Finding MNEMON: Reviving Memories of Node Embeddings

Yun Shen*
NetApp

Yufei Han
Inria

Zhikun Zhang
CISPA Helmholtz Center for Information Security

Min Chen
CISPA Helmholtz Center for Information Security

Ting Yu
QCRI

Michael Backes
CISPA Helmholtz Center for Information Security

Yang Zhang
CISPA Helmholtz Center for Information Security

Gianluca Stringhini
Boston University

ABSTRACT

Previous security research efforts orbiting around graphs have been exclusively focusing on either (de-)anonymizing the graphs or understanding the security and privacy issues of graph neural networks. Little attention has been paid to understand the privacy risks of integrating the output from graph embedding models (e.g., node embeddings) with complex downstream machine learning pipelines. In this paper, we fill this gap and propose a novel model-agnostic graph recovery attack that exploits the implicit graph structural information preserved in the embeddings of graph nodes. We show that an adversary can recover edges with decent accuracy by only gaining access to the node embedding matrix of the original graph without interactions with the node embedding models. We demonstrate the effectiveness and applicability of our graph recovery attack through extensive experiments.

CCS CONCEPTS

• **Security and privacy**; • **Computing methodologies** → **Machine learning**;

KEYWORDS

Machine Learning Security and Privacy; Graph Embedding

ACM Reference Format:
Yun Shen, Yufei Han, Zhikun Zhang, Min Chen, Ting Yu, Michael Backes, Yang Zhang, Gianluca Stringhini. 2022. Finding MNEMON: Reviving Memories of Node Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3559358

*Work partially done while the author was with NortonLifeLock.

1 INTRODUCTION

Many complex systems can be represented as graphs, e.g., social networks, communication networks, function call graphs, biomedical graphs, and the World Wide Web [37, 44, 58]. Graph embedding algorithms [6, 21, 79] have been long researched to obtain graph representations to concisely represent these networks in low dimensional Euclidean vectors. Upon such transformation, these embedding vectors can make graph analytics tasks efficient and facilitate numerous solutions to real world problems, e.g., node classification [71], community detection [50], link prediction/recommendation [45], binary similarity detection [20, 85, 88], malware detection [18, 54], fraud detection [72], and bot detection [2].

It is well recognized that graphs contain sensitive and private information about the nodes (e.g., node attributes, the relationships among the nodes, etc.). Previous security research efforts orbiting around graphs have been focusing on either (de-)anonymizing the graphs [33, 47, 87] or understanding the security and privacy issues of graph neural networks [11, 28, 64, 68, 73, 74, 82, 83]. Specifically, graph anonymization methods [33, 47, 87] perturb the original graph data to protect users' privacy while preserving as much data utility as possible. In contrast, graph de-anonymization methods focus on unveiling sensitive private information from graphs. In recent years, inspired by the membership inference attack [10, 65], we have witnessed several successful link re-identification attacks against graph neural networks that extract private links contained in the training data via these GNN models [28, 73, 81, 82]. Note that the node embeddings are not privacy preserving by design. Yet, they are pervasively used in many graph analytics tasks as aforementioned. To our surprise, understanding and quantifying the privacy risks of integrating them with the complex ML pipeline in a model-agnostic setting remains unexplored.

In this paper, we fill this gap and quantify the privacy risks of integrating node embeddings with downstream data analytics/machine learning pipelines. Our attack's application scenarios (see Section 3.2) lie in the complex ML systems where raw graph data is part of the learning process but cannot be directly obtained by the attackers due to data segregation policy and/or privacy policy. Instead, the attackers only gain access to the transformed graph data (i.e., the node embeddings of the original graph). They cannot interact with the node embedding models since such pipelines usually operate in one direction. For instance, the data holder may have integrated with the malicious machine learning solution providers (i.e., MLaaS providers) from the AWS Marketplace [49, 67], or the data holder is part of a vertical federated learning environment in an enterprise [73]. In both cases, the node embeddings are part of the learning process and can be obtained by either the malicious MLaaS providers [49, 67] or the insiders [73] in the pipeline.

Concretely, our study addresses two research questions - *can we recover the edges with decent accuracy from the node embedding matrix* and *can we recover a graph structure that is similar to the original graph with respect to the graph properties?* - without knowledge of and the interactions with the node embedding models. These two research questions were discussed in the link re-identification attacks [16, 28, 73]. They, however, follow the adversarial machine learning methodology and assume the interaction with the target model using shadow datasets and the supervision information from the feedback. Our attack does not assume such capabilities (see Section 3), which is more practical in the real world.

Our Contributions. In this paper, we propose MNEMON - a joint graph metric learning and self-supervised learning based graph recovery attack - to tackle these two questions. MNEMON first leverages the background information (i.e., the origin of the node embedding matrix) to estimate the average node degree. It then uses graph metric learning with a multi-head attention mechanism to construct a data specific distance metric from a given node embedding matrix. Coupling with graph metric learning, MNEMON employs graph autoencoder framework to iteratively optimize a graph structure through self-supervised graph regularization (i.e., the learning objectives are generated from the data itself). Upon the termination of the process, the learned graph structure constitutes the recovered graph from the node embedding matrix. We stress that our goal is not perfectly recovering a graph from its node embedding matrix. Rather, we focus on understanding and quantifying the privacy risks of integrating them with the complex ML pipeline. A successful graph recovery attack can lead to severe consequences. For instance, in the context of social networks, MNEMON allows an adversary to gain direct knowledge of sensitive and private social relationships. Also, certain graph data is often expensive to obtain (e.g., protein interaction networks collected from lab studies). MNEMON can pose a direct threat to the intellectual property of the data holder as well. In summary, we make the following contributions.

- We propose a novel model-agnostic graph recovery attack that exploits the implicit graph structural information preserved in the node embedding vectors. We show that the attacker can unveil the private and sensitive graph structural information with decent accuracy from the node embeddings.
- We systematically define the threat model to characterize an adversary's background knowledge and realistic application scenarios. Extensive evaluation of four popular node embedding models using four benchmark graph datasets demonstrates the efficacy of our attacks.
- We discuss a preliminary mitigation mechanism to defend against the graph recovery attack. Our results demonstrate that MNEMON could be partially mitigated with some utility trade-off.

Table 1: Summary of the notations. We use lowercase letters to denote scalars, bold lowercase letters to denote vectors and bold uppercase letters to denote matrices.

Notation	Description
$\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{X})$	graph (network)
$\mathbf{G}_O$ / $\mathbf{G}_R$	original/recovered graph
n	number of nodes
$\mathbf{A} \in \mathbb{R}^{n\times n}$	(weighted) adjacency matrix
$\mathbf{X}$	node features
v, u	node
d	dimension of node embeddings
$\mathbf{H} \in \mathbb{R}^{n\times d}$	node embedding matrix
$\mathbf{h}_v$	node embedding of node v
t	t-th iteration
k	(estimated) average node degree
f	node embedding model
ϕ	learnable embedding distance function
$\mathcal{L}$	loss function

2 PRELIMINARIES

2.1 Notations

We denote an undirected, attributed graph as $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{X})$, where $\mathbf{V} = \{v_i\}_{i=1}^n$ represents the nodes, $\mathbf{E} \subseteq \{(v, u)|v, u \in \mathbf{V}\}$ denotes the edges, and $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^n$ denotes the node features, where $\mathbf{x}_i$ represents the node feature of v_i. $|\mathbf{E}|$ denotes the graph size (i.e., the number of edges). The original and the recovered graphs are denoted as $\mathbf{G}_O$ and $\mathbf{G}_R$ respectively. Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ represent the (weighted) adjacency matrix. As such, $\mathbf{G}$ can also be represented as $\mathbf{G} = (\mathbf{A}, \mathbf{X})$. The notations introduced here and used in the following sections are summarized in Table 1.

2.2 Node Embedding

Definition. In this paper, we focus on *node embedding*, which plays a central role in graph embedding techniques. As the name suggests, a node embedding model f maps nodes to d-dimensional vectors that capture their structural properties and node features (if available). Formally, a node embedding model is defined as $f : \mathbf{G} \to \mathbf{H}$, where $\mathbf{H} \in \mathbb{R}^{n\times d}$ represents node embedding matrix where d denotes the dimension of the embeddings ($d \ll n$) and $\mathbf{h}_v \in \mathbf{H}$ denotes the node embedding vector of node v. The node embeddings of connected nodes maintain "approximate closeness" to each other in the latent space (e.g., $\mathbf{h}_v$ and $\mathbf{h}_u$ should be close in the Euclidean space if v and u are connected in the graph).

Overview. There exists abundant previous work on node embedding models [6, 21, 79]. Broadly speaking, these techniques can be grouped into two categories - matrix factorization based approaches and deep learning based approaches.

- *Matrix factorization based approaches.* The essence of these approaches is treating node embedding as a dimensionality reduction problem and factorizing graph adjacency matrix or node proximity/similarity matrix to obtain node embedding [34]. The core idea of these approaches is that the graph property to be preserved can be interpreted as pairwise node similarities or node proximity in a low dimensional space by matrix factorization. In general, matrix factorization methods can be classified into two categories - node proximity matrix factorization and graph Laplacian eigenmaps factorization.

- *Deep learning (DL) based approaches.* The pioneer DL-based approaches include DeepWalk [55], Node2Vec [22], and their variants. These approaches first generate a set of truncated random walk paths sampled from a graph, then apply deep learning techniques (e.g., SkipGram) to the sampled paths, consequently learning node embeddings. In recent years, we also witnessed the rise of Graph Neural Networks (GNNs). These GNN models are a type of Neural Network which directly operates on the Graph structure via message passing between the nodes of graphs, and encoding the nodes into a low dimensional space (e.g., GCN [38], GraphSAGE [25]). They can take node features into consideration and do not need random walk paths.

We refer the audience to [6, 21, 79] for the overview of node embedding techniques and other graph tasks (e.g., graph-level embedding, graph-level classification, etc.).

3 THREAT MODEL

3.1 Attack Setting

We frame our attack in a *model agnostic* setting. We assume that the adversary only has access to the *node embedding matrix* **H** together with the *background information* of the origin from which the embedding matrix was leaked (see Section 3.2 for detailed application scenario discussion). The attackers do not have any knowledge of the node embedding model, and they cannot tamper with its internals (e.g., model parameters, model architecture). We strictly require that the attackers cannot interact with the target model, and do not have the auxiliary data to train a shadow model using the feedback from the target model.

Remarks. It is important to note that our attack setting is different from the existing adversarial machine learning settings, whereas the interaction with the target model (i.e., querying the target model via publicly accessible API) and the availability of auxiliary data (e.g., nodes with features and labels, etc.) are indispensable. Our attack, however, assumes neither. That is, *we strictly require that the attackers cannot interact with the target model, and do not have the auxiliary data to query a target model and use the query results to train a shadow model.* In other words, this setting eliminates the supervise information from the target model and consequently renders the previous link re-identification attacks inapplicable, hence the novelty of our attack. We provide a detailed discussion in Section 7 to distinguish our attack from the existing ones.

3.2 Attack Scenarios

We consider our attack's application scenarios lie in those complex ML systems where graph data is part of the learning process but cannot be directly obtained by the attackers due to data segregation policy and/or privacy policy. As such, we discuss three real world scenarios below.

- The first attack scenario is the insider threat in a complex enterprise ML environment. In this scenario, a company enforces rigid data protection and segregation policies to guard the security of raw data. As a result, one department may have sensitive user private profile and relationship information (i.e., graph data with node features), and another department has the user purchase history. To train a joint model (e.g., a personalized recommender system) that leverages the data from different departments, the company needs to perform vertical federated learning [77]. Instead of supplying the graph data to the central model, the department that holds the graph data may generate node embeddings that preserve the utility (i.e., user closeness without disclosing the exact edges) and facilitate the learning task. The insider then obtains the node embeddings during this learning process and leaks them to the attackers. This attack scenario is in line with the setting recently discussed by Wu et al. [73].
- The second attack scenario is the malicious third-party provider that is already part of the data holder's data analytics or machine learning pipeline. For example, the data holder may have integrated with the malicious machine learning solution providers (i.e., MLaas providers) from the AWS Marketplace. In this case, the upstream data holder, without knowing the implications, passes the node embedding matrix to the rouge provider for downstream analytical tasks, such as data visualization, link prediction, node classification, profiling, etc. The attackers can then obtain the node embedding matrix from the data holder through the rouge provider. This attack scenario is in line with the malicious machine learning provider scenario discussed by Song et al. [67] and Malekzadeh et al. [49].
- The third attack scenario is security misconfiguration in the ML environment. For instance, researchers may leverage the free computing resources (e.g., GPUs) offered by Colab, and connect it to their private Github repository. Due to such misconfigurations, the notebooks containing the node embeddings are leaked (i.e., wrongly using "anyone on the Internet with this link can view" instead of "send to the specific users"). This attack scenario is in line with the real world misconfigured S3 buckets leakage discussed by Continella et al. [14].

Background Information Acquisition. Besides, given the first two attack scenarios, the attackers can easily obtain the background information of the origin of the embedding matrix (e.g., from which companies the matrices come from). With fair reconnaissance efforts (e.g., correlating the owner of Colab notebooks with Github handles), the attacker may also infer the origin of the embedding matrix in the third scenario. In summary, these three attack scenarios are tangible and match our attack setting.

3.3 Attack Goals

The primary goal of the attackers is *uncovering the edges with decent accuracy from the node embedding matrix.* Attaining this goal would enable the attacker to expose private and sensitive relationships among the nodes rather than the "approximate closeness" offered by the node embeddings (see Section 2). Nevertheless, due to the strict attack setting, it is impractical for the attackers to faultlessly retrieve all the edges from the node embedding matrix. As a result, the secondary goal of the attackers is *recovering a graph structure* $\mathbf{A}_R$ *that is similar to the original graph* $\mathbf{A}_O$ *with respect to the graph properties.* Achieving this goal would enable the attackers to gain additional knowledge of the original graph as a whole and perform graph mining tasks, which in turn violates the intellectual property of the data holder or can facilitate advanced attacks, such as re-identifying individuals [33], structural data de-anonymization [32], etc. For example, recovering a graph with similar triangle counts and joint degree distribution to the original graph would enable the attacker to gain insights into the underlying user engagement in a social network. This information itself is sensitive and proprietary.

Non-goals. Recall our attack setting in Section 3.1 that the attackers only have the node embedding matrix and the background information of the origin of the embedding matrix, and cannot interact with the target model with auxiliary data. We thereby cannot infer node features (i.e., attribute inference attack) since we do not have any auxiliary data (i.e., we do not know the format of the original node features). Similarly, we cannot steal the target model (i.e., model extraction attack) nor can we understand the privacy leakage from the target model itself as we do not interact with it. Finally, our attack focuses on the node-level embeddings. We thus do not attack the graph-level embeddings [6, 21, 79].

4 MNEMON: GRAPH RECOVERY ATTACK

4.1 Attack Overview

At a high level, MNEMON contains three main components.

- The first component (see Section 4.2) leverages the background information (i.e., the origin of the node embedding matrix) to estimate the average node degree. The goal is to estimate a rough average node degree k and the graph size (i.e., $|\mathbf{E}| = \frac{k \times n}{2}$).
- The second component (see Section 4.3) leverages graph metric learning (GML) to learn a data-specific distance function since it is often difficult to choose a standard metric that fits all the datasets. The goal is to learn multi-head attention weights and tailor the distance function on a per node embedding matrix basis.
- The third component (see Section 4.4) learns a graph structure through Graph AutoEncoder (GAE) framework using self supervised graph regularization. The goal is to optimize the graph structure and reduce the false positive edges incurred by the learned metric from the second component.

We iteratively optimize the second and third components as they are inter-connected. MNEMON's workflow is outlined in Figure 1. Specifically, GML learns a distance function to measure the closeness of two nodes and builds the input graph for GAE ($T = t$ in Figure 1). GAE then learns to reconstruct this input graph. If GAE finds certain parts of the input graph are hard to reconstruct, which is reflected by the self supervised graph learning loss, it may be due to the input graph built by GML partially capturing the graph structure. We then merge the graph structures by combining both the input graph and output graph of GAE, which enables us to retain the most confident edges (the transition from $T = t$ to $T = t + 1$ in Figure 1). The combined graph is then used to guide GML to update its metric learning process in the next iteration. In the following sections, we discuss the technical details of our attack.

4.2 Estimate the Average Node Degree

The only clue that the attackers have is the background information about the origin of the node embedding matrix. For instance, in our first attack scenario where the node embedding matrix is leaked by an insider, it is trivial for the attackers to obtain such background information. The attacker's immediate task is thereby estimating the average node degree k. The rationale is straightforward. The

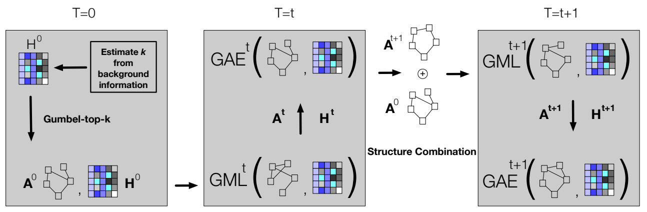


Figure 1: Overview of MNEMON**. At timestamp $T = 0$,** MNEMON **estimates the average node degree k and initializes the seed graph using Gumbel-Top-k trick. At timestamp $T = t$, it iteratively learns a data-specific graph distance metric using GML and optimizes a graph structure in a self-supervised way using GAE.**

attackers already know the number of nodes from the embedding matrix (i.e., n). Yet, due to the combinatorial nature of the graph, there exist $n(n-1)^2$ possible edges. As such, if the attacker can estimate the average node degree k, they can obtain the estimated size of the original graph (i.e., the number of edges) which is equivalent to $(k \times n)/2$. In this way, the estimated graph size enables them to effectively learn the graph structure (see Section 4.3 and Section 4.4).

Abundant previous work [15, 26, 27, 40, 43] has already exemplified that the graphs of similar origins may share similar graph properties (e.g., node degree, graph density, small world phenomenon, local clustering coefficient, etc.). Our core idea is that the attackers can estimate the average node degree from the graphs of similar origins and transfer the estimated node degree from these graphs to facilitate the attack. This alleviates the attackers from stealing a sample training data from the data holder, which, in turn, makes our attack realistic. For instance, if the attackers know that the node embeddings come from a Facebook network, they can leverage graph sampling methods to sample Facebook networks publicly available in the Network Data Repository [61] and estimate the average node degree of the network that they target (see Section 5 for how we use graph sampling to sample real world data). These graph sampling methods have been proven accurate in estimating the average node degree [62]. In this paper, we use the state-of-the-art spikyball sampling [43] implemented in the latest Little Ball of Fur python library [62] to estimate the average node degree. Additional details can be found in Section 5.2.

Notes. The graph sampling process does not interact with the original node embedding models. It also does not need the supervision information from the target models as required by previous research [16, 28, 73]. We also stress that MNEMON does not estimate or require the precise average node degree. MNEMON can accommodate the inevitable estimation error. We provide a detailed study in Section 5.5 to illustrate this capability. For instance, we show that our attack can still achieve good performance even when the estimated average node degree is twice the real average node degree (see Section 5.5) thanks to graph metric learning (see Section 4.3) and self-supervised graph structure learning (see Section 4.4).

4.3 Graph Metric Learning

Upon estimating the average node degree, the common approach to recover a graph from the node embedding matrix is using kNN algorithm. kNN builds a graph in which two nodes v and u are connected by an edge if the distance between the embedding vectors h_v and h_u is among the k-th smallest distances. The drawback of kNN algorithm is that it requires a manually predefined distance function for neighbor selection. However, it is often difficult to choose a standard metric that fits all the datasets and tasks of interest. Take a barbell graph for example, which consists of two dense cliques connected by a long chain. Reflected in the latent space, the node embedding vectors from two dense cliques are close to each other (i.e., dense regions), while those from the long chain are relatively farther to each other (i.e., sparse regions). A standard distance function, such as Euclidean or cosine distance, used by kNN may not recover the edges from the long chain as the distances among them are inevitably large. Yet, they are equally connected from a graph perspective. As such, we propose to leverage graph metric learning in the node embedding space to learn a data-specific distance function and automatically adjust for both the dense and sparse regions in the node embedding matrix.

Graph Initialization with Gumbel-Top-*k* Trick. We follow the approach discussed by Kazi [36] to initialize a seed graph. We first generate a fully connected graph with edge normalized distance score using Equation 1.

$$p_{vu} = e^{-\tau\delta(h_v,h_u)}, \forall v,u \in \mathbf{V} \tag{1}$$

Here δ is a distance function and τ is a temperature parameter controlling the smoothness of the distance scores between node embedding vectors. Instead of using the Euclidean distance function adopted by Kazi [36], we opt for cosine distance as the distance function δ, i.e., $\delta(h_v,h_u) = 1 - cos(h_v,h_u)$. Note that most of the node embeddings are normalized to facilitate downstream tasks. In this case, Euclidean distance is proportional to cosine distance, given a well normalized value range of the node embeddings. When node embeddings are not normalized, our framework can also be adjusted to Euclidean distance. Let $\mathbf{P} = \{p_{vu}\}$ denote the edge probability matrix. We then leverage Gumbel-Top-k trick [41] to sample from $\mathbf{P}$, which generalizes Gumbel-Max trick [24] to draw an ordered sample of size k without replacement from a categorical distribution by taking the indices of the k largest perturbed log-probabilities. That is, we perturb each p_{vu} by adding a Gumbel random variate $\vartheta_{vu} \sim Gumbel(0,1)$. We then select the indices of the k largest perturbed log-probabilities without replacement. This process makes the sampling a stochastic relaxation of kNN [36]. This sampled adjacency matrix (denoted as $\mathbf{A}^0$) constitutes our seed graph structure. Note that δ is used for sampling purposes only and is not part of our learning targets. This corresponds to $T = 0$ in Figure 1.

Learnable Distance Function (ϕ). Due to the stochastic nature of Gumbel-Top-k trick, we inevitably obtain an initial noisy graph structure from the above graph initialization process. That is, an edge (v,u) in $\mathbf{A}^0$ may not exist in the original graph $\mathbf{G}_O$, i.e., a false positive edge. To reduce such false positives, we propose a learnable distance function ϕ to learn a better graph structure. The core idea is that, instead of using a predefined distance function, we leverage metric learning [75] to learn a distance metric for the input space of data (i.e., the node embedding matrix $\mathbf{H}$) from the adjacency matrix $\mathbf{A}$ that preserves the node relationships (i.e., $\mathbf{A}$ is used to supervise the distance learning). In this paper, we adopt a weighted cosine distance (defined in Equation 2) [8, 86] as our learnable distance function ϕ.

$$\phi(\mathbf{h}_v,\mathbf{h}_u) = 1 - cos(\mathbf{w} \circ \mathbf{h}_v, \mathbf{w} \circ \mathbf{h}_u) \tag{2}$$

Here $\mathbf{w}$ is a learnable weight vector that is the same dimension as $\mathbf{h}_v$ and $\mathbf{h}_u$, and $\circ$ denotes the Hadamard product. Following the procedure discussed in [12, 70], we further extend Equation 2 to a multi-head version as in Equation 3 to increase the expressiveness and stablize the learning process.

$$\phi(\mathbf{h}_v,\mathbf{h}_u) = 1 - \frac{1}{m}\sum_{i=1}^{m} cos(\mathbf{w}_i \circ \mathbf{h}_v, \mathbf{w}_i \circ \mathbf{h}_u) \tag{3}$$

Here m refers to the number of attention heads. In this way, we can learn the distance function from multiple perspectives. Note that all node embeddings share the same metric parameters $\mathbf{W} = \{\mathbf{w}_i\}_{i=1}^m$.

Graph Sparsification. We plug ϕ into Equation 1 (i.e., replacing δ) and use the aforementioned Gumbel-Top-k-based sampling trick to extract an adjacency matrix. Our graph sparsification method is different from the ϵ-neighborhood approach used by [12] which cannot easily control the graph size (i.e., ϵ is fixed and may lead to different graph sizes as the learned weighted adjacency matrix also evolves during the learning process).

4.4 Self-Supervised Graph Structure Learning

Having introduced how we apply the graph metric learning technique to tune a data-specific distance function in the previous section, we move on to discuss how we optimize a graph structure and learn the graph distance metric jointly via self-supervised learning. The core idea is that we refine the initial noisy graph structure through self-supervised graph regularization. To this end, we propose to use Graph AutoEncoder [39] (GAE) with an adaptive graph structure combination mechanism to iteratively refine the graph structure learned from the node embedding matrix.

Graph Autoencoder (GAE). Given the adjacency matrix roughly estimated using the multi-headed distance metric in Eq.3 and the node embedding vectors as the input, GAE learns to refine the adjacency matrix as the output. The initial input of our GAE is $\mathbf{G}^0 = (\mathbf{A}^0, \mathbf{H}^0)$. Here $\mathbf{H}^0$ represents the node embedding matrix obtained by the attackers. $\mathbf{A}^0$ represents the initialized seed graph. In this way, we treat $\mathbf{H}^0$ as the node features $\mathbf{X}$ of $\mathbf{G}^0$. Note that we add a superscript for ease of description of the following iterative learning process.

- *Encoder.* The encoder is a Z-layer graph convolutional network (GCN) [38]. At the t-th iteration, its input is a graph $\mathbf{G}^t = (\mathbf{A}^t, \mathbf{H}^t)$. The encoder (see Equation 4) learns a latent representation $\mathbf{H}^{t+1} \in \mathbb{R}^{n\times d}$ where each row represents a node v's latent representation after encoding.

$$\mathbf{H}^{t+1} = GCN(\mathbf{A}^t, \mathbf{H}^t) \tag{4}$$

- *Decoder.* We use an inner-product decoder in this paper [38]. The adjacency matrix can be reconstructed using Equation 5, where

Algorithm 1: Graph recovery attack MNEMON

```
Input   : Node embedding matrix $\mathbf{H}^0$, background
          information $B$, maximum iteration $T$,
          hyperparameters $\tau, \alpha, \beta, \eta, m$
Output  : Learned graph structure $\mathbf{A}^T$ (i.e., $\mathbf{G}_R$)
1  $k \leftarrow EstimateAvgDegree(B)$
2  $\mathbf{A}^0 \leftarrow$ Apply Gumbel-Top-$k$ trick on the fully connected
     probabilistic graph $\mathbf{P}$ with $\tau$ (Equation 1) to generate the
     initial seed graph
3  for $t \leftarrow 0$ to $T-1$ do
4      $\mathbf{A}^t \leftarrow GML(\mathbf{A}^t, \mathbf{H}^t, m)$
5      $\mathbf{A}^{t+1}, \mathbf{H}^{t+1} \leftarrow GAE(\mathbf{A}^t, \mathbf{H}^t)$                    }
6      $\mathcal{L} \leftarrow \mathcal{L}_{lap}(\mathbf{A}^{t+1}, \mathbf{H}^0) + \mathcal{L}_{spa}(\mathbf{A}^{t+1}, \alpha, \beta)$  }
7            $+ \mathcal{L}_{rec}(\mathbf{A}^{t+1}, \mathbf{A}^t)$                          } GAE
8      Backpropagate $\mathcal{L}$                                   }
9      $\mathbf{A}^{t+1} \leftarrow Combine(\mathbf{A}^0, \mathbf{A}^{t+1}, \eta)$
10     $\mathbf{A}^{t+1} \leftarrow Binarize(\mathbf{A}^{t+1})$
11 end
12
```

$\sigma(x) = 1/(1+e^{-x})$ and the output $\mathbf{A}^{t+1}$ is a weighted adjacency matrix.

$$\mathbf{A}^{t+1} = \sigma(\mathbf{H}^{t+1}\mathbf{H}^{t+1^T}) \tag{5}$$

Note that GAE is a generic framework. We follow the design by Kipf et al. [39] and use GCN as the encoder and inner-product as the decoder. This design allows us to use linear GCN [63] to accelerate the computation and compare to our baseline [16] in Section 5. The adversary can plug in other GNN models into GAE framework. The audience can use different architectures as encoders and decoders.

Self-Supervised Graph Regularization. MNEMON cannot interact with the target model (i.e., the node embedding model). We therefore rely on several graph regularization objectives to guide the above GAE-based learning process in a self-supervised way.

- Graph Laplacian regularization ($\mathcal{L}_{lap}$) [4]. A graph Laplacian regularization assumes that the learned weighted adjacency matrix is smooth with respect to a set of node features. In our case, the weighted adjacency matrix is $\mathbf{A}^{t+1}$ and the set of node features is the node embedding matrix $\mathbf{H}^0$. Note that our goal is to optimize the graph structure (i.e., $\mathbf{A}^{t+1}$). As we can see in Equation 6, we stress that we always force that the learned weighted adjacency matrix $\mathbf{A}^{t+1}$ is smooth with respect to the initial node embedding matrix $\mathbf{H}^0$. As such, graph Laplacian regularization can be interpreted that two connected nodes in the learned graph structure should be close enough in the latent node embedding space defined by $\mathbf{H}^0$.

$$\mathcal{L}_{lap}(\mathbf{A}^{t+1}, \mathbf{H}^0) = \frac{1}{2n^2}\sum_{v,u}\mathbf{A}^{t+1}_{vu}\|h^0_v - h^0_u\| = \frac{1}{2}\mathbf{tr}(\mathbf{H}^{0^T}\mathbf{L}^{t+1}\mathbf{H}^0) \tag{6}$$

where $\mathbf{tr}$ denotes trace of matrix, $\mathbf{L}^{t+1} = \mathbf{D}^{t+1} - \mathbf{A}^{t+1}$ and $\mathbf{D}^{t+1} = \sum_v \mathbf{A}^{t+1}_{vu}$.

- Graph sparsity regularization ($\mathcal{L}_{spa}$) [35]. In the real world, the graphs are normally sparse. We use graph sparsity regularization proposed by Kalofolias et al. [35] to learn graphs that meet such expectations. As we can see in Equation 7, graph sparsity regularization encourages that each node connects to at least another node in the first term, and penalizes large degrees in the second term naturally arising from the first term. Graph sparsity regularization can be interpreted as using α to force the graph degrees to be positive and β to control the graph sparsity.

$$\mathcal{L}_{spa}(\mathbf{A}^{t+1}, \alpha, \beta) = -\alpha\mathbf{1}^T log(\mathbf{A}^{t+1}\mathbf{1}) + \frac{\beta}{2}\|\mathbf{A}^{t+1}\| \tag{7}$$

where $\alpha > 0$ and $\beta \geq 0$ are two controlling hyperparameters.

- Graph reconstruction loss ($\mathcal{L}_{rec}$) [39]. Graph reconstruction loss forces GAE to learn a latent representation $\mathbf{H}^{t+1}$ to faithfully rebuild the input adjacency matrix $\mathbf{A}^t$. In this paper, we adopt the link prediction as the way to interpret the reconstruction loss [46] and minimize the binary cross entropy loss between negative (i.e., non-existing edges) and positive samples (i.e., existing edges). The loss function can be found in Equation 8.

$$\mathcal{L}_{rec} = \frac{1}{2n^2}\|\mathbf{A}^t \circ log(A^{t+1}) + (\mathbf{1} - \mathbf{A}^t)\circ log(\mathbf{1} - \mathbf{A}^{t+1})\|^2_F \tag{8}$$

where $\circ$ is elementwise product and $\mathbf{1}$ is an all-ones matrix.

To summarize, $\mathcal{L}_{rec}$ forces GAE to learn simultaneously the updated latent representation $\mathbf{H}^{t+1}$ and a graph adjacency matrix $\mathbf{A}^{t+1}$ decoded from $\mathbf{H}^{t+1}$ to faithfully rebuild the input adjacency matrix $\mathbf{A}^t$. $\mathcal{L}_{lap}$ and $\mathcal{L}_{spa}$ makes the learned graph smooth and sparse. Note that all these three supervisory signals are from the data itself.

Learning Objective. Equation 9 summarizes the objective function of the self-supervised graph structure learning.

$$\begin{aligned} \mathcal{L} &= \mathcal{L}_{lap} + \mathcal{L}_{spa} + \mathcal{L}_{rec} \\ \mathbf{W}^*,\ \mathbf{A}^* &= \underset{\mathbf{W},\ \mathbf{A}}{\arg\min}\ \mathcal{L}(\mathbf{W}, \mathbf{A}, \mathbf{H}^0) \end{aligned} \tag{9}$$

By minimizing Equation 9, we can jointly refine the graph structure and the graph metric function ϕ. The learning process is executed with two alternating steps. First, we refine the distance metric ϕ by updating the multi-heads parameters $\mathbf{w}_{i=1...m}$, given the current estimation of the graph structure (Section 4.3). Second, we estimate the graph structure using the current distance metric ϕ (Section 4.4). The two steps are complementary to each other and boost the overall accuracy of graph structure recovery. It is worth noting that all three loss functions are empirically comparable in magnitude in our evaluation. The adversary can weigh the losses to accommodate their specific attack targets in Equation 9.

Adaptive Graph Structure Combination. The learned weighted graph structure $\mathbf{A}^{t+1}$ is then combined with the input graph structure $\mathbf{A}^0$ using Equation 10. This structure combination step can be interpreted as a denoising function to reduce false positive edges incurred by the initial adjacency matrix $\mathbf{A}^0$. That is, GAE learns to reconstruct a graph structure $\mathbf{A}^{t+1}$ given its structure $\mathbf{A}^t$ and node feature $\mathbf{H}^t$. The edges reconstructed with high confidence are likely to appear the original graph. we use η in Equation 10 to control the update rate of $\mathbf{A}^0$ using $\mathbf{A}^{t+1}$, and iteratively filter out the false positive edges from the initial graph structure.

Table 2: Summary of datasets.

Dataset	Category	|V|	|E|	|X|	⌈|E|/|V|⌉	Density
Cora	Citation	2,708	5,429	1,433	4	0.0014
Citeseer	Citation	4,230	5,358	602	3	0.0006
Actor	Co-Occurrence	7,600	33,544	931	9	0.0011
Facebook	Social	4,039	88,234	1,283	43	0.011

$$\mathbf{A}^{t+1} = (1-\eta)\mathbf{A}^{0} + \eta\mathbf{A}^{t+1} \tag{10}$$

Note that $\mathbf{A}^{t+1}$ remains a weighted adjacency matrix after combination. At the end of each iteration, however, we need to obtain the learned graph structure in a binary form to guide graph metric learning in the next iteration. To this end, we first apply an entry-wise clipping function, $clip(x) = min(max(0,x),1)$, to $\mathbf{A}^{t+1}$. We then use the same Bernoulli binarization strategy outlined in [9] to obtain the binary adjacency matrix $\mathbf{A}^{t+1}$. Specifically, we treat each element of the weighted adjacency matrix $\mathbf{A}^{t+1}$ as the parameter of a Bernoulli distribution and sample independently to produce the final binary adjacency matrix.

Summary. We summarize the whole learning process (i.e., Section 4.2, Section 4.3 and Section 4.4) in Algorithm 1.

5 EVALUATION

5.1 Experimental Setup

Datasets. We use 4 public benchmark datasets to evaluate the performance of our graph reconstruction attack, including Cora [78], Citeseer [78], Actor [53], and Facebook [51]. Cora and Citeseer are citation networks with nodes representing publications and edges indicating citations among these publications. Actor is the actor-only induced subgraph of the film-director-actor-writer network used in [53]. Each node corresponds to an actor, and the edge between two nodes denotes co-occurrence on the same Wikipedia page. Facebook is a social network where nodes represent Facebook users and edges are friendships. We use these datasets to verify the efficacy of our attack given graphs with different characteristics (e.g., origin, graph size, density, node feature size, etc.). For example, Facebook is a social network, it has a well known small world phenomenon and tight community structures among the nodes while the other networks are relatively sparse. Statistics of these datasets are summarized in Table 2.

Node Embedding Models (f). We use four popular node embedding models - network embedding as sparse matrix factorization (NetSMF) [56], Deepwalk (abbreviated as DW) [55], Node2Vec (abbreviated as N2V) [22] and graph convolutional network (GCN) [38] - to generate node embeddings for our evaluation. These four node embedding models are representative of the existing node embedding model families. Network embedding as sparse matrix factorization (NetSMF) [56] improves NetMF [57] and represents the state-of-the-art matrix factorization based approach to generate node embeddings. Deepwalk and Node2Vec are two well known shallow neural network-based (i.e., a neural network with one hidden layer) node embedding techniques. Graph convolutional network (GCN) is a widely used deep neural network based approach for graph representation learning. Note that NetMF, Deepwalk, and Node2Vec generate node embedding using graph structural information only, while GCN considers both node feature and graph structure. As such, these models also cover different real world use cases whereas node embeddings can be generated with different inputs. For reproducibility purposes, we outline their details below.

- **NetSMF.** We use the Pytorch implementation by the original authors [7]. The window size of approximate matrix is 10. The number of negative nodes in sampling is 1. We run the path sampling algorithm for 100 iterations.
- **Deepwalk.** We use the DGL implementation of Deepwalk. The learning rate is set to 0.1. The number of negative nodes in sampling is 5. The random walk length is fixed at 80, and we run 10 random walks per node.
- **Node2Vec.** We also use the DGL implementation of Node2Vec. The number of negative nodes in sampling is 5. The random walk length is fixed at 50, and we run 100 random walks per node. p and q are set to 0.25 and 4 respectively by default.
- **Graph Convolutional Network (GCN).** We use the Pytorch Geometric implementation of GCN. Our GCN model consists of 2 layers as suggested by the original authors. For the first hidden layer, we set the hidden unit size to twice the size of input vectors. For the second layer, we set the hidden unit size to the embedding size. We use ReLU as the activation function between layers. Node embeddings are generated using link prediction as the objective function. We train the GCN model for 400 epochs.

For all node embedding models, we set their output embedding size (i.e., d) to 64, 128, and 256 for our evaluation. These sizes are commonly used in the real world practices balancing between the expressiveness of the node embeddings and the computational complexity of the downstream tasks. Besides, we use the largest connected components from all four datasets to accommodate these node embedding models in our evaluation.

Competitors. We implement three baseline methods detailed below for comparison study.

- **Direct Recovery.** This baseline computes the pairwise similarity matrix from the embeddings of the original graph and reconstructs the graph by choosing the top $k \times n/2$ pairs (i.e., edges) of the largest pairwise similarity scores. It is a straightforward attack strategy that can be leveraged by the adversaries since the embeddings of similar nodes should be close in the latent spaces (see Section 2). Note that our implementation of direct recovery is identical to the decoder used by Duddu et al. [16] to reconstruct graphs.
- ***k*NN Graph.** We employ the widely used kNN algorithm (see Section 2) as the second baseline. kNN builds a graph in which two nodes v and u are connected by an edge if the distance between h_v and h_u is among the k-th smallest distances. We use cosine similarity as the distance function.
- **Invert Embedding [9].** We adapt the optimization algorithm (Algorithm 2&3 in Chanpuriya et al. [9]) as our third baseline to recover a graph from the node embeddings. Since the attackers cannot obtain the real eigenvalues from the PPMI matrix in a model agnostic setting, we thereby use a random diagonal eigenvalue matrix together with the node embedding matrix to generate the low-rank approximation matrix. We set the other

hyperparameters as outlined in Chanpuriya et al. [9]. Additional discussion about invert embedding can be found in Section 7.

The graph size (i.e., the number of edges) of all baselines are set to $k \times n/2$. We detail how we estimate k in Section 5.2 and how k influences the graph recovery performance in Section 5.5.

Hyperparamter Configurations. We set the number of attention heads m to 16. The temperature τ, graph sparsity hyperparameters α and β, and the update rate η are set to 1, 0.3, 0.1 and 0.5 respectively. We set the maximum iteration T to 400. We use a linear graph autoencoder (i.e., Z is set to 1) proposed by Salha et al. [63], which is an effective alternative to multilayer GCNs. These hyperparameter values offer consistent performance across different datasets and models in our evaluation.

Evaluation Metrics. Recall that the attackers have two main goals. Their primary goal is uncovering the edges with decent accuracy from the node embedding matrix, and their secondary goal is recovering a graph structure that is similar to the original graph with respect to the graph properties. Bearing them in mind, we use two categories of metrics to evaluate MNEMON's performance.

- **Edge Metrics.** We first use four edge related metrics - precision (P), recall (R), F1 score (F1), and joint degree distribution (JDD) - to measure how MNEMON attains the primary goal. Precision, recall, and F1 are commonly used, and we apply them to measure the overall capability of MNEMON recovering the exact edges. The joint degree distribution is a metric relating to the edge distribution and provides an additional measurement about 1-hop neighborhoods around a node. It examines each pair of connected nodes and notes their respective nodal degrees. It is defined as $P(k_1, k_2) = \mu(k_1, k_2) \times m(k_1, k_2)$, where $\mu(k_1, k_2) = 1$ if $k_1 = k_2$ otherwise 2, and $m(k_1, k_2)$ denotes the number of edges connecting nodes of degree k_1 and k_2. We use SecGraph [31] to calculate the Jaccard similarity among two JDDs. For all edge metrics, values close to 1 are the best.
- **Global Metrics.** We then employ three global metrics - relative Frobenius error, relative triangle error, and relative average clustering coefficient error - to measure how MNEMON achieves its secondary goal. The relative error is defined as the absolute error (i.e., the difference between the measured value and ground truth value) divided by the ground truth value. It gives an indication of how good a measurement is relative to the ground truth value, or in other words, how much the observed value deviates from actual value. We use the relative Frobenius error, which measures the difference between the adjacency matrix $\mathbf{A}_O$ and $\mathbf{A}_R$, i.e., $\|\mathbf{A}_O - \mathbf{A}_R\|_F / \|\mathbf{A}_O\|_F$. Similarly, we count the absolute difference between the number of triangles (respectively average clustering coefficient) of $\mathbf{G}_R$ and that of $\mathbf{G}_O$, then divided by the number of triangles (respectively average clustering coefficient) of $\mathbf{G}_O$ to calculate the relative triangle error (the relative average clustering coefficient error). For all global metrics, values close to 0 are the best. Similar relative error metrics are also used in Chanpuriya et al. [9].

In practice, the audience could potentially leverage Narayanan-Shmatikov's attack [52] (and other appropriate de-anonymization attacks) to measure, to what extent, the recovered graph can assist the graph de-anonymization task given different graphs and different levels of background knowledge.

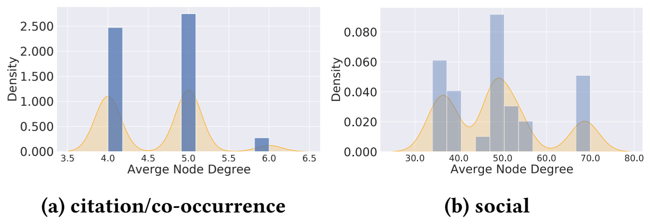


Figure 2: Distribution of estimated average node degrees. The mean and standard deviation of our estimated average node degree of the citation/co-occurrence graphs (Figure 2a) are 4.6 and 0.8. The respective values of social networks (Figure 2b) are 45.7 and 8.4.

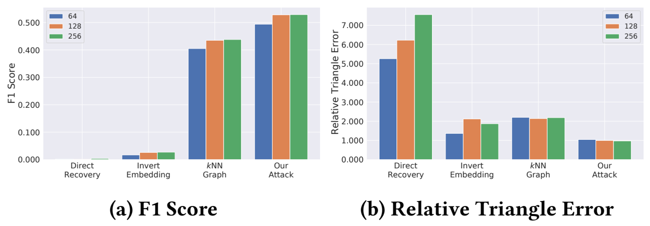


Figure 3: F1 scores and relative triangle error scores of all baseline methods and MNEMON **when given different node embedding sizes (i.e., 64, 128 and 256). We use Node2Vec to generate node embedding matrices.**

Runtime Configuration. All the experiments in this paper are repeated 5 times. For each run, we follow the same experimental setup laid out before. We report the mean and standard deviation of each metric to evaluate the attack performance. In this way, we can delineate objective performance results without reporting opportunistically optimal results.

5.2 How to Estimate the Average Node Degree?

Recall that the only clue that the attackers have is the background information about the origin of the node embedding matrix. In this section, we exemplify how the attackers can estimate the average node degree k from the graphs of similar origins by leveraging state-of-the-art graph sampling methods. Note that the attackers can estimate the graph size (i.e., the number of edges) which equals to $k \times n/2$. We use the state-of-the-art spikyball sampling [60] to estimate the average node degree for our evaluation. It generalizes several exploration-based sampling schemes (e.g., Snowball sampling, Forest Fire sampling, graph-expander sampling etc.), and can be applied to any large graphs due to its flexibility [60].

Specifically, for citation/co-occurrence graphs (e.g., Cora, Citeseer, and Actor), we use the publicly available citation graphs - Pubmed and DBLP - to estimate the average node degree. For each graph, we use spikyball sampling to sample 30% of the whole graph then estimate the average node degree from the sampled graph. This process is repeated 300 times. We calculate the mean average node degree as our final estimation of citation/co-occurrence graphs. For social network graphs (e.g., Facebook), we randomly select six graphs (e.g., socfb-BU10, socfb-Carnegie49, socfb-JMU79, socfb-Lehigh96, socfb-Maine59 and socfb-UCSC68) from the publicly available FB100 dataset [59] plus one Twitter graph [51] from SNAP. We also sample 30% of each graph to estimate the average node degree and repeat this process 300 times per graph. This strategy enables the adversary to sample enough graphs to cover a wide spectrum of graph properties.

The estimation results are shown in Figure 2. What can be seen in Figure 2 is that the estimated average node degrees may not exactly match the real values but are roughly within the same order of magnitude. For instance, the mean and standard deviation of our estimated average node degree of the citation/co-occurrence graphs (Figure 2a) are 4.6 and 0.8, while the mean and standard deviation of our estimated average node degree of social networks are 45.7 and 8.4 respectively. Comparing to the real values in Table 2, the estimated values are not precise. For instance, the above graph sampling process overestimates the average node degree for the Cora and Citeseer datasets, while underestimating the average node degree of the Actor dataset. However, they can offer the attackers a reasonable starting point to estimate the graph sizes. We use these estimated values (i.e., 5 for citation/co-occurrence graphs and 46 for social network graphs) in the rest of our evaluation. We provide a detailed study on how MNEMON can attain good performance even when the estimated average node degree is almost twice the ground truth value in Section 5.5.

Takeaways. When only having the background information about the origin of the node embedding matrix, our sampling process represents a feasible way that the attackers take to estimate the average node degree. The estimated average node degrees are in the vicinity of the ground truth values but not exactly matching them.

5.3 Is MNEMON Better than the Baselines?

In this section, we aim at studying whether MNEMON is effective to recover a graph from the node embedding matrix, or whether the existing baseline methods would be enough for the task at hand. To address this research question, we compare MNEMON to the baseline methods discussed in Section 5.1. All methods use the estimated average node degrees outlined in Section 5.2. The node embedding size is fixed to 256. Due to space limitations, we only report the attack results on the Cora dataset.

Performance. The performance comparison results are shown in Table 3. Overall, direct graph recovery and invert embedding graph recovery cannot recover graphs from the node embedding matrices given all four node embedding models. For instance, the F1 scores of these two methods are no greater than 0.027, indicating that they cannot attain the attacker's primary goal. At the same time, the global metrics of these two baselines are equally underwhelming. Our results show that such optimization based approach is less effective in a model agnostic setting. *k*NN algorithm represents a *de facto* approach to recover the edges from node embeddings. Our results show that *k*NN graph can partially recover the edges from the node embedding matrix. For example, it can recover the edges from the node embeddings generated by Node2Vec with a 0.438 F1 score. As we can see in Table 3, MNEMON outperforms all baseline methods. Take the node embeddings generated by Node2Vec for example, MNEMON achieves 0.529 F1 score, which is 0.091 higher than that of *k*NN graph recovery. In other words, MNEMON's F1 score relatively improves that of *k*NN graph recovery by 0.208 (i.e., 0.091/0.438=0.208). If we take the edges recovered by *k*NN graph as the upper bound of the existing privacy risk assessment, MNEMON empirically improves this upper bound by 0.208 per our evaluation results. Given the combinatorial nature of graph edges (i.e., n(n-1)/2 possibilities) and our strict attack setting (i.e., no interaction with the node embedding models), such 0.208 relative improvement by MNEMON is substantial. Practically speaking, if we position MNEMON in the privacy risk assessment framework, it would lead to 0.208 increase of the estimated privacy loss than the de facto risk assessment using *k*NN algorithm.

Impact of Node Embedding Size. We use two metrics - F1 score and relative triangle error - to understand the impact of node embedding size on both baselines and MNEMON. We use Node2Vec to generate node embedding matrices. The results are shown in Figure 3. It is straightforward to see that MNEMON consistently performs better than the baselines given different node embedding sizes.

Takeaways. The proposed learnable distance function and adaptive graph structure combination can reduce a reasonable amount of false edges. They, in turn, enable MNEMON to recover better graph structure from the node embedding matrix given different node embedding models and embedding sizes. Besides, *k*NN graph remains a viable approach to recover edges from the node embedding matrix. However, due to its non-learning-based nature, *k*NN graph is outperformed by MNEMON.

5.4 How Effective is MNEMON?

In this section, we evaluate MNEMON on all four datasets to understand its overall performance. The results are summarized in Table 4. Due to space limitations, we only show the results when the node embedding size is fixed to 256.

Edge Metrics. Recall that the primary goal of the attackers is uncovering the edges with decent accuracy from the node embedding matrix. We thereby use edge metrics outlined in Section 5.1 to measure MNEMON's performance. Besides, *k*NN graph remains a viable approach to recover edges from the node embedding matrix as we explicate in Section 5.3. We also show the relative improvement scores in Table 4 to demonstrate to what extent MNEMON can relatively improve from *k*NN graph. We add a positive sign (+) next to the relative improvement score to highlight the improvement. As we can see from Table 4, MNEMON can enable the adversary to recover edges from all node embedding matrices generated by all four node embedding models with good precision and recall. Take the Cora dataset and the node embedding matrix generated by NetMF for example. MNEMON achieves 0.579 precision, 0.640 recall and 0.608 F1 scores. These scores relatively improve 0.235, 0.113, and 0.176 from those of *k*NN graph recovery. Similarly, take the Actor dataset and the node embedding matrix generated by Deepwalk for example, MNEMON achieves 0.687 precision, 0.435 recall, and 0.533 F1 scores. These scores relatively improve 0.222, 0.088, and 0.138 from those of *k*NN graph recovery. The other datasets given all node embedding models follow similar patterns. At the same time, MNEMON overwhelmingly outperforms *k*NN graph given joint degree distribution similarity metric. Given the above two

Table 3: Comparison of all baseline methods and MNEMON. We use the Cora dataset and the node embedding size is 256.

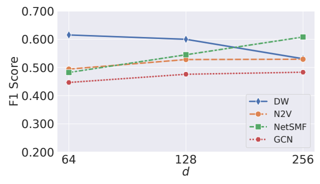
Graph Reocvery Method	f	Edge Metric				Global Metric		
		Precision	Recall	F1	JDD	Frobenius Error	Triangle Error	Clustering Coef. Error
Direct Recovery	DW	0.001±0.000	0.002±0.000	0.001±0.000	0.000±0.000	1.647±0.000	6.867±0.000	0.753±0.000
	N2V	0.003±0.000	0.006±0.000	0.004±0.000	0.000±0.000	1.645±0.000	7.559±0.000	0.759±0.000
	NetSMF	0.013±0.000	0.022±0.000	0.016±0.000	0.311±0.000	1.647±0.000	0.621±0.000	0.228±0.000
	GCN	0.001±0.000	0.002±0.000	0.002±0.000	0.000±0.000	1.647±0.000	6.391±0.000	0.653±0.000
Invert Embedding	DW	0.007±0.005	0.012±0.010	0.009±0.007	0.667±0.092	1.660±0.010	0.866±0.661	0.198±0.008
	N2V	0.021±0.008	0.037±0.014	0.027±0.010	0.665±0.030	1.645±0.008	1.869±0.160	0.148±0.005
	NetSMF	0.003±0.001	0.005±0.003	0.004±0.002	0.462±0.064	1.676±0.007	0.842±0.789	0.221±0.011
	GCN	0.015±0.003	0.026±0.006	0.019±0.004	0.675±0.004	1.657±0.005	0.255±0.158	0.188±0.005
***k*NN Graph**	DW	0.401±0.000	0.492±0.000	0.442±0.000	0.340±0.000	1.114±0.000	3.029±0.000	0.286±0.000
	N2V	0.397±0.000	0.487±0.000	0.438±0.000	0.338±0.000	1.119±0.000	2.185±0.000	0.276±0.000
	NetSMF	0.469±0.000	0.575±0.000	0.517±0.000	0.334±0.000	1.037±0.000	2.379±0.000	0.325±0.000
	GCN	0.378±0.000	0.463±0.000	0.416±0.000	0.333±0.000	1.140±0.000	2.172±0.000	0.286±0.000
MNEMON	DW	0.492±0.004	0.578±0.003	0.531±0.003	0.840±0.011	1.010±0.005	1.118±0.036	0.215±0.006
	N2V	0.506±0.001	0.554±0.003	0.529±0.001	0.724±0.007	0.993±0.001	0.973±0.027	0.228±0.004
	NetSMF	0.579±0.002	0.640±0.003	0.608±0.003	0.732±0.006	0.908±0.003	1.263±0.019	0.288±0.004
	GCN	0.462±0.002	0.506±0.001	0.483±0.001	0.753±0.006	1.040±0.003	0.864±0.032	0.230±0.005

Table 4: The performance results of MNEMON using all four datasets. We fix the node embedding size to 256. We show the relative improvement scores in edge metrics to demonstrate to what extent MNEMON can relatively improve from *k*NN graph. We add a positive sign (+) next to the relative improvement score to highlight the improvement. We also show the relative error reduction scores in global metrics to demonstrate to what extent MNEMON can relatively reduce errors incurred by *k*NN graph. We add a negative sign (-) next to the relative error reduction score to highlight the difference.

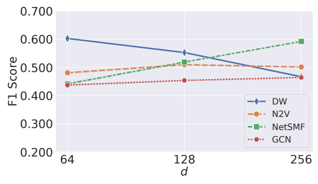
Dataset	f	Edge Metrics				Global Metrics		
		Precision	Recall	F1	JDD	Frobenius Error	Triangle Error	Clustering Coef. Error
Cora	DW	0.492±0.004 (+0.226)	0.578±0.003 (+0.174)	0.531±0.003 (+0.201)	0.840±0.011 (+1.471)	1.010±0.005 (-0.104)	1.118±0.036 (-1.911)	0.215±0.006 (-0.071)
	N2V	0.506±0.001 (+0.276)	0.554±0.003 (+0.137)	0.529±0.001 (+0.208)	0.724±0.007 (+1.143)	0.993±0.001 (-0.126)	0.973±0.027 (-1.212)	0.228±0.004 (-0.048)
	NetSMF	0.579±0.002 (+0.235)	0.640±0.003 (+0.113)	0.608±0.003 (+0.176)	0.732±0.006 (+1.191)	0.908±0.003 (-0.129)	1.263±0.019 (-1.116)	0.288±0.004 (-0.037)
	GCN	0.462±0.002 (+0.223)	0.506±0.001 (+0.092)	0.483±0.001 (+0.162)	0.753±0.006 (+1.260)	1.040±0.003 (-0.100)	0.864±0.032 (-1.308)	0.230±0.005 (-0.056)
Citeseer	DW	0.403±0.002 (+0.193)	0.555±0.005 (+0.149)	0.467±0.003 (+0.174)	0.617±0.011 (+1.635)	1.125±0.003 (-0.085)	1.877±0.075 (-2.651)	0.341±0.009 (-0.080)
	N2V	0.445±0.001 (+0.271)	0.575±0.002 (+0.149)	0.502±0.001 (+0.217)	0.506±0.007 (+1.137)	1.069±0.002 (-0.127)	1.734±0.039 (-1.665)	0.357±0.005 (-0.059)
	NetSMF	0.530±0.002 (+0.229)	0.672±0.001 (+0.091)	0.592±0.001 (+0.168)	0.461±0.005 (+1.048)	0.961±0.002 (-0.133)	2.001±0.056 (-1.419)	0.432±0.003 (-0.056)
	GCN	0.414±0.003 (+0.206)	0.529±0.002 (+0.080)	0.465±0.001 (+0.153)	0.527±0.011 (+1.344)	1.105±0.004 (-0.099)	1.467±0.057 (-1.734)	0.330±0.004 (-0.067)
Actor	DW	0.687±0.001 (+0.222)	0.435±0.002 (+0.088)	0.533±0.002 (+0.138)	0.417±0.001 (+0.587)	0.874±0.001 (-0.081)	0.203±0.009 (-0.105)	0.229±0.002 (-0.017)
	N2V	0.465±0.001 (+0.356)	0.313±0.000 (+0.282)	0.374±0.000 (+0.312)	0.473±0.003 (+0.293)	1.023±0.001 (-0.083)	0.179±0.007 (-0.387)	0.176±0.001 (-0.035)
	NetSMF	0.562±0.002 (+0.240)	0.366±0.001 (+0.136)	0.443±0.001 (+0.179)	0.457±0.003 (+0.406)	0.959±0.001 (-0.074)	0.147±0.013 (-0.758)	0.285±0.002 (-0.025)
	GCN	0.373±0.001 (+0.226)	0.263±0.000 (+0.211)	0.308±0.001 (+0.218)	0.505±0.003 (+0.446)	1.086±0.001 (-0.045)	0.280±0.008 (-0.349)	0.153±0.002 (-0.049)
Facebook	DW	0.441±0.001 (+0.028)	0.471±0.001 (+0.066)	0.456±0.001 (+0.046)	0.519±0.006 (+1.745)	1.061±0.001 (-0.009)	0.494±0.002 (+0.213)	0.077±0.001 (+0.006)
	N2V	0.468±0.000 (+0.018)	0.487±0.001 (+0.026)	0.477±0.001 (+0.022)	0.444±0.002 (+1.581)	1.033±0.001 (-0.007)	0.545±0.001 (+0.499)	0.090±0.001 (+0.050)
	NetSMF	0.454±0.001 (+0.022)	0.502±0.002 (+0.098)	0.476±0.001 (+0.059)	0.457±0.002 (+0.570)	1.050±0.001 (-0.006)	0.424±0.007 (+0.418)	0.081±0.001 (+0.041)
	GCN	0.342±0.001 (+0.061)	0.364±0.001 (+0.100)	0.352±0.001 (+0.078)	0.371±0.004 (+1.026)	1.157±0.001 (-0.012)	0.452±0.002 (+0.380)	0.056±0.001 (-0.031)

examples, JDD similarity scores of MNEMON respectively improve 1.191 and 0.587 from those of *k*NN graph. Our results demonstrate that MNEMON can recover a graph in which each pair of connected nodes share similar 1-hop neighborhood as they are in the original graph.

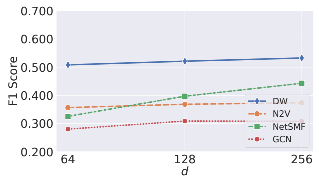
Global Metrics. Recall that the secondary goal of the attackers is recovering a graph structure that is similar to the original graph with respect to the graph properties. We use the global metrics outlined in Section 5.1 to understand MNEMON's performance. Similar to the above edge metrics, we also compare MNEMON to kNN graph. We show the relative error reduction scores in Table 4 to demonstrate to what extent MNEMON can relatively reduce errors incurred by *k*NN graph. We add a negative sign (-) next to the relative error reduction score to highlight the difference between MNEMON and *k*NN graph. As we can see from Table 4, MNEMON can incur relatively low error scores given all three global metrics. Take the Actor dataset and the node embedding matrix generated by GCN for example. MNEMON's relative triangle error is 0.280. This indicates that the graph recovered by MNEMON contains a similar number of triangles to that of the original graph. At the same time, this score reduces the relative error made by *k*NN graph for 0.349. Note that the estimated average node degree (i.e., 5) is larger than the ground truth values of both Cora and Citeseer. This leads to higher relative triangle errors. However, combined with the edge metrics, we can assert that such error is due to a combination of reorientation of the specific edges between the true and the recovery networks, and extra edges incurred by the overestimation of k. Besides, we compare MNEMON with the invert embedding using the overlapping Citeseer dataset and its 256-dimensional NetSMF



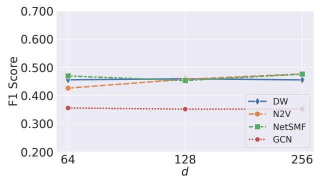
(a) Cora



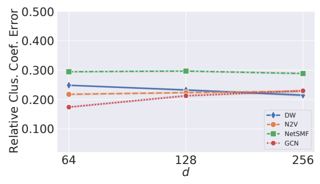
(b) Citeseer



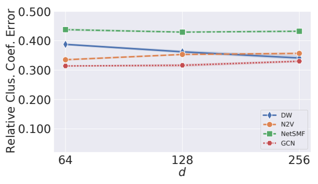
(c) Actor



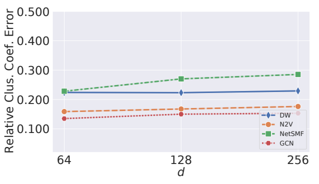
(d) Facebook



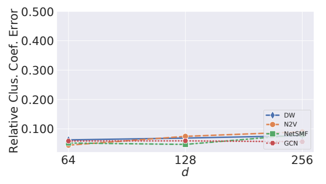
(e) Cora



(f) Citeseer

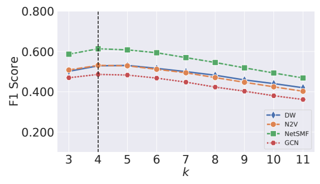


(g) Actor

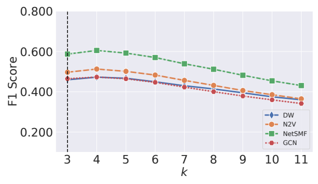


(h) Facebook

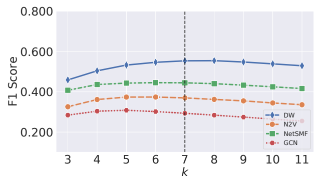
Figure 4: F1 scores and relative average clustering coefficient error scores of MNEMON **given all four datasets. We fix the node embedding size to 256.**



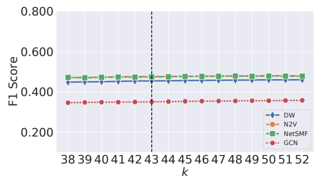
(a) Cora



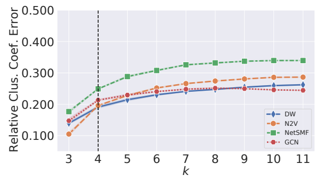
(b) Citeseer



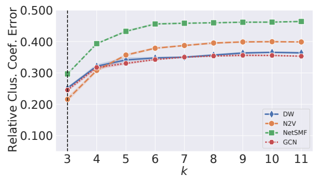
(c) Actor



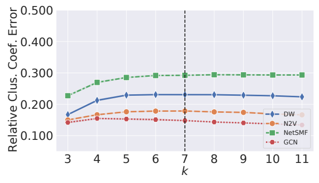
(d) Facebook



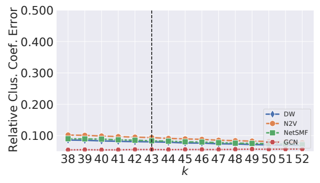
(e) Cora



(f) Citeseer



(g) Actor



(h) Facebook

Figure 5: F1 scores and relative average clustering coefficient error scores of MNEMON **given all four datasets and various average node degrees. We fix the node embedding size to 256. The vertical bar indicates the actual average node degree.**

node embedding matrix. MNEMON can achieve 0.908 relative Frobenius error score which is close to that of the invert embedding (see Figure 4 in [9]). Note that the invert embedding in [9] is under the white box setting while MNEMON is under the black box setting. In summary, our performance results demonstrate that MNEMON can also recover a graph that is structurally similar to the original graph with respect to the global graph properties. Note that the clustering coefficient of a node c_v is defined as $c_v = \frac{2*\mathcal{N}(v)}{deg(v)(deg(v)-1)}$ where $\mathcal{N}(v)$ represents the number of edges between the neighbors of v and $deg(v)$ represents the degree of v. The average clustering coefficient of the whole graph c_G is defined as $c_G = \frac{1}{n}\sum_{v=1}^{n} c_v$. A smaller k may lead to the increasing possibility that the number of triangles recovered from a graph drops closer to 0. We therefore use different relative errors to objectively evaluate the performance from multiple perspectives.

Impact of Node Embedding Size. We use two metrics - F1 score and relative average clustering coefficient error - to understand the impact of node embedding size MNEMON across all four datasets. The results are shown in Figure 4. As we can see in the figure, MNEMON can offer stable graph recovery performance given different embedding sizes and all embedding models. We only observe a marginal F1 score decrease given the Deepwalk node embedding model in Cora and Citeseer datasets. Overall, our results imply that reducing the embedding size (a common defense mechanism) may not work for MNEMON. More details can be found in Section 6.

Stability. We run MNEMON 5 times on a given node embedding matrix. Such runtime configuration enables us to measure how widely all those metric values are dispersed from the average value (i.e., standard deviation). At the same, it eliminates the chance of reporting opportunistically good results. A low standard deviation indicates low volatility. As we can observe in Table 4, the standard deviation values are low in all cases. The results show that MNEMON can recover graphs from node embedding matrices with statistically stable performance.

Table 5: Ablation study on the impact of GML to MNEMON's performance. We use Citeseer dataset and fix the node embedding size to 256.

Method	Precision	Recall	F1
*k*NN	0.338	0.483	0.398
MNEMON w/o GML	0.391	0.511	0.443
MNEMON	0.404	0.557	0.468

Ablation Study. We also carry out an ablation study to understand the impact of GML on the performance of MNEMON. To this end, we customize MNEMON and remove GML from the optimization. Specifically, we initialize a graph using Gumbel-Top-k trick and run GAE once. We use edge metrics in this study and summarize the results in Table 5. We observe that MNEMON performs better given all edge metrics. The results exemplify that jointly optimizing GAE and GML enables us to learn more information from the node embedding matrix and further reduce noise from the recovered graph.

Takeaways. We can observe that MNEMON achieves good performance on all datasets. Such results demonstrate that jointly optimizing the learnable distance function and adaptive graph structure combination is effective for recovering graphs from node embeddings.

5.5 How does k Affect the Attack Performance?

Recall that the attackers use the graph sampling algorithms to estimate the average node degree from the graphs of similar origins and transfer the estimated node degree from these graphs to facilitate the attack (see Section 4.2). We show that the adversary cannot obtain the precise average node degree in Section 5.2. However, the estimated average node degree k directly affects the graph size of the recovered graph $\mathbf{G}_R$. More importantly, our attack uses this estimated k to seed the initial graph using the Gumbel-Top-k trick and iterative graph structure optimization during the learning process. It is therefore essential to study the impact of the estimated average node degree k on the graph recovery performance. To this end, we run MNEMON 5 times on every k that falls within at least one standard deviation of the estimated average node degree in Section 5.2. For instance, the mean and standard deviation of our estimated average node degree of the Facebook dataset are 45.7 and 8.4 respectively. In this case, we run MNEMON 5 times for every value between 38 and 52. We use two metrics - F1 score and relative average clustering coefficient error - to understand the impact of k across all four datasets. Due to space limitations, we only show the attack results when the node embedding size is fixed to 256.

Performance. The results are shown in Figure 5. We show a vertical line in each figure to mark the ground truth value of the average node degree for each dataset (see Table 2). In general, we can see that MNEMON can accommodate the inevitable estimation error. Take the Citeseer dataset and the node embedding matrix generated by NetSMF for example. The ground truth average node degree is 3, and the estimated average node degree is 5. The F1 scores achieved by MNEMON are respectively 0.592 and 0.605. This means that, even though our estimated k is almost twice the ground truth value, MNEMON can still deal with such estimation error and attain good results (i.e., the F1 score difference is only 0.013). Similar to the F1 score metric, we can see that MNEMON can also achieve low relative average clustering coefficient error. Take the Citeseer dataset and the node embedding matrix generated by NetSMF for instance, the relative average clustering coefficient errors of MNEMON are 0.432 and 0.332. The difference is approximately 0.100.

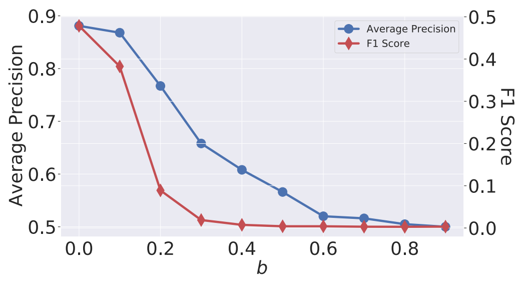


Figure 6: Trade-off between node embedding utility (average link prediction precision) and MNEMON's graph recovery performance (F1 score). We use the Cora dataset and GCN as the node embedding model. The node embedding size is fixed to 256.

Observations. We observe that the node embedding matrices generated by GCN are relatively harder to recover than those by the other three models. Two factors make the graph recovery task difficult. First, GCN considers both node features and graph structure to generate node embeddings while the other three models only use graph structures. Second, we use ReLU as the activation function between layers. This non-linear, element-wise function outputs the input directly if it is positive, otherwise, it outputs zero. It is computationally efficient but leads to sparse representation, making the graph recovery harder.

Takeaways. Our evaluation results show that MNEMON can accommodate the inevitable estimation error of k. The root cause of the moderate decrease (increase) of F1 scores (relative average clustering coefficient error) in Figure 5 is due to the increasing graph size. However, given the real world application scenarios outlined in Section 3.2, we show that the estimated average node degrees can be in the vicinity of the real values as shown in Section 5.2. Combining with the adaptive learning process outlined in Section 4.4, MNEMON remains practical to recover graphs in the wild as exemplified by the results in Figure 5.

6 DEFENSE

In this section, we discuss node embedding perturbation as a tentative defense mechanism and empirically evaluate its effectiveness.

Embedding Perturbation. One possible defense of MNEMON is adding perturbations (i.e., noise) to the original node embeddings $\mathbf{H}_O$. As such, the data holder only passes on a noisy but usable version $\widetilde{\mathbf{H}}_O$ to the ML pipeline. Formally, $\widetilde{\mathbf{H}}_O = \mathbf{H}_O + \Delta(\mu, b)$ where Δ denotes the Laplace distribution, μ is a location parameter and $b > 0$ is a scale parameter. However, adding noise inevitably distorts the information contained in the node embeddings and can lead to utility loss. We therefore focus on evaluating the trade-off between the utility and the defense in this section. Similar defense mechanisms were also discussed in the previous literature [28, 82, 84].

Table 6: Difference between our attack and the close work.

Method	Supervision from Auxiliary data	Shadow model	Interaction w/ target model	Attack setting
Chanpuriya et al. [9]	✓	✗	N/A	whitebox
Link reidentification [16, 28, 73]	✓	✓	✓	blackbox
Zhang et al. [82]	✓	✓	✓	blackbox
MNEMON	✗	✗	✗	model-agnostic

Experimental Setup. We use the Cora dataset and the 256 dimensional node embedding matrix generated by GCN for our evaluation. We fix μ to 0 and choose 10 evenly distributed values between 0 and 1 for b. We use average link prediction precision as the utility metric and F1 score as the attack performance metric.
Results. The results are shown in Figure 6. As we can see in the figure, adding perturbations could work with noticeable utility loss. For instance, when $b = 0.2$, the average link prediction precision using the perturbated node embedding matrix drops from 0.881 to 0.761. In turn, we can see that the MNEMON's F1 score drops from 0.486 to 0.088. This result shows that the data holder might choose the noise level to defend against MNEMON while preserving some utility. However, it is a delicate process. For instance, if the data holder chooses $b = 0.1$, the average link prediction precision drops from 0.881 to 0.868. In this case, MNEMON's F1 score drops from 0.486 to 0.401. In short, our results show that the trade-off is inevitable if using added perturbations to defend against MNEMON. We plan to explore such research direction in the future.
Notes. The node embedding size affects the expressiveness of the node embeddings. As such, another prospective defense mechanism is to reduce the dimension of the node embedding. Its core idea is reducing the knowledge that the attackers can obtain and consequently lessening the capability of the graph recovery attack. However, we show that MNEMON achieves stable graph recovery performance given different embedding sizes and all embedding models in Figure 4. Our results indicate that reducing the embedding size may not work for MNEMON. We plan to expand such research direction in the future.

7 RELATED WORK

Graph Theory Based Graph Restoration. Graph restoration algorithms in the graph theory realm restore a hidden graph by repeatedly querying an oracle for certain types of information about the graph structure [42]. Depending on the algorithm, different types of information can be revealed by the oracle, including node betweenness [1], distance or shortest path between nodes [29], edge counting [5], edge detection [3], etc. The common goal among these research is identifying strategies that recover the graph with low worst-case query complexity. However, those approaches are not learning-based and require the existence of an oracle knowing the structural information of the original graph. They cannot be adapted to reconstruct graphs from the node embeddings.
Graph Completion. Graph completion [13, 23] infer the unobserved part of the network (i.e., missing edges and nodes) given the partially observed network. Link prediction algorithms (see [17, 48] for an overview) have been successfully applied to identify missing edges [69, 80]. Probabilistic and deep learning models [30, 66] have also been investigated to deduce the missing nodes. However, these algorithms require the graphs to be substantially observed and high-quality attributes provided. Our attack assumes neither.
Deep Graph Structure Learning for Robust Representations. This line of research centers on Graph Structure Learning (GSL) that jointly learns an optimized graph structure and corresponding representations [89]. The goal of GSL is to generate node representations robust to noisy graph structures. GSL methods assume the availability of node features, incomplete graph structure, and node labels. Different approaches then leverage metric learning [12], probabilistic modeling [19], direct optimization [76], etc. to learn an adjacency matrix as well as the corresponding node representations. In contrast to GSL, our attack does not assume the availability of node features and node labels. Besides the goal difference, our attack is self-supervised while GSL approaches use node labels to supervise the learning process.
Close Work. To our best knowledge, there exist five pieces of close work to our attack [9, 16, 28, 73, 82]. The closest work is Chanpuriya et al. [9] presenting two optimization algorithms to recover a graph from its node embeddings generated by NetMF [57]. Their algorithms assume the knowledge of the NetMF algorithm (i.e., the target model), window size T, the low-ranking approximation of the finite-T PPMI matrix, and the exact degree of each node, hence a white-box attack against NetMF. Another closely related work is link reidentification attack [16, 28, 73]. In theory, those attacks can be used to reconstruct a graph upon querying the target model n^2 times. However, they train a shadow model using auxiliary data and their posterior scores obtained from the target model. Our attack assumes the attackers can not interact with the target model using auxiliary data, which renders this link stealing attack infeasible in our setting. In addition to link re-identification attacks using node-level information, Zhang et al. [82] also introduce a reconstruction attack to rebuild a graph from its graph-level embedding within the context of graph classification. This attack suffers from the same pitfalls of the link re-identification attacks, and cannot be used in our setting. Our attack is fundamentally different from the existing work by removing the assumptions of the availability of supervision information from auxiliary data, the shadow model, and the interaction with the target model. (see Table 6 for the summary).

8 CONCLUSION

In this paper, we presented a model-agnostic attack that uses the node embedding matrices to recover graphs. Extensive experiments show that an adversary can recover graphs with decent accuracy by only gaining access to the node embeddings of the original graph. Our results highlight the need for the data holders to rethink the privacy implications when integrating node embeddings for downstream analysis, even when the third party has extremely limited knowledge of the data.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their feedback and our shepherd Gergely Acs for his help in improving our paper. This work is partially funded by the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4) and by the National Science Foundation under grant CNS-2127232.

REFERENCES

[1] Mikkel Abrahamsen, Greg Bodwin, Eva Rotenberg, and Morten Stöckel. 2016. Graph Reconstruction with a Betweenness Oracle. In *STACS*.

[2] Seyed Ali Alhosseini, Raad Bin Tareaf, Pejman Najafi, and Christoph Meinel. 2019. Detect Me If You Can: Spam Bot Detection Using Inductive Representation Learning. In *WWW*.

[3] Dana Angluin and Jiang Chen. 2008. Learning a Hidden Graph Using O(log n) Queries Per Edge. *J. Comput. System Sci.* 74, 4 (2008).

[4] Mikhail Belkin and Partha Niyogi. 2001. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In *NeurIPS*.

[5] Mathilde Bouvel, Vladimir Grebinski, and Gregory Kucherov. 2005. Combinatorial Search on Graphs Motivated by Bioinformatics Applications: A Brief Survey. In *WG*.

[6] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018).

[7] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yizhen Luo, Xingcheng Yao, Aohan Zeng, Shiguang Guo, Peng Zhang, Guohao Dai, Yu Wang, Chang Zhou, Hongxia Yang, and Jie Tang. 2021. CogDL: Toolkit for Deep Learning on Graphs. *arXiv preprint arXiv:2103.00959* (2021).

[8] Jianxin Chang, Chen Gao, Yu Zheng, Yiqun Hui, Yanan Niu, Yang Song, Depeng Jin, and Yong Li. 2021. Sequential Recommendation with Graph Neural Networks. In *SIGIR*.

[9] Sudhanshu Chanpuriya, Cameron Musco, Konstantinos Sotiropoulos, and Charalampos E Tsourakakis. 2021. DeepWalking Backwards: From Embeddings Back to Graphs. In *NeurIPS*.

[10] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. 2021. When Machine Unlearning Jeopardize Privacy. In *ACM CCS*. ACM, 896–911.

[11] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. 2022. Graph Unlearning. In *ACM CCS*. ACM.

[12] Yu Chen, Lingfei Wu, and Mohammed Zaki. 2020. Iterative Deep Graph Learning for Graph Neural Networks: Better and Robust Node Embeddings. In *NeurIPS*.

[13] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. 2008. Hierarchical Structure and the Prediction of Missing Links in Networks. *Nature* (2008).

[14] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. 2018. There's a Hole in that Bucket! A Large-scale Analysis of Misconfigured S3 Buckets. In *ACSAC*.

[15] Anirban Dasgupta, Ravi Kumar, and Tamas Sarlos. 2014. On Estimating the Average Degree. In *WWW*.

[16] Vasisht Duddu, Antoine Boutet, and Virat Shejwalkar. 2020. Quantifying Privacy Leakage in Graph Embedding. *arXiv preprint arXiv:1912.10979* (2020).

[17] Daniel M Dunlavy, Tamara G Kolda, and Evrim Acar. 2011. Temporal Link Prediction using Matrix and Tensor Factorizations. *TKDD* 5, 2 (2011).

[18] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. 2019. Graph Embedding based Familial Analysis of Android Malware using Unsupervised Learning. In *ICSE*.

[19] Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. 2019. Learning Discrete Structures for Graph Neural Networks. In *ICML*.

[20] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *ASE*.

[21] Palash Goyal and Emilio Ferrara. 2018. Graph Embedding Techniques, Applications, and Performance: A Survey. *Knowledge-Based Systems* 151 (2018).

[22] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD*.

[23] Roger Guimerà and Marta Sales-Pardo. 2009. Missing and Spurious Interactions and the Reconstruction of Complex Networks. *PNAS* (2009).

[24] Emil Julius Gumbel. 1954. *Statistical Theory of Extreme Values and Some Practical Applications: A Series of Lectures.*

[25] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*.

[26] Stephen J Hardiman and Liran Katzir. 2013. Estimating Clustering Coefficients and Size of Social Networks via Random Walk. In *WWW*.

[27] Michael Hay, Chao Li, Gerome Miklau, and David Jensen. 2009. Accurate Estimation of the Degree Distribution of Private Networks. In *ICDM*.

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

[29] Danny Hermelin, Avivit Levy, Oren Weimann, and Raphael Yuster. 2011. Distance Oracles for Vertex-Labeled Graphs. In *ICALP*.

[30] Darko Hric, Tiago P Peixoto, and Santo Fortunato. 2016. Network Structure, Metadata, and the Prediction of Missing Nodes and Annotations. *Physical Review X* (2016).

[31] Shouling Ji, Weiqing Li, Prateek Mittal, Xin Hu, and Raheem Beyah. 2015. Secgraph: A uniform and open-source evaluation system for graph data anonymization and de-anonymization. In *USENIX Security*.

[32] Shouling Ji, Weiqing Li, Mudhakar Srivatsa, and Raheem Beyah. 2014. Structural Data De-anonymization: Quantification, Practice, and Implications. In *ACM CCS*.

[33] Shouling Ji, Prateek Mittal, and Raheem Beyah. 2016. Graph Data Anonymization, De-Anonymization Attacks, and De-Anonymizability Quantification: A Survey. *IEEE Communications Surveys & Tutorials* (2016).

[34] Ian T Jolliffe and Jorge Cadima. 2016. Principal Component Analysis: A Review and Recent Developments. *Philos. Trans. A Math. Phys. Eng Sci.* (2016).

[35] Vassilis Kalofolias. 2016. How to Learn a Graph from Smooth Signals. In *AISTATS*.

[36] Anees Kazi, Luca Cosmo, Nassir Navab, and Michael Bronstein. 2020. Differentiable Graph Module (DGM) for Graph Convolutional Networks. *arXiv preprint arXiv:2002.04999* (2020).

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

[38] Thomas N Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[39] Thomas N Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *arXiv preprint arXiv:1611.07308* (2016).

[40] Jon Kleinberg. 2000. The Small-World Phenomenon: An Algorithmic Perspective. In *ACM STOC*.

[41] Wouter Kool, Herke Van Hoof, and Max Welling. 2019. Stochastic Beams and Where to Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement. In *ICML*.

[42] Josef Lauri and Raffaele Scapellato. 2016. *Topics in Graph Automorphisms and Reconstruction*. Cambridge University Press.

[43] Jure Leskovec and Christos Faloutsos. 2006. Sampling from Large Graphs. In *KDD*.

[44] Xiaoxiao Li, João Saúde, Prashant Reddy, and Manuela Veloso. 2020. Classifying and Understanding Financial Data Using Graph Neural Network. In *The AAAI Workshop on Knowledge Discovery from Unstructured Data in Financial Services (KDF)*. AAAI.

[45] David Liben-Nowell and Jon Kleinberg. 2007. The Link-Prediction Problem for Social Networks. *JAIST* 58, 7 (2007), 1019–1031.

[46] Jenny Liu, Aviral Kumar, Jimmy Ba, Jamie Kiros, and Kevin Swersky. 2019. Graph normalizing flows. *NeurIPS* 32 (2019).

[47] Kun Liu and Evimaria Terzi. 2008. Towards Identity Anonymization on Graphs. In *SIGMOD*.

[48] Linyuan Lü and Tao Zhou. 2011. Link Prediction in Complex Networks: A Survey. *Physica A* 390, 6 (2011).

[49] Mohammad Malekzadeh, Anastasia Borovykh, and Deniz Gündüz. 2021. Honest-but-Curious Nets: Sensitive Attributes of Private Inputs can be Secretly Coded into the Entropy of Classifiers' Outputs. In *ACM CCS*.

[50] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. Clustering and Community Detection in Directed Networks: A Survey. *Physics reports* 533, 4 (2013).

[51] Julian J McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NeurIPS*.

[52] Arvind Narayanan and Vitaly Shmatikov. 2009. De-anonymizing social networks. In *IEEE S&P*.

[53] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. 2020. Geom-GCN: Geometric Graph Convolutional Networks. *arXiv preprint arXiv:2002.05287* (2020).

[54] Abdurrahman Pektaş and Tankut Acarman. 2020. Deep Learning for Effective Android Malware Detection using API Call Graph Embeddings. *Soft Computing* 24, 2 (2020).

[55] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online Learning of Social Representations. In *KDD*.

[56] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. 2019. NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization. In *WWW*.

[57] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *WSDM*.

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

[59] Veronica Red, Eric D Kelsic, Peter J Mucha, and Mason A Porter. 2011. Comparing Community Structure to Characteristics in Online Collegiate Social Networks. *SIAM review* 53, 3 (2011).

[60] Benjamin Ricaud, Nicolas Aspert, and Volodymyr Miz. 2020. Spikyball Sampling: Exploring Large Networks via an Inhomogeneous Filtered Diffusion. *Algorithms* (2020).

[61] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

[62] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. Little Ball of Fur: A Python Library for Graph Sampling. In *CIKM*.

[63] Guillaume Salha, Romain Hennequin, and Michalis Vazirgiannis. 2020. Simple and Effective Graph Autoencoders with One-Hop Linear Models. In *ECML-PKDD*.

[64] Yun Shen, Xinlei He, Yufei Han, and Yang Zhang. 2022. Model Stealing Attacks Against Inductive Graph Neural Networks. In *IEEE S&P*.

[65] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks against Machine Learning Models. In *IEEE S&P*.

[66] Sigal Sina, Avi Rosenfeld, and Sarit Kraus. 2013. Solving the Missing Node Problem using Structure and Attribute Information. In *ASONAM*.

[67] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Machine Learning Models that Remember Too Much. In *ACM CCS*.

[68] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S Yu, Lifang He, and Bo Li. 2018. Adversarial Attack and Defense on Graph Data: A Survey. *arXiv preprint arXiv:1812.10528* (2018).

[69] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex Embeddings for Simple Link Prediction. In *ICML*.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*.

[71] Binghui Wang, Jinyuan Jia, and Neil Zhenqiang Gong. 2019. Graph-based Security and Privacy Analytics via Collective Classification with Joint Weight Learning and Propagation. In *NDSS*.

[72] Jianyu Wang, Rui Wen, Chunming Wu, Yu Huang, and Jian Xion. 2019. FdGars: Fraudster Detection via Graph Convolutional Networks in Online App Review System. In *WWW*.

[73] Fan Wu, Yunhui Long, Ce Zhang, and Bo Li. 2022. LinkTeller: Recovering Private Edges from Graph Neural Networks via Influence Analysis. In *IEEE S&P*.

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

[75] Liu Yang and Rong Jin. 2006. Distance Metric Learning: A Comprehensive Survey. *Michigan State Universiy* 2, 2 (2006).

[76] Liang Yang, Zesheng Kang, Xiaochun Cao, Di Jin, Bo Yang, and Yuanfang Guo. 2019. Topology Optimization based Graph Convolutional Network. In *IJCAI*.

[77] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* (2019).

[78] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. 2016. Revisiting Semi-Supervised Learning with Graph Embeddings. In *ICML*.

[79] Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang. 2018. Network Representation Learning: A Survey. *IEEE transactions on Big Data* (2018).

[80] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *NeurIPS*.

[81] Minxing Zhang, Zhaochun Ren, Zihan Wang, Pengjie Ren, Zhunmin Chen, Pengfei Hu, and Yang Zhang. 2021. Membership Inference Attacks Against Recommender Systems. In *ACM CCS*.

[82] Zhikun Zhang, Min Chen, Michael Backes, Yun Shen, and Yang Zhang. 2022. Inference Attacks Against Graph Neural Networks. In *USENIX Security*.

[83] Zaixi Zhang, Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. 2021. Backdoor Attacks to Graph Neural Nnetworks. In *SACMAT*.

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

[85] Gang Zhao and Jeff Huang. 2018. Deepsim: Deep Learning Code Functional Similarity. In *ESEC/FSE*.

[86] Jianan Zhao, Xiao Wang, Chuan Shi, Binbin Hu, Guojie Song, and Yanfang Ye. 2021. Heterogeneous Graph Structure Learning for Graph Neural Networks. In *AAAI*.

[87] Bin Zhou, Jian Pei, and WoShun Luk. 2008. A Brief Survey on Anonymization Techniques for Privacy Preserving Publishing of Social Network Data. *ACM SIGKDD Explorations Newsletter* (2008).

[88] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS*.

[89] Yanqiao Zhu, Weizhi Xu, Jinghao Zhang, Qiang Liu, Shu Wu, and Liang Wang. 2021. Deep Graph Structure Learning for Robust Representations: A Survey. *arXiv preprint arXiv:2103.03036* (2021).