BÁO CÁO PROJECT

NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification

Các bước thực hiện:

Table of Contents

1. NETWORK FLOW GRAPH EXTRACTION

3 NETWORK FLOW GRAPH EXTRACTION

To decide whether a particular candidate instance of an application is malicious, we collect all network traffic generated during the execution of that application instance within a given time interval after installation. The resulting data consists of a set of network flows described by feature vectors that can be extracted from pcap-files using tools such as *CICFlowMeter* [13]. Thereby, each flow F describes network traffic associated with one particular `(Source-IP, Source-Port, Destination-IP, Destination-Port, Protocol)`-tuple during the considered time frame and has a feature vector $f \in \mathbb{R}^d$ attached to it. Typical features include the number of packets sent, mean and standard deviation of the packet length, or minimum and maximum interarrival time of the packets.

From the resulting set of flows $\mathcal{F}$, we extract a flow graph, where the nodes correspond to endpoints in the network and edges model communication between these endpoints. Instead of considering `(IP, Port)`-tuples, we factor out the port information and consider IP-endpoints for two main reasons:

(1) Apart from standard ports, port selection is often arbitrary and could even be subject to port randomization techniques, leading to arbitrary and potentially misleading graph structure.
(2) Empirically, we found `(IP, Port)`-graphs to be very sparse and rather uninformative. In comparison, IP-graphs exhibit much more interesting topologies.

More specifically, from a set of flows $\mathcal{F}$, we extract a directed graph $G = (V, E)$ where the nodes correspond to endpoints involved in any flow $F \in \mathcal{F}$ and a directed edge is added for all pairs (s_i, t_i) for which there exists a flow $F_i \in \mathcal{F}$ with source and target IP s_i and t_i, respectively. The feature vector assigned to this edge aggregates the feature vectors $f_i \in \mathbb{R}^d$ of all flows F_i along this edge using a set of five aggregation functions. For each feature, the shape of the distribution of values along the edge is described using the first four moments, namely the mean, standard deviation, skew and kurtosis, and the median value. The aggregate values are computed for each feature and then concatenated, resulting in a feature vector $x_i \in \mathbb{R}^{5d}$ for each edge $e_i \in E$. The flow graph extraction procedure is illustrated in Figure 1a. Figure 1b shows an exemplary graph extracted from real data.

Intuitively, the resulting graph captures how network traffic flows between different endpoints in the monitored network during a specific time frame. The graph structure reveals where traffic is flowing, and the additional edge attributes describe how it is flowing. Connecting individual flows in a graph provides a much richer relational representation compared to treating flows individually. Thus, we expect models learning from these graphs to perform significantly better at detection and classification tasks than models which classify individual flows. Our experimental results confirm this intuition.

We wish to note that such graphs could potentially be used for further applications, such as intrusion detection or identifying devices generating malicious traffic.

4 NETWORK FLOW GRAPH NEURAL NETWORKS

From a machine learning perspective, we consider two different problems: Supervised graph classification and unsupervised graph anomaly detection. To solve these problems, we introduce a novel graph neural network model, which learns expressive representations from network flow graphs, along with three different variants of that base model, a supervised graph classifier, an unsupervised graph autoencoder, and an unsupervised one-class graph neural network. The different model variants are illustrated in Figure 2c.

5.1 Dataset Preparation

Our extracted dataset consists of 2126 samples, where each sample corresponds to one instance of an android application installed and executed on a mobile phone. For each sample, all network flows within the network during execution are captured. For each flow, 80 features are recorded, including, e.g., the number of packets sent, mean and standard deviation of the packet length, and minimum and maximum interarrival time of the packets. For a more detailed description of the data collection process, we refer to [26].

For each sample, 3 different labels are available, a binary label indicating whether the application is malicious or not, a category label with 5 possible values indicating the general class of malware, and a family label with 36 different values indicating the specific type of malware. Malware families with fewer than 9 samples have been removed to ensure a reasonable split into train, validation, and test sets. Consequently, for the family prediction task, only 2071 samples are available.

For each sample, we extract a graph as described in Section 3 and remove the flow-id, timestamp, and endpoint IP and port information from the feature set. Additionally, we remove all features that are constant among all edges of all graphs, leaving 330 edge features.

1.1: Xử lí dữ liệu trước khi chuyển thành đồ thị

1. Loại bỏ các cột có giá trị giống nhau trong tất cả các flow và file csv, và các cột không cần thiết

2. Xử lí địa chỉ IP ở 2 cột Source IP và Destination IP chuyển sang dạng số

3. Gán lại nhãn cho mỗi flow, có 5 loại nhãn từ 0 đến 4 tương ứng với bình thường và 4 loại malware

```
def preprocessing_data(csv_file):
    # drop features that all of flow have value equal to zero
    columns_to_drop = [
        " ECE Flag Count",
        " Fwd Avg Packets/Bulk ",
        " Fwd Avg Bulk Rate",
        " Bwd Avg Bytes/Bulk",
        " Bwd Avg Packets/Bulk",
        " Bwd Avg Bulk Rate",
        "Flow ID",
        " Source Port",
        " Destination Port",
        " Timestamp",
    ]
    # Tạm thời lấy 200 flow đầu
    data = pd.read_csv(csv_file, usecols=lambda column: column not in columns_to_drop,
nrows=200)

    data.columns = data.columns.str.strip()

    # loại bỏ dòng mà có địa chỉ ip không hợp lệ
    def is_valid_ipv4(ip):
        try:
            ipaddress.IPv4Address(ip)
            return True
        except ipaddress.AddressValueError:
            return False

    # Lọc dữ liệu dựa trên địa chỉ IP không hợp lệ
    valid_ip_mask = data.apply(
        lambda row: is_valid_ipv4(row["Source IP"])
        and is_valid_ipv4(row["Destination IP"]),
        axis=1,
    )
    data = data[valid_ip_mask]

    # Filter out rows with None values (invalid IP addresses)
    data = data.dropna(subset=["Source IP", "Destination IP"])

    def ip_address(ip):
        ip_integer = int(ipaddress.IPv4Address(ip))
```

```
        return ip_integer

    data["Source IP"] = data["Source IP"].apply(lambda i: ip_address(i))
    data["Destination IP"] = data["Destination IP"].apply(lambda i: ip_address(i))
    data = data.dropna()
    #if data[] is None delete data[]
    # encode label for binary classification task
    def encode_label(label):
        if label == "BENIGN":
            return 0
        elif "ADWARE" in label:
            return 1
        elif "RANSOMWARE" in label:
            return 2  # Assuming Ransomware is also considered malicious
        elif "SCAREWARE" in label:
            return 3  # Assuming Scareware is also considered malicious
        elif "SMSMALWARE" in label:
            return 4  # Assuming SMSMalware is also
    data["Label"] = data["Label"].apply(lambda i: encode_label(i))
    graph_label = data["Label"].unique()
    return data, graph_label
```

1.2: Hàm tạo flow graph

Từ một tập hợp các luồng F, trích xuất một đồ thị có hướng G = (V, E) trong đó:

- Các nút tương ứng với các điểm cuối được liên quan đến bất kỳ luồng F nào và một cạnh có hướng được thêm vào cho tất cả các cặp (s_i, t_i) mà tồn tại một luồng F_i trong F có nguồn và đích IP là s_i và t_i tương ứng.
- Vectơ đặc trưng được gán cho cạnh này tổng hợp các vectơ đặc trưng $f_i \in R^d$ = của tất cả các luồng F_i dọc theo cạnh này bằng cách sử dụng một tập hợp năm hàm tổng hợp. Đối với mỗi đặc trưng, hình dạng của phân phối các giá trị dọc theo cạnh được mô tả bằng cả bốn độ lệch chuẩn đầu tiên, cụ thể là **trung bình, độ lệch chuẩn, độ lệch và độ nhọn, và giá trị trung vị**. Các giá trị tổng hợp được tính toán cho mỗi đặc trưng và sau đó được nối lại, kết quả là một vectơ đặc trưng $x_i \in R^{5d}$ cho mỗi cạnh $e_i \in E$.

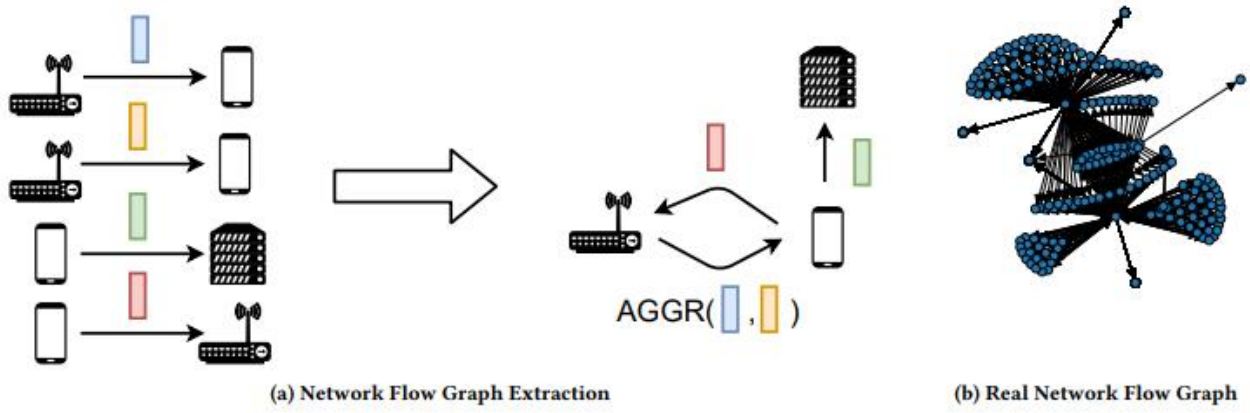

Figure 1: From a set of network flows recorded during a specific time frame, a flow graph is constructed by adding directed edges between all pairs of endpoints that communicated with at least one flow. Each flow is described by a feature vector summarizing its corresponding network traffic. Edges in the flow graph are annotated with these feature vectors, where feature vectors of parallel flows are aggregated. The topology of a real flow graph extracted from network traffic generated during execution of a *FakeAV* Scareware application is shown in (1b).

```python
def create_flow_graph_from_csv(data, idx, graph_label):

    end_point = data["Source IP"].astype(str) + data["Destination IP"].astype(str)
    nodes_list = end_point.unique()
    num_nodes = len(nodes_list)

    # Create edge_index using mapped indices
    edges_list = data[["Source IP", "Destination IP"]].values.tolist()
    edge_index = np.array(
        edges_list
    ).T

    edge_attr_list = []
    for edge in edges_list:
        src_ip = edge[0]
```

```
        dst_ip = edge[1]

        # Lọc dữ liệu theo cạnh hiện tại
        edge_data = data[
            (data["Source IP"] == src_ip) & (data["Destination IP"] == dst_ip)
        ]
        if not edge_data.empty:
            # Compute aggregation functions for features
            edge_attr = []
            for feature in edge_data.columns:
                if feature not in ["Source IP", "Destination IP"]:
                    values = edge_data[feature]

                    #Tính trung bình
                    if not np.isnan(np.nanmean(values)):
                        mean_value = np.nanmean(values)
                    else:
                        mean_value = -1

                    #Tính độ lệch chuẩn
                    if not np.isnan(np.nanstd(values)):
                        std_value = np.nanstd(values)
                    else:
                        std_value = -1

                    #Tính độ lệch
                    if not np.isnan(values.skew()):
                        skew_value = values.skew()
                    else:
                        skew_value = -1

                    # Tính độ nhọn
                    if not np.isnan(values.kurtosis()):
                        kurtosis_value = values.kurtosis()
                    else:
                        kurtosis_value = -1

                    #Tính giá trị trung vị
                    if not np.isnan(np.nanmedian(values)):
                        median_value = np.nanmedian(values)
                    else:
                        median_value = -1

                    edge_attr.extend(
                        [
                            mean_value,
                            std_value,
                            skew_value,
                            kurtosis_value,
                            median_value,
```

```
                        ]
                    )

            edge_attr_list.append(edge_attr)
    node_attr = torch.zeros(num_nodes, 375)
    edge_attr = torch.tensor(edge_attr_list, dtype=torch.float)
    label = torch.tensor(graph_label)
    flow_graph = Data(x = node_attr, edge_index=edge_index, edge_attr=edge_attr, y = label)
    torch.save(flow_graph, f"Graph_dataset/data{idx}.pt")
    return flow_graph
```

1.3: Extracting flow graph

Mỗi file csv tương ứng với 1 flow graph

```
directories = [
    "Data/Adware/*/*.csv",
    "Data/Ransomware/*/*.csv",
    "Data/Scareware/*/*.csv",
    "Data/SMSmalware/*/*.csv",
    "Data/Benign/*/*.csv",
]
```

```
all_graphs = []
graph_labels = []
idx = 1
for directory in directories:
    file_paths = glob.glob(directory)
    for file in file_paths:
        graph_data, graph_label = preprocessing_data(file)
        flow_graph = create_flow_graph_from_csv(graph_data, idx, graph_label)
        idx = idx + 1
```

2. XÂY DỰNG AUTOENCODER MODEL

4.2 Learning Representations of Network Flow Graphs

Each input instance for our model is a directed graph $G = (V, E)$ with adjacency matrix $A \in \{0,1\}^{n \times n}$ and an edge feature matrix $X \in \mathbb{R}^{m \times d}$, where $m := |E|$. The representation learning part of our model computes latent representations of the edges and nodes in the graph and finally outputs a latent feature vector $h^{(1)} \in \mathbb{R}^h$ for each node in the graph. Such a vector intuitively describes how the corresponding endpoint interacts with other endpoints in the network. Depending on the availability of labels, variants of our model compute either predictions or an anomaly score for an input graph from its latent node feature vectors. The model is trained end-to-end such that the latent representations are optimized towards the specific task. Given an input graph with edge attributes, our model performs the following feature transformation and propagation steps to sequentially compute latent representations of the graph's nodes and edges:

$$E^{(0)} = f_1(X) \qquad \in \mathbb{R}^{m \times h} \quad (1)$$

$$H^{(0)} = f_2\left(\left[\hat{\tilde{\mathrm{B}}}_{in} E^{(0)}, \hat{\tilde{\mathrm{B}}}_{out} E^{(0)}\right]\right) \qquad \in \mathbb{R}^{n \times h} \quad (2)$$

$$E^{(1)} = f_3\left(\left[\hat{\tilde{\mathrm{B}}}_{in}^T H^{(0)}, \hat{\tilde{\mathrm{B}}}_{out}^T H^{(0)}, E^{(0)}\right]\right) \qquad \in \mathbb{R}^{m \times h} \quad (3)$$

$$H^{(1)} = f_4\left(\left[\hat{\tilde{\mathrm{B}}}_{in} E^{(1)}, \hat{\tilde{\mathrm{B}}}_{out} E^{(1)}, H^{(0)}\right]\right) \qquad \in \mathbb{R}^{n \times h}, \quad (4)$$

where $[\cdot, \cdot]$ denotes concatenation and $f_1, \ldots, f_4$ are Multi-Layer Perceptrons (MLPs) with appropriate input and output dimensions. As per default, we use single-layer MLPs

$$f_i(X) = q\left(XW_i + b_i\right), \quad (5)$$

where W_i and b_i are the learnable parameters of the model and q is a non-linear activation. We use *ReLU* activations and add batch normalization. The propagation matrices $\hat{\tilde{\mathrm{B}}}_{in}, \hat{\tilde{\mathrm{B}}}_{out} \in \mathbb{R}^{n \times m}$ are obtained from the node-edge incidence matrices $B_{in}, B_{out} \in \{0,1\}^{n \times m}$ with

$$(B_{in})_{ij} = \begin{cases} 1 & \text{if} \quad \exists v_k \in V : e_j = (v_k, v_i) \\ 0 & \text{else} \end{cases} \quad (6)$$

and

$$(B_{out})_{ij} = \begin{cases} 1 & \text{if} \quad \exists v_k \in V : e_j = (v_i, v_k) \\ 0 & \text{else} \end{cases}, \quad (7)$$

indicating in- and out-going edges for each node, by substituting non-zero entries with normalized edge weights. Normalization is applied to preserve the scale of the feature vectors. In particular, we apply symmetric normalization to the adjacency as in [25] before computing the node-edge incidence matrices, where the normalized adjacency matrix is given as $\hat{\tilde{\mathrm{A}}} = \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}$ with $\tilde{A} = A + I$ and degree matrix $\tilde{D}$. Self-loops added for normalization are removed again such that the graph structure remains unchanged. Illustrations of the performed node and edge feature updates are provided in Figure 2a and 2b, respectively.

The first network layer learns how endpoints interact with each other directly by first applying a learnable feature transformation to the original edge feature vectors (Equation 1) and subsequently computing node representations by aggregating feature vectors from neighboring edges (Equation 2). Notably, incoming and outgoing traffic is modeled separately for each node.

The second layer enables the model to learn how endpoints communicate indirectly with their 2-hop neighbors. In a first step, the edge features are updated again by transforming the concatenated feature vectors of the source and destination node and the edge features from the previous layer (Equation 3). Concatenating the edge features from the previous layer as residual connections [19] gives this layer direct access to previously learned features and aids in optimization. Such skip-connections have shown to improve the performance of graph neural networks when applied to node features [41], motivating us to apply them to edge features as well. The edge feature update is followed by an update of the node features using features of incoming and outgoing edges and skip-connected node features from the first layer (Equation 4). These node representations constitute the final output of our representation learning module.

In principle, one could add more layers to the model in a similar fashion to model interaction between more distant endpoints. However, the flow graphs considered in this paper usually have a relatively small diameter, such that additional layers might not result in improved performance but rather lead to over-fitting. In our experiments, we observed the best performance with either one or two layers.

4.4 Network Flow Graph Autoencoder

For unsupervised anomaly detection, autoencoder models often perform well in practice [11]. In general, an autoencoder consists of two neural network modules. While an encoder learns compact and expressive representations of the model inputs, a decoder is supposed to reconstruct the original inputs from their learned representations. If the model is trained with exclusively or mostly normal data, the reconstruction loss can be interpreted as an anomaly score, where instances incurring a larger reconstruction loss are considered more anomalous.

We propose a graph autoencoder model where our representation learning module acts as an encoder. The latent node representations $H^{(1)}$ are then used to reconstruct the original edge feature vectors of the graph using a decoder, which is a mirrored version of the encoder:

$$E^{(2)} = f_5\left(\left[\hat{\tilde{\mathrm{B}}}_{in}^T H^{(1)}, \hat{\tilde{\mathrm{B}}}_{out}^T H^{(1)}\right]\right) \qquad \in \mathbb{R}^{m \times h} \quad (11)$$

$$H^{(2)} = f_6\left(\left[\hat{\tilde{\mathrm{B}}}_{in} E^{(2)}, \hat{\tilde{\mathrm{B}}}_{out} E^{(2)}, H^{(1)}\right]\right) \qquad \in \mathbb{R}^{n \times h} \quad (12)$$

$$E^{(3)} = f_7\left(\left[\hat{\tilde{\mathrm{B}}}_{in}^T H^{(2)}, \hat{\tilde{\mathrm{B}}}_{out}^T H^{(2)}, E^{(2)}\right]\right) \qquad \in \mathbb{R}^{m \times h} \quad (13)$$

$$E^{(4)} = f_8\left(E^{(3)}\right) \qquad \in \mathbb{R}^{m \times h} \quad (14)$$

If the encoder uses only a single layer, the first layer of the decoder (Equation 11–12) is dropped and the node embeddings $H^{(0)}$ are used as input instead. The model parameters are optimized w.r.t. a reconstruction loss

$$\mathcal{L}_{AE}(\mathcal{G}) = \frac{1}{N} \sum_{G_i \in \mathcal{G}} \frac{1}{m_i} \left\| X_i - E_i^{(4)} \right\|_F^2, \quad (15)$$

where $\|\cdot\|_F$ denotes the Frobenius norm. A similar loss was used in [10] to reconstruct node attributes, whereas our model operates on edge-attributed graphs. We denote this variant of our model as *NF-GNN-AE*.

TÓM TẮT: Autoencoder:

2.1. Encode

$$E^{(0)} = f_1(X) \qquad \in \mathbb{R}^{m \times h} \qquad (1)$$

$$H^{(0)} = f_2\left(\left[\hat{\hat{\mathrm{B}}}_{in}\, E^{(0)}, \hat{\hat{\mathrm{B}}}_{out}\, E^{(0)}\right]\right) \qquad \in \mathbb{R}^{n \times h} \qquad (2)$$

$$E^{(1)} = f_3\left(\left[\hat{\hat{\mathrm{B}}}_{in}^{T}\, H^{(0)}, \hat{\hat{\mathrm{B}}}_{out}^{T}\, H^{(0)}, E^{(0)}\right]\right) \qquad \in \mathbb{R}^{m \times h} \qquad (3)$$

$$H^{(1)} = f_4\left(\left[\hat{\hat{\mathrm{B}}}_{in}\, E^{(1)}, \hat{\hat{\mathrm{B}}}_{out}\, E^{(1)}, H^{(0)}\right]\right) \qquad \in \mathbb{R}^{n \times h}, \qquad (4)$$

Từ E là thuộc tính của cạnh -> tổng hợp ra H là thuộc tính của các node

Đề bài có nhắc đến là xây dựng 2 layer, nhưng ở đây là 4 layer để tính các công thức 1,2, 3,4

```
class NF_GNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(NF_GNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim * 2, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim * 3, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim * 3, output_dim)

    def forward(self, edge_attr, Bin, Bout):
        E_0 = F.relu(self.fc1(edge_attr))
        H_0 = F.relu(
            self.fc2(
                torch.cat((torch.matmul(Bin, E_0), torch.matmul(Bout, E_0)), dim=1)
            )
        )
        E_1 = F.relu(
            self.fc3(
                torch.cat(
                    (torch.matmul(Bin.t(), H_0), torch.matmul(Bout.t(), H_0), E_0),
                    dim=1,
                )
            )
        )

        H_1 = F.relu(
            self.fc4(
                torch.cat((torch.matmul(Bin, E_1), torch.matmul(Bout, E_1), H_0), dim=1)
            )
        )
        return H_1
```

2.2. Decode

Đầu ra của encode (H1) là đầu vào của decode:

We propose a graph autoencoder model where our representation learning module acts as an encoder. The latent node representations $H^{(1)}$ are then used to reconstruct the original edge feature vectors of the graph using a decoder, which is a mirrored version of the encoder:

$$E^{(2)} = f_5\left(\left[\hat{\tilde{B}}_{in}^{T} H^{(1)}, \hat{\tilde{B}}_{out}^{T} H^{(1)}\right]\right) \quad \in \mathbb{R}^{m \times h} \quad (11)$$

$$H^{(2)} = f_6\left(\left[\hat{\tilde{B}}_{in} E^{(2)}, \hat{\tilde{B}}_{out} E^{(2)}, H^{(1)}\right]\right) \quad \in \mathbb{R}^{n \times h} \quad (12)$$

$$E^{(3)} = f_7\left(\left[\hat{\tilde{B}}_{in}^{T} H^{(2)}, \hat{\tilde{B}}_{out}^{T} H^{(2)}, E^{(2)}\right]\right) \quad \in \mathbb{R}^{m \times h} \quad (13)$$

$$E^{(4)} = f_8\left(E^{(3)}\right) \quad \in \mathbb{R}^{m \times h} \quad (14)$$

```
class NF_GNN_decode(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(NF_GNN_decode, self).__init__()
        self.fc1 = nn.Linear(input_dim * 2, input_dim)
        self.fc2 = nn.Linear(input_dim * 3, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim * 2 + input_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, output_dim)

    def forward(self, H_1, Bin, Bout):
        E_2 = F.relu(
            self.fc1(
                torch.cat(
                    (torch.matmul(Bin.t(), H_1), torch.matmul(Bout.t(), H_1)), dim=1
                )
            )
        )
        H_2 = F.relu(
            self.fc2(
                torch.cat((torch.matmul(Bin, E_2), torch.matmul(Bout, E_2), H_1), dim=1)
            )
        )
        E_3 = F.relu(
            self.fc3(
                torch.cat(
                    (torch.matmul(Bin.t(), H_2), torch.matmul(Bout.t(), H_2), E_2),
                    dim=1,
                )
            )
        )
        E_4 = F.relu(self.fc4(E_3))
        return E_4
```

2.3. Autoencoder

- Hàm tính ma trận cạnh ra (B_{out}) và ma trận cạnh vào (B_{in}) từ ma trận kề, edge_index (danh sách cạnh)

```
def create_bin_bout(adj_matrix, edge_index, num_nodes):
    m = edge_index.shape[1]  # Number of edges
    bin_matrix = torch.zeros(num_nodes, m)
    bout_matrix = torch.zeros(num_nodes, m)

    for i in range(num_nodes):
        for j in range(num_nodes):
            if adj_matrix[i, j] == 1:
                bin_matrix[j, i] = 1
                bout_matrix[i, j] = 1

    return bin_matrix, bout_matrix
```

- Model

```
class NF_GNN_AE(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(NF_GNN_AE, self).__init__()
        self.encoder = NF_GNN(input_dim, hidden_dim, output_dim)
        self.decoder = NF_GNN_decode(output_dim, hidden_dim, input_dim)

    def forward(self, x, edge_index, edge_attr):
        num_nodes = x.shape[0]  # Added num_nodes definition
        adj_matrix = torch.zeros(num_nodes, num_nodes, dtype=torch.float)
        node_labels = {}
        label_counter = 1  # Start label counter from 1
        for edge in edge_index.T:
            src = edge[0].item()
            dst = edge[1].item()
            # Check if src node has been assigned a label
        if src not in node_labels:
            node_labels[src] = label_counter
            label_counter += 1

        if dst not in node_labels:
            node_labels[dst] = label_counter
            label_counter += 1

        adj_matrix[node_labels[src], node_labels[dst]] = 1

        Bin, Bout = create_bin_bout(adj_matrix, edge_index, num_nodes)
        encoded = self.encoder(edge_attr, Bin, Bout)
        decoded = self.decoder(encoded, Bin, Bout)
        return decoded
```

```
input_dim = 375  ## có 77 thuộc tính
hidden_dim = 128
output_dim = 32

model = NF_GNN_AE(input_dim, hidden_dim, output_dim)
criterion_ae = nn.MSELoss()
optimizer_ae = optim.Adam(model.parameters(), lr=0.001)
```

2.4. Hàm đánh giá

If the encoder uses only a single layer, the first layer of the decoder (Equation 11–12) is dropped and the node embeddings $H^{(0)}$ are used as input instead. The model parameters are optimized w.r.t. a reconstruction loss

$$\mathcal{L}_{AE}(\mathcal{G}) = \frac{1}{N} \sum_{G_i \in \mathcal{G}} \frac{1}{m_i} \left\| X_i - E_i^{(4)} \right\|_F^2, \tag{15}$$

where $||\cdot||_F$ denotes the Frobenius norm. A similar loss was used in [10] to reconstruct node attributes, whereas our model operates on edge-attributed graphs. We denote this variant of our model as *NF-GNN-AE*.

```
def anomalyScores_NF_GNN_AE(original_tensors, reduced_tensors):
    loss = []
    for i in range(len(original_tensors)):
        original_tensor = original_tensors[i]
        reduced_tensor = reduced_tensors[i]

        diff = original_tensor - reduced_tensor

        squared_diff = diff ** 2
        sum_squared_diff_per_sample = torch.sum(squared_diff, dim=1)
        sum_squared_diff_per_sample = torch.sum(sum_squared_diff_per_sample)

        frobenius_norm_per_sample = torch.sqrt(sum_squared_diff_per_sample) /
original_tensor.shape[0]
        loss.append(frobenius_norm_per_sample)

    # Chuẩn hóa scores chạy từ 0 - 1
    min_score = min(loss)
    max_score = max(loss)
    normalized_scores = torch.tensor([(score - min_score) / (max_score - min_score) for
score in loss])

    return normalized_scores
```

5.3 Unsupervised Malware Detection

We further evaluate our approach in a more realistic unsupervised setting, where no labels are available for training.

5.3.1 Experimental Setup. Our experimental setup is the same as in the supervised case with a few adjustments. To ensure a realistic distribution of benign and malicious samples, we perform stratified sampling to first split 20% of the samples for training and then 10% of the remaining samples for validation. The remaining samples are used for testing. All algorithms obtain access to the labeled validation set for hyper-parameter optimization, but training is still unsupervised. We evaluate detection performance using AUROC since it inherently adjusts for class imbalance [9].

Sử dụng AUROC (Area Under the Receiver Operating Characteristic Curve) để đánh giá hiệu quả của việc phát hiện

Ngoài ra trong code còn sử dụng precision_recall_curve để đánh giá

```
def plotResults(trueLabels, anomalyScores, returnPreds=False):
    # Combine true labels and anomaly scores into a DataFrame
    preds = pd.concat([trueLabels, anomalyScores], axis=1)
    preds.columns = ['trueLabel', 'anomalyScore']

    # Precision-Recall curve
    precision, recall, thresholds = precision_recall_curve(preds['trueLabel'],
preds['anomalyScore'])
    average_precision = average_precision_score(preds['trueLabel'], preds['anomalyScore'])
    plt.step(recall, precision, color='k', alpha=0.7, where='post')
    plt.fill_between(recall, precision, step='post', alpha=0.3, color='k')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])
    plt.title('Precision-Recall curve: Average Precision =
{0:0.2f}'.format(average_precision))
    plt.show()  # Show the plot

    # ROC curve
    fpr, tpr, thresholds = roc_curve(preds['trueLabel'], preds['anomalyScore'])
    areaUnderROC = auc(fpr, tpr)
    plt.plot(fpr, tpr, color='r', lw=2, label='ROC curve')
    plt.plot([0, 1], [0, 1], color='k', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic: Area under the curve =
{0:0.2f}'.format(areaUnderROC))
    plt.legend(loc="lower right")
    plt.show()

    if returnPreds==True:
        return preds
```

3. CHIA DỮ LIỆU THÀNH TẬP TRAIN VÀ TEST

```
4. directory = "Graph_dataset/"
5. file_list = [file for file in os.listdir(directory) if file.endswith(".pt")]
6. graph_data = []
7. for file in file_list:
8.     path = os.path.join(directory, file)
9.     graph = torch.load(path)
10.    graph_data.append(graph)
```

- Chia tập train 20%, test 80%

```
from sklearn.utils import shuffle
X = shuffle(graph_data)

X_train, X_test = train_test_split(X, test_size=0.8, random_state=42, shuffle=True)


class GraphDataset(Dataset):
    def __init__(self, data_list):
        self.data_list = data_list

    def __len__(self):
        return len(self.data_list)

    def __getitem__(self, idx):
        return self.data_list[idx]


train_dataset = GraphDataset(X_train)
test_dataset = GraphDataset(X_test)
```

```
def custom_collate(batch):
    if isinstance(batch[0], Data):
        return batch
    else:
        return torch.utils.data._utils.collate.default_collate(batch)

batch_size = 32
train_loader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True, collate_fn=custom_collate
)
test_loader = DataLoader(test_dataset, batch_size=batch_size, collate_fn=custom_collate)
```

4. DETECTING

```
# Extract features using the encoder
model.eval()
reconstructed_data = []
original_data = []
encoded_graphs = []
anomaly_scores = []
with torch.no_grad():
    for batch in train_loader:
        for data in batch:
            edge_attr, edge_index, x = (
                data.edge_attr,
                data.edge_index,
                data.x,
            )
            num_nodes = x.shape[0]
            original_data.append(edge_attr)
            adj_matrix = torch.zeros(num_nodes, num_nodes, dtype=torch.float)
            node_labels = {}
            label_counter = 1
            for edge in edge_index.T:
                src = edge[0].item()
                dst = edge[1].item()

            if src not in node_labels:
                node_labels[src] = label_counter
                label_counter += 1
            if dst not in node_labels:
                node_labels[dst] = label_counter
                label_counter += 1

            adj_matrix[node_labels[src], node_labels[dst]] = 1

            Bin, Bout = create_bin_bout(adj_matrix, edge_index, num_nodes)

            #encoded_graph for cluster task
            encoded_graph = model.encoder(edge_attr, Bin, Bout)
            encoded_graphs.append(encoded_graph)

            #reconstructed graph
            reconstructed = model(x, edge_index, edge_attr)

            reconstructed_data.append(reconstructed)
```

Tính anomaly score

```
anomaly_scores = anomalyScores_NF_GNN_AE(original_data, reconstructed_data)
```

```
## Chuyển sang dạng data frame
y = []
for label in y_train:
   if label == 0:
       y.append(0)
   else:
       y.append(1)
y  = pd.DataFrame(y)
anomaly_scores = pd.DataFrame(anomaly_scores)
```

Hiển thị đánh giá:

```
preds = plotResults(y, anomaly_scores, True)
```

5. CLUTERING

5.1: Model tổng hợp các vector node attributes của 1 đồ thị thành 1 vector 1 chiều

```
def Embedded_Flow_Graph(input_shape, vector_dimension):
    model = models.Sequential()
    model.add(layers.Input(shape=input_shape))
    model.add(layers.Flatten())  # Flatten the input into a single vector
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dense(vector_dimension, activation='linear'))
    return model
```

6.2. Embedding flow graphs

```
vector_dimension = 10  # Dimensionality of the output vector
graph_embeddings = []
for encoded_graph in encoded_graphs:
    input_shape = (encoded_graph.shape[0] *encoded_graph.shape[1],)  # Flatten input shape
    encoded_graph_np = encoded_graph.numpy()
    model = Embedded_Flow_Graph(input_shape, vector_dimension)
    embedding = model(encoded_graph_np.reshape(1, -1))  # Reshape node_features into a
single vector
    graph_embeddings.append(embedding.numpy())
```

```
# Convert graph_embeddings to TensorFlow tensor
graph_embeddings_tensor = tf.convert_to_tensor(graph_embeddings)
graph_embeddings_tensor = tf.squeeze(graph_embeddings_tensor, axis=1)
```

6.3: Cluster by kmeans algorithm

```
# Perform KMeans clustering
n_clusters = 5
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(graph_embeddings_tensor)

# Visualize the clustering result
plt.figure(figsize=(12, 10))  # Adjust the size here as needed
plt.scatter(graph_embeddings_tensor[:, 0], graph_embeddings_tensor[:, 1], c=cluster_labels,
cmap='viridis')
plt.title('Graph Clustering Result')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster')
plt.show()
```

- Xem nhãn được gán

```
cluster_labels
unique_labels, label_counts = np.unique(cluster_labels, return_counts=True)

# Print the counts
for label, count in zip(unique_labels, label_counts):
    print(f"Cluster {label}: {count} samples")
```