Attack ($\mathcal{F}_{AP}$). Given the target graph embedding $H_{\mathcal{G}_T}$, the attack goal is to infer the basic properties of $\mathcal{G}_T$, such as the number of nodes, the number of edges, the density, etc. Note that the primary goal of GNN is learning information from graphs for downstream tasks, e.g., protein toxicity prediction. Many graph properties, such as node numbers, are not related to the downstream tasks, and successful property inference attacks imply such properties are overlearned [47, 49] by GNNs. These properties can be proprietary when the graph contains valuable information such as molecules. Inferring such properties can directly violate the intellectual property (IP) of the data owner.

Subgraph Inference Attack ($\mathcal{F}_{AS}$). Given the target graph embedding $H_{\mathcal{G}_T}$ and a subgraph of interest $\mathcal{G}_S$, the attack goal is to infer whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$. For instance, an attacker can infer whether a specific chemical compound structure ($\mathcal{G}_S$) is contained in a molecular graph ($\mathcal{G}_T$) if gaining access to its graph embedding ($H_{\mathcal{G}_T}$). Note that we consider the scenario where the subgraph constituting a major part of the target graph. Small graphs, such as triangles or stars, are universal for almost all graphs, hence not taking part in our subgraph inference attack.

Graph Reconstruction Attack ($\mathcal{F}_{AR}$). Given the graph embedding $H_{\mathcal{G}_T}$, the attack goal is to reconstruct a graph $\mathcal{G}_R$ that shares similar graph structural statistics, such as degree distribution and local clustering coefficient, with $\mathcal{G}_T$. Concretely, we aim to reconstruct an adjacency matrix A of $\mathcal{G}_T$. Knowing the high-level structural quantities of the molecular

[6] https://developer.twitter.com/en/docs/twitter-api
[7] https://chrsmrrs.github.io/datasets
[8] http://snap.stanford.edu/gnn-pretrain/

graphs may lead to IP loss of the companies creating them. For instance, the adversary can develop generic drugs with much lower cost than the famous pharmaceutical companies by exploiting the high-level structural quantities of the reconstructed molecular graphs to narrow down the search space.

4 Property Inference Attack

4.1 Attack Overview

Given the target graph embedding $H_{\mathcal{G}_T}$, the goal of the property inference attack is to infer the basic properties of the target graph $\mathcal{G}_T$, such as the number of nodes, the number of edges, and density. Figure 2 illustrates the general attack pipeline of the property inference attack. Our attack model $\mathcal{F}_{AP}$ takes as input the target graph embedding $H_{\mathcal{G}_T}$ and outputs all the interested graph properties of $\mathcal{G}_T$ simultaneously.

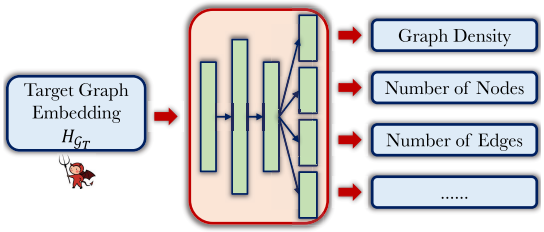

Figure 2: Attack pipeline of the property inference attack. The attack model $\mathcal{F}_{AP}$ is a multi-task classifier, which consists of multiple output layers, each predicts one graph property.

4.2 Attack Model $\mathcal{F}_{AP}$

Model Definition. Formally, the property inference attack $\mathcal{F}_{AP}$ is defined as

$$\mathcal{F}_{AP} : H_{\mathcal{G}_T} \rightarrow \{\text{graph properties}\}$$

Concretely, the attack model consists of a feature extractor $\mathcal{E}$ (multiple sequential linear layers), and multiple parallel prediction layers $\mathcal{M}$, each is responsible for predicting one property. We outline the technical details of building $\mathcal{F}_{AP}$ below.

Training Data. To train the attack model, we need a set of graph embeddings $H_{\mathcal{G}}$ and a set of properties of interest $\mathbb{P}$. As discussed in Section 3, the adversaries have access to an auxiliary dataset $\mathcal{D}_{aux}$ that comes from the same distribution of $\mathcal{G}_T$. The adversaries can obtain the auxiliary graph embedding $H_{\mathcal{G}_{aux}}$ of the auxiliary graph $\mathcal{G}_{aux} \in \mathcal{D}_{aux}$ by querying the target embedding model. Finally, we use the graph properties of $\mathcal{G}_{aux}$ to label $H_{\mathcal{G}_{aux}}$. We further bucketize the domain of the property values into k bins. For instance, the density of a graph is in the range of $[0, 1]$ and $k = 5$, we bucketize the graph density into 5 bins, which results in 5 classes in the classification. Note that modeling the inference of continuous value into multi-class classification is commonly used, such as demographic properties prediction in social networks [33] and dropout rate prediction [37].

Training Attack Model. Recall that the attack model $\mathcal{F}_{AP}$ is the combination of a feature extractor $\mathcal{E}$ and multiple prediction layers $\mathcal{M}$, we can train the attack model by optimizing the following optimization problem:

$$\min \mathop{\mathbb{E}}_{\mathcal{G}_{aux} \in \mathcal{D}_{aux}} \left[\sum_{p \in \mathbb{P}} \mathcal{L} \left[\mathcal{M}^p(\mathcal{E}(H_{\mathcal{G}_{aux}})), p \right] \right]$$

where $\mathbb{P}$ is the set of properties that the attackers interested, p is a property in $\mathbb{P}$, $\mathcal{L}$ is the cross-entropy loss. Notice that all properties share the same parameters for $\mathcal{E}$, and use different parameters for $\mathcal{M}^p$.

5 Subgraph Inference Attack

5.1 Attack Overview

Given the target graph embedding $H_{\mathcal{G}_T}$ and a subgraph of interest $\mathcal{G}_S$, the attack goal is to infer whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$. Here, we assume that $\mathcal{G}_S$ constitutes a major part of the target graph $\mathcal{G}_T$.[9] That is, we do not focus on small subgraphs, such as triangles or stars, as they appear in almost all the graphs, thus not worth the adversary's efforts. The general attack pipeline of subgraph inference attack is illustrated in Figure 3.

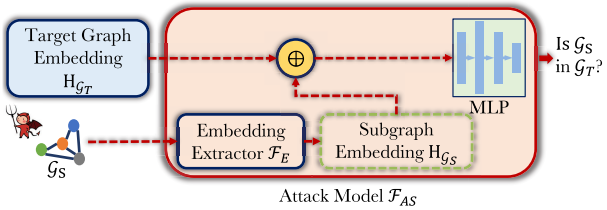

Figure 3: Attack pipeline of the subgraph inference attack. The attack model $\mathcal{F}_{AS}$ has two inputs with different formats, namely target graph embedding and subgraph. The subgraph is transformed to a subgraph embedding by an embedding extractor integrated in the attack model, aggregated with the target embedding, and sent to a binary classifier for prediction.

Note that subgraph inference attack is more challenging than the property inference attack $\mathcal{F}_{AP}$. First, subgraph isomorphism is known to be NP-complete [17]. Second, the attack model $\mathcal{F}_{AS}$ has two inputs with different formats, namely the embedding ($H_{\mathcal{G}_T}$) and the graph ($\mathcal{G}_S$), and cannot be directly compared. To make the two inputs comparable, we integrate a *graph embedding extractor* $\mathcal{F}_E$ in the attack model to transform the subgraph $\mathcal{G}_S$ to a subgraph embedding $H_{\mathcal{G}_S}$. The architecture of $\mathcal{F}_E$ can be either the same with (when the target embedding model is known) or different from (when the target embedding model is unknown) the target embedding model $\mathcal{F}_T$. Finally, the target graph embedding $H_{\mathcal{G}_T}$ and the subgraph embedding $H_{\mathcal{G}_S}$ are aggregated, using the approaches introduced in Section 5.2, and sent to a binary classifier for prediction.

[9] We experiment with subgraphs containing from 20% to 80% of the target graph's nodes (see Section 7.3).

5.2 Attack Model $\mathcal{F}_{AS}$

Attack Definition. Formally, the subgraph inference attack is defined as

$$\mathcal{F}_{AS} : \langle H_{\mathcal{G}_T}, \mathcal{G}_S \rangle \rightarrow \{\mathcal{G}_S \in \mathcal{G}_T, \mathcal{G}_S \notin \mathcal{G}_T\}$$

Concretely, the attack model $\mathcal{F}_{AS}$ is a binary classifier to determine if a given subgraph $\mathcal{G}_S$ is contained in the target graph $\mathcal{G}_T$. We outline the technical details of building $\mathcal{F}_{AS}$ below.

Generating Positive and Negative Samples. Similar to the property inference attack, we use the auxiliary dataset $\mathcal{D}_{aux}$ to obtain the training data for the attack model $\mathcal{F}_{AS}$. To generate ground truth for $\mathcal{F}_{AS}$, given an auxiliary graph $\mathcal{G}_{aux} \in \mathcal{D}_{aux}$, we generate a *positive subgraph* $\mathcal{G}_S \in \mathcal{G}_{aux}$ and a *negative subgraph* $\bar{\mathcal{G}}_S \notin \mathcal{G}_{aux}$. The positive subgraph $\mathcal{G}_S$ is generated by sampling a subgraph from the auxiliary graph $\mathcal{G}_{aux}$ using the graph sampling method, such as *random walk*. To generate the negative subgraph $\bar{\mathcal{G}}_S$, we use the same sampling method to sample a subgraph from another auxiliary graph $\mathcal{G}'_{aux} \in \mathcal{D}_{aux}$ and $\mathcal{G}'_{aux} \neq \mathcal{G}_{aux}$. As aforementioned, the subgraph of interest constitutes a major part of the target graph, the sampled negative subgraph $\bar{\mathcal{G}}_S$ is unlikely to be contained in $\mathcal{G}_{aux}$.

For each auxiliary graph $\mathcal{G}_{aux}$, we have one positive subgraph $\mathcal{G}_S$ and one negative subgraph $\bar{\mathcal{G}}_S$. The adversary first obtains the auxiliary graph embedding $H_{\mathcal{G}_{aux}}$ by querying the target embedding model. They then have a *positive sample* $\langle H_{\mathcal{G}_{aux}}, \mathcal{G}_S \rangle$, which is labeled as 1, and a negative sample $\langle H_{\mathcal{G}_{aux}}, \bar{\mathcal{G}}_S \rangle$, which is labeled as 0, for the attack model.

Constructing Features. The attack model first uses a graph embedding extractor to transform the subgraph $\mathcal{G}_S$ into a subgraph embedding $H_{\mathcal{G}_S}$ to make the two inputs comparable. The attack model then aggregates the target graph embedding $H_{\mathcal{G}_T}$ and the subgraph embedding $H_{\mathcal{G}_S}$ to generate an *attack feature vector* χ. In this paper, we propose the following three aggregation strategies:

- **Concatenation.** A commonly used approach is to concatenate the two graph embeddings, i.e., $\chi = H_{\mathcal{G}_T} || H_{\mathcal{G}_S}$, where $||$ is the concatenation operation.
- **Element-wise Difference.** An alternative is to calculate the element-wise difference of two graph embeddings, i.e., $\chi = H_{\mathcal{G}_T} - H_{\mathcal{G}_S}$.
- **Euclidean Distance.** Another approach is to calculate the Euclidean distance between two graph embeddings, i.e., $\chi = ||H_{\mathcal{G}_T} - H_{\mathcal{G}_S}||_2$.

We empirically evaluate the effectiveness of these three strategies in Section 7.3.

Training Attack Model. The final step of the attack is to send the attack feature vector χ to a binary classifier, which is modeled as an MLP (multi-layer perceptron), to determine whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$. We use the cross entropy loss and gradient decent algorithm to train the attack model. Note that the binary classifier and the graph embedding extractor in the attack model $\mathcal{F}_{AS}$ are trained simultaneously.

6 Graph Reconstruction Attack

6.1 Attack Overview

Given the target graph embedding $H_{\mathcal{G}_T}$, the attack goal is to reconstruct a graph $\mathcal{G}_R$ that have similar graph statistics, such as degree distribution and local clustering coefficient, with the target graph $\mathcal{G}_T$. Figure 4 shows the overall attack pipeline of graph reconstruction attack. The graph reconstruction attack is the most challenging task because we are rebuilding the whole graph from a single vector H_G. To this end, the attack model $\mathcal{F}_{AR}$ leverages a tailored graph auto-encoder [46] and puts its decoder into service to transform the graph embedding to a graph. Once trained, the adversary feeds $H_{\mathcal{G}_T}$ to the decoder, and the decoder would output reconstructed graph $\mathcal{G}_R$ that have similar graph statistics with the target graph $\mathcal{G}_T$.

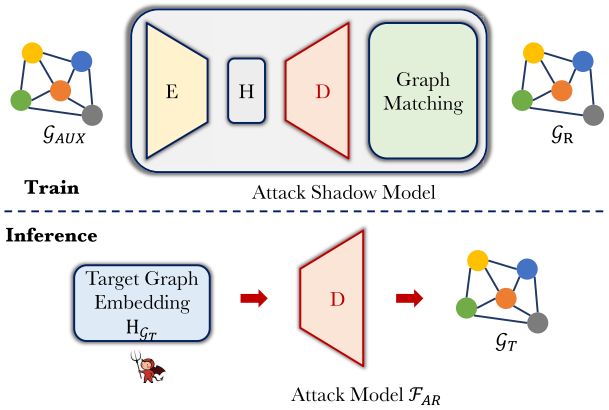

Figure 4: Attack pipeline of the graph reconstruction attack. The attack model $\mathcal{F}_{AR}$ is a decoder that can transform the graph embedding to a graph. The decoder can be obtained from the graph auto-encoder paradigm.

6.2 Attack Model $\mathcal{F}_{AR}$

Attack Definition. Formally, the graph reconstruction attack is defined as

$$\mathcal{F}_{AR} : H_{\mathcal{G}_T} \rightarrow \mathcal{G}_R$$

Essentially, the graph reconstruction attack $\mathcal{F}_{AR}$ is the decoder of a customized graph auto-encoder. We outline the technical details of building $\mathcal{F}_{AR}$ below.

Graph Auto-encoder Design. We use the graph auto-encoder paradigm to train the attack model. The architecture is shown in the training phase of Figure 4. We use an auxiliary dataset $\mathcal{D}_{aux}$ to train the graph auto-encoder. Different from the auto-encoder in the image domain, the graph auto-encoder has an additional component named *graph matching* except for the encoder and decoder. The reason for introducing the graph matching component is that neither the auxiliary graph $\mathcal{G}_{aux} \in \mathcal{D}_{aux}$ nor the reconstructed graph $\mathcal{G}_R$ imposes node orderings (i.e., graph isomorphism), making the calculation of loss between $\mathcal{G}_{aux}$ and $\mathcal{G}_R$ inaccurate. For instance, an auxiliary graph $\mathcal{G}_{aux}$ and a reconstructed graph $\mathcal{G}_R$ with the

same structure and completely different node orderings can have different adjacency matrix, such that the loss between $\mathcal{G}_{aux}$ and $\mathcal{G}_R$ is large while it is expected to be zero. Besides, the encoder in the graph auto-encoder can transform a graph to the graph embedding, which can be modeled as a GNN model. The decoder can transform the graph embedding back to graph in the form of an adjacency matrix, which can be modeled as a multi-layer perceptron.

Graph Matching. Following the same strategy as in [46], we adopt the maximum pooling matching method in our implementation. The main idea is to find a transformation matrix $Y \in \{0,1\}^{n\times n}$ between $\mathcal{G}_T$ and $\mathcal{G}_R$, where $Y_{a,i} = 1$ if node $v_a \in \mathcal{G}_T$ is assigned to $v_i \in \mathcal{G}_R$, and $Y_{a,i} = 0$ otherwise. Due to space limitation, we refer the readers to [46] for the detailed calculation of Y.

Training Attack Model. To train the graph auto-encoder, we use the cross entropy to calculate the loss between $\mathcal{G}_{aux}$ and $\mathcal{G}_R$, which calculates the cross entropy between each pair of elements in $\mathcal{G}_{aux}$ and $\mathcal{G}_R$. Formally, denote the adjacency matrix of $\mathcal{G}_{aux}$ and $\mathcal{G}_R$ as $A_{\mathcal{G}_{aux}}$ and $A_{\mathcal{G}_R}$ respectively. For each training sample, we first conduct the graph matching to obtain Y, then use the cross entropy between $A_{\mathcal{G}_{aux}}$ and $YA_{\mathcal{G}_R}Y^T$ to update the graph auto-encoder.

Fine-tuning Decoder. Note that the structure or the parameters of the encoder can be different from the target embedding model; thus, the decoder may not perfectly capture the correlation between the auxiliary graph $\mathcal{G}_{aux}$ and its graph embedding $H_{\mathcal{G}_{aux}}$ generated by the target embedding model. To address this issue, we use the auxiliary graph $\mathcal{G}_{aux}$ to query the target embedding model and obtain the corresponding graph embedding $H_{\mathcal{G}_{aux}}$. Then, the graph-embedding pairs $\langle \mathcal{G}_{aux}, H_{\mathcal{G}_{aux}} \rangle$ obtained from the target embedding model are used to fine-tune the decoder using the same procedure of graph matching and loss function as aforementioned [42].

Discussion. Both the space and time complexity of the graph matching algorithm are $O(n^4)$; thus, our attack can be only applied to graphs with tens of nodes. This is enough in many real-world datasets, such as bioinformatics graphs and mocecular graphs. In the future, we plan to investigate more advanced methods to extend our attacks to larger graphs. Besides, our current attack can only restore the graph structure of the target graph. We plan to reconstruct the node features and the graph structure simultaneously in the future.

7 Evaluation

7.1 Experimental Setup

Datasets. We conduct our experiments on five public graph datasets from TUDataset [35], including DD, ENZYMES, AIDS, NCI1, and OVCAR-8H. These datasets are widely used as benchmark datasets for evaluating the performance of GNN models [7, 12, 14, 59]. DD and ENZYMES are bioinformatics graphs, where the nodes represent the secondary structure elements, and an edge connects two nodes if they are neighbors along the amino acid sequence or one of three nearest neighbors in space. The node features consist of the amino acid type, i.e., helix, sheet, or turn, as well as several physical and chemical information. AIDS, NCI1, and OVCAR-8H are molecule graphs, where nodes and edges represent atoms and chemical bonds, respectively. The node features typically consist of one-hot encoding of the atom type, e.g., hydrogen, oxygen, carbon, etc. Each dataset has multiple independent graphs with a different number of nodes and edges, and each graph is associated with a label. For instance, the label of the molecule datasets indicates the toxicity or biological activity determined in drug discovery projects. Table 1 summarizes the statistics of all the datasets.

Table 1: Dataset statistics, including the type of graphs, the total number of graphs in the dataset, the average number of nodes, the average number of edges, and the number of classes associated with each dataset. The datasets with * are used for dataset transfer attacks.

Dataset	Type	# Graphs	Avg. Nodes	Avg. Edges	# Feats	# Classes
DD	Bioinformatics	1,178	284.32	715.66	89	2
ENZYMES	Bioinformatics	600	32.63	62.14	21	6
AIDS	Molecules	2,000	15.69	16.20	42	2
NCI1	Molecules	4110	29.87	32.30	37	2
OVCAR-8H	Molecules	4052	46.67	48.70	65	2
PC3*	Molecules	2751	26.36	28.49	37	2
MOLT-4H*	Molecules	3977	46.70	48.74	65	2

Graph Embedding Models. As discussed in Section 2.2, the graph embedding models typically consist of node embedding modules and graph pooling modules (see Section 2). In our experiments, we use a 3-layer SAGE [21] module to implement node embedding. For graph pooling, we consider the following three methods.

- **MeanPool [20].** Given all the node embeddings $H_u, \forall u \in \mathcal{G}$, MeanPool directly averages all the node embeddings to obtain the graph embedding, i.e., $H_{\mathcal{G}} = \frac{1}{|\mathcal{G}|}\sum_{u\in\mathcal{G}} H_u$, where $|\mathcal{G}|$ is the number of nodes in $\mathcal{G}$.
- **DiffPool [60].** This is a hierarchical pooling method, which relies on multiple layers of graph pooling operations to obtain the graph embedding $H_{\mathcal{G}}$. Concretely, we use three layers of graph pooling operations in our implementation. The first and second graph pooling layers narrow down the number of nodes to $0.25 \cdot |\mathcal{G}|$ and $0.25^2 \cdot |\mathcal{G}|$, respectively, using DiffPool operation. In the last layer of graph pooling, we use the mean pooling operation to generate the final graph embedding $H_{\mathcal{G}}$.
- **MinCutPool [3].** This is also a hierarchical graph pooling method. Similar to DiffPool, we use three layers of graph pooling operations. The first two graph pooling layers narrow down the number of nodes to $0.5 \cdot |\mathcal{G}|$ and $0.5^2 \cdot |\mathcal{G}|$, respectively, using MinCutPool operation, and the last layer uses the mean pooling operation.

For presentation purpose, we use the name of graph pooling methods, namely MeanPool, DiffPool, and MinCutPool, to represent the graph embedding models in this section.

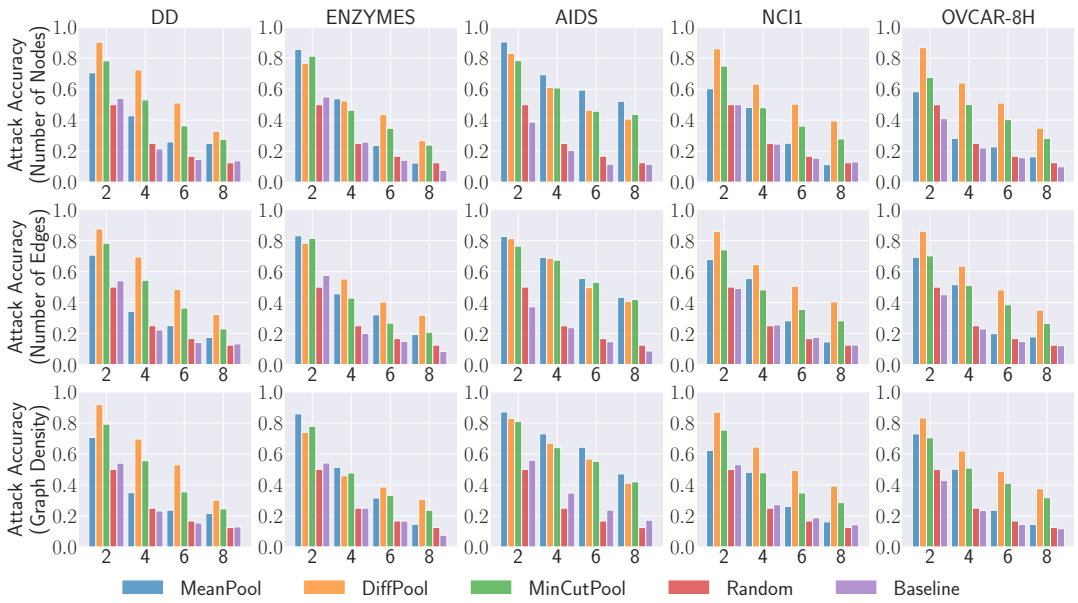

Figure 5: [Higher means better attack performance.] Attack accuracy for property inference. Different columns represent different datasets, and different rows represent different graph properties to be inferred. In each figure, different legends stand for different graph embedding models, different groups stand for different bucketization schemes. The Random and Baseline method represent the random guessing and summarizing auxiliary dataset baseline, respectively.

Implementation. We use the PyTorch Geometric[10] library to implement all the graph embedding models. All the attacks are implemented with Python 3.7, and conducted on an NVIDIA DGX-A100 server with 2TB memory.

Experimental Settings. For each dataset $\mathcal{D}$, we split it into three disjoint parts, target dataset $\mathcal{D}_T$, attack training dataset $\mathcal{D}_A^{train}$, and attack testing dataset $\mathcal{D}_A^{test}$. The target dataset $\mathcal{D}_T$ (40%) is used to train the target embedding model $\mathcal{F}_T$, which is shared by all the three inference attacks. The attack training dataset $\mathcal{D}_A^{train}$ (30%) corresponds to the auxiliary dataset $\mathcal{D}_{aux}$, which is used to generate the training data for the attack model. The attack testing dataset $\mathcal{D}_A^{test}$ (30%) corresponds to the target graph $\mathcal{G}_T$ in the attack phase. By default, we set the graph embedding dimension d_H as 192, which is the default setting of PyTorch Geometric.

7.2 Property Inference Attack

Evaluation Metrics. As the attack goal of property inference attack is to infer the basic graph properties of the target graph $\mathcal{G}_T$, a commonly used metric to measure the attack performance is the *attack accuracy*, which calculates the proportion of graphs being correctly inferred.

Attack Setup. We conduct extensive experiments on five real-world graph datasets and three state-of-the-art GNN-based graph embedding models. In our experiments, we consider five different graph properties: Number of nodes, number of edges, graph density, graph diameter, and graph radius. For each graph property, we bucketize its domain into k bins,

[10] https://github.com/rusty1s/pytorch_geometric

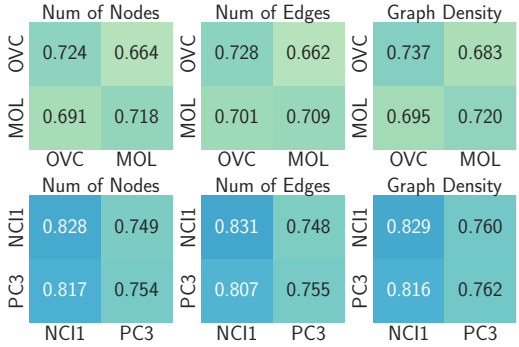

Figure 6: Datasets transferability for property inference attack between OVCAR-8H (OVC) and MOLT-4H (MOL), as well as between NCI1 and PC3.

which transforms the attack into a multi-class classification problem. Concretely, for the number of nodes (edges) and the graph diameter (radius), the property domain is from 1 to the maximum number of nodes (edges) and the maximum graph diameter (radius) in the auxiliary dataset $\mathcal{D}_{aux}$. For the graph density, the property domain is $[0.0, 1.0]$. In our experiments, we consider four different *bucketization schemes*, i.e., $k \in \{2, 4, 6, 8\}$.

Competitors. To validate the effectiveness of our proposed attack, we need to compare with two baseline attacks.

- **Random Guessing (Random).** The most straightforward baseline is random guessing, which varies for different bucketization schemes. For instance, the attack accuracy of random guessing for $k = 2$ and $k = 8$ are 0.5 and 0.125.

- **Directly Summarizing Auxiliary Dataset (Baseline).** Another baseline attack is directly summarizing the properties from the auxiliary dataset $\mathcal{D}_{aux}$ instead of training a classifier. Concretely, we calculate the average property values from $\mathcal{D}_{aux}$, and use them for predicting the properties of the target graphs.

Experimental Results. Figure 5 illustrates the attack performance, where different rows represent different graph properties, and different columns represent different datasets. Due to space limitation, we defer the results of graph diameter and graph radius to Appendix C.1.

In general, the experimental results show that our attack outperforms two baseline attacks in most of the settings. For instance, when the bucketization scheme $k = 2$, on the number of nodes property, we can achieve an attack accuracy of 0.904 on the DD dataset for the DiffPool model, while the attack accuracy of random guessing and summarizing auxiliary dataset baseline is 0.500 and 0.541, respectively. We further observe that a larger bucketization scheme k leads to worse attack accuracy. This is expected because larger k requires higher granularity of graph structural information, and is more difficult for the classifier to distinguish. In addition, we note that, in most of the cases, the attack accuracy on the MeanPool model is worse than that of the other two graph embedding models, and sometimes even close to that of the random guessing baseline. This can be explained by the fact that the MeanPool model directly averages all the node embeddings, which might lose some graph structural information.

Datasets Transferability. In previous experiments, we assume the auxiliary dataset $\mathcal{D}_{aux}$ comes from the same distribution as the target graphs. To relax this assumption, we conduct additional experiments when $\mathcal{D}_{aux}$ comes from different distribution than the target graphs. We evaluate the transferability between OVCAR-8H (OVC) and MOLT-4H (MOL), as well as between NCI1 and PC3 on MinCutPool with $k = 2$. The experimental results in Figure 6 show that our property inference attack is still effective when $\mathcal{D}_{aux}$ and the target graphs come from different distributions.

7.3 Subgraph Inference Attack

Evaluation Metrics. Recall that the subgraph inference attack is a binary classification task; thus we use the *AUC* metric to measure the attack performance, which is widely used to measure the performance of binary classification in a range of thresholds [2, 6, 16, 24, 40, 61]. The higher AUC value implies better attack performance. An AUC value of 1 implies maximum performance (true-positive rate of 1 with a false-positive rate of 0) while an AUC value of 0.5 means performance equivalent to random guessing.

Attack Setup. We conduct extensive experiments on five graph datasets and three graph embedding models to evaluate the effectiveness of our proposed attack. To obtain the subgraph, we rely on three graph sampling methods: Random walk sampling, snowball sampling, and forest fire sampling. We refer the readers to Appendix B for detailed descriptions of these sampling methods. For each sampling method, we consider four different *sampling ratios*, i.e., $\{0.2, 0.4, 0.6, 0.8\}$, which determines how many nodes are contained in the subgraph. In practice, the sampling ratio is determined by the size of the subgraph of interest. We use element-wise difference method to generate the feature vector χ. We generate the same number of positive samples and negative samples in both training and testing datasets to learn balanced model.

Competitor. Recall that we integrate a graph embedding extractor in the attack model to transform the subgraph into subgraph embedding in Section 5. The embedding extractor is jointly trained with the binary classifier in the attack model. An alternative for subgraph inference is to generate the subgraph embedding from the target model together with the target graph embedding, and then train an isolated binary classifier as attack model. To validate the necessity of integrating embedding extractor in the attack model, we compare with the baseline attack that obtains subgraph embeddings from the target model.

Experimental Results. Figure 7 illustrates the attack performance, where different rows represent different datasets, and different columns represent different sampling methods. Due to space limitation, we defer the results of other datasets to Appendix C.2. The experimental results show that our attack is effective in most of the settings, especially when the sampling ratio is 0.8. For instance, we can achieve 0.982 attack AUC on the DD dataset and MeanPool model with FireForest sampling method. Besides, we observe that when the sampling ratio decreases, the attack AUC decreases for most of the settings. This is expected as the positive samples and the negative samples tend to be more similar to each other on smaller subgraphs, making the attack model more difficult to distinguish between them. Despite this, our attack can still achieve 0.859 attack AUC on ENSYMES and MeanPool with Snowball when the sampling ratio is 0.2.

Comparing different graph embedding models, we further observe that the subgraph inference attack performs the best on the MeanPool model in most of the settings, which is opposite to the property inference attack. We suspect this is because DiffPool and MinCutPool decompose the graph structure during their pooling process; thus, the subgraph as a whole might never be seen by the target model. This makes it harder for graph embedding matching to be effective.

Necessity of Embedding Extractor. Comparing with the baseline, we observe that our subgraph inference attack consistently outperforms the baseline attack in most of the cases, especially when the sampling ratio is small. For instance, on the DD dataset, when the sampling ratio is 0.2, our attack achieves 0.821 AUC on MeanPool model and FireForest sampling method, while the baseline attack achieves AUC of 0.515. We further observe that when the sampling ratio increases, the baseline attack can gradually achieve comparable

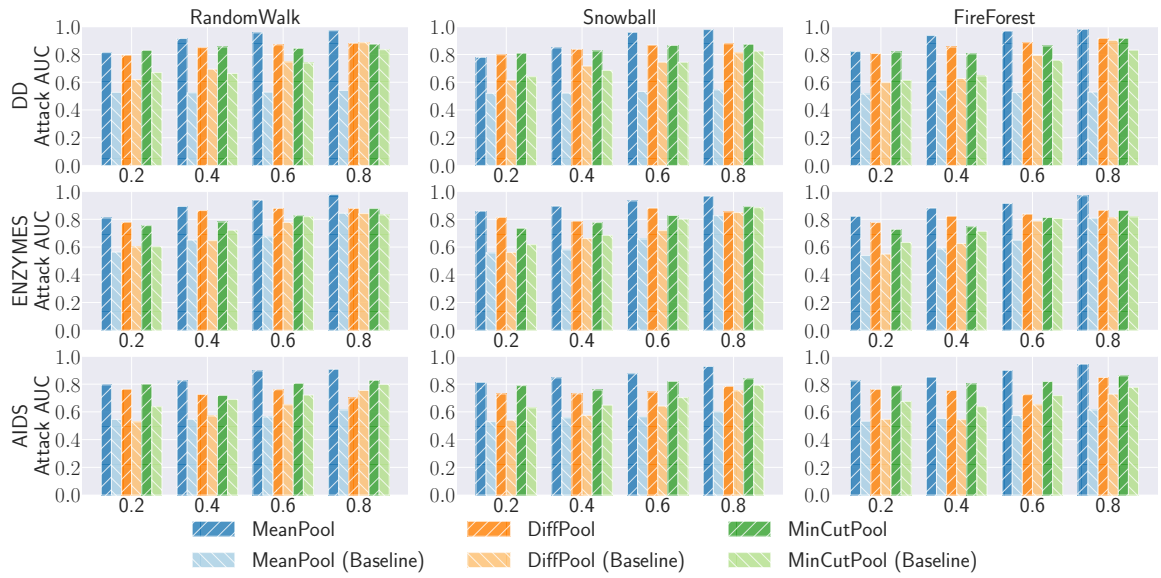

Figure 7: [Higher means better attack performance.] Attack AUC for subgraph inference attack. Different rows and columns represent different datasets and graph sampling methods. In each figure, different legends and groups stand for different graph embedding models and different sampling ratios. We use element-wise difference method to generate the feature vector χ.

Table 2: Attack AUC for different feature construction methods in subgraph inference attack. The graph embedding model is DiffPool and the graph sampling method is RandomWalk. Due to space limitation, we use Concat, EDist, and EDiff to represent Concatenation, Euclidean Distance, and Element-wise Difference, respectively.

Dataset	0.8 Concat	0.8 EDist	0.8 EDiff	0.6 Concat	0.6 EDist	0.6 EDiff	0.4 Concat	0.4 EDist	0.4 EDiff	0.2 Concat	0.2 EDist	0.2 EDiff
DD	0.53 ± 0.01	0.81 ± 0.06	$\mathbf{0.88 \pm 0.01}$	0.51 ± 0.01	0.79 ± 0.04	$\mathbf{0.87 \pm 0.01}$	0.52 ± 0.01	0.79 ± 0.02	$\mathbf{0.85 \pm 0.01}$	0.50 ± 0.02	0.71 ± 0.08	$\mathbf{0.80 \pm 0.00}$
ENZYMES	0.49 ± 0.02	0.63 ± 0.10	$\mathbf{0.88 \pm 0.03}$	0.52 ± 0.03	0.71 ± 0.10	$\mathbf{0.88 \pm 0.03}$	0.54 ± 0.02	0.56 ± 0.07	$\mathbf{0.86 \pm 0.01}$	0.48 ± 0.02	0.53 ± 0.03	$\mathbf{0.78 \pm 0.01}$
AIDS	0.51 ± 0.01	0.53 ± 0.04	$\mathbf{0.78 \pm 0.04}$	0.55 ± 0.01	0.51 ± 0.02	$\mathbf{0.76 \pm 0.05}$	0.54 ± 0.01	0.51 ± 0.03	$\mathbf{0.73 \pm 0.06}$	0.56 ± 0.02	0.50 ± 0.00	$\mathbf{0.76 \pm 0.05}$
NCI1	0.51 ± 0.00	0.51 ± 0.02	$\mathbf{0.70 \pm 0.06}$	0.49 ± 0.02	0.52 ± 0.01	$\mathbf{0.67 \pm 0.06}$	0.50 ± 0.01	0.51 ± 0.01	$\mathbf{0.64 \pm 0.03}$	0.49 ± 0.01	0.51 ± 0.01	$\mathbf{0.64 \pm 0.00}$
OVCAR-8H	0.54 ± 0.01	0.63 ± 0.12	$\mathbf{0.89 \pm 0.02}$	0.50 ± 0.04	0.69 ± 0.09	$\mathbf{0.88 \pm 0.02}$	0.51 ± 0.03	0.74 ± 0.02	$\mathbf{0.84 \pm 0.01}$	0.54 ± 0.01	0.60 ± 0.13	$\mathbf{0.82 \pm 0.02}$

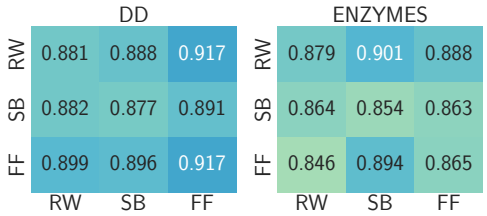

Figure 8: Sampling methods transferability for subgraph inference attack. RW, SB, and FF are abbreviations for RandomWalk, Snowball, and FireForest, respectively.

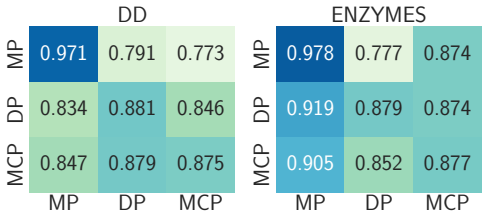

Figure 9: Embedding models transferability for subgraph inference attack. MP, DP, and MCP are abbreviations for MeanPool, DiffPool, and MinCutPool, respectively.

attack AUC as our attack. This is expected as distinguishing between the positive subgraph and negative subgraph is much easier when the sampling ratio is large.

Comparison of Feature Construction Methods. We propose three strategies to aggregate the graph embeddings of the target graph and the subgraph of interest in the attack model $\mathcal{F}_{AS}$, namely concatenation, element-wise difference, and Euclidean distance, in Section 5. We now compare the performance of different strategies. Table 2 shows the experimental results on five datasets when the graph embedding model is DiffPool and the graph sampling method is RandomWalk.

We observe that the element-wise difference method achieves the best performance, while the concatenation method has an attack AUC close to random guessing. This indicates that the discrepancy information between two graph embeddings (element-wise difference method) is more in-

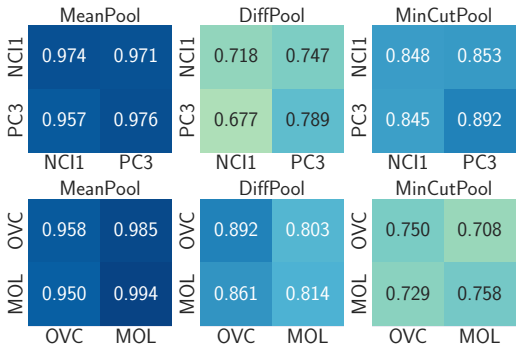

Figure 10: Dataset transferability for subgraph inference between OVCAR-8H (OVC) and MOLT-4H (MOL), as well as between NCI1 and PC3.

formative than the plain graph embeddings (concatenation method) in terms of subgraph inference attack. Note that the Euclidean distance also implicitly captures the discrepancy information of two graph embeddings, while it relies on one scalar value and loses other rich discrepancy information.

Sampling Methods Transferability. So far, our experiments use the same sampling method for the auxiliary graph to train the attack model and the target graph to test the attack model. We conduct additional experiments to show whether our attack still works when the sampling methods are different. Figure 8 illustrates the experimental results on DD and ENZYMES datasets. We use DiffPool as the graph embedding model and adopt a sampling ratio of 0.8. As we can see, in most cases, the sampling methods do not have a significant impact on the attack performance.

Embedding Models Transferability. In previous experiments, the architecture of the graph embedding extractor in the attack model is the same as the target embedding model. In practice, the model architecture of the target embedding model might be unknown to the adversaries. To understand whether our attack still works when the architectures are different, we conduct experiments on the DD and ENZYMES datasets. Figure 9 illustrates the experimental results of RandomWalk sampling method with a sampling ratio of 0.8. We observe that the attack performance slightly drops when the model architectures are different. Despite this, we can still achieve 0.773 attack AUC in the worse case.

Datasets Transferability. Similar to property inference attack, to relax the assumption that $\mathcal{D}_{aux}$ comes from the same distribution of the target graphs, we conduct additional experiments when $\mathcal{D}_{aux}$ and the target graphs come from different distributions. We experiment on the RandomWalk method with a sampling ratio of 0.8. The experimental results in Figure 10 show that our subgraph inference attack is still effective for dataset transfer.

7.4 Graph Reconstruction Attack

Evaluation Metrics. We evaluate the performance of graph reconstruction from two perspectives:

Table 3: [Higher means better attack performance.] Attack performance of graph reconstruction measured by graph isomorphism.

Dataset	DiffPool	MeanPool	MinCutPool
AIDS	0.875 ± 0.003	0.794 ± 0.003	0.869 ± 0.002
ENZYMES	0.670 ± 0.019	0.653 ± 0.022	0.704 ± 0.012
NCI1	0.752 ± 0.005	0.771 ± 0.010	0.693 ± 0.007

- **Graph Isomorphism**. The graph isomorphism compares the structure of the reconstructed graph $\mathcal{G}_R$ with the target graph $\mathcal{G}_T$, and determines their similarity. The graph isomorphism problem is well-known to be intractable in polynomial time; thus, approximate algorithms such as *Weisfeiler-Lehman (WL) algorithm* are widely used for addressing it [36, 44, 59]. The general idea of WL algorithm is to iteratively calculate the WL graph kernel of two graphs. We normalize the WL graph kernel in the range of [0.0, 1.0], and a WL graph kernel of 1.0 means two graphs perfectly match. We adopt the DGL implementation of WL algorithm in our experiments.[11]
- **Macro-level Graph Statistics**. Recall that the objective of the graph reconstruction attack is to generate a graph $\mathcal{G}_R$ that has similar graph statistics with the target graph $\mathcal{G}_T$. In practice, there are a plethora of graph structural statistics to analyze a graph. In this paper, we adopt four widely used graph statistics: Degree distribution, local clustering coefficient (LCC), betweenness centrality (BC), and closeness centrality (CC). We refer the readers to Appendix B for detailed descriptions of these statistics.

Note that the number of nodes in $\mathcal{G}_R$ might be different from the target graph $\mathcal{G}_T$ due to the graph auto-encoder architecture, and there are no node orderings imposed for $\mathcal{G}_R$ and $\mathcal{G}_T$; thus we cannot directly compare the node-level graph statistics including LCC, CC, and BC. To address this issue, we bucketize the statistic domain into 10 bins and measure their distributions. For each graph statistic, we use three metrics to measure the distribution similarity between the target graph $\mathcal{G}_T$ and the reconstructed graph $\mathcal{G}_R$: *Cosine similarity*, *Wasserstein distance*, and *Jensen-Shannon (JS) divergence*. Intuitively, higher cosine similarity and lower Wasserstein distance/JS divergence mean better attack performance. The ranges of cosine similarity, Wasserstein distance, and JS divergence are $[-1.0, 1.0]$, $[0.0, 1.0]$, and $[0.0, 1.0]$, respectively.

Attack Setup. Recall that both space and time complexity of the graph matching algorithm are $O(n^4)$, we conduct our experiments on three small datasets in Table 1, i.e., AIDS, ENZYMES, and NCI1, and three graph embedding models. We run all the experiments five times with the mean and standard deviation reported.

[11] https://github.com/InkToYou/WL-Kernel-DGL

Table 4: [Higher means better attack performance.] Attack performance of graph reconstruction measured by macro-level graph statistics, the similarity of which is measured by cosine similarity.

Dataset	Target Model	Degree Dist.	LCC Dist.	BC Dist.	CC Dist.
AIDS	MeanPool	0.651 ± 0.001	0.999 ± 0.001	0.987 ± 0.001	0.876 ± 0.002
	DiffPool	0.894 ± 0.001	0.999 ± 0.001	0.983 ± 0.001	0.787 ± 0.002
	MinCutPool	0.888 ± 0.003	0.999 ± 0.001	0.983 ± 0.001	0.785 ± 0.006
ENZYMES	MeanPool	0.450 ± 0.070	0.646 ± 0.005	0.959 ± 0.001	0.516 ± 0.037
	DiffPool	0.519 ± 0.007	0.661 ± 0.008	0.958 ± 0.001	0.504 ± 0.005
	MinCutPool	0.467 ± 0.019	0.490 ± 0.009	0.916 ± 0.001	0.414 ± 0.009
NCI1	MeanPool	0.736 ± 0.003	0.999 ± 0.001	0.877 ± 0.001	0.402 ± 0.001
	DiffPool	0.633 ± 0.002	0.999 ± 0.001	0.877 ± 0.001	0.495 ± 0.002
	MinCutPool	0.570 ± 0.002	0.999 ± 0.001	0.877 ± 0.001	0.496 ± 0.001

Experimental Results. Table 3 and Table 4 illustrate the attack performance in terms of graph isomorphism and macro-level graph statistics (measured by cosine similarity), respectively. Due to space limitation, we defer the results of the macro-level graph statistics measured by Wasserstein distance and JS divergence to Appendix C.3. In general, our attack achieves strong performance. For instance, the WL graph kernel on AIDS and DiffPool achieves 0.875. Besides, the cosine similarity of the betweenness centrality distribution is larger than 0.85 for all the settings. We can also achieve 0.99 cosine similarity for local clustering coefficient distribution for the AIDS and NCI1 datasets. For degree distribution and closeness centrality distribution, the attack performance is slightly worse; however, we can still achieve cosine similarity larger than or close to 0.5.

To investigate the impact of the quality of the auto-encoder on the attack performance, we conduct additional experiments on the auto-encoders trained in different epochs. Due to space limitation, we defer the experimental results to Appendix C.3.

8 Defenses

Graph Embedding Perturbation. A commonly used defense mechanism for inference attacks is adding perturbation to the output of the model [62]. In this paper, we propose to add perturbations to the target graph embedding H_{G_T} to defend our proposed inference attacks. Formally, given the target graph embedding H_{G_T}, the data owner only shares a noisy version of graph embedding $\tilde{H_{G_T}} = H_{G_T} + \mathsf{Lap}(\beta)$ to the third party, where $\mathsf{Lap}(\beta)$ denotes a random variable sampled from the Laplace distribution with scale parameter β; that is, $\Pr[\mathsf{Lap}(\beta) = x] = \frac{1}{2\beta} e^{-|x|/\beta}$. Notice that adding noise to the graph embedding vector may destroy the graph structural information, thus affect the normal tasks such as graph classification. Therefore, we need to choose a moderate level of noise to tradeoff the defense effectiveness and the performance of the normal tasks.

Defense Evaluation Setup. We conduct experiments to validate the effectiveness of our proposed defense against all the inference attacks, as well as the impact on normal graph classification task. For property inference attack, we evaluate the performance of graph density with bucketization scheme $k = 2$. For subgraph inference attack, we consider the RandomWalk sampling method with sampling ratio of 0.8. We conduct our experiments on DD and ENZYMES datasets and three graph embedding models. Due to space limitation, we refer the readers to Appendix C.4 for experimental results for other datasets and graph reconstruction attack.

Defense Evaluation Results. Figure 11 illustrates the experimental results, where the first and second column represents the attack performance of property inference attack and subgraph inference attack respectively, the last column represents the accuracy of the normal graph classification task. In each figure, the x-aixs stands for the scaling parameter β of Laplace noise, where larger β means larger noise. We observe that when the noise level increases, the attack performance for both property inference and subgraph inference attack decreases. This is expected since more noise will hide more structural information contained in the graph embedding. On the other hand, the accuracy of the graph classification tasks will also decrease when the noise level increase. To defend against the inference attacks while preserving the utility for normal tasks, one needs to carefully choose the noise level. For instance, when we set the standard deviation of Laplace noise to 2, the performance of subgraph inference attack significantly drops while the graph classification accuracy only slightly decreases.

9 Related Work

In this section, we review the research work close to our proposed attacks. We refer the readers to [18, 63] for in-depth overview of different GNN models, and [8, 25, 50, 57] for comprehensive surveys of existing adversarial attacks and defense strategies on GNNs.

Causative Attacks on GNNs. Causative attack allows attackers to manipulate training dataset in order to change the parameters of the target model. In the context of causative attacks on GNNs, Zügner *et al.* [64] was the first research work that introduced unnoticeable adversarial perturbations targeting the node's features and the graph structure to reduce the accuracy of node classification via graph convolutional networks. Following this direction, researchers investigated different adversarial attack strategies (*i.e.* edge/node-level/structure/attribute perturbation) to achieve various attack objectives, such as reducing the accuracy of node classification [4, 13, 32, 51, 55, 58], link prediction [4, 31], graph classification [8, 56], etc. Our attacks do not tamper with the training data that is used to construct the GNN models.

Exploratory Attacks on GNNs. Exploratory attack does not change the parameters of the target model. Instead, the attacker sends new data to the target model and observes

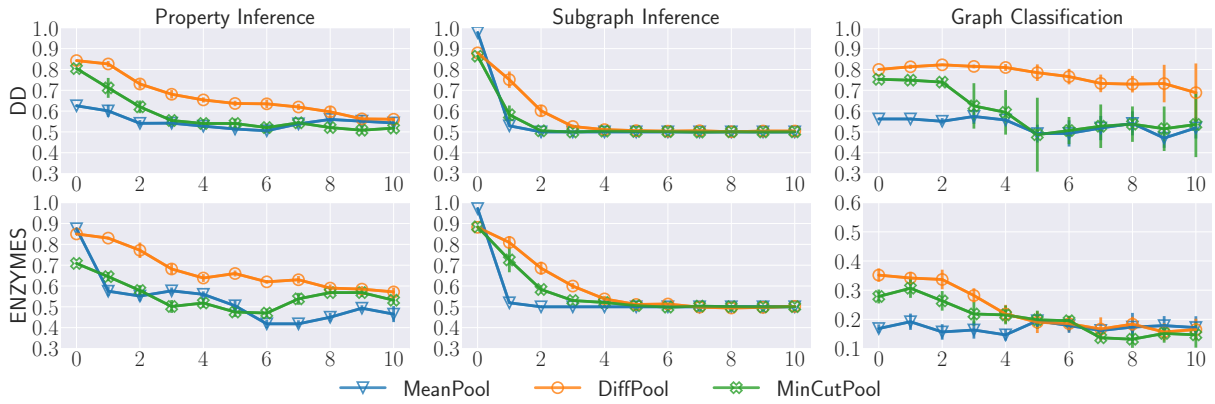

Figure 11: Graph embedding perturbation defense on the DD and ENZYMES datasets (different rows). The first two columns represent the attack performance of property inference and subgraph inference respectively, the last column represents the accuracy of normal graph classification task. In each figure, the x-axis stands for the scaling parameter β for Laplace noise, where larger β means higher noise level. The y-axis stands for the attack performance/normal graph classification accuracy.

the model's decisions on these carefully crafted input data. However, graph-based machine learning under adversarial exploratory setting is much less explored. In particular, only a few studies [11, 22, 54] focused on exploratory attacks on GNNs. For instance, He *et al.* [22] proposed link stealing attack to infer, from the outputs of a GNN model, whether there exists a link between any pair of nodes in the graph used to train the model. Wu *et al.* [54] discussed GNN model extraction attack, given various levels of background knowledge, by gathering both the input-output query pairs and the graph structure to reconstruct a duplicated model. Duddu *et al.* [11] proposed a graph reconstruction attack against *node embeddings*; however, there are several difference from our graph reconstruction attack. First, the task is different, [11] aims to reconstruct a graph from a set of node embeddings, while ours is to reconstruct the graph from a graph embedding. Also, the node embeddings targeted by [11] are generated from traditional node embedding method such as Deepwalk [38] and node2vec [19], while ours focus on state-of-the-art GNN. In addition, our threat model is more general and practical as we are only given one embedding vector of the target graph instead of all embeddings of all the nodes. In this sense, our adversary has much less background knowledge than that of [11]. Besides, their method uses the non-learnable dot product as the decoder. Our approach leverages a learnable decoder and can be further fine-tuned to enhance graph reconstruction performance.

Defense of Adversarial Attacks on GNNs. The emerging attacks on GNNs leads to an arm race. To mitigate those attacks, several defense strategies (*e.g.* graph sanitization [55], adversarial training [9, 15] and certification of robustness [5]) have been proposed. One important direction of those defense strategies is to reduce the sensitivity of GNNs via adversarial training so that the train GNNs are robust to structure perturbation [9] and attribution perturbation [15]. Beside, robustness certification [5] is an emerging research direction that measure and reason the safety of graph neural networks under adversarial perturbation. Note that aforementioned defense mechanisms focus on mitigating causative attacks on GNNs, hence they are are not design to protect GNNs from exploratory attacks.

10 Conclusion

In this paper, we investigate the information leakage of graph embedding. Concretely, we propose three different attacks to extract information from the target graph given the graph embedding. First, we can successfully infer graph properties, such as the number of nodes, the number of edges, and graph density, of the target graph. Second, given a subgraph of interest and the graph embedding, we can determine with high confidence that whether the subgraph is contained in the target graph. Third, we propose a novel graph reconstruction attack that can reconstruct a graph that has similar graph statistics with the target graph. We further propose an embedding perturbation based defense to mitigate the inference attacks without noticeable accuracy degradation.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work is partially funded by the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4).

References

[1] James Atwood and Don Towsley. Diffusion-Convolutional Neural Networks. In *NIPS*, pages 1993–2001, 2016.

[2] Michael Backes, Mathias Humbert, Jun Pang, and Yang Zhang. walk2friends: Inferring Social Links from Mobility Profiles. In *CCS*, pages 1943–1957, 2017.

[3] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral Clustering with Graph Neural Networks for Graph Pooling. In *ICML*, pages 874–883, 2020.

[4] Aleksandar Bojchevski and Stephan Günnemann. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *ICML*, pages 695–704, 2019.

[5] Aleksandar Bojchevski and Stephan Günnemann. Certifiable Robustness to Graph Perturbations. In *NeurIPS*, pages 8317–8328, 2019.

[6] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. When Machine Unlearning Jeopardizes Privacy. In *CCS*, 2021.

[7] Zhengdao Chen, Soledad Villar, Lei Chen, and Joan Bruna. On the equivalence between graph isomorphism testing and function approximation with GNNs. In *NeurIPS*, pages 15868–15876, 2019.

[8] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial Attack on Graph Structured Data. In *ICML*, pages 1123–1132, 2018.

[9] Quanyu Dai, Xiao Shen, Liang Zhang, Qiang Li, and Dan Wang. Adversarial Training Methods for Network Embedding. In *WWW*, pages 329–339, 2019.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*, pages 3837–3845, 2016.

[11] Vasisht Duddu, Antoine Boutet, and Virat Shejwalkar. Quantifying Privacy Leakage in Graph Embedding. *CoRR abs/2010.00906*, 2020.

[12] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking Graph Neural Networks. *CoRR abs/2003.00982*, 2020.

[13] Negin Entezari, Saba A. Al-Sayouri, Amirali Darvishzadeh, and Evangelos E. Papalexakis. All You Need Is Low (Rank): Defending Against Adversarial Attacks on Graphs. In *WSDM*, pages 169–177, 2020.

[14] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A Fair Comparison of Graph Neural Networks for Graph Classification. In *ICLR*, 2020.

[15] Fuli Feng, Xiangnan He, Jie Tang, and Tat-Seng Chua. Graph Adversarial Training: Dynamically Regularizing Based on Graph Structure. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[16] Matt Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security*, pages 17–32, 2014.

[17] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[18] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge Based Systems*, 2018.

[19] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, pages 855–864, 2016.

[20] William L. Hamilton. *Graph Representation Learning*. Morgan and Claypool, 2020.

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, pages 1025–1035, 2017.

[22] Xinlei He, Jinyuan Jia, Michael Backes, Neil Zhenqiang Gong, and Yang Zhang. Stealing Links from Graph Neural Networks. In *USENIX Security*, pages 2669–2686, 2021.

[23] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High Accuracy and High Fidelity Extraction of Neural Networks. In *USENIX Security*, pages 1345–1362, 2020.

[24] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. MemGuard: Defending against Black-Box Membership Inference Attacks via Adversarial Examples. In *CCS*, pages 259–274, 2019.

[25] Wei Jin, Yaxin Li, Han Xu, Yiqi Wang, and Jiliang Tang. Adversarial Attacks and Defenses on Graphs: A Review and Empirical Study. *CoRR abs/2003.00653*, 2020.

[26] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS*, pages 615–629, 2017.

[27] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular Graph Convolutions: Moving Beyond Fingerprints. *Journal of Computer-Aided Molecular Design*, 2016.

[28] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.

[29] Nicholas D. Lane and Petko Georgiev. Can Deep Learning Revolutionize Mobile Sensing? In *HotMobile*, pages 117–122, 2015.

[30] Xiaoxiao Li, João Saúde, Prashant Reddy, and Manuela Veloso. Classifying and Understanding Financial Data Using Graph Neural Network. In *KDF*, 2020.

[31] Wanyu Lin, Shengxiang Ji, and Baochun Li. Adversarial Attacks on Link Prediction Algorithms Based on Graph Neural Networks. In *ASIACCS*, pages 370–380, 2020.

[32] Jiaqi Ma, Shuangrui Ding, and Qiaozhu Mei. Towards More Practical Adversarial Attacks on Graph Neural Networks. In *NeurIPS*, 2020.

[33] Eric Malmi and Ingmar Weber. You Are What Apps You Use: Demographic Prediction Based on User's Apps. In *ICWSM*, pages 635–638, 2016.

[34] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *S&P*, pages 497–512, 2019.

[35] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. In *GRL*, 2020.

[36] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. In *AAAI*, pages 4602–4609, 2019.

[37] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards Reverse-Engineering Black-Box Neural Networks. In *ICLR*, 2018.

[38] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *KDD*, pages 701–710, 2014.

[39] Trang Pham, Truyen Tran, Dinh Q. Phung, and Svetha Venkatesh. Column Networks for Collective Classification. In *AAAI*, pages 2485–2491, 2017.

[40] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *NDSS*, 2018.

[41] Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. DeepInf: Social Influence Prediction with Deep Learning. In *KDD*, pages 2110–2119, 2018.

[42] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning. In *USENIX Security*, pages 1291–1308, 2020.

[43] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 2009.

[44] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 2011.

[45] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *S&P*, pages 3–18, 2017.

[46] Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards Gen-

eration of Small Graphs Using Variational Autoencoders. In *ICANN*, pages 412–422, 2018.

[47] Congzheng Song and Ananth Raghunathan. Information Leakage in Embedding Models. In *CCS*, pages 377–390, 2020.

[48] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine Learning Models that Remember Too Much. In *CCS*, pages 587–601, 2017.

[49] Congzheng Song and Vitaly Shmatikov. Overlearning Reveals Sensitive Attributes. In *ICLR*, 2020.

[50] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S. Yu, Lifang He, and Bo Li. Adversarial Attack and Defense on Graph Data: A Survey. *CoRR abs/1812.10528*, 2018.

[51] Yiwei Sun, Suhang Wang, Xianfeng Tang, Tsung-Yu Hsieh, and Vasant Honavar. Non-target-specific Node Injection Attacks on Graph Neural Networks: A Hierarchical Reinforcement Learning Approach. In *WWW*, pages 673–683, 2020.

[52] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: Large-scale Information Network Embedding. In *WWW*, pages 1067–1077, 2015.

[53] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *ICLR*, 2018.

[54] Bang Wu, Xiangwen Yang, Shirui Pan, and Xingliang Yuan. Model Extraction Attacks on Graph Neural Networks: Taxonomy and Realization. *CoRR abs/2010.12751*, 2020.

[55] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. Adversarial Examples for Graph Data: Deep Insights into Attack and Defense. In *IJCAI*, pages 4816–4823, 2019.

[56] Zhaohan Xi, Ren Pang, Shouling Ji, and Ting Wang. Graph Backdoor. In *USENIX Security*, 2021.

[57] Han Xu, Yao Ma, Haochen Liu, Debayan Deb, Hui Liu, Jiliang Tang, and Anil K. Jain. Adversarial Attacks and Defenses in Images, Graphs and Text: A Review. *International Journal of Automation and Computing*, 2020.

[58] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective. In *IJCAI*, pages 3961–3967, 2019.

[59] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? In *ICLR*, 2019.

[60] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical Graph Representation Learning with Differentiable Pooling. In *NeurIPS*, pages 4805–4815, 2018.

[61] Yang Zhang, Mathias Humbert, Bartlomiej Surma, Praveen Manoharan, Jilles Vreeken, and Michael Backes. Towards Plausible Graph Anonymization. In *NDSS*, 2020.

[62] Zhikun Zhang, Tianhao Wang, Jean Honorio, Ninghui Li, Michael Backes, Shibo He, Jiming Chen, and Yang Zhang. PrivSyn: Differentially Private Data Synthesis. In *USENIX Security*, pages 929–946, 2021.

[63] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[64] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial Attacks on Neural Networks for Graph Data. In *KDD*, pages 2847–2856, 2018.

A Notations

The frequently used notations used in this paper is summarized in Table 5.

Table 5: Summary of the notations used in this paper.

Notation	Description
$\mathcal{G} = \langle \mathcal{V}, A, X \rangle$	Graph
$u, v \in \mathcal{V}$	Nodes in $\mathcal{G}$
$n = \lvert\mathcal{V}\rvert$	Number of nodes
d_X / d_H	Dimension of attributes / embeddings
$A \in \{0,1\}^{n \times n}$	Adjacency matrix of $\mathcal{G}$
$X \in \mathbb{R}^{n \times d_X}$	Attributes associated with $\mathcal{V}$
$\mathcal{N}_u$	Neighborhood nodes of u
$\mathcal{G}_S$	Subgraph of $\mathcal{G}$
$\mathcal{G}_T$ / $\mathcal{G}_{aux}$	Target / auxiliary graph
$\mathcal{D}_{aux}$	Auxiliary dataset ($\mathcal{G}_{aux} \in \mathcal{D}_{aux}$)
H_u / $H_\mathcal{G}$	Node / graph embedding
$\mathcal{F}_T$ / $\mathcal{F}_A$	Target / attack model
$\mathcal{F}_{AP}$	Attack model of property inference
$\mathcal{F}_{AS}$	Attack model of subgraph inference
$\mathcal{F}_{AR}$	Attack model of graph reconstruction
Φ	Aggregation operation
Ψ	Updating operation
Σ	Graph pooling operation
$\mathbf{m}$	Message received from neighbors
χ	Feature vector of subgraph inference

B Experimental Details

B.1 Graph Sampling Methods

- **Random Walk Sampling.** The main idea of RandomWalk is to randomly pick a starting node, and then simulate a random walk on the graph until we obtain the desired number of nodes.
- **Snowball Sampling.** The main idea of Snowball is to randomly select a set of seed nodes, and then iteratively select a set of neighboring nodes of the selected nodes until we obtain the desired number of nodes.
- **Forest Fire Sampling.** The main idea of FireForest is to randomly select a seed node, and begin "burning" outgoing edges and the corresponding nodes. Here, a node "burns" its outgoing edges and the corresponding nodes means these edges and nodes are sampled. If an edge gets burned, the node at the other endpoint gets a chance to burn its own edges, and so on recursively until we obtain the desired number of nodes.

B.2 Macro-level Graph Statistics

- **Degree Distribution.** The degree distribution $P(k)$ of a graph is defined to be the fraction of nodes in the graph with degree k. It is the most widely used graph statistic to quantify a graph.
- **Local Clustering Coefficient (LCC).** The LCC of a node quantifies how close its neighbors are to being a cluster. It is primarily introduced to determine whether a graph is a small-world network.
- **Betweenness Centrality (BC).** The betweenness centrality is a measure of centrality in a graph based on the shortest

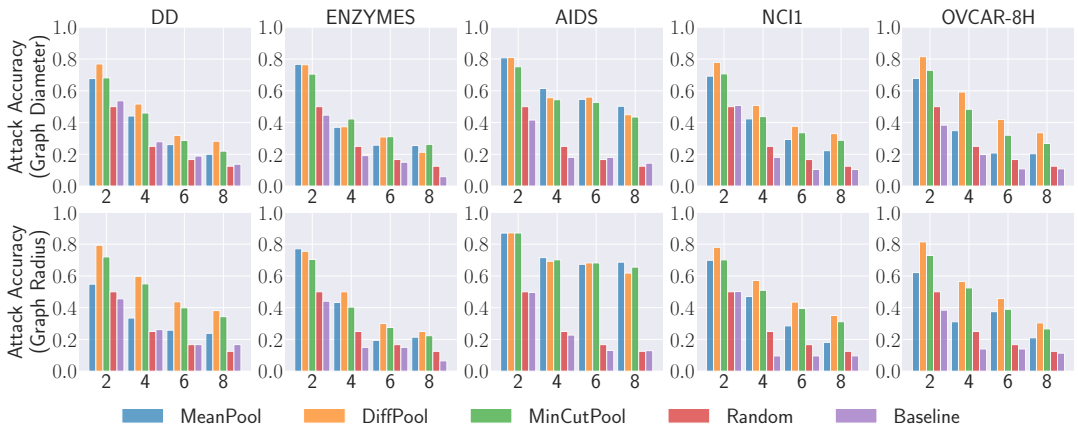

Figure 12: [Higher means better attack performance.] Attack accuracy of additional properties for property inference. Different columns represent different datasets, and different rows represent different graph properties to be inferred. In each figure, different legends stand for different graph embedding models, different groups stand for different bucketization schemes. The Random and Baseline method represent the random guessing and summarizing auxiliary dataset baseline, respectively.

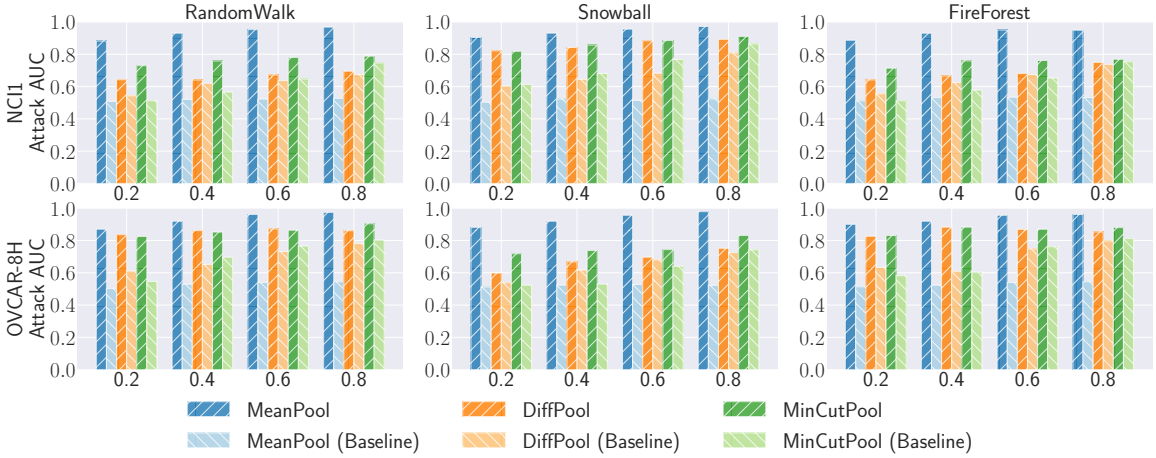

Figure 13: [Higher means better attack performance.] Attack AUC for subgraph inference attack. Different rows represent different datasets, and different columns represent different graph sampling methods. In each figure, different legends stand for different graph embedding models, different groups stand for different sampling ratios.

paths. For every pair of nodes in a graph, there exists at least one shortest path between the nodes such that either the number of edges that the path passes through is minimized. The betweenness centrality for each node is the number of these shortest paths that pass through the node.

- **Closeness Centrality (CC).** The CC of a node is a measure of centrality in a graph, which is calculated as the reciprocal of the sum of the length of the shortest paths between the node and all other nodes in the graph. Intuitively, the more central a node is, the closer it is to all other nodes.

C Additional Experimental Results

C.1 Property Inference Attack

Additional Properties. Figure 12 illustrates the attack performance on the graph diameter and the graph radius properties. The experimental results show that our attack is still effective on these two properties in most of the settings. The conclusions are consistent with that of Section 7.2.

C.2 Subgraph Inference Attack

Addtional Datasets. Figure 13 illustrates the comparison with baseline subgraph inference attacks on the NCI1 and OVCAR-8H datasets. The conclusions are consistent with that of Section 7.3.

C.3 Graph Reconstruction Attack

Additional Metrics. Table 6 and Table 7 illustrate the attack performance in terms of macro-level graph statistics measured by Wasserstein distance and JS divergence, respectively. The experimental results show that our graph reconstruction achieves small Wasserstein distance and JS divergence for most of the settings, indicating our graph reconstruction attack is effective.

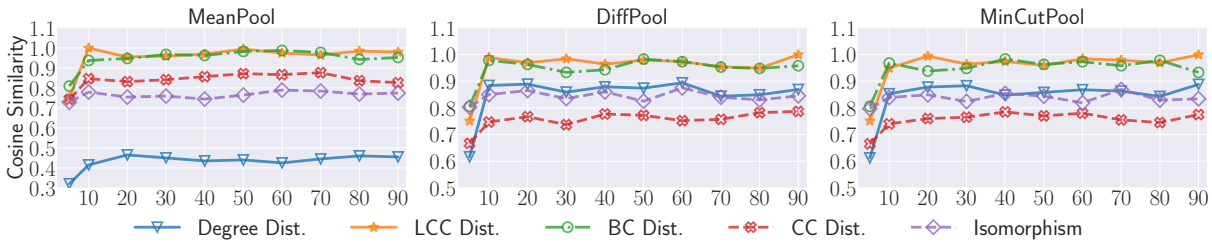

Figure 14: Impact of the quality of graph auto-encoder on the AIDS dataset.

Table 6: [Lower means better attack performance.] Attack performance of graph reconstruction measured by macro-level graph statistics, the similarity of which is measured by Wasserstein distance.

Dataset	Target Model	Degree Dist.	LCC Dist.	BC Dist.	CC Dist.
AIDS	DiffPool	0.040 ± 0.001	0.055 ± 0.002	0.011 ± 0.000	0.038 ± 0.001
	MeanPool	0.073 ± 0.000	0.020 ± 0.001	0.027 ± 0.001	0.067 ± 0.001
	MinCutPool	0.046 ± 0.000	0.067 ± 0.002	0.012 ± 0.000	0.047 ± 0.001
ENZYMES	DiffPool	0.125 ± 0.004	0.201 ± 0.009	0.039 ± 0.001	0.258 ± 0.005
	MeanPool	0.060 ± 0.006	0.188 ± 0.018	0.039 ± 0.001	0.086 ± 0.009
	MinCutPool	0.085 ± 0.006	0.199 ± 0.005	0.040 ± 0.003	0.171 ± 0.013
NCI1	DiffPool	0.063 ± 0.001	0.091 ± 0.004	0.056 ± 0.001	0.084 ± 0.003
	MeanPool	0.045 ± 0.001	0.049 ± 0.004	0.062 ± 0.001	0.067 ± 0.001
	MinCutPool	0.087 ± 0.000	0.119 ± 0.003	0.055 ± 0.001	0.138 ± 0.001

Table 7: [Lower means better attack performance.] Attack performance of graph reconstruction measured by macro-level graph statistics, the similarity of which is measured by JS divergence.

Dataset	Target Model	Degree Dist.	LCC Dist.	BC Dist.	CC Dist.
AIDS	DiffPool	0.120 ± 0.003	0.052 ± 0.002	0.029 ± 0.001	0.080 ± 0.005
	MeanPool	0.253 ± 0.001	0.019 ± 0.000	0.056 ± 0.002	0.132 ± 0.004
	MinCutPool	0.136 ± 0.000	0.068 ± 0.003	0.029 ± 0.001	0.106 ± 0.001
ENZYMES	DiffPool	0.341 ± 0.007	0.279 ± 0.012	0.071 ± 0.006	0.540 ± 0.014
	MeanPool	0.201 ± 0.015	0.213 ± 0.009	0.073 ± 0.003	0.165 ± 0.019
	MinCutPool	0.280 ± 0.004	0.248 ± 0.003	0.073 ± 0.006	0.354 ± 0.028
NCI1	DiffPool	0.210 ± 0.001	0.103 ± 0.002	0.093 ± 0.003	0.206 ± 0.006
	MeanPool	0.159 ± 0.004	0.048 ± 0.003	0.105 ± 0.001	0.149 ± 0.003
	MinCutPool	0.275 ± 0.000	0.160 ± 0.003	0.085 ± 0.001	0.345 ± 0.005

Impact of Graph Auto-encoder. To investigate the impact of the quality of the graph auto-encoder on the attack performance, we conduct additional experiments on the graph auto-encoders trained with different epochs. Figure 14 shows the experimental results. We observe that with the number of epochs increases, our attack performance increases, indicating the quality of the graph auto-encoder has positive impact on our attack. When the number of epochs exceeds 10, the attack performance remains unchanged for most of the settings. Thus, we train the graph auto-encoder for 10 epochs in our experiments.

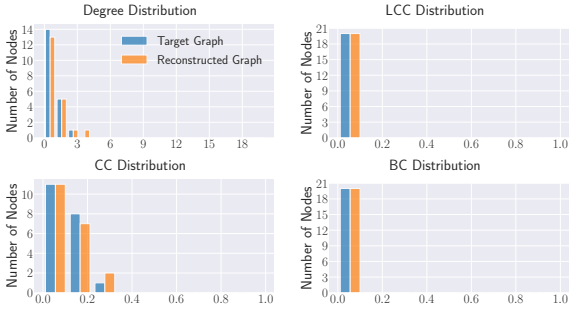

Figure 15: Visualization of macro-level graph statistic distribution for graph reconstruction attack on the AIDS dataset.

Visualization. To better illustrate the effectiveness of our graph reconstruction attack on preserving the macro-level graph statistics, we provide a distribution visualization of the AIDS dataset in Figure 15. We experiment on the MinCutPool model. The visualization results show that our graph reconstruction attack can effectively preserve the macro-level graph statistics.

C.4 Defense

Additional Datasets. Figure 16 illustrates the defense performance on the ADIS and OVCAR-8H datasets for property inference and subgraph inference attack. The conclusions are consistent with that of Section 8 for these datasets.

Defense against Graph Reconstruction. Figure 17 illustrates the defense performance for graph reconstruction attack. The experimental results show that our defense mechanism is still effective for graph reconstruction attack.

D Impact of Node Features

To evaluate the impact of node features, we conduct additional experiments on graphs without node features. Concretely, for each dataset in Table 1, we replace all its original node features with one-hot encodings of node degrees. This follows the setting of [59] which aims to investigate the expressiveness of graph structure. Figure 18 shows the experimental results for the subgraph inference attack. The experimental results show that the attack performance of graphs with and without node features is similar for most of the settings, indicating the robustness of our subgraph inference attack.

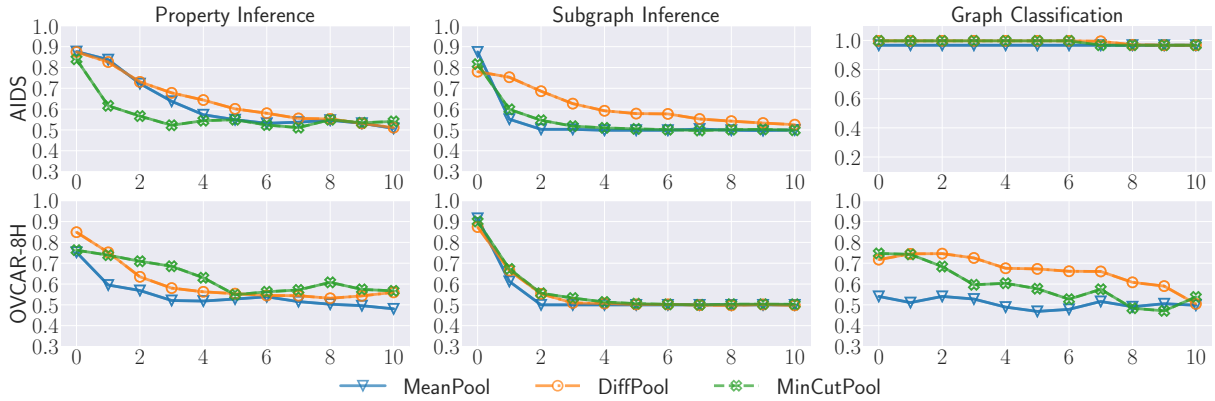

Figure 16: Graph embedding perturbation defense on the AIDS and OVCAR-8H datasets. The first and second column represents the attack performance of property inference and subgraph inference respectively, the last column represents the accuracy of normal graph classification task. In each figure, the x-axis stands for the scaling parameter β for Laplace noise, where larger β means higher noise level. The y-axis stands for the attack performance/normal graph classification accuracy.

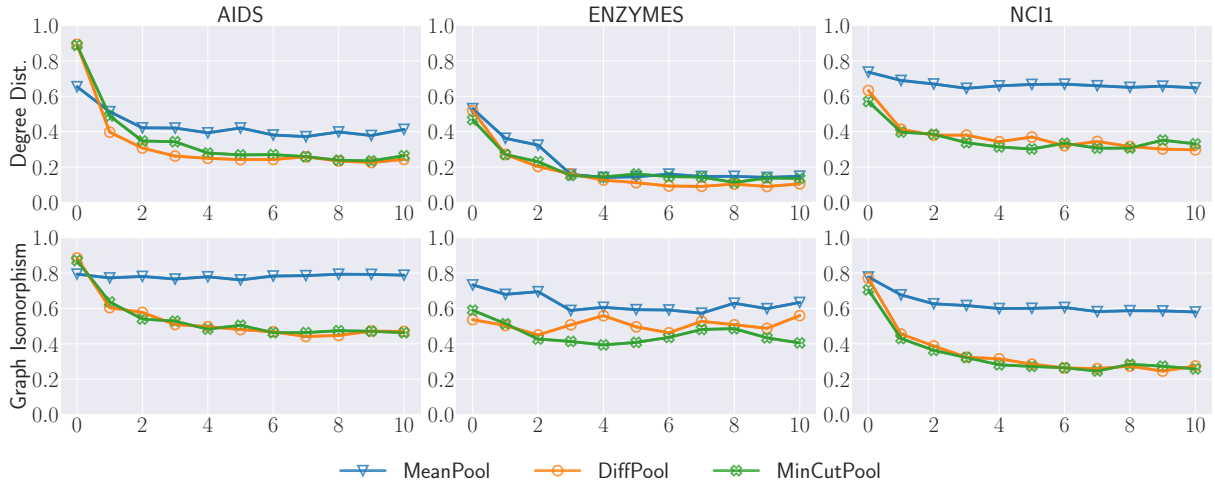

Figure 17: Graph embedding perturbation defense against the graph reconstruction attack. In each figure, the x-axis stands for the scaling parameter β for Laplace noise, where larger β means higher noise level. The y-axis stands for the cosine similarity of degree distribution and graph isomorphism, respectively.

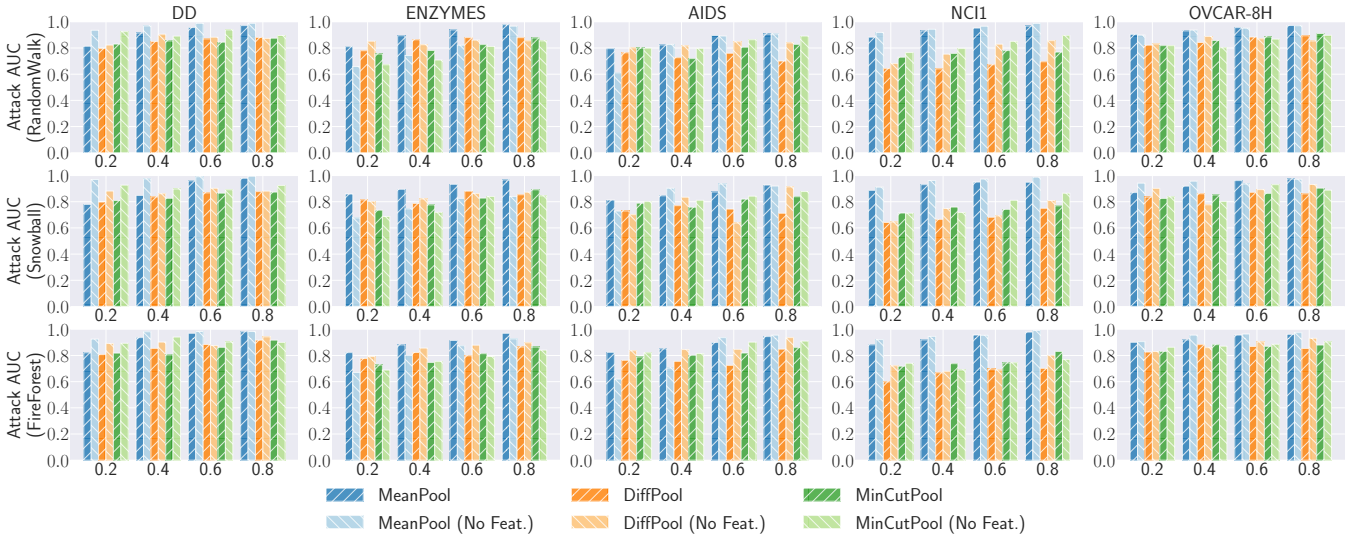

Figure 18: Comparison of attack AUC between graphs with and without node features for subgraph inference attack.