LONG-DISTANCE TARGETED POISONING ATTACKS ON GRAPH NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

GNNs are vulnerable to targeted poisoning in which an attacker manipulates the graph to cause a target node to be mis-classified to a label chosen by the attacker. However, most existing targeted attacks inject or modify nodes within the target node's k-hop neighborhood to poison a k-layer GNN model. In this paper, we investigate the feasibility of *long-distance* attacks, i.e., attacks where the injected nodes lie outside the target node's k-hop neighborhood. We show such attacks are feasible by developing a bilevel optimization-based approach, inspired by meta-learning. While this principled approach can successfully attack small graphs, scaling it to large graphs requires significant memory and computation resources, and is thus impractical. Therefore, we develop a much less expensive, but approximate, heuristic-based approach that can attack much larger graphs, albeit with lower attack success rate. Our evaluation shows that long-distance targeted poisoning is effective and difficult to detect by existing GNN defense mechanisms. To the best of our knowledge, our work is the first to study long-distance targeted poisoning attacks.

1 Introduction

Many recent papers have proposed attacks on GNNs that allow an attacker to cause mis-predictions by strategically modifying the graph structure (Zügner & Günnemann, 2019; Xu et al., 2019; Dai et al., 2018; Chen et al., 2018; 2020; Zang et al., 2020; Chang et al., 2020; Bojchevski & Günnemann, 2019; Wang & Gong, 2019; Geisler et al., 2021a; Mujkanovic et al., 2022; Wang et al., 2022), modifing the features of an existing node in the graph (Zügner et al., 2018; Liu et al., 2019; Wu et al., 2019; Ma et al., 2020), or injecting new nodes with carefully crafted features (Sun et al., 2020; Jiang et al., 2022; Zou et al., 2021; Tao et al., 2021; Chen et al., 2022; Ju et al., 2022; Wang et al., 2020; 2018; Nguyen Thanh et al., 2023). Depending on when the adversarial perturbation occurs, these attacks can be classified as poisoning (training time) attacks or evasion (test time) attacks. Furthermore, depending on whether the attacker aims to misclassify a specific node or degrade the overall prediction accuracy, the attacks can be targeted (local) or untargeted (global) attacks. In this paper, we investigate targeted poisoning on node classification GNNs, where an attacker's goal is to flip a selected a target node's label to an attacker-chosen label.

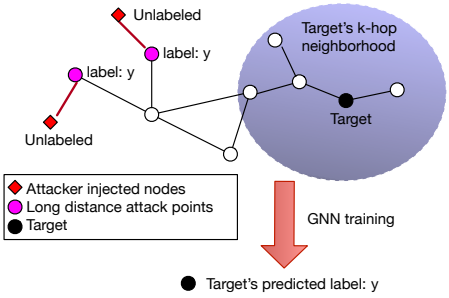

Figure 1: *Targeted poisoning attack via long distance node injection.*

Existing attacks of this type assume that attackers have great flexibility in how they modify the graph, e.g., they can add edges between any chosen pairs of nodes (Zügner et al., 2018; Chang et al., 2020; Dai et al., 2018; Chen et al., 2018; Xu et al., 2019; Zang et al., 2020; Geisler et al., 2021a; Wang & Gong, 2019; Bojchevski & Günnemann, 2019) or connect a new fake node directly to the target node (Wang et al., 2020; Chen et al., 2022; Dai et al., 2023; Xi et al., 2021). However, changing nodes close to the target is undesirable because it make attack detection easy: many existing tools (Ying et al., 2019; Huang et al., 2022; Luo et al., 2020; Yuan et al., 2021; Duval & Malliaros, 2021) measure the influence of nodes within a target's k-hop neighborhood and attack nodes generally have high influence values.

In this paper, we consider a more practical but restricted scenario, where an attacker can inject new nodes but can only connect them to nodes beyond a threshold distance from the target. In other words, we seek attacks that, given a threshold k and target node v_t, can insert (inject) new nodes or add edges connecting injected nodes to existing nodes that lie outside v_t's k-hop neighborhood. We call such attacks *long-distance* targeted poisoning attacks.

Inspired by meta learning (Bengio, 2000; Finn et al., 2017), we design a bilevel optimization-based method to craft a long distance attack, called MetaLDT (aka **Meta** learning based **L**ong **D**istance **T**argeted poisoning). Like the previous application of meta learning to GNN poisoning (Zügner & Günnemann, 2019), we treat the graph perturbation, which includes injected nodes' features and their connections to existing nodes, as a hyperparameter to learn. However, unlike (Zügner & Günnemann, 2019), we introduce several important constraints to the optimization to make the attack hard to detect. These include constraints that prevent changes to the target's k-hop neighborhood, that avoid noticeably changing node degree distribution, and that maintain graph homophilly. Our evaluation shows that MetaLDT achieves excellent poison success rates over regular GNNs ($> 84\%$) as well as robust versions that have been fortified with state-of-the-art defenses. However, MetaLDT has high computation and memory requirements, and thus cannot scale: we found that we could only use it with small graphs with at most a few thousand nodes (e.g., Cora and Citeseer), and even in these cases an attack could take a day or more. Thus, the attack cannot target most practical graphs.

To address this, we examine the graphs generated by MetaLDT to learn common patterns. We find that in order to flip a target node's label to y, MetaLDT's optimization tries to make the target node's embedding—the last GNN layer's output before the softmax classifier—"collide" with the embedding of nodes labelled y. Based on this observation, we designe MimicLDT, a scalable heuristic attack that mimics MetaLDT's effect. MimicLDT attaches a few fake nodes to a small subset of existing nodes (aka points of attack) with target label y and uses gradient-based optimization to craft the features of the fake nodes so that the points of attack become close to the target node in the embedding space. While less effective than MetaLDT, our evaluation shows that MimicLDT nevertheless achieves decent poison success rate (>55%) and can scale to graphs with hundreds of thousands of nodes (e.g. arXiv).

In summary, we make the following contributions:

- We study a new type of targeted poisoning attack on GNNs that does not modify the target node's k-hop neighborhood. We call such an attack *long distance* poisoning.
- We propose two optimization-based method, MetaLDT and MimicLDT, to achieve targeted poisoning by injecting fake nodes that lie beyond the target node's k-hop neighborhood. MetaLDT is based on the principled approach of meta learning but is too expensive to use with graphs containing more than a few thousand nodes. MimicLDT builds upon the insights discovered by MetaLDT to perform direct optimization without meta learning and can scale to much larger graphs.
- We evaluate our attack on different graphs and defenses. To the best of our knowledge, we are the first to show the existence and effectiveness of long distance targeted poisoning attacks.

2 Related work

We discuss related work on targeted attacks for GNN-based node classification. For a broader discussion including untargeted attacks, attacks on tasks other than node classification as well as GNN defenses, we refer readers to existing surveys (Jin et al., 2020a; Zheng et al., 2021).

Targeted attacks can occur during training time (poisoning attacks) or test time (evasion attacks). For any given k-layer GNN model architecture, the target node's label prediction is a function of (1) model weights, (2) the input features of the target node itself, and (3) the input features of the target's k-hop neighbors. Thus, in order to manipulate the target's label prediction, the attacker can try to corrupt any of these three factors. Attacks of type (2), aka corrupting the target's input features, is a well-studied problem in non-graph domains (Goodfellow et al., 2015; Shafahi et al., 2018; Madry et al., 2018) such as images, text, and time series, and can be straightforwardly extended to the graph setting. Thus, existing GNN attacks, including both poisoning and evasion attacks, focus on the adversarial manipulation of (3), aka the target's k-hop neighborhood, achieved through adding/removing edges (referred to as structure perturbation attacks), or adding fake nodes (referred to as injection attacks). The manipulation of (3) can be further categorized into direct vs. indirect attacks depending on whether the target's direct or k-hop neighborhood is modified. There is a vast collection of attacks of type (3); most perturb graph structure, e.g. NetAttack (Zügner et al., 2018), FGA (Chen et al., 2018),

MGA (Chen et al., 2020), PGD (Xu et al., 2019), DICE (Waniek et al., 2018), GUA (Zang et al., 2020), RL-S2V (Dai et al., 2018), Bojchevski et al. (Bojchevski & Günnemann, 2019), GF-Attack (Chang et al., 2020), GAFNC (Jiang et al., 2022), IG-JSMA (Wu et al., 2019), Wang et al. (Wang et al., 2022) and PR-BCD/GR-BCD (Geisler et al., 2021a). Some also modify existing nodes' labels or features, e.g. (Liu et al., 2019; Zügner et al., 2018; Wu et al., 2019). Others inject fake nodes, e.g., Wang et al. (Wang et al., 2018), TDGIA (Zou et al., 2021), AFGSM (Wang et al., 2020), G^{2}A2C (Ju et al., 2022) and G-NIA (Tao et al., 2021).

Our work differs from existing ones in that we aim to achieve targeted poisoning by changing the graph $\mathcal{G}$ used to train the GNN model. The key advantage of this approach is that we can make the resulting attack *long distance*, by completely avoiding any modification to the target's k-hop neighborhood. This is desirable for two reasons: one, it makes the attack difficult to detect, foiling analysis tools like GNNExplainer (Ying et al., 2019) and others (Luo et al., 2020; Yu & Gao, 2022; Yuan et al., 2021; Duval & Malliaros, 2021). Two, it makes the attack easier to launch since there are many more potential attack points beyond the target's k-hop neighborhood.

Recent work such as HAO (Chen et al., 2022) and ADIMA (Tao et al.) aim to make attacks hard to detect; the former adds a homophily constraint and the latter uses a GAN-style framework to train a discriminator to distinguish subgraphs that include fake nodes from those that don't. Like HAO, our attack also tries to preserve homophily. Existing attacks all assume that the attacker can add/remove edges to *any* existing node. A recent proposal tries to make attacks more realistic by assuming that the attacker can only use a subset of nodes as attack points (Ma et al., 2020). Our work can also be extended to this setting by restricting the set of attack points.

Our attack assumes that the attacker knows the process used to train the model being attacked, this information includes knowledge of what defenses are used. Prior work (Zügner et al., 2018; Zügner & Günnemann, 2019) has made similar assumptions, and our approach is inspired by these proposals.

3 Background and Problem Definition

In this section, we define the terminology we use, formalize our setting, and state our assumptions about the attacker's capabilities. We use the standard terminology and notation for graphs and GNNs:

Graphs We use $\mathcal{G}=(\mathcal{V},\mathcal{E})$ to denote a graph with nodes $\mathcal{V}=\{v_1,v_2,..,v_n\}$ and edges $\mathcal{E}=\{e_1,e_2,..e_m\}$, use $\mathbf{A}\in\{0,1\}^{n\times n}$ to denote graph $\mathcal{G}$'s adjacency matrix, and $\mathbf{X}\in\mathbb{R}^{n\times d}$ to denote the d-dimensional feature matrix for each node $v\in\mathcal{V}$.

GNN based node classification Our work focuses on node classification in the *transductive learning* setting: our training data consists of a graph $\mathcal{G}$, a subset of whose nodes are labeled, denoted as $\mathcal{V}_L\subset\mathcal{V}$. Given this training data, node classification aims to learn a function that can predict labels for unlabeled nodes $\mathcal{V}_U:=\mathcal{V}\setminus\mathcal{V}_L$.

For node classification, the computation can be viewed as consisting of two parts: first, one uses a GNN to compute an embedding for each node that represents the node's features and neighborhood, and second, one feeds this embedding to a classification layer to get a set of *logits* for label prediction. In what follows, we use f_θ to represent the GNN model with θ representing its weights, and we use the term *node embedding* to represent the embedding computed by f_θ.

3.1 Attack Model

Attacker's goal We consider *targeted* label-flipping attacks, where the attacker selects a target node v_t and target label y_t, and the attack aims to alter the graph used to train the GNN f_θ so that it predicts label y_t for node v_t.

Attacker's knowledge We assume the attacker has access to the training data, including the original graph G, node features, and labels, and also knows the training procedure (including any changes made to the training to improve model robustness). Note that we do not assume knowledge of model weights, but if available this can be used to further reduce the cost of MimicLDT.

Attacker's capability We constrain the attacker to long-distance node injection attacks. This means that the attacker cannot modify an existing node's features, remove existing edges, or add edges that connect two nodes that are already present in the graph. In addition, constraining to long-distance attacks means the attacker can only add edges to nodes that are outside the target's k-hop neighborhood. The attacks we consider add (inject) one or more new nodes, and add edges connecting these injected

nodes to each other or to existing nodes that are outside the target's k-hop neighborhood. Finally, we constrain the number of nodes and edges that an attacker can add.

More formally, we define **node injection attacks** as follows: an attack that generates a poisoned graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ by injecting a set of malicious nodes $\mathcal{V}_{inj}$ as $\mathbf{A}' = \begin{bmatrix} \mathbf{A} & \mathbf{B}_{inj} \\ \mathbf{B}_{inj}^T & \mathbf{O}_{inj} \end{bmatrix}$, $\mathbf{X}' = \begin{bmatrix} \mathbf{X} \\ \mathbf{X}_{inj} \end{bmatrix}$, where $\mathbf{X}_{inj}$ are the injected nodes features, $\mathbf{B}_{inj}$ is the adjacency matrix between injected and existing nodes in $\mathcal{G}$. We refer to an existing node that connects to any injected node as an attack point. We limit the number of injected nodes, and their degree, i.e., we require that $|\mathcal{V}_{inj}| \leq \triangle \in \mathbb{Z}$ and $1 \leq deg(i) \leq b \in \mathbb{Z}, \forall i \in \mathcal{V}_{inj}$ for some threshold $\triangle$ and b.

We define **long-distance node injection** attacks as follows: a node injection attack where no attack point is within the target node v_t's k-hop neighborhood. More formally, in a long distance attack on a k-layer GNN, $\forall v_a \in \mathcal{V}_a$, $d(v_a, v_t) > k$ where $\mathcal{V}_a$ is the set of existing-graph nodes connected to injected nodes (aka attack points), and $d(v_a, v_t)$ is the path length from v_a to v_t.

3.2 Problem Formulation

We start by formalizing attacks as an optimization problem, which we can solve using a meta-learning inspired approach. GNN attacks can generally be formalized as:

$$\min_{\mathcal{G}'} \quad \mathcal{L}_{atk}(f_{\theta^*}(\mathcal{G}')) \quad s.t. \quad \theta^* = \underset{\theta}{\operatorname{argmin}} \quad \mathcal{L}_{\text{train}}(f_\theta(\mathcal{G}')). \tag{1}$$

where $\mathcal{L}_{train}$ is the general loss function used when training model f_θ, which we assume the attacker knows. Therefore, our goal is to find a graph, $\mathcal{G}'$, that minimizes the attacker's loss $\mathcal{L}_{atk}$.

A targeted label-flipping attack requires incorporating the target node and desired label into the loss-function. More precisely, we need a loss function that maximize the target node's *logit* (i.e., the model's confidence score for a label) for the attacker-chosen label. Therefore, we use $\mathcal{L}_{atk} = -\mathcal{M}_{\mathcal{G}'}(v_t)[y_t]$, where $\mathcal{M}_{\mathcal{G}'}(v_t) = f_{\theta^*}(v_t; \mathcal{G}')$, which maximizes the probability that target node v_t has label y_t. Beyond this, and similar to prior work (Chen et al., 2022), we want to ensure that the attack is stealthy and injected nodes do not differ significantly from existing nodes. To do so, we incorporate a homophily term in $\mathcal{L}_{atk}$ that minimizes feature differences between an injected node and its neighbors. Our final attacker loss function, $\mathcal{L}_{atk}$, is thus:

$$\mathcal{L}_{atk} = -\mathcal{M}_{\mathcal{G}'}(v_t)[y_t] - \beta C(\mathcal{G}') \tag{2}$$

$$C(\mathcal{G}') = \frac{1}{|\mathcal{V}_{inj}|} \sum_{u \in \mathcal{V}_{inj}} sim(r_u, X_u), \text{ where } r_u = \sum_{j \in \mathcal{N}(u)} \frac{1}{\sqrt{d_j}\sqrt{d_u}} X_j \tag{3}$$

where β is a hyperparameter that controls how important homophily is, $sim(\cdot)$ measures *cosine similarity*, $\mathcal{N}(u)$ is the set of nodes neighboring node u, and d_u is the node degree. The homophily formulation above is based on Chen et al. (2022).

4 The MetaLDT Attack via Optimization

We start by describing an optimization approach to solving the problem defined in the previous section. Given information about how the model is trained as well as access to the original graph $\mathcal{G}$, a target node v_t, and a target label y_t, our optimization algorithm produces an attack graph $\mathcal{G}'$. To do this, we start with an initial attack graph $\mathcal{G}'_0$ and iteratively modify it to minimize the attacker's loss function $\mathcal{L}_{atk}$. Our iterative approach, inspired by meta-learning (Bengio, 2000; Zügner & Günnemann, 2019), treats the attack graph's edges and features as hyperparameters, which it optimizes.

At the start of the process, MetaLDT produces an initial graph $\mathcal{G}'_0$ by injecting Δ new nodes into the input graph $\mathcal{G}$. These injected nodes have zeroed-out features, and no edges connecting them to any other node. In each iteration i ($i \geq 0$), MetaLDT updates graph $\mathcal{G}'_i$ and produces the graph $\mathcal{G}'_{i+1}$, which the next iteration operates on. When producing $\mathcal{G}'_{i+1}$, we can either alter $\mathcal{G}'_i$'s adjacency matrix (thus adding or removing edges) or feature matrix (thus changing node features), and MetaLDT uses alternating minimization to update both. Specifically, this means that our iterations alternate between changing the adjacency matrix and changing the feature matrix.

In each iteration i, we determine updates to the feature or adjacency matrix (as appropriate) using a computed meta-gradient $\nabla^{meta}_{\mathcal{G}'_i}$, which we compute by unrolling the model training loop for T epochs. Formally:

$$\begin{aligned} \nabla^{meta}_{\mathcal{G}'_i} &= \nabla_{\mathcal{G}'_i} \mathcal{L}_{atk}(f_{\theta_T}(\mathcal{G}'_i)) \\ &= \nabla_f \mathcal{L}_{atk}(f_{\theta_T}(\mathcal{G}'_i)) \cdot \left[\nabla_{\mathcal{G}'_i} f_{\theta_T}(\mathcal{G}'_i) + \nabla_{\theta_T} f_{\theta_T}(\mathcal{G}'_i) \cdot \nabla_{\mathcal{G}'_i} \theta_T \right] \end{aligned} \tag{4}$$

where the last term is recursively defined as $\nabla_{\mathcal{G}'_i}\theta_{t+1} = \nabla_{\mathcal{G}'_i}\theta_t - \alpha\nabla_{\mathcal{G}'_i}\nabla_{\theta_t}\mathcal{L}_{train}(f_{\theta_t}(\mathcal{G}'_i))$, and α is the learning rate. Observe that computing this meta-gradient does not require access to the model used by the attack's victim, but requires running T training epochs, using the same training setting (i.e., the same algorithm and approach) as used by the victim. In the rest of the paper, we use the term *surrogate model* to refer to models trained by the attacker using the same process as the victim.

4.1 Changing Graph Structure

Iterations that alter the adjacency matrix assume that the node feature matrix is a constant. Consequently, we can treat $\nabla^{meta}_{\mathcal{G}'_i}$ as the meta-gradient for the graph $\mathcal{G}'_i$'s adjacency matrix A_i, and can compute a *meta-score* (Zügner & Günnemann, 2019), $S(u,v) = \nabla^{meta}_{\mathcal{G}'_i}[a_{uv}] \cdot (-2 \cdot a_{uv} + 1)$ for each pair-of-nodes (u,v), where $[a_{uv}]$ indicates that we indexed the value at position (u,v) in $\nabla^{meta}_{\mathcal{G}'_i}$'s adjacency matrix. Our approach is predicated on the observation that altering the adjacency matrix for the pair (u,v) with the highest computed meta-score $S(u,v)$ is likely to best decrease the attacker's loss $\mathcal{L}_{atk}$.

However, our assumptions (§3) limit what adjacency matrix modifications the attacker can perform, and so we only consider a subset of node pairs in this process. In particular, we impose the following constraints on the node-pairs we consider: (a) either u or v must be an injected node; (b) neither u nor v can be within v_t's k-hop neighborhood, thus ensuring that the attacks are long-distance; and (c) that an injected node u has no more than one-edge connecting it to a node in the original graph $\mathcal{G}$, a constraint we add to avoid cases where the optimization spends all of its time optimizing a single injected node. We evaluate the effect of the last optimization in Appendix C.3.

Thus, iterations that change the adjacency matrix compute a score $S(u,v)$ for any pair of nodes (u,v) that meet our constraints, identify the pair (u_m,v_m) with the largest score, and then adds edge (u_m,v_m) if none exists or removes it if it already exists.

4.2 Changing Node Features

Similarly, iterations where node-features are changed assume that the adjacency matrix is a constant, and therefore use $\nabla^{meta}_{\mathcal{G}'_i}$ as the meta-gradient for the feature matrix. However, in this case, we do not use $\nabla^{meta}_{\mathcal{G}'_i}$ to compute a scoring function to select and then update node features for a single node. Instead, we use $\nabla^{meta}_{\mathcal{G}'_i}$ to compute feature gradients which we use to update $\mathcal{G}_i$'s feature matrix. Care must be taken when doing so, since we assume attacker cannot change features for any nodes already present in the input graph $\mathcal{G}$ (§3). We impose this constraint by zeroing out the corresponding elements in $\nabla^{meta}_{\mathcal{G}'_i}$'s feature matrix, and in what follows we refer to the resulting matrix as $X_{\nabla_{\mathcal{G}'_i}}$. Given this, we compute: $\hat{X}_{\mathcal{G}'_{i+1}} = X_{\mathcal{G}'_i} - \alpha X_{\nabla_{\mathcal{G}'_i}}$ where $X_{\mathcal{G}'_i}$ is $\mathcal{G}'_i$'s feature matrix, and α is the learning rate.

Empirically, we found that a single gradient update is often insufficient, so in practice each iteration repeats this process q times (and we compute a new $\nabla^{meta}_{\mathcal{G}'_i}$ after each update).

5 The MimicLDT Attack via Embedding Collision

Our evaluation (§6) shows that MetaLDT is effective, but has such high memory and computational requirements (Table 2) that it is impractical to use it with common graphs (e.g., the Arxiv dataset). MimicLDT is a cheaper attack that uses heuristics to mimic MetaLDT's behavior. Our heuristics derive from the following observations about MetaLDT generated attack graphs ($\mathcal{G}'$):

(a) MetaLDT iterations reduce the embedding space distance (as determined by the surrogate GNN) between v_t and existing nodes with the attacker's chosen target label y_t. We empirically demonstrate this phenomenon in Figure 2 by using MetaLDT to attack a GCN that uses the Cora dataset. We give more details about this setting in §6. The graph shows how the average L2-distance, in each iteration's surrogate model's embedding space, between v_t and nodes whose ground truth label is y_t varies across iterations, and we observe that the optimization minimizes this distance.
(b) Edges between existing and injected nodes in $\mathcal{G}'$ tend to connect injected nodes to nodes labeled y_t. We hypothesize that this is in support of the previous observation: an injected node, that connects to a node v labeled y_t, can reduce the embedding distance between v_t (the target) and v.
(c) The edges connecting pairs of injected nodes in $\mathcal{G}'$ do not appear to have any noticeable patterns. This leads us to hypothesize that it is sufficient to randomly connect injected nodes with each other.

Next, we describe how MimicLDT's heuristics allow it to efficiently generate attack graphs. Similar to MetaLDT, MimicLDT takes as input an initial graph $\mathcal{G}$, a target node v_t, a target label y_t, and assumes

knowledge of how the attacked GNN has been trained. MimicLDT works as follows: (a) First, it trains a surrogate model f_θ using $\mathcal{G}$, which is used for the entire optimization. Much of MimicLDT's performance improvement is because it only needs to train one surrogate model. (b) Next, MimicLDT generates the structure of the attack graph $\mathcal{G}'$ based on heuristicsc) Finally, it optimizes injected node features in $\mathcal{G}'$ to produce the final attack graph.

5.1 Determining Graph Structure

Similar to MetaLDT, MimicLDT generates an initial attack graph $\mathcal{G}'$ from $\mathcal{G}$. To construct $\mathcal{G}'$ MimicLDT first selects a set of nodes, $\mathcal{V}_a$, in $\mathcal{G}$ whose label is y_t, and who lie outside the target's (i.e., v_t's) k-hop neighborhood. We refer to the nodes $\mathcal{V}_a$ as attack points, and a hyperparameter r determines $|\mathcal{V}_a|$, the number of attack points chosen.[1]

Next, for each attack point, $v_a \in \mathcal{V}_a$, MimicLDT injects Φ nodes $\mathcal{V}_{g_v}$ and connects them (directly or indirectly) to v_a. To do so, MimicLDT iterates over possible edges connecting nodes in the set $\mathcal{V}_{g_v} \cup \{v_a\}$ (i.e., edges that either connect injected nodes to each other or to the attack point), and add each edge to the graph with probability $p = 0.5$. Finally, it prunes any nodes in $\mathcal{V}_{g_v}$ not reachable to any v_a. Therefore, the final set of injected nodes may have fewer than Φ nodes.

The final graph structured produced by MimicLDT thus consists of all nodes and edges in $\mathcal{G}$, and a set of injected nodes that are connected to each other and attack points. In the rest of this section, we use $\mathcal{V}_i$ to refer to the set of injected nodes in $\mathcal{G}'$, and $B(v_i)$ to refer to the attack point whose k-hop neighborhood contains the injected node $v_i \in \mathcal{V}_i$.

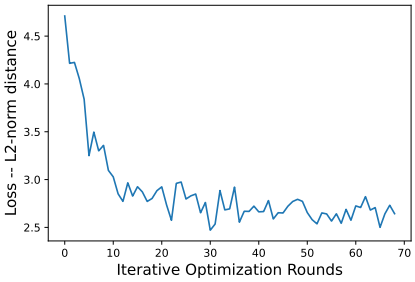

Figure 2: *The embedding space distance between the target node and existing nodes whose ground-truth label is the same as the attack's target label, as MetaLDT's optimization progresses.*

5.2 Determining Injected Node Features

The algorithm above generates a graph $\mathcal{G}'$ that contains all nodes in $\mathcal{G}$ and the set $\mathcal{V}_i$ of injected nodes. We do not assign labels to any of the injected nodes, and formulate the problem of assigning feature vectors to these injected nodes as an optimization problem, which we describe below.

Optimization formulation. Our feature optimization problem aims to meet two goals. First, based on our observations form MetaLDT, we try to ensure that the final node embedding for any selected attack point $v_a \in \mathcal{V}_a$, $\mathbf{h}_{v_a}^{(L)}$, is close to the target's final node embedding $\mathbf{h}_{v_t}^{(L)}$. Second, similar to MetaLDT, we try to ensure that an injected node v_i's feature vector $\mathbf{X}_{v_i}$ is similar to that of its attack point $B(v_i)$.

Taken both optimization goals into account, our final optimization formulation is:

$$\mathbf{X}^*_{\mathcal{V}_i} = \underset{\mathbf{X}_{\mathcal{V}_i}}{\operatorname{argmin}} \mathcal{L}_{atk}, \tag{5}$$

$$\mathcal{L}_{atk} = -\left(\frac{1}{|\mathcal{V}_a|} \sum_{v_a} Sim_f(\mathbf{h}_{v_a}^{(L)}, \mathbf{h}_{v_t}^{(L)}) + \beta * \frac{1}{|\mathcal{V}_i|} \sum_{v_i} Sim_{in}(\mathbf{X}_{v_i}, \mathbf{X}_{B(v_i)}) \right) \tag{6}$$

In this formulation, $\mathbf{X}_{\mathcal{V}_i}$ represents a $|\mathcal{V}_i| \times d$ dimensional matrix whose columns are features of nodes in $\mathcal{V}_i$; Sim_f is a metric function (specific to the GNN used) that measures similarity between a pair of final node embeddings; and Sim_{in} is a metric function (specific to the input graph $\mathcal{G}$) that measures similarity between a pair of nodes' feature vectors.

The second term of the optimization in Eq 5 aims to preserve homophilly, but differs from the formulation used in MetaLDT, and prior work (Chen et al., 2022): rather than maximizing the node-centric homophily score which aggregating with neighborhood for all injected nodes, this formulation maximizes similarity between the attack points and injected nodes. We found that this change in formulation improved our performance, and our empirical results (in Appendix A.3) show that it does not noticeably impact the homophily scores for injected nodes. Similar to MetaLDT, the β hyperparameter allows attackers to decide how much the generated attack prioritizes homophily.

[1] In the evaluation we use $0 < r < 1$, and select $r|\mathcal{V}_L|$ points ($\mathcal{V}_L$ is the set of labeled training nodes in $\mathcal{G}$).

Computing feature vectors. We use a stochastic gradient descent based optimizer to compute feature vectors. Our evaluation uses GNNs to classify nodes in citation graphs, and consequently we use cosine similarity to measure similarity between node features (i.e., Sim_{in} is cosine similarity), and we use $L2$-norm to measure similarity between final node embeddings (Sim_f).

We considered various initial values for $\mathbf{X}_{\mathcal{V}_i}$, including using the input features of the neighboring attack point ($X_{v_i} = X_{B(v_i)}$), input features of a random neighbor, and the target's input features. Empirically, we found no noticeable difference between these options since the embedding space distance as well as the feature space distance will converge to the similar place despite being initialized differently. During the optimization, we use the surrogate model f_θ to compute $\mathbf{h}_{v_a}^{(L)}$ and $\mathbf{h}_{v_t}^{(L)}$.

6 Experiments

We run experiments on NVIDIA V100 GPUs, with 32GB memory limitation. Our evaluation aims to answer the following questions:

- Can our attacks poison existing GNN models and their fortified versions?
- Compared to MetaLDT, how effective is MimicLDT and can it scale to larger graphs?
- How do long-distance attacks compare to existing short-distance ones?
- Are our attacks stealthy? e.g. do the attacks impact graph homophily?
- Can we launch effective end-to-end attacks?

Datasets We use four graph datasets: Cora (Yang et al., 2016), Citeseer (Yang et al., 2016), PubMed (Yang et al., 2016) and Ogbn-arXiv (Hu et al., 2021). The largest graph, arXiv, is almost two orders of magnitude larger than the smallest, Cora. Appendix§B provides details.

GNN models. We use three popular GNN models: **GCN** (Kipf & Welling, 2017), **GraphSAGE** (Hamilton et al., 2017), **GAT** (Veličković et al., 2018). We use 3-layer models for most datasets, the exception is Cora, which is a small graph and for which we use a 2-layer model. We provide detailed model settings, the training process, and hyperparameters in Appendix§B. In addition to vanilla models, we also evaluate our attacks against models that use the following 5 GNN defense mechanisms: **ProGNN** (Jin et al., 2020b), **GNNGuard** (Zhang & Zitnik, 2020), **Soft-Median-GDC** (Geisler et al., 2021b), **Jaccard GCN** (Wu et al., 2019) and **SVD GCN** (Entezari et al., 2020).

Comparison with short-distance attacks Existing attacks perturb the target's k-hop neighborhood and are thus *short-distance* attacks. We compare against three short-distance attacks: **Nettack** (Zügner et al., 2018), **FGA** (Chen et al., 2018) and **IG-FGSM** (Wu et al., 2019). In all cases, we use a loss function designed for our goal of changing a target node's label to a specified one (details in Appendix. E.1).

6.1 Effectiveness of MetaLDT Attack

We evaluate MetaLDT on Cora. Table. 1 reports MetaLDT's poison success rate for different GNN models, both vanilla ones as well as the their fortified versions. The poison success rate is calculated over 200 experiments, each with a randomly chosen target node and target poison label. We configure $\Delta = 68$, which limits the number of changes on the adjacency matrix. For each step of adjacency matrix optimization, MetaLDT performs $q = 1000$ optimization steps on injected nodes' feature.

Table 1 shows that MetaLDT can achieve high attack success rate ($84\%\sim96\%$) over vanilla GNN models. When evaluating MetaLDT against robust models, we assume the attacker is aware of the defensive mechanism used and adapt MetaLDT accordingly (Mujkanovic et al., 2022). However, doing so comes at the cost of increased memory consumption and computational overhead. Hence, for some robust models (GNNGuard, SoftMedianGDC, ProGNN), we stop MetaLDT's inner-training loop early before its convergence after 50 instead of the regular 200 epochs, in order to avoid OOM. From Table 1, we can see that when MetaLDT's inner training loops is allowed converge (JaccardGCN, SVDGCN), its success rate remains high. However, stopping the inner training loop early comes at a significant cost of poison success rate. It is crucial that MetaLDT adapts to the underlying GNN defense. We report the results of nonadaptive MetaLDT in §C.5.

Comparing with short-distance attacks. Table 1 also shows the range of performance achieved by existing short-distance modification attacks, including Nettack-direct (modifying the target's immediate neighbors), Nettack-indirect(modifying the target's k-hop neighborhood), FGA and IG-FGSM. Detailed results are in §F (Table 14). We set the short-distance attack budget to be 68 and leave experiments with varying perturbation budgets in §E.2). From Table 1, we can see that short-distance attacks can achieve higher success rate, often at 100%. However, short-distance attacks are susceptible

		MetaLDT	MimicLDT	Short Distance
Vanilla	GCN	0.96	0.67	0.79—1.00
	GraphSAGE	0.87	0.63	0.42—0.96
	GAT	0.84	0.60	0.53—0.97
Robust	GNNGuard	(0.53)	0.70	0.94—1.00
	SoftMedianGDC	(0.58)	0.55	0.46—1.00
	JaccardGCN	0.91	0.66	0.47—1.00
	SVDGCN	0.83	0.74	0.18—1.00
	ProGNN	(0.55)	0.59	0.60—1.00

Table 1: *Success rate of MetaLDT, MimicLDT and short-distance attacks (Nettack, FGA and IG-FGSM) over Cora. Numbers in parentheses indicate cases where MetaLDT could not complete, and we instead ran a variant where inner-training runs for 50 epochs. More detailed numbers are in Appendix§F.*

	Graph size		MetaLDT		MimicLDT	
Dataset	Nodes	Edges	Time(s)	Mem.	Time(s)	Mem.
Cora	2708	5429	82198.92	2.91GB	43.17	1.38GB
Citeseer	3312	4536	82510.19	3.01GB	41.08	1.54GB
PubMed	19717	44338	—	OOM	105.82	1.89GB
ArXiv	169343	1157799	—	OOM	692.44	9.51GB

Table 2: *Total running time (in seconds) and GPU memory cost of generating one poisoned graph for various datasets on GCN model.*

	Vanilla			Robust				
	GCN	GraphSAGE	GAT	GNNGuard	SoftMedianGDC	JaccardGCN	SVDGCN	ProGNN
Citeseer	0.72	0.69	0.66	0.70	0.59	0.64	0.67	0.61
PubMed	0.71	0.69	0.69	0.70	0.56	0.67	0.60	0.57
ArXiv	0.74	0.73	0.70	0.64	0.59	0.63	0.62	0.58

Table 3: *Poison success rate of MimicLDT. More detailed numbers are in Appendix§F*

	MetaLDT	MimicLDT
Degree changes	0.0419 ± 0.0055	0.0393 ± 0.0021
Homophily changes	0.0142 ± 0.0015	0.0205 ± 0.0010

Table 4: *Changes in the distribution of graph node degrees and homophiliy, measured using Earth Mover's Distance, for the Cora dataset. The values represent the average distance between each poisoned graph and the original input. We report on other datasets in Appendix. D.2 and D.3.*

to GNN analysis tools such as GNNExplainer (Ying et al., 2019). In §E.3, we show that GNNExplainer could detect the short-distance attacks with reasonable accuracy (0.45∼0.85) and recall (0.32∼0.84).

MetaLDT is much more expensive than MimicLDT. MetaLDT requires a lot of compute and memory resources due to its extensive unrolling process. Table. 2 compares the running time (on a V100) and the memory cost of MetaLDT and MimicLDT for the GCN model over various datasets. Not only MetaLDT is 3 orders-of-magnitude slower than MimicLDT, but it can only handle small graphs (Cora, Citeseer) of several thousands nodes due to OOM. Thus, unless otherwise mentioned, the rest of our evaluation uses MimicLDT and Arxiv, our largest dataset.

Ablation Study and more analysis. We perform extensive ablation study and leave its discussion to the Appendix. In particular, we study the effects of hyperparameters (§C.4§ C.1), explore the design rationale of the optimization process (§C.1) and optimization constraints (§C.3). Finally, in §C.2, we show that there are benefits to optimizing the adjacency matrix. Nevertheless, MimicLDT's heuristic of randomly connecting to existing nodes labelled with the target label is fast although imperfect alternative, reducing success rate to 79.5% from 96%.

6.2 Effectiveness of MimicLDT Attack

Because it is feasible, we run MimicLDT over a larger variety of datasets. Table. 3 shows MimicLDT's poison success rate over Citeseer, PubMed and arXiv. For comparison with MetaLDT, we show MimicLDT's Cora results in Table 1. We set the number of attack points to $r*|\mathcal{V}_L|$, where $\mathcal{V}_L$ is the

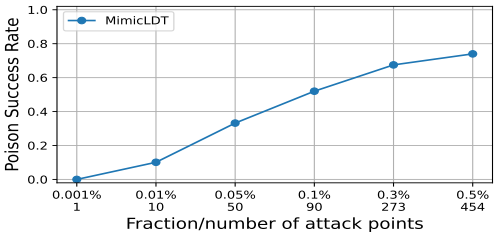

Figure 3: *Poison success rate with varying number of attack points for GraphSAGE on arXiv.*

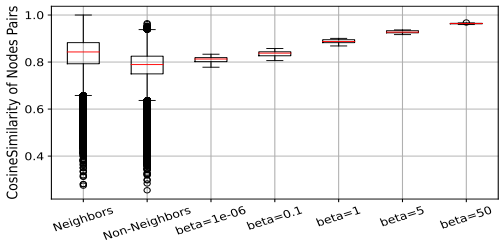

Figure 4: *Similarity between injected nodes (varying β), their attack points, and between neighboring and non-neighboring nodes for GraphSAGE on arXiv.*

set of labeled training nodes, and we use $r = 0.5\%$ for arXiv and PubMed, and $r = 1\%$ for Cora and Citeseer (which are smaller). MimicLDT injects a maximum of $\Delta = \Phi * r|\mathcal{V}_L|$ nodes, and we use $\Phi = 4$ for all experiments. Due to time constraint, instead of training a surrogate model, our experiments directly use the weights of models under attack. We have evaluated both ways over Cora and found they result in similar success rates. As shown in Table. 3, MimicLDT can achieve decent poison success rate, for vanilla models ($66\%\sim74\%$) as well as robust models ($56\%\sim70\%$).

Effect of varying the number of attack points: We study the effect of varying the number of attack points. Fig.3 shows poison success rate as r varies (while keeping Φ fixed). Increasing r, and thus the number of attack points, improves attack success rate. §D.1 studies the effects of varying Φ. The total number of injected nodes is determined by both r and Φ.

Attack stealthiness: degree distribution: We examine whether poisoned graphs can preserve the node degree distribution. We measure the changes to degree distribution using the Earth Mover's Distance (EMD) metric. The average distance between each poisoned graph with original clean graph for Cora is 0.039 ± 0.002. Statistics on other datasets can be found in § D.2. The attacks only cause slight changes on the node degree distribution.

Attack stealthiness: homophily: The second term of MimicLDT's loss function (Eq. 5) keeps injected nodes similar to the attack points they attach to. It serves a similar goal as prior work Chen et al. (2022), which is to ensure that injected nodes do not significantly impact graph homophily. The hyperparameter β controls the importance of the second term. Figure 4 measures the similarity of neighboring and non-neighboring nodes in the arXiv graph, and compare them to the similarity between injected nodes and their attack points with varying β. Appendix§ D.3 gives the detailed setup. We can observe that the larger the value of β, the more similar injected nodes appear to their attack points. As we note in §5, we use feature vector similarity as a proxy for the standard node-centric homophily metric (Chen et al., 2022). In §D.3, we show this does not affect the homophily results.

6.3 End-to-end attacks

The attacks generated by MetaLDT and MimicLDT inject fake nodes whose features lie in continuous space. Thus, they are not end-to-end attacks for graphs with discrete features, such as citation graphs whose raw node features are natural language texts. Thus, an end-to-end attack needs to inject nodes with textual features. We extend our design to perform such an attack.

Suppose some language model such as SciBERT Beltagy et al. (2019) is used to encode a node's raw texts to an embedding vector in continuous space. Our extension trains a decoder that can generate texts given an embedding vector, which corresponds to some fake node's feature as computed by MimicLDT (or MetaLDT). We provide more details on the design and evaluation of end-to-end attack in Appendix§G and give example texts generated for the fake nodes (§G Fig 11).

7 Conclusion

Our work shows that GNNs are susceptible to, long-distance injection attacks, a type of attack that (to the best of our knowledge) have not been investigated in the past. When compared to short-distance attacks, where the attacker modifies the target's neighborhood, long-distance attacks require injecting a larger number of nodes. However, detecting these nodes is challenging, they lie outside the target's k-hop neighborhood and thus defender must consider the influence of all nodes in the graph.

References

Iz Beltagy, Kyle Lo, and Arman Cohan. Scibert: A pretrained language model for scientific text. *arXiv 1903.10676*, 2019.

Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8): 1889–1900, 2000. doi: 10.1162/089976600300015187.

Aleksandar Bojchevski and Stephan Günnemann. Adversarial attacks on node embeddings via graph poisoning. In *International Conference on Machine Learning (ICML)*, 2019.

Heng Chang, Yu Rong, Tingyang Xu, Wenbing Huang, Honglei Zhang, Peng Cui, Wenwu Zhu, and Junzhou Huang. A restricted black-box adversarial framework towards attacking graph embedding models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 3389–3396, 2020.

Jinyin Chen, Yangyang Wu, Xuanheng Xu, Yixian Chen, Haibin Zheng, and Qi Xuan. Fast gradient attack on network embedding. *arXiv preprint arXiv:1809.02797*, 2018.

Jinyin Chen, Yixian Chen, Haibin Zheng, Shijing Shen, Shanqing Yu, Dan Zhang, and Qi Xuan. Mga: momentum gradient attack on network. *IEEE Transactions on Computational Social Systems*, 8(1): 99–109, 2020.

Yongqiang Chen, Han Yang, Yonggang Zhang, Kaili Ma, Tongliang Liu, Bo Han, and James Cheng. Understanding and improving graph injection attack by promoting unnoticeability. In *International Conference on Learning Representations (ICLR)*, 2022.

Enyan Dai, Minhua Lin, Xiang Zhang, and Suhang Wang. Unnoticeable backdoor attacks on graph neural networks. 2023.

Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial attack on graph structured data. In *International conference on machine learning (ICML)*, 2018.

Alexandre Duval and Fragkiskos D Malliaros. Graphsvx: Shapley value explanations for graph neural networks. In *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part II 21*, pp. 302–318. Springer, 2021.

Negin Entezari, Saba A Al-Sayouri, Amirali Darvishzadeh, and Evangelos E Papalexakis. All you need is low (rank) defending against adversarial attacks on graphs. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pp. 169–177, 2020.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Simon Geisler, Tobias Schmidt, Hakan Şirin, Daniel Zügner, Aleksandar Bojchevski, and Stephan Günnemann. Robustness of graph neural networks at scale. In *Advances in Neural Information Processing Systems*, 2021a.

Simon Geisler, Tobias Schmidt, Hakan Şirin, Daniel Zügner, Aleksandar Bojchevski, and Stephan Günnemann. Robustness of graph neural networks at scale. In *Neural Information Processing Systems (NeurIPS 2021)*, 2021b.

Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Neural Information Processing Systems (NIPS 2017)*, 2017.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv 2005.00687*, 2021.

Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, and Yi Chang. Graphlime: Local interpretable model explanations for graph neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

Chao Jiang, Yi He, Richard Chapman, and Hongyi Wu. Camouflaged poisoning attack on graph neural networks. In *Proceedings of the 2022 International Conference on Multimedia Retrieval*, pp. 451–461, 2022.

Wei Jin, Yaxin Li, Han Xu, Yiqi Wang, and Jiliang Tang. Adversarial attacks and defenses on graphs: A review and empirical study. *CoRR*, abs/2003.00653, 2020a.

Wei Jin, Yao Ma, Xiaorui Liu, Xianfeng Tang, Suhang Wang, and Jiliang Tang. Graph structure learning for robust graph neural networks. In *Knowledge Discovery and Data Mining (KDD)*, 2020b.

Mingxuan Ju, Yujie Fan, Chuxu Zhang, and Yanfang Ye. Let graph be the go board: Gradient-free node injection attack for graph neural networks via reinforcement learning. *arXiv preprint arXiv:2211.10782*, 2022.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. 2017.

Xuanqing Liu, Si Si, Xiaojin Zhu, Yang Li, and Cho-Jui Hsieh. A unified framework for data poisoning attack to graph-based semi-supervised learning. 2019.

Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. *Advances in neural information processing systems*, 33:19620–19631, 2020.

Jiaqi Ma, Shuangrui Ding, and Qiaozhu Mei. Towards more practical adversarial attacks on graph neural networks. In *Advances in neural information processing systems (NeurIPS)*, 2020.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

Felix Mujkanovic, Simon Geisler, Stephan Günnemann, and Aleksandar Bojchevski. Are defenses for graph neural networks robust? In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

Toan Nguyen Thanh, Nguyen Duc Khang Quach, Thanh Tam Nguyen, Thanh Trung Huynh, Viet Hung Vu, Phi Le Nguyen, Jun Jo, and Quoc Viet Hung Nguyen. Poisoning gnn-based recommender systems with generative surrogate-based attacks. *ACM Transactions on Information Systems*, 41(3): 1–24, 2023.

Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *Advances in neural information processing systems*, 31, 2018.

Yiwei Sun, Suhang Wang, Xianfeng Tang, Tsung-Yu Hsieh, and Vasant Honavar. Adversarial attacks on graph neural networks via node injections: A hierarchical reinforcement learning approach. In *Proceedings of the Web Conference (WWW)*, 2020.

Shuchang Tao, Qi Cao, Huawei Shen, Yunfan Wu, Liang Hou, and Xueqi Cheng. Rethinking the imperceptibility of node injection attack on graphs. *arXiv 2208.01819*.

Shuchang Tao, Qi Cao, Huawei Shen, Junjie Huang, Yunfan Wu, and Xueqi Cheng. Single node injection attack against graph neural networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pp. 1794–1803, 2021.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

Binghui Wang and Neil Zhenqiang Gong. Attacking graph-based classification via manipulating the graph structure. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

Binghui Wang, Youqi Li, and Pan Zhou. Bandits for structure perturbation-based black-box attacks to graph neural networks with theoretical guarantees. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13379–13387, 2022.

Jihong Wang, Minnan Luo, Fnu Suya, Jundong Li, Zijiang Yang, and Qinghua Zheng. Scalable attack on graph data by injecting vicious nodes. *Data Mining and Knowledge Discovery*, 34:1363–1389, 2020.

Xiaoyun Wang, Minhao Cheng, Joe Eaton, Cho-Jui Hsieh, and Felix Wu. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751*, 2018.

M. Waniek, T.P. Michalak, M.J. Wooldridge, and T. Rahwan. Hiding individuals and communities in a social network. *Nature Human Behaviour*, (2), 2018.

Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. Adversarial examples on graph data: Deep insights into attack and defense. In *International Joint Conference on Artificial Intelligence*, 2019.

Zhaohan Xi, Ren Pang, Shouling Ji, and Ting Wang. Graph backdoor. In *USENIX Security Symposium*, pp. 1523–1540, 2021.

Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. Topology attack and defense for graph neural networks: An optimization perspective. In *International Joint Conference on Artificial Intelligence*, 2019.

Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. 2016.

Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. GNNExplainer: A tool for post-hoc explanation of graph neural networks. In *Advances in neural Information Processing Systems (NeurIPS)*, 2019.

Zhaoning Yu and Hongyang Gao. Motifexplainer: a motif-based graph neural network explainer. *arXiv preprint arXiv:2202.00519*, 2022.

Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. On explainability of graph neural networks via subgraph explorations. In *International Conference on Machine Learning*, pp. 12241–12252. PMLR, 2021.

Xiao Zang, Yi Xie, Jie Chen, and Bo Yuan. Graph universal adversarial attacks: A few bad actors ruin graph learning models. 2020.

Xiang Zhang and Marinka Zitnik. Gnnguard: Defending graph neural networks against adversarial attacks. *Advances in neural information processing systems*, 33:9263–9275, 2020.

Qinkai Zheng, Xu Zou, Yuxiao Dong, Yukuo Cen, Da Yin, Jiarong Xu, Yang Yang, and Jie Tang. Graph robustness benchmark: Benchmarking the adversarial robustness of graph machine learning. *CoRR*, abs/2111.04314, 2021.

Xu Zou, Qinkai Zheng, Yuxiao Dong, Xinyu Guan, Evgeny Kharlamov, Jialiang Lu, and Jie Tang. Tdgia: Effective injection attacks on graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 2461–2471, 2021.

Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. In *ACM SIGKDD international conference on knowledge discovery & data mining*, 2018.

Daniel Zügner and Stephan Günnemann. Adversarial attacks on graph neural networks via meta learning. In *International Conference on Learning Representations (ICLR)*, 2019.

A Design details of MetaLDT and MimicLDT

In this section, we provide pseudocode for our proposed attacks and give a more detailed explanation of MimicLDT.

A.1 Pseudocode of MetaLDT

MetaLDT performs its alternating optimization in rounds (Alg. 1). Within each round, it does one optimization step of the adjacency matrix optimization (Alg. 2) based on meta-gradient with constraints, and q optimization steps of the features matrix optimization (Alg. 3) based on the gradient descent across all injected nodes' all feature dimensions.

Algorithm 1 MetaLDT

Input: graph $\mathcal{G} = (\mathbf{A},\mathbf{X})$, labels for labeled nodes C_L, injected nodes $\mathcal{V}_{inj}$, target node v_t, optimization rounds I, feature optimization iterations q, inner-train epochs T.
Initialize: $\mathcal{G}' = (\mathbf{A}',\mathbf{X}') \leftarrow$ inject Δ nodes into $\mathcal{G}$; $\mathbf{X}'_{inj} = \vec{0}$; no linkages between $\mathcal{V}_{inj}$ to nodes in $\mathcal{G}$
for $rounds \in \{1,2,...,I\}$ **do**
    Adjacency_Matrix_Optimization $(\mathcal{G}',C_L,v_t,y_t,T)$
    for $iter \in \{1,2,...,q\}$ **do**
        Feature_Matrix_Optimization $(\mathcal{G}',C_L,v_t,y_t,T)$
return $\mathcal{G}' = (\mathbf{A}',\mathbf{X}')$

Algorithm 2 Adjacency_Matrix_Optimization $(\mathcal{G}',C_L,v_t,y_t,T)$

Define: $\mathcal{N}_k(v_t)$: k-neighboring nodes of the target node v_t; FLAG=True;
for $t \in \{1,2,...,T\}$ **do**
    $\theta_{t+1} \leftarrow step(\theta_t, \nabla_{\theta_t}\mathcal{L}_{train}(f_{\theta_t}(\mathcal{V}_L;\mathcal{G}');C_L))$
$\nabla^{meta}_{A'} \leftarrow \nabla_{A'}\mathcal{L}^h_{atk}(f_{\theta_T}(v_t;\mathcal{G}');y_t)$
$S(u,v) = \nabla^{meta}_{a_{u,v}}(-2\cdot a_{u,v}+1)$
// Constraints on edges
$S[\mathcal{V}\setminus\mathcal{V}_{inj}, \mathcal{V}\setminus\mathcal{V}_{inj}] = -\infty$
$S[\mathcal{V}_{inj}, \mathcal{N}_k(v_t)] = S[\mathcal{N}_k(v_t), \mathcal{V}_{inj}] = -\infty$
while FLAG == True **do**
    $\hat{e} \leftarrow \text{argmax} S(u,v)$ ▷ $\hat{e} = (u,v), u \in \mathcal{V}_{inj}$
    if $v \in \mathcal{V}$ **then** ▷ v is an original node
        if exist $e = (u,w) \in \mathcal{E}', w \in \mathcal{V}$ **then**
            $\hat{e} \leftarrow \text{argmax}[S(u,v)\setminus\hat{e}]$
        else FLAG=False
    else FLAG=False
$\mathbf{A}' \leftarrow$ insert or remove $\hat{e}$ to or from $\mathbf{A}$

Algorithm 3 Feature_Matrix_Optimization $(\mathcal{G}',C_L,v_t,y_t,T)$

for $t \in \{1,2,...,T\}$ **do**
    $\theta_{t+1} \leftarrow step(\theta_t, \nabla_{\theta_t}\mathcal{L}_{train}(f_{\theta_t}(\mathcal{V}_L;\mathcal{G}');C_L))$
$\nabla^{meta}_{X'} \leftarrow \nabla_{X'}\mathcal{L}^h_{atk}(f_{\theta_T}(v_t;\mathcal{G}');y_t)$
$\mathbf{X}'_{inj} = \mathbf{X}'_{inj} - \alpha\nabla^{meta}_{X'_{inj}}$
$\mathbf{X}' \leftarrow update\ \mathbf{X}'_{inj}$, other nodes remain unchanged

A.2 Pseudocode of MimicLDT

As described in Sec.5, to generate a poison graph $\mathcal{G}'$, we first inject fake nodes in $\mathcal{G}$ (Alg. 4) and then optimize the injected nodes' features using the gradient method (Alg. 5).

Algorithm 4 Inject Nodes – MimicLDT

procedure INJECTNODES($\mathcal{G}$,$\mathcal{V}_s$,Φ) ▷ $\mathcal{G}$ is the benign graph, $\mathcal{V}_s$ is the list of attack points, Φ is the upper bound of injected nodes for each attack point
  Let $\mathcal{G}' = \mathcal{G}$
  for $v_s \in \mathcal{V}_s$ **do**
    $\mathcal{V}_{\text{subg\_s}} = \{v_1, v_2, ..., v_\Phi\}$ ▷ First inject Φ nodes
    $\mathcal{G}' = \mathcal{G}' \cup \mathcal{V}_{\text{subg\_s}}$
    for each possible linkage $(v_i, v_j), v_i, v_j \in \mathcal{V}_{\text{subg\_s}}$ **do**
      add edge $e(v_i, v_j)$ to $\mathcal{G}'$ with probability p ▷ $p = 0.5$
    while True **do** ▷ randomly generate linkage to the attack point (at least one)
      link_to_ap = random.binomial(1, p, Δ) ▷ $p = 0.5$
      if sum(link_to_ap) > 0 **then**
        Break
    for i in $1, ..., \Delta$ **do**
      add edge $e(v_s, v_i)$ to $\mathcal{G}'$ if link_to_ap[i] == 1
    for $v_i \in \mathcal{V}_{\text{subg\_s}}$ **do**
      if deg(v_i) == 0 **then**
        remove v_i from $\mathcal{G}'$
  return $\mathcal{G}'$

Algorithm 5 Injected Nodes Feature Optimization – MimicLDT

Input: graph $\mathcal{G} = (\mathbf{A}, \mathbf{X})$, injected nodes $\mathcal{V}_i$, attack points $\mathcal{V}_s$, target node $\mathcal{V}_t$, pretrained GNN model $f_\theta(\cdot)$, loss L_{atk}
▷ For each inject node v_i, $B(v_i)$ represents its associated attack point
Parameter: β, MaxIters, learning rate α
Initialize: $\mathbf{X}_{v_i} = \mathbf{X}_{B(v_i)}, \forall v_i \in \mathcal{V}_i$
Define: $\mathcal{L}_{atk} = -\left(\frac{1}{|\mathcal{V}_s|} \sum_{v_s \in \mathcal{V}_s} Sim_f\left(\mathbf{h}_{v_s}^{(L)}, \mathbf{h}_{v_t}^{(L)}\right) + \beta * \frac{1}{|\mathcal{V}_i|} \sum_{v_i \in \mathcal{V}_i} Sim_{in}\left(\mathbf{X}_{v_i}, \mathbf{X}_{B(v_i)}\right) \right)$
for $t \in \{1, 2, ..., MaxIters\}$ **do**
  $\mathbf{h}_{v_s}^{(L)} = f_\theta\left(v_s; \mathcal{G}^{(t-1)}\right)$, $\mathbf{h}_{v_t}^{(L)} = f_\theta\left(v_t; \mathcal{G}^{(t-1)}\right)$ ▷ $\mathcal{G}^{(t-1)} = \left(\mathbf{A}, \mathbf{X}^{(t-1)}\right)$
  $\mathbf{X}_{\mathcal{V}_i}^{(t)} = \mathbf{X}_{\mathcal{V}_i}^{(t-1)} - \alpha \nabla_{\mathbf{X}_{\mathcal{V}_i}} \mathcal{L}_{atk}\left(\mathbf{X}_{\mathcal{V}_i}^{(t-1)}\right)$ ▷ For $v_j \notin \mathcal{V}_i, \mathbf{X}_{v_j}^{(t)} = \mathbf{X}_{v_j}^{(t-1)}$
return $\mathcal{G}^* = (\mathbf{A}, \mathbf{X}^*)$

A.3 More Detailed Explanation for MimicLDT Attack

In MimicLDT attack, our feature optimization problem takes Eq.5 as the optimization formulation and uses a stochastic gradient descent based optimizer to compute feature vectors.

Injected nodes influence the *attack point*'s embedding through message passing.: The first term of the loss function aims to make a attack point's embedding close to the target node's embedding. The optimization can only change features for injected nodes, because we assume that the attacker cannot modify others. But injected nodes are in the attack point's k-hop neighborhood, and can thus influence the attack point's final embedding due to GNN message passing. More formally, given a pre-trained GNN classifier f_θ and graph $\mathcal{G}'$, the embedding of nodes are $\mathbf{h}_{v_i}^{(L)} = f_\theta(v_i; \mathcal{G}'), \forall v_i \in \mathcal{V}(\mathcal{G}')$. Our goal is to ensure that the *target node*'s embedding v_t, $\mathbf{h}_{v_t} = f_\theta(v_t; \mathcal{G}')$, is close to the attack point v_s's embedding $\mathbf{h}_{v_s} = f_\theta(v_s; \mathcal{G}')$. Concretely, if we consider a 2-layer GCN model, and attack point v_s, we compute $\mathbf{h}_{v_s}$ as:

$$\mathbf{h}_{v_s} = \sigma\left(\sum_{j \in \mathcal{N}_{v_s}} \frac{1}{c_{v_s,j}} \sigma\left(\sum_{m \in \mathcal{N}_j} \frac{1}{c_{j,m}} \mathbf{h}_{v_s}^{(0)} \theta^{(0)} \right) \theta^{(1)} \right) \tag{7}$$

where $c_{j,m} = \sqrt{\hat{d}_j \hat{d}_m}$, $\hat{d}_j = 1 + deg(j)$. When injected nodes are within the 2-hop neighborhood of the attack point v_s, they will pass their information to their neighbors and finally the attack point to influence the $\mathbf{h}_{v_s}$.

Dataset	Nodes($\|V\|$)	Edges($\|E\|$)	Classes($\|Y\|$)	Labeled nodes
Ogbn-arXiv	169343	1157799	40	90941
SciBERT-embed-arXiv	169343	1157799	40	90941
Cora	2708	5429	7	1708
PubMed	19717	44338	3	18217
Citeseer	3312	4536	6	1812

Table 5: *Dataset statistics.*

GNN	Dropout	Weight Decay	Inner-train-epochs	Others
GCN	0.5	0.0005	200	
GraphSAGE	0.5	0.0005	200	
GAT	0.6	0.0005	200	
GNNGuard	0.5	0.0005	50	$\epsilon=1e-6$
SoftMedianGDC	0.5	0.0005	50	$k=64, \alpha=0.15, T=1.0$
JaccardGCN	0.5	0.0005	200	$\epsilon=0.01$
SVDGCN	0.5	0.0005	200	Rank$=50$
ProGNN	0.5	0.0005	50	$\alpha=0.1, \beta=10.0, \gamma=1.0, \lambda\_=0, \phi=0, \text{lr}_{adj}=0.01$

Table 6: *Hyperparameters for MetaLDT.*

Attack stealthiness.: The second term of equation 5 tries to ensure that injected nodes are similar to other nodes in their neighborhood. We use feature vector similarity Sim_{in} as a proxy for homophily Chen et al. (2022), though we also used node-centric homophily Chen et al. (2022) in §6. To formulate in detail:

We use the following *CosineSimilarity* formulation:

$$Sim_{in} = \frac{\mathbf{X}_{v_i} \cdot \mathbf{X}_{B(v_i)}}{||\mathbf{X}_{v_i}||_2 ||\mathbf{X}_{B(v_i)}||_2} \tag{8}$$

We also did evaluation by using the following *node-centric homophily* formulation:

$$Sim_{in} = \frac{1}{|\mathcal{V}_i|} \sum_{v_i} sim(\sum_{j \in \mathcal{N}_{v_i}} \frac{1}{\sqrt{d_j}\sqrt{d_{v_i}}} \mathbf{X}_j, \mathbf{X}_{v_i}) \tag{9}$$

where $sim(\cdot)$ here is the *CosineSimilarity* and d_j represents the degree of node j.

Our empirical evaluation shows that both similarity metrics perform similarly.

B Details of experimental setup

Dataset statistics: Table. 5 shows detailed statistics for the datasets used in the evaluation.

Model settings: By default, all GNNs used in the experiments have 3 layers (except 2 layers GNN models for Cora), a hidden dimension of 16 for Cora, Citeseer, PubMed, and a hidden dimension of 256 for the ArXiv dataset. We adopt dropout with dropout rate of 0.5 between each layer (i.e., 0.6 for GAT model). We use $5e-4$ weight decay for models except the training for ArXiv graph. By default, we set the maximum GNN models training epochs as 1000 and do the early stopping of 100 epochs by examining the validation accuracy.

Hyperparameter settings: The hyperparameters for MetaLDT and MimicLDT are shown in Table 6 and Table 7), respectively.

C Additional evaluations on MetaLDT

C.1 Alternating vs. combined optimization of adj. matrix and node feature

Apart from adding constraints to enforce long-distance, a key difference of MetaLDT over existing meta-learning based GNN attack (Zügner & Günnemann, 2019) is that MetaLDT performs alternating

GNN	Dropout	Weight Decay	LR	Max epochs	Patience	Others
GCN	0.5	0.0005	0.01	1000	100	
GraphSAGE	0.5	0.0005	0.01	1000	100	
GAT	0.6	0.0005	0.01	1000	100	
GNNGuard	0.5	0.0005	0.01	1000	50	$\epsilon = 1e-2$
SoftMedianGDC	0.5	0.0005	0.01	1000	50	$k = 64, \alpha = 0.15, T = 1.0$
JaccardGCN	0.5	0.0005	0.01	1000	50	$\epsilon = 0.01$
SVDGCN	0.5	0.0005	0.01	1000	50	Rank = 2000 (ArXiv) and 50 (others)
ProGNN	0.5	0.0005	0.01	1000	50	$\alpha = 5e-4, \beta = 1.5, \gamma = 1.0, \lambda\_ = 0, \phi = 0, \text{lr}_{adj} = 0.01$

Table 7: *Hyperparameters for MimicLDT.*

q	1	10	1000
Poison Success Rate	40.5%	52%	96%

Table 8: *Poison success rate with varying q for GCN over Cora. $I = 68$.*

optimization of the adjacent matrix and node features as opposed to combining both in one step. We also tried the combined optimization used in (Zügner & Günnemann, 2019) and achieved poison success rate of 65.5% for GCN on Cora, which is significantly less than 96% achieved by the alternating optimization approach. We find that the combined optimization is heavily biased towards modifying the adjacency matrix as opposed to node feature, as the gain from changing a node feature dimension is much smaller than that of adding/deleting an edge.

C.2 Benefits of optimizing the adjacency matrix

Compared to MetaLDT, our faster attack MimicLDT does not optimize the connections between injected and existing nodes but simply connects injected nodes to each other and to randomly chosen attack points with the target label. To understand the additional benefits of optimizing the adjacency matrix, we run MetaLDT over a fixed adjacency matrix like that used by MimicLDT. MetaLDT is only allowed to optimize the fake nodes' features, like MimicLDT. This feature-only MetaLDT achieves poison success rate of 79.5% for GCN over Cora, compared to 96% achieved by the full MetaLDT. However, if we are to connect fake nodes to attack points with random labels instead of target labels, the poison success rate drops dramatically to 22%. This shows that MimicLDT's heuristic of forming connections is a good, albeit still imperfect, strategy.

C.3 Importance of constraining edges between injected and existing nodes

Apart from constraining each injected nodes to only connect to existing nodes outside of the target's k-hop neighborhood, MetaLDT also constrains each injected node to connect to at most one existing node. We run experiments without this constraint. Not only the resulting poison success rate is worse (71%), the generated poisoned graphs tend to connect only a very small number of fake nodes (i.e, ≤ 3) each of which have a large number of connections with different existing nodes. Thus, due to their high degree, the injected fake nodes are hardly inconspicuous.

C.4 Effects of hyperparameters

We evaluate the effects of q and I. Hyperparameter q refers to the number of optimization steps on node features for each each optimization step of the adjacency matrix. Hyperparameter I refers to the total number of rounds where each round takes one optimization step on adjacency matrix and q steps on node feature. Table. 8 shows the effects of varying q while keeping $I = 68$ unchanged. As we can see, the success rate improves with larger q. We believe this is because larger q allows feature optimization to converge better for each given adjacency matrix change. By default, our experiments use $q = 1000$.

Increasing the rounds of optimization could make MetaLDT convergence to a higher poison success rate at the cost of extra running time. As shown in Figure. 5, MetaLDT's optimization is quite efficient: at $I = 68$, the poison success rate has mostly converged. Since the number of optimization rounds must match or exceed the number of allowed adjacency matrix modification ($I \geq \Delta = 68$), $I = 68$ is the smallest sensible I.

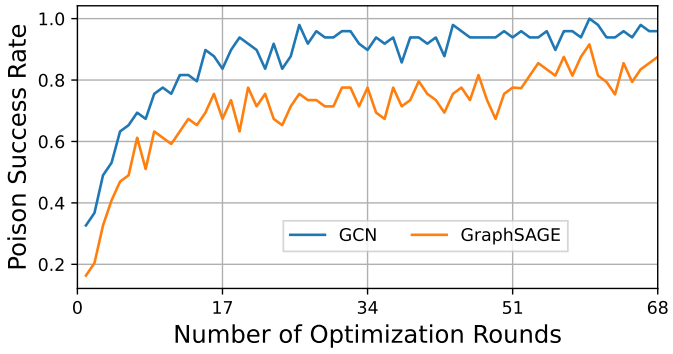

Figure 5: *Poison success rate as the rounds of optimization increases.*

	Vanilla			Robust				
	GCN	GraphSAGE	GAT	GNNGuard	SoftMedianGDC	JaccardGCN	SVDGCN	ProGNN
MetaLDT (non-adaptive)	0.96	0.76	0.51	0.19	0.30	0.72	0.14	0.31
MetaLDT-Adaptive.	0.96	0.87	0.84	OOM	OOM	0.91	0.83	OOM
MetaLDT-Adaptive (ET)	0.69	0.73	0.68	0.53	0.58	0.86	0.62	0.55

Table 9: *Poison success rate of MetaLDT on Cora. In the adaptive setting, MetaLDT's inner training loops takes into account the GNN defense used, and vice versa. To handle OOM cases, we also evaluate MetaLDT in a setting with fewer inner training epochs (50) instead of the default (200). This setting is referred to as (ET, early termination).*

Dataset	Cora	ArXiv	PubMed	Citeseer
EMD (degree)	0.0393 ± 0.0021	0.1189 ± 0.0007	0.0550 ± 0.0008	0.0186 ± 0.0021

Table 10: *The average graph degree distribution changes for different datasets according to the EMD metric.*

C.5 Importance of adapting MetaLDT to GNN defenses

In Section 6.1 (Table 1), we show MetaLDT's performance when its inner training adapts to the GNN defense mechanisms used. The implementation of the adaption follows the work (Mujkanovic et al., 2022). We also experimented with non-adaptive MetaLDT its surrogate model is the vanilla GCN, no matter what defense mechanism is. As shown in Table. 9, non-adaptive MetaLDT fares much worse than MetaLDT, e.g. lowering success rate to 14% from 62% for SVD. Our finding is consistent with that reported by (Mujkanovic et al., 2022).

D Additional evaluations on MimicLDT

D.1 Effect of Φ

We evaluate how the value of Φ, which determines the number of nodes injected per attack point, affects poison success rate. We show the results in Figure 6 show the poison success rate with varying Φ for GraphSAGE over arXiv. We observe that the success rate increases as the value of Φ increases, but the additional benefit is small beyond $\Phi = 4$, which is our default value.

D.2 More on attack graph's node degree distribution

Table. 10 shows the node degree distribution change as a result of the MimicLDT attack for different datasets. The change is measured in the Earth Mover's Distance (EMD) metric.

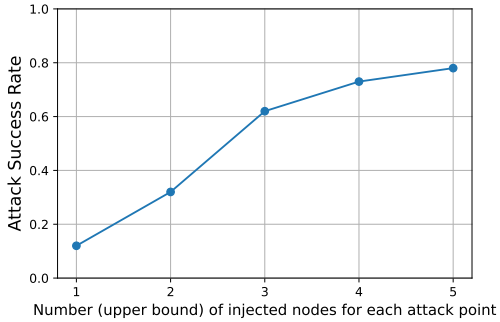

Figure 6: *Attack success rate of varying Φ, which is the upper bound number of injected nodes for each attack point. keep other settings and hyperparameters to be the same.*

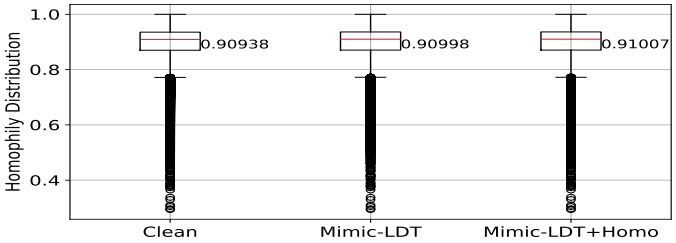

Figure 7: *Homophily distribution for clean and poisoned graph over arXiv. MimicLDT-homo refers to a variant of MimicLDT that uses the node-centric homophily metric for its loss function.*

Dataset	Cora	ArXiv	PubMed	Citeseer
EMD (homophily)	0.0205 ± 0.0010	0.0008 ± 0.0003	0.0123 ± 0.0004	0.0164 ± 0.0014

Table 11: *The average graph homophily distribution changes for different datasets according to the EMD metric.*

D.3 More on attack graph homophily

In MimicLDT, we use feature vector similarity as a proxy for homophily. We also evaluate a variant of MimicLDT, called MimicLDT-homo, that uses the original node homophily metric (Chen et al., 2022) in the loss function. Figure 7 shows that both MimicLDT and MimicLDT-homo can preserve graph homophily well.

We additionally measure changes in node homophily distribution after the attack using the EMD metric for all datasets, shown in Table 11. As can be seen, the attacks only result in slight changes on graph homophily distribution.

E Comparison to short-distance baselines

We compare against three existing short-distance attacks in the targeted poisoning setting: **Nettack** Zügner et al. (2018), **FGA** Chen et al. (2018) and **IG-FGSM** Wu et al. (2019).

E.1 Experiment settings

We use the implementation from DeepRobust's[2], but modify the loss function so that the attacker's goal is the same as ours, which is to flip the target node's label to a specific label of the attacker's choosing instead of *any* arbitrary incorrect label.

[2]Open-sourced DeepRobust library: https://github.com/DSE-MSU/DeepRobust

		Mimic-LDT	Direct attacks	Indirect attacks	
Dataset	Avg. Degree	r	perturbations	perturbations	influencers
Cora	3.84	0.01	34	34	10
ArXiv	13.67	0.005	N/A	N/A	N/A
PubMed	4.50	0.01	182	182	20
Citeseer	2.61	0.005	36	36	10

Table 12: *Hyperparameter setups for baseline attacks. Direct attacks includes Nettack-direct, FGA and IG-FGSM; Indirect attack includes Nettack-indirect.*

The baselines are modification attacks, aka they can either perturb the graph structure or existing nodes' features. By contrast, MetaLDT and MimicLDT are injection attacks. Thus, it is impossible to directly compare them. Therefore, to make these baseline attacks somewhat comparable to ours, we limit the existing attacks to perturb the graph structure only. From a practical perspective, it is much harder (or even impossible) for an attacker to modify an existing node's features than to create a link to it.

Baseline attacks setups.: In order to compare with MimicLDT attack, we did experiments on baseline attacks (i.e., Nettack, FGA, IG-FGSM) according to the Table. 12. Because for each graph in MimicLDT attack, in expectation, the new edges created to link with the real nodes is $p*\Delta = p*\Phi*r|\mathcal{V}_L|$. In our experiment, $\Phi = 4$, $p = 0.5$, therefore the average newly created edges is $2r|\mathcal{V}_L|$. Thus, correspondingly, we allow the number of perturbations in each baseline attack to be $2r|\mathcal{V}_L|$ (i.e., $r = 1\%$ in Cora and Citeseer; 0.5% for arXiv and PubMed). As for the indirect attack (i.e., Nettack-indirect), the number of influencers (i.e., the attack perturbs the edges that connect to up to how many of the target's neighbors) should be at least greater than average degree. As mentioned in (Zügner et al., 2018), increasing the number of influencers will greatly increase the running time for attacks. We set the number of influencers to be 10 for Cora and Citeseer, and 20 for PubMed.

E.2 Comparing to baselines with varying edge perturbation budgets

Here, we show a comparison by varying the edge perturbation budgets in baseline attacks and MimicLDT attack for Cora dataset and GCN model. We used three settings (i.e., add one more setting) for Nettack here: direct where the target node's edges are perturbed; indirect with 2 influencers, where the attack perturbs the edges that connect to up to 2 of the target's direct neighbors; and indirect with 10 (which is more than Cora's average node degree) influencers.

Figure 8 shows baseline attacks' poison success rate as we vary their edge perturbation budget, while Figure 9 shows MimicLDT's success rate as we vary the number of edges between attack points and injected nodes. As we probabilistically connect fake nodes to their associated attack points, the number of edges between the two, as shown as the x-axis in Figure 9, is in expectation. We can see that all baseline attacks, especially direct attacks (Nettack-direct, FGA, IG-FGSM), are much more efficient than our attack in terms of the number of edge budget required. Indirect (aka influence) attacks are much less efficient than direct ones, but they can still be more efficient than our attack under some settings (e.g., Nettack indirect with 10 influencers when budget is 17). However, we note that an indirect attack is not a long-distance attack, since it must modify the target's neighbors. This shows that long-distance attacks, which cannot change the target neighborhood, carry an efficiency cost.

E.3 Detecting baseline attacks using GNN explanability tools

By modifying the target node's k-hop neighborhood, baseline attacks are vulnerable to existing GNN explanability tools. We evaluate this by measuring the likelihood that GNNExplainer Ying et al. (2019) can detect perturbations from the baseline attacks.

GNNExplainer takes as input a target node, graph and GNN model, and returns the subgraph that has the largest influence on the target node's prediction. Our experiments use the GNNExplainer implementation included in the PyTorch-Geometric library[3] to find this subgraph for the targeted node, and we use this returned subgraph to compute precision and recall. The baseline attacks can both and

[3] `https://pytorch-geometric.readthedocs.io/en/latest/modules/explain.html`

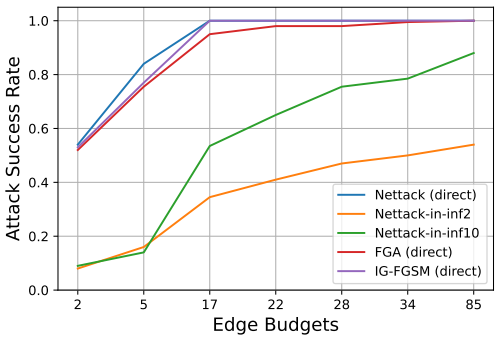

Figure 8: *Attack success rate when varying the edge perturbation budget for direct Nettack, indirect Nettack with 2 or 10 influencers, FGA and IG-FGSM.*

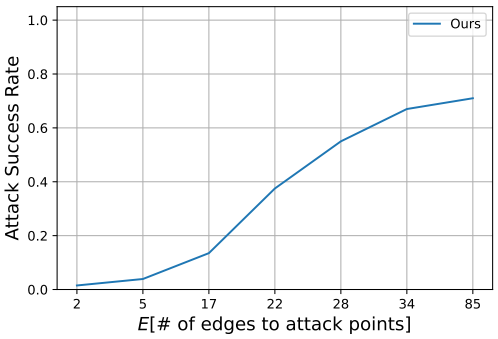

Figure 9: *Attack success rate as a function of the number of edges connecting injected nodes and attack points.*

Attacks	Settings	thres.:0 Avg Rec.	thres.:0 Avg Prec.	thres.:0.6 Avg Rec.	thres.:0.6 Avg Prec.	thres.:0.8 Avg Rec.	thres.:0.8 Avg Prec.
Nettack	Direct; budget=5	0.99	0.19	0.98	0.27	0.84	0.45
Nettack	Direct; budget=34	0.98	0.31	0.52	0.82	0.32	0.85
Nettack	Indirect; budget=34 n_influencers=2	0.74	0.44	0.74	0.51	0.52	0.54
Nettack	Indirect; budget=34 n_influencers=10	0.79	0.48	0.79	0.49	0.44	0.57
FGA	Direct; budget=34	0.99	0.27	0.55	0.68	0.55	0.85
IG-FGSM	Direct; budget=34	0.98	0.30	0.56	0.71	0.53	0.83

Table 13: *Run GNNExplainer to detect short-distance baseline attacks.*

remove edges. However, GNNExplainer cannot indicate removed edges in its explanation. Therefore, we only consider edges added by the attack when computing precision and recall: *recall* is the fraction of edges added by the attack contained in the GNNExplainer subgraph, while *precision* is the fraction of edges in the subgraph that were added by the attack:

$$recall = \frac{\text{Number of attack-added edges exist in GNNExplainer subgraph}}{\text{Number of total attack-added edges}} \tag{10}$$

$$precision = \frac{\text{Number of attack-added edges exist in GNNExplainer subgraph}}{\text{Number of total edges in GNNExplainer subgraph}} \tag{11}$$

Table 13 shows the average recall and precision for different attacks and settings. GNNExplainer allows users to specify a threshold, and selects only those edges whose importance score is above the specified threshold. We show results for three thresholds: 0.0 (the default value), 0.6 and 0.8. We find that GNNExplainer can find a significant fraction of the perturbed edges, and that at higher thresholds it has a recall that is above 0.5 (so it finds more than half the added edges), while also having a precision ranging from 0.4 — 0.85 thus making it a feasible tool for automated detection and defense against such attacks.

F Detailed results over different datasets and short-distance baselines

In this section, we give the detailed results of MetaLDT over Cora and MimicLDT over all four datasets. We report the results by datasets: Table 14 (Cora), Table 15 (Citeseer), Table 16 (Pubmed), and Table 17 (arXiv). These tables correspond to the same experiments as the summary tables presented in §6 (Table 1 and Table 3).

		Long-distance		Short-distance (targeted, modification)			
		MetaLDT	MimicLDT	Nettack-direct	Nettack-indirect	FGA	IGSM
Vanilla	GCN	0.96	0.67	1.00	0.79	0.98	1.00
	GraphSAGE	0.87	0.63	0.96	0.42	0.58	0.70
	GAT	0.84	0.60	0.97	0.53	0.72	0.86
Robust	GNNGuard	(0.53)	0.70	1.00	0.98	0.94	0.96
	SoftMedianGDC	(0.58)	0.55	1.00	0.46	0.88	0.94
	JaccardGCN	0.91	0.66	1.00	0.47	0.90	0.96
	SVDGCN	0.83	0.74	1.00	0.18	0.96	1.00
	ProGNN	(0.55)	0.59	1.00	0.60	0.90	0.97

Table 14: *Detailed Results on Cora.*

		Long-distance	Short-distance (targeted, modification)			
		MimicLDT	Nettack-direct	Nettack-indirect	FGA	IGSM
Vanilla	GCN	0.72	0.98	0.82	1.00	0.88
	GraphSAGE	0.69	1.00	0.74	0.94	0.98
	GAT	0.66	0.97	0.75	0.92	0.94
Robust	GNNGuard	0.70	0.88	0.98	1.00	1.00
	SoftMedianGDC	0.59	1.00	0.38	0.98	0.95
	JaccardGCN	0.64	1.00	0.74	0.96	0.94
	SVDGCN	0.67	0.96	0.78	0.98	1.00
	ProGNN	0.61	0.98	0.92	1.00	0.97

Table 15: *Detailed Results on Citeseer.*

		Long-distance	Short-distance (targeted, modification)			
		MimicLDT	Nettack-direct	Nettack-indirect	FGA	IGSM
Vanilla	GCN	0.71	1.00	1.00	1.00	1.00
	GraphSAGE	0.69	1.00	0.84	1.00	OOM
	GAT	0.69	1.00	0.82	1.00	OOM
Robust	GNNGuard	0.70	1.00	1.00	1.00	OOM
	SoftMedianGDC	0.56	1.00	0.45	1.00	OOM
	JaccardGCN	0.67	1.00	1.00	1.00	1.00
	SVDGCN	0.60	1.00	0.09	1.00	0.97
	ProGNN	0.57	1.00	0.58	1.00	OOM

Table 16: *Detailed Results on PubMed. OOM means out-of-memory under the GPU limitation.*

		Long-distance	Short-distance (targeted, modification)			
		MimicLDT	Nettack-direct	Nettack-indirect	FGA	IGSM
Vanilla	GCN	0.74	OOM	OOM	OOM	OOM
	GraphSAGE	0.73	OOM	OOM	OOM	OOM
	GAT	0.70	OOM	OOM	OOM	OOM
Robust	GNNGuard	0.64	OOM	OOM	OOM	OOM
	SoftMedianGDC	0.59	OOM	OOM	OOM	OOM
	JaccardGCN	0.63	OOM	OOM	OOM	OOM
	SVDGCN	0.62	OOM	OOM	OOM	OOM
	ProGNN	0.58	OOM	OOM	OOM	OOM

Table 17: *Detailed Results on ArXiv. OOM means out-of-memory under the GPU limitation.*

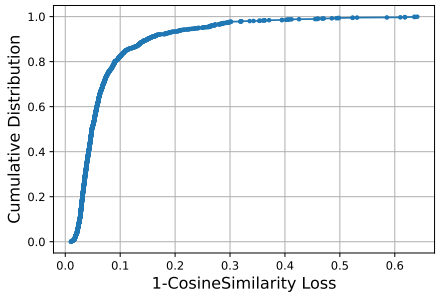

Figure 10: *An example of successfully poisoned end-to-end attack: the CDF of the 1-CosineSimilarity loss of f_{opt} and f'_{opt} for all injected nodes.*

G Design and evaluation of end-to-end attacks

We describe our extension to MimicLDT that allows it to generate textual features for fake nodes to attack GNN models over citation graphs.

G.1 End-to-end attack design

We assume that the attacker has access to the encoder model weights used by the GNN to embed textual node features. This assumption is reasonable as pretrained language models, such as SciBERT (Beltagy et al., 2019), are widely available for embedding purpose.

Our design requires the attacker to use the encoder and some corpus of text to train a *decoder model* that can be used to generate fake texts from arbitrary embedding. The *decoder* model is modified from an existing encoder-decode architecture such as SciBERT. We change the decoder to take a single embedding vector as input instead of an embedding matrix containing a vector for each token. We also modify the overall encoder-decoder so that the encoder's output is the final embedding of the classification token (`[CLS]`) which is then used as the decoder's input.

To train the decoder, we fine-tune a pre-trained encoder-decoder model. During the fine-tuning process, we use the victim's encoder weights (which are provided as an input) and freeze them. The fine-tuning process can thus only update decoder weights, and must do so to ensure that the output sequence is close to the input sequence.

Once we have trained the decoder model, we can use it for end-to-end attack as follows. First, we generate the fake nodes using MimicLDT (or MetaLDT) and attach them to the existing graph accordingly. Then, we take the continuous feature vectors of each injected node and pass them as input to the decoder model, which generates text that the attacker can use to create the corresponding fake nodes with raw textual features.

G.2 End-to-end attack evaluation

We evaluate the end-to-end attack for GraphSAGE over arXiv with SciBERT embedding. We refer to this dataset as SciBERT-embed-arXiv which is generated by passing the title and abstract of each paper in the Ogbn-arXiv dataset through SciBERT, and using the encoder's output (i.e., the [CLS] token embedding after the final pooling layer) as the node's feature in the graph. As we previously stated (§5), across 50 experiments, we observed a poison success rate of 84%. Furthermore, our manual inspection showed that the generated title and abstract are readable, and we show examples in Table 18 and Table 19.

Finally, we also found that the text generated by the decoder lead to node features that are close to what was generated by the attack optimization formulation: Figure 10 shows a CDF of node feature similarity between nodes injected by the end-to-end attack (after they have been passed through an encoder) and injected node features generated by the attack optimization, and over 80% of the nodes have a cosine distance smaller than 0.1.

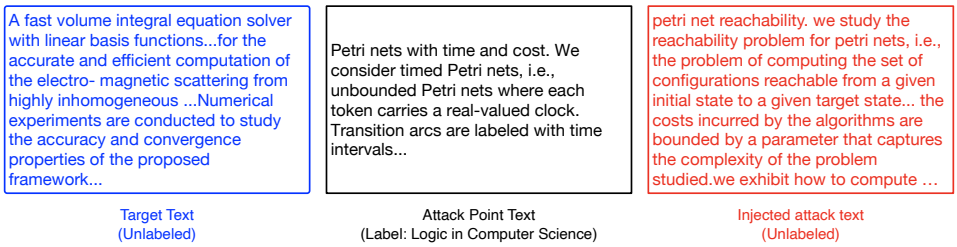

Figure 11: *An example text generated for a fake node. The attack manages to flip the target node's label from* Numerical Analysis *to* Logic in Computer Science*.*

Case study. In Fig 11, we show an example text generated for an unlabeled fake node to flip the target node's label from "Numerical Analysis" to "Logic in Computer Science". Addtional examples can be found in Table 18 and Table 19.

Table 18: *More examples (Part-1) of target node, one base node, and one of the generated fake-text link to this base. The category (i.e., labels) of the base/target node shown in first column, generated injected node is unlabelled. Due to space limit, only show partial of the contents.*

	Generated fake-text *(link to base)*	*petri net reachability. we study the reachability problem for petri nets, i.e., the problem of computing the set of configurations reachable from a given initial state to a given target state... the costs incurred by the algorithms are bounded by a parameter that captures the complexity of the problem studied.we exhibit how to compute optimal solutions...*
Logic in Computer Science	Base: Title and Abstract	Petri nets with time and cost. We consider timed Petri nets, i.e., unbounded Petri nets where each token carries a real-valued clock. Transition arcs are labeled with time intervals...
Numerical Analysis	Target: Title and Abstract	A fast volume integral equation solver with linear basis functions...for the accurate and efficient computation of the electromagnetic scattering from highly inhomogeneous ...Numerical experiments are conducted to study the accuracy and convergence properties of the proposed framework...
	Generated fake-text *(link to base)*	*on the performance of real time text and speech enhancement in mobile communication systems. in this paper, we present a comprehensive study on the effectiveness of a real - time text - and - speech enhancement technique for automatic speech recognition (asr) applications. we consider a scenario in which a mobile phone communicates with a user through a background noise - free background noise channel, while the phone continuously monitors the speech signal and provides feedback to the user about the quality of its speech enhancement...as a complement to an extensive performance evaluation in real - life applications...*
Information Theory	Base: Title and Abstract	On the performance of selection cooperation with imperfect channel estimation. In this paper, we investigate the performance of selection cooperation in the presence of imperfect channel estimation. In particular, we consider a cooperative scenario with multiple relays and amplifyand-forward protocol over frequency flat fading channels...outage probability and average capacity per bandwidth of the received signal in the presence of channel estimation errors. A simulation study...
Multimedia	Target: Title and Abstract	High quality low delay music coding in the opus codec. The IETF recently standardized the Opus codec as RFC6716. Opus targets a wide range of real-time Internet applications by combining a linear prediction coder with a transform coder. We describe the transform coder, with particular attention to the psychoacoustic knowledge built into the format. The result out-performs existing audio codecs that do not operate under real-time constraints.

Table 19: *More examples (Part-2) of target node, one base node, and one of the generated fake-text link to this base. The category (i.e., labels) of the base/target node shown in first column, generated injected node is unlabelled. Due to space limit, only show partial of the contents.*

	Generated fake-text *(link to base)*	*uncertainty quantification in automated planning and verification of automated control systems. this paper presents an uncertainty quantification framework for automated control system verification. uncertainty quantification is carried out in three steps ... verification of the generated probabilistic model against real - world constraints. the uncertainty quantification problem is formulated as a mixed integer linear program (milp) and solved using a branch - and - bound approach. the results are applied to a case study of the automated control of a heating, ventilation and air conditioning (hvac) system in a residential building...*
Computational Engineering	Base: Title and Abstract	Heuristic optimization for automated distribution system planning in network integration studies. Network integration studies try to assess the impact of future developments, such as the increase of Renewable Energy Sources or the introduction of Smart Grid Technologies... This allows the estimation of the expected cost in massive probabilistic simulations of large numbers of real networks...
Numerical Analysis	Target: Title and Abstract	Numerical verification of affine systems with up to a billion dimensions. Affine systems reachability is the basis of many verification methods. With further computation, methods exist to reason about richer models with inputs, nonlinear differential equations, and hybrid dynamics... In this paper, we improve the scalability of affine systems verification... this direct approach requires an intractable amount of memory while using an intractable amount of computation time...
	Generated fake-text *(link to base)*	*reactive planning and control of autonomous vehicles in uncertain environments. this paper deals with the problem of reactively driving an autonomous vehicle in a uncertain environment. in particular, we assume that the vehicle is equipped with a collision avoidance system, and that the environment is uncertain... the proposed reactive strategy is able to perform reactively and reliably, while taking into consideration the uncertainties of both the environment and the vehicle dynamics.*
Machine Learning	Base: Title and Abstract	Neural networks trained with wifi traces to predict airport passenger behavior. The use of neural networks to predict airport passenger activity choices inside the terminal is presented in this paper. Three network architectures are proposed...A real-world case study exemplifies the application of these models, using anonymous WiFi traces collected at Bologna Airport to train the networks...
Systems and Control	Target: Title and Abstract	An unconditionally stable first order constraint solver for multibody systems. This article describes an absolutely stable, first-order constraint solverfor multi-rigid body systems that calculates (predicts) constraint forces for typical bilateral and unilateral constraints, contact constraints with friction, and many other constraint types...I assess the approach on some fundamental multibody dynamics problems.