6-12-2023

Graph Representation Learning for Context-Aware Network Intrusion Detection

Augustine Premkumar

Madeline Schneider

Carlton Spivey

John Pavlik

Nathaniel D. Bastian
Army Cyber Institute, U.S. Military Academy, nathaniel.bastian@westpoint.edu

PROCEEDINGS OF SPIE

SPIEDigitalLibrary.org/conference-proceedings-of-spie

Graph representation learning for context-aware network intrusion detection

Augustine Premkumar, Madeleine Schneider, Carlton Spivey, John Pavlik, Nathaniel Bastian

Augustine Premkumar, Madeleine Schneider, Carlton Spivey, John Pavlik, Nathaniel D. Bastian, "Graph representation learning for context-aware network intrusion detection," Proc. SPIE 12538, Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications V, 125380H (12 June 2023); doi: 10.1117/12.2663162

SPIE.

Event: SPIE Defense + Commercial Sensing, 2023, Orlando, Florida, United States

Graph Representation Learning for Context-Aware Network Intrusion Detection

Augustine Premkumar[a,b], Madeleine Schneider[b], Carlton Spivey[a,b], John A. Pavlik[b], and Nathaniel D. Bastian[b,a]

[a]Mathematical Sciences Department, United States Military Academy, West Point, NY, USA
[b]Army Cyber Institute, United States Military Academy, West Point, New York, USA

ABSTRACT

Detecting malicious activity using a network intrusion detection system (NIDS) is an ongoing battle for the cyber defender. Increasingly, cyber-attacks are sophisticated and occur rapidly, necessitating the use of machine/deep learning (ML/DL) techniques for network intrusion detection. Traditional ML/DL techniques for NIDS classifiers, however, are often unable to sufficiently find context-driven similarities between the various network flows and/or packet captures. In this work, we leverage graph representation learning (GRL) techniques to successfully detect adversarial intrusions by exploiting the graph structure of NIDS data to derive context awareness, as graphs are a universal language for describing entities and their relationships. We explore several methods for NIDS data graph representation at both the network flow and packet level utilizing the CIC-IDS2017 dataset. We leverage graph neural networks and graph embedding algorithms to create a context-aware network intrusion detection system. Results indicate that adding context derived from GRL improves performance for detecting attacks. Our highest-scoring classifier incorporated both GNN embeddings and flow-level features and achieved an accuracy of 99.9%. Adding GRL methods to augment the flow/packet features improved accuracy by as much as 52.41%.

Keywords: Graph Representation Learning, Network Intrusion Detection, Deep Learning, Context-awareness

1. INTRODUCTION

In the domain of cyberspace operations, protecting computer networks from malicious intrusions is a priority for cyber defenders. Traditional network defense systems use signature-based software to detect intrusions that share similarities with previous intrusions. Increasingly, cyber attacks are sophisticated and occur rapidly, necessitating the use of machine/deep learning techniques for network intrusion detection. Using traditional machine/deep learning techniques to train a classifier as part of a network intrusion detection system (NIDS) would require features from the network flow (e.g., number of packets, duration of the flow)[1] or the raw packet capture (e.g., protocol, payload)[2] for model training. Use of these traditional network features, however, often lacks context. We hypothesize that representing the network traffic graphically may provide a means to learn symbolic context to then augment traditional network features in order to train an intelligent, context-aware NIDS to detect malicious activity with greater reliability and performance.

A generic overview of our proposed modeling architecture is displayed in Figure 1. As input, we use CIC-IDS2017, which is a preexisting NIDS dataset that provides raw packet captures (PCAP) and network flows. We perform PCAP labeling and additional data pre-processing and then conduct graph data representation to prepare the NIDS data for graph representation learning to create a low-dimensional vector of each graph. These vectors are then used as features to augment PCAP and/or network flow features as inputs to train and evaluate several classification models (logistic regression, decision tree, extreme gradient boosting) for network intrusion detection. As a result, we make the following contributions in this paper.

- Present two different graph data representation methods for NIDS data for both network flow and raw packet capture level intrusion detection.

Send correspondence to: nathaniel.bastian@westpoint.edu.

- Conduct graph representation learning using Graph Neural Networks and Graph Embeddings algorithms to create low-dimensional vectors of each graph and then build multiple deep graph classification models.
- Perform computational experimentation to evaluate developed models in terms of effectiveness for context-aware network intrusion detection.

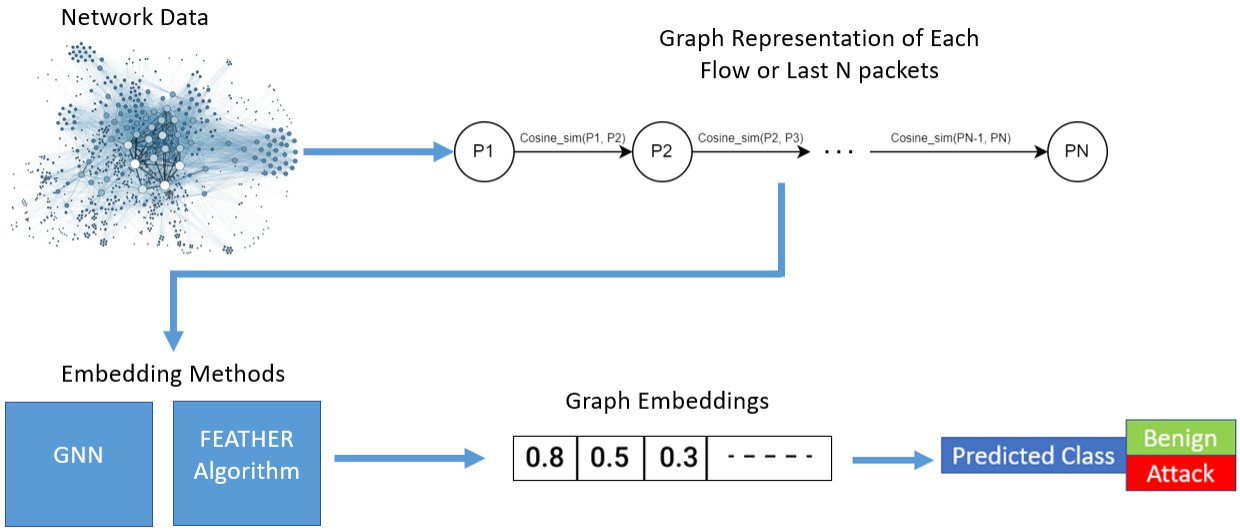

Figure 1. Graph Representation Learning Architecture for Context-Aware Network Intrusion Detection

The rest of this paper is organized as follows. In Section 2, we explore and discuss related works, while in Section 3 we discuss our methodology including data selection and preparation, graph data representation methods, graph representation learning techniques, and classification modeling algorithms. In Section 4, we provide and discuss the results of our experimentation, followed by conclusions in Section 5.

2. RELATED WORKS

The use of machine/deep learning techniques for network intrusion detection is a heavily researched area, whether that be for labeled NIDS data (detecting known attack vectors)[3] or for unlabeled NIDS data (detecting anomalies or outliers).[4,5] Representing network traffic as a graph can be a useful way to represent network data for the purpose of learning context, given the relationship between nodes and edges in the graph.[6] Network traffic can also be represented as a time series with features related to particular aspects of the network flow, such as duration, size of payload shared, number of packets, and more.[7] To convert this data into a graph, internet protocol (IP) addresses/port numbers are used as nodes with edges between them representing a transfer of data between two devices on the same network. Nodes are represented as a list of attributes that are relevant to identifying a particular device or to take an attribute list, using some aggregator function to represent its features as an embedding. Edges can also be represented as a collection of multiple attributes of a transmission between two devices on a network or as an aggregate of those features. An important consideration when converting from time series to graph structure is determining a fixed time interval to use to sample the network flow to avoid creating a graph that is too large to input/train by a trained model.[8]

Xiao et. al.[9] represent network traffic data using first-order and second-order graphs. The first order graph learns latent features from a host's perspective, and the second-order graph learns latent features from a global perspective. By combining the two representations, they were able to improve detection accuracy and detect unknown attacks. The first-order graph represent the distance between two nodes as the similarity between the respective port distributions. The second-order graph essentially encodes the context of nodes using a

hypergraph. In experimentation, network intrusion detection using both graph representations outperformed scenarios where only one representation was used. When the detection models were tested against unknown attacks, only variants of known attacks were detected and brand-new attacks went undetected.[9] Lin et al.[6] use hypergraphs to capture evolving patterns of port scan attacks via the set of IP addresses and destination ports. They derive a set of hypergraph-based metrics that are then used to augment traditional network flow features to train a machine learning based ensemble NIDS that effectively monitors and detects port scanning activities and adversarial intrusions while evolving intelligently in real-time. Lopes-Martin et. al.[10] represent source and destination IP and port addresses as low dimensional vectors, or embeddings. The probability that a source and destination address share the same connection is interpreted as the distance between two embeddings. Seven network elements are represented, each as a separate embedding: IP address, port number, concatenated IP and port addresses, and the four IP components. These distances are used to calculate an aggregate value that represents the probability that the source and destination IP and port belong to the same connection. This probability is an additional feature, with a final list of eight features. We aim to incorporate their distance representations with other context representations for the purpose of context-aware network intrusion detection.

In addition to hypergraphs and embeddings, graph neural networks show promise in network intrusion detection due to their ability to leverage the structure of a graph,[8] which can be leveraged to classify individual nodes, edges or make inferences on the whole graph. Lo et. al.[11] demonstrate the capabilities of network intrusion detection using graph neural networks,[11] notably using flow-based data for an edge classification approach; this can accurately label a malicious flow but ignores the relationships between flow records. Outside of the NIDS setting, Roy et. al.[12] developed a graph convolutional neural network approach (with one model for node representation and the other for label dependence) that incorporates predictive context, where they measure context from co-occurrence, spatial distance, geometric and appearance similarity, and classified object classes with relative sizes and location with other objects. Ancharya et. al.[13] use a similar graph-based approach to detect out-of-context objects in images using graph contextual reasoning networks. The first graph, a representation graph, learns object features based on neighboring objects, whereas the second graph, a context graph, captures contextual cues from neighboring objects. The paper recognizes three contextual relations: co-occurrence, location, and shape similarity, like Roy et. al.[12] These two works demonstrate the effectiveness in using graph representation learning techniques for deriving symbolic context to improve classification capabilities and accuracy, proving useful for our context-aware network intrusion detection approach.

3. METHODOLOGY

In this section, we describe the NIDS data used, two graph data representation methods, two graph representation learning approaches, classification modeling techniques, and computational experimentation setup.

3.1 Data

We leverage CIC-IDS2017,[14] which is a well-known network traffic dataset that is often used for cybersecurity research. This dataset provides benign and malicious labeled network flows spanning a range of attack types and network interactions, as well as the raw PCAP associated with these network flows. It includes network flow features and raw payload data.

Unfortunately, the ample data available by having both flow and packet-level information has historically been difficult to use. This is because CIC-IDS2017 is known to suffer from a lack of labeling standardization. Namely, the full network flow information is labeled, but the individual packets are not. Due to ambiguity, including missing values and labeling inconsistencies, it can be difficult to accurately match labeled flows to unlabeled packets. To handle this, we used Payload-Byte to extract and label the PCAP files of the two NIDS datasets mentioned above.[15] This tool takes raw PCAP data and parses the data into bytes and packet header features. Bytes that do not reach the default size of 1500 are padded with zeros. The bytes are transformed from their hex value to an integer in the range 0-255. The other features included are time-to-live, protocol, and time between packets.[15] As such, we are able to consider both packet and flow information in this work.

CIC-IDS2017 The CIC-IDS2017 dataset was created by the Canadian Institute for Cybersecurity (CIC) over five days from 03 July to 07 July 2017. The CIC generated this data to supply a modern, updated dataset that addressed a historic lack of traffic diversity, volume, packet anonymization, attack variety, and feature span. Their experiment implemented two networks, a victim and attack network, between which attacks were conducted and recorded. The dataset includes seven common types of attacks: brute force, heart-bleed, botnet, denial-of-service, distributed denial-of-service, web, and infiltration, out of which over 80 features were extracted. The resulting dataset covers the following evaluation criteria: complete network configuration, complete traffic, labeled dataset, complete interaction, complete capture, available protocols, attack diversity, heterogeneity, feature set, and meta data.[16] Table 3.1 summarizes the frequency of each attack type with the data set.

DoS	63.28%
DDoS	32.19%
Web	0.49%
Brute Force	3.48%
Bot	0.49%
Heartbleed	0.003%
Infiltration	0.009%

Table 1. Proportion of each attack type in the CIC-IDS2017 dataset

3.2 Graph Data Representation

To describe how we represented the NIDS data set as graphs, we first provide the definition of graphs.

Definition 3.1 (Graphs). *A graph can be described as $G = (V, E)$, where V is the set of nodes and E is set of edges. Let $v_i \in V$ denote a node and $e_{ij} = (v_i, v_j) \in E$ denote an edge between node v_i and node v_j. We let $\boldsymbol{A} \in \{0,1\}^{n \times n}$ represent the adjacency matrix of graph G, where $n = |V|$ is the total number of nodes. $\boldsymbol{A}_{ij} = 1$ implies that there exists an edge between node v_i and node v_j, otherwise $\boldsymbol{A}_{ij} = 0$. The neighborhood of node v_i is defined as $N(v_i) = \{v_j \in V | (v_i, v_j) \in E\}$. For graph data with node features, we use $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ to denote the node feature matrix where d is the number of node features.*

We represented the NIDS dataset as graphs using two different approaches, further detailed below.

3.2.1 Method 1

Method 1 represents full network flow information as a graph. To do this, pre-processed Payload-Byte data was grouped by the source IP address, destination IP address, source port, destination port, and protocol. Then, full flow information was evaluated for generating a flow graph. In this representation, each packet in the entire network flow represents a node, and each node is connected by an edge to the next packet in the flow. This generates a simple directed acyclic graph (a tree graph) as depicted in Figure 2. Both the nodes and edges of our graphs contain attributes. Each node has 1500 attributes, one for each byte. The attribute is simply the byte value. Edges are attributed with the cosine similarity (see Equation 1) between the bytes of sequential packets. Each graph is then labeled as either malicious or benign. If there was a single malicious packet in a flow, the entire graph was labeled malicious.

$$CS = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|} \tag{1}$$

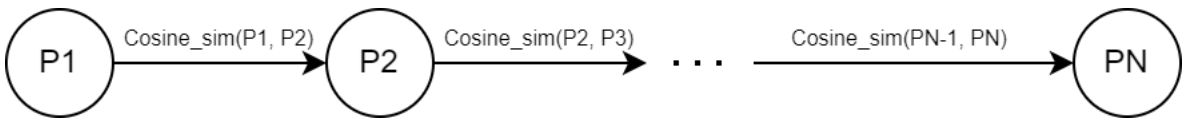

Figure 2. A graph of N payloads is made of N nodes. The node attributes are the byte representation of each payload. The edge attributes are the cosine similarities between the payloads.

3.2.2 Method 2

Method 2 attempts to enable network intrusion detection in real-time. Instead of grouping packets by network flow, this method groups n sequential packets/nodes in a linear fashion. Each packet represented a node in a graph, and edge values between nodes were given the value of the cosine similarly between the two subsequent packets. Node attribute are created in the same manner as Method 1. Finally, the graph label is the attack type label for the packet in the graph with the latest arrival based on timestamp.

3.3 Graph Representation Learning

Graph representation learning, which is essentially machine learning on graph-structured data, has multiple prediction tasks, such as node classification, edge prediction, graph classification, and community detection. Node classification aims to predict properties of nodes within the graph, whereas edge prediction predicts properties of the edge between two nodes. Graph classification entails learning over the whole graph, which takes the graph as input and performs classification based on it. Finally, community detection aims to identify dense clusters of nodes within the graph.[17] We now provide a more formal definition of a graph machine learning model.

Definition 3.2 (Graph Machine Learning Model). *Given a graph $G = (V, E)$, a graph machine learning model f_ω parameterized by ω learns the node representations $\boldsymbol{H} \in \mathbb{R}^{n \times d_e}$ with respect to G for downstream tasks, where d_e is the dimension of node embeddings*

$$\boldsymbol{H} = f_\omega(G). \tag{2}$$

For node classification tasks, we can employ a softmax function to obtain the probability vector for each node based on its embedding, and then a loss function (such as cross-entropy loss) is applied to measure the difference between predictions and the given node labels. For graph classification tasks, which is the case in our research, a graph-level representation $\boldsymbol{h}_G$ can be pooled from node representations

$$\boldsymbol{h}_G = \text{readout}(\boldsymbol{H}), \tag{3}$$

where readout$(\cdot)$ is a pooling function (such as mean pooling and sum pooling) that aggregates the embeddings of all nodes in the graph into a single embedding vector, which is also known as the graph embedding. This can be used as a feature to build machine learning models using logistic regression, decision trees, XGBoost, etc.

When it comes to Graph Neural Networks (GNNs) as the graph machine learning, each node v_i typically gathers the information from its neighbors $N(v_i)$ and aggregates them with its own information to update its representation $\boldsymbol{h}_i$. It should be noted that an L-layer GNN f_ω can be formulated as

$$\boldsymbol{h}_i^l = \sigma(\omega^l \cdot (\boldsymbol{h}_i^{l-1}, \text{Agg}(\{\boldsymbol{h}_j^{l-1} | v_j \in N(v_i)\}))) \tag{4}$$

for $l = 1, 2, \ldots, L$, where $\boldsymbol{h}_i^l$ is the representation of node v_i after the l-layer of f_ω and $\boldsymbol{h}_i^0 = \boldsymbol{X}_i$ is the raw feature of node v_i. Note that ω^l is the learnable parameters in the l-layer of f_ω, Agg$(\cdot)$ is the aggregation operation (e.g., mean pooling), and σ is the activation function. More detail on GNNs and graph embeddings are below.

3.3.1 Graph Neural Networks

Graph neural networks (GNNs) are a subclass of neural networks that are capable of working with data represented as a graph. Unlike many other types of neural networks, GNNs can understand the unordered, complex relationships of nodes and edges through neighborhood aggregation.[18] Neighborhood aggregation is an iterative process in which each node in the graph takes on some of the information stored in neighboring nodes and edges through aggregation, pooling, and message passing.[17]

As input, a GNN usually takes a graph represented as three multi dimensional arrays: node features, edge features, and an adjacency matrix. If the graph is unattributed, some of these arrays may not be necessary. As output, the GNN gives probability of classification to a specific group. There are three types of classification used in GNNs: node classification, edge classification, and graph classification.[17]

Our research considered binary, graph-level classification, where we attempt to classify a flow or sequential packet graph as malicious or benign. The architecture of our graph neural network consists of two edge-conditioned convolutional layers (ECCConv[19]), one batch normalization layer, one global sum pooling layer, and a dense layer as depicted in Figure 3. The first ECCConv layer has 32 channels and a ReLU activation function. The second ECCConv layer has 500 channels also with a ReLU activation function. Following the ECCConv layers we added a batch normalization layer to get a normalized output when using the GNN to create a graph embedding. ECCConv layers were selected due to their unique ability to include edge attributes, as well as node attributes and graph structure in graph convolutions.[19]

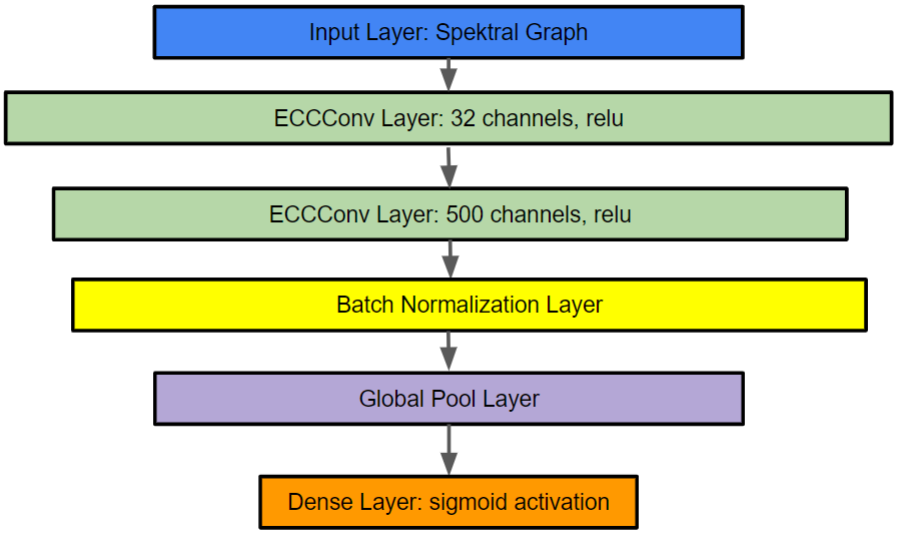

Figure 3. The architecture of the employed graph neural network.

3.3.2 Graph Embeddings

Graph embeddings look to capture the relevant information stored in the nodes and edges of a graph into a lower dimensional vector space. By performing classification techniques on the embeddings, instead of the graph itself, we can reduce the computational complexity of classifying graphs. Embeddings are superior to traditional adjacency matrix representations of graphs because the size of the embedding is not dependent on the size of the graph it represents. This means each graph can be represented as a fixed-length non-sparse vector, allowing us to utilize traditional machine learning classification methods. In our work, we consider two methods of embedding, FEATHER,[20] and sub-modeling a trained GNN model to extract the output of the global pooling layer.

FEATHER This graph embedding algorithm uses characteristic functions of node features with random walk weights to describe node neighborhoods. The node features are combined using mean pooling to create graph-level embeddings.[20] Pseudo code of the algorithm is shown below in Algorithm 1. For our purposes, we utilized the implementation of the FEATHER algorithm in the KarateClub[21] library.

GNN Global Pooling Layer Utilizing the trained GNN, we are able to create a graph embedding by taking the output of the model's global pool layer for each graph (equation 3). Due to our model architecture, this output is a 500-length vector, making it comparable to the 500-length vector received from using the FEATHER algorithm. Since our model included a batch normalization layer prior to the global pooling layer, the outputted graph embedding is normalized. We do this using Keras submodeling.

Algorithm 1 FEATHER Algorithm

Data: Å– Normalized adjacency matrix.
$\chi = x^1, ..., x^k$ – Set of node feature vectors.
$e^\theta = \theta^1, 1, ..., \theta^{1,r}, \theta^{2,1}, ..., \theta^{k,r}$ – Set of evaluation point vectors.
r – Scale of empirical graph characteristic function.
Result: Node embedding matrix Z.
$Z_{Re} \leftarrow$ Initialize Real Features()
$Z_{Im} \leftarrow$ Initialize Imaginary Features()
for i **do** in 1 : k do
  for j **do** in 1 : r do
    for l **do** in 1 : j do
      if l **then** = 1 then
        $H \leftarrow x^i \otimes \theta^{i,j}$
        $H_{Re} \leftarrow cos(H)$
        $H_{Im} \leftarrow sin(H)$
      end if
      $H_{Re} \leftarrow \mathring{A} H_{Re}$
      $H_{Im} \leftarrow \mathring{A} H_{Im}$
    end for
    $Z_{Re} \leftarrow [Z_{Re}|H_{Re}]$
    $Z_{Im} \leftarrow [Z_{Im}|H_{Im}]$
  end for
end for
$Z \leftarrow [Z_{Im}|Z_{Re}]$
Output Z.

3.4 Machine Learning Classifiers

Logistic Regression Since we are dealing with a binary classification problem between benign and malicious packets, logistic regression presents itself as a computationally efficient and easy-to-implement method for classification. Thus, we used a logistic regression model, with each feature of our embedding representing a variable. Once the model was fit to our training data set, we used a threshold of 0.5 to differentiate between malicious and benign packets. This was implemented with the sklearn library with the LogisticRegression module.[22]

Decision Trees Decision trees consist of decision nodes and leaf nodes. Data is split at decision nodes and homogeneous at leaf nodes. At each decision node, the data is split with the goal of reducing the randomness of the resulting groups. Once all resulting groups are homogeneous, the model is complete and ready for test data. In our implementation, we used entropy as our splitting criteria, a max depth of 3, and a minimum of 5 sample leaves. This was implemented with the sklearn library with the DecisionTreeClassifier module.[22]

XGBoost Extreme gradient boosting is an ensemble model which uses multiple shallow decision trees and aggregates them to make a prediction. It is sequential in nature and incorporates the error of the previous tree when creating the subsequent one. This is done iteratively until the error ceases to decrease. This method is extremely powerful and computationally efficient. For our implementation, we used the XGBoost library[23] along with its default hyperparameters.

3.5 Computational Experimentation

To conduct our computational experiments, we created a pipeline in which the data representation method could be toggled. Once selected, our pipeline consisted of five main phases: data pre-processing, graph construction, GNN/embedding model learning, classifier learning, and classifier testing.

In phase 1, packets were grouped based on the graph data representation method. With method 1, packets were grouped by flow, identified as packets sharing a source IP address, destination IP address, source port, destination port, and protocol. With method 2, packets were grouped sequentially, for our implementation every 4 packets were grouped together. Next, to create a balanced dataset, benign groups were downsampled by random selection. The data was then broken into a training set (80%), validation set (10%), and testing set (10%).

In phase 2, graphs were constructed in two ways. First, graphs were constructed for GNN implementation using the Spektral[24] Python package. Here, each packet's payload was represented as a node feature, its edge weights represented the cosine similarity between it and the next sequential packet, and a binary adjacency matrix was used to indicate an edge. Next, graphs were constructed for FEATHER embedding using the NetworkX[25] library. These graphs were generated using the cosine similarity between sequential packets, and this information was stored in a weighted adjacency matrix. This matrix was then used to create a NetworkX graph object.

In phase 3, the training data was used to train the GNN and the FEATHER embedding model. The validation data was used for early stopping on the GNN, where worsening validation loss (min delta = .01) for five epochs would signal for the training to end. The best weights from the previous epoch were then restored. We used an Adam optimizer with a 1e-3 learning rate and a categorical cross-entropy loss function. FEATHER was trained using default parameters as defined in the KarateClub library.[21]

In phase 4, classifiers were trained using the GNN global pooling layer embedding and the FEATHER embedding. Additionally, the classifiers were trained using embeddings concatenated with the additional flow or packet-level features as well as just the additional features themselves. Packet-level features included: (1) time to live, (2) the one-hot-encoded protocol, and (3) the payload of the most recent packet. Flow-level features included: (1) the duration of the flow in milliseconds, (2) the total number of packets in the forward direction, (3) the total number of packets in the backward direction, (4) the size of the packets in the forward direction, (5) the size of the packets in the backward direction, (6) number of flow bytes per second, (7) number of flow packets per second, and (8) download and upload ratio. Each of these features was normalized between 0 and 1 during pre-processing using min-max scaling.

In phase 5, we tested the classification models. The different classifiers were evaluated using test FEATHER embeddings, test GNN embeddings, test FEATHER embeddings with additional flow/packet features, test GNN embeddings with additional flow/packet features, and just the flow/packet features. In this phase, we consider accuracy, precision, recall, and F1 score to measure the effectiveness of each model and embedding approach.

4. RESULTS AND DISCUSSION

Our results shown in Tables 2 and 3 indicate the best-performing classifier is XGBoost, which consistently showed higher accuracy, recall, and precision than its counterparts. Comparing graph data representations, Method 1 on average showed better performance than Method 2 and had a higher largest accuracy of 0.9995 compared to 0.9892 for Method 2. GNN embeddings performed better than FEATHER embeddings for each of the classification models tested. Concatenating additional flow/packet features showed varying levels of improvement based on the methodology being tested. Method 1 showed a negligible increase in model performance with the incorporation of flow-level features. Method 2, on the other hand, occasionally did show an increase in accuracy, recall, and precision when adding additional packet-level features. Flow-level features and packet-level features by themselves, had comparable results to the embedding approaches when using the XGBoost classifier; however, they consistently showed worse results compared to when they were concatenated with the embedding approaches. Due to the balanced dataset used, precision and recall were similar for both benign and malicious packets.

Adding context to flow-level features or packet-level features through the use of graph representation learning is effective in improving the classification of malicious and benign packets as evidenced by our results in Tables 2 and 3. Note that in what follows we make comparisons between methodologies using the XGBoost classifier, given that it consistently performed the best of the classifiers used. For both methodologies used to represent the NIDS data, adding embeddings learned from the graph structure of the data improved the accuracy of classification. In Method 1, adding FEATHER embeddings improved the accuracy from 0.9781 to 0.9989, whereas adding GNN embeddings improved it to 0.9995. For Method 2, adding FEATHER embeddings improved accuracy from 0.9585

Method 1	Accuracy	Precision Benign	Recall Benign	F1 Score Benign	Precision Malicious	Precision Benign	F1 Score Malicious
(Flow features) Logistic Regression	0.4746	0.4681	0.3941	0.4279	0.4793	0.5546	0.5142
(Flow features) Decision Tree	0.8564	0.8600	0.8505	0.8552	0.8529	0.8623	0.8576
(Flow features) XGBoost	0.9781	0.9789	0.9772	0.9780	0.9773	0.9791	0.9782
(FEATHER embeddings) Logistic Regression	0.8557	0.9070	0.7918	0.8455	0.8162	0.9193	0.8647
(FEATHER embeddings) Decision Tree	0.8885	0.9098	0.8619	0.8852	0.8695	0.9150	0.8917
(FEATHER embeddings) XGBoost	0.9697	0.9705	0.9686	0.9696	0.9688	0.9707	0.9698
(GNN embeddings) Logistic Regression	0.9987	0.9986	0.9988	0.9987	0.9988	0.9986	0.9987
(GNN embeddings) Decision Tree	0.9940	0.9913	0.9966	0.9940	0.9966	0.9913	0.9940
(GNN embeddings) XGBoost	**0.9995**	**0.9995**	**0.9995**	**0.9995**	**0.9995**	**0.9995**	**0.9995**
(Flow features, FEATHER embeddings) Logistic Regression	0.8657	0.9164	0.8039	0.8565	0.8262	0.9271	0.8737
(Flow features, FEATHER embeddings) Decision Tree	0.9230	0.9371	0.9063	0.9215	0.9098	0.9395	0.9244
(Flow features, FEATHER embeddings) XGBoost	0.9989	0.9986	0.9993	0.9989	0.9993	0.9986	0.9989
(Flow features, GNN embeddings) Logistic Regression	0.9987	0.9988	0.9986	0.9987	0.9986	0.9988	0.9987
(Flow features, GNN embeddings) Decision Tree	0.9940	0.9913	0.9966	0.9940	0.9966	0.9913	0.9940
(Flow features, GNN embeddings) XGBoost	**0.9995**	**0.9995**	**0.9995**	**0.9995**	**0.9995**	**0.9995**	**0.9995**

Table 2. Classification model performance on GRL embeddings and flow-level features as defined in Method 1

Method 2	Accuracy	Precision Benign	Recall Benign	F1 Score Benign	Precision Malicious	Precision Benign	F1 Score Malicious
(Payload, TTL, Protocol) Logistic Regression	0.7699	0.9053	0.6027	0.7237	0.7024	0.9370	0.8029
(Payload, TTL, Protocol) Decision Tree	0.9573	0.9858	0.9280	0.9560	0.9320	0.9866	0.9586
(Payload, TTL, Protocol) XGBoost	0.9585	0.9858	0.9304	0.9573	0.9342	0.9866	0.9597
(FEATHER embeddings) Logistic Regression	0.6688	0.6902	0.6119	0.6487	0.6517	0.7256	0.6867
(FEATHER embeddings) Decision Tree	0.8170	0.7716	0.9002	0.8310	0.8803	0.7338	0.8004
(FEATHER embeddings) XGBoost	0.9091	0.8789	0.9488	0.9125	0.9445	0.8693	0.9053
(GNN embeddings) Logistic Regression	0.9886	0.9867	0.9906	0.9886	0.9905	0.9867	0.9886
(GNN embeddings) Decision Tree	0.9831	0.9833	0.9828	0.9831	0.9829	0.9833	0.9831
(GNN embeddings) XGBoost	**0.9892**	**0.9874**	**0.9910**	**0.9892**	**0.9909**	**0.9874**	**0.9892**
(Payload, TTL, Protocol, FEATHER embeddings) Logistic Regression	0.7364	0.7458	0.7170	0.7311	0.7278	0.7558	0.7415
(Payload, TTL, Protocol, FEATHER embeddings) Decision Tree	0.9494	0.9769	0.9205	0.9479	0.9249	0.9783	0.9508
(Payload, TTL, Protocol, FEATHER embeddings) XGBoost	0.9696	0.9703	0.9688	0.9696	0.9689	0.9703	0.9696
(Payload, TTL, Protocol, GNN embeddings) Logistic Regression	0.9887	0.9867	0.9906	0.9887	0.9906	0.9867	0.9886
(Payload, TTL, Protocol, GNN embeddings) Decision Tree	0.9831	0.9833	0.9828	0.9831	0.9829	0.9833	0.9831
(Payload, TTL, Protocol, GNN embeddings) XGBoost	**0.9892**	**0.9875**	**0.9910**	**0.9892**	**0.9909**	**0.9875**	**0.9892**

Table 3. Classification model performance on GRL embeddings and packet-level features as defined in Method 2

to 0.9696, whereas adding GNN embeddings improved it to 0.9892. This increase in accuracy from incorporating embeddings retrieved from GRL was most pronounced in Method 1 when using logistic regression where we saw accuracy increase by 52.41%. This leads us to believe that our model is learning from the context added by representing the NIDS data as a graph, as hypothesized. In fact, our GNN embeddings perform so well a case can be made for not including flow/packet-level features at all in the model. Performance for the GNN embeddings with and without flow/packet level features is identical. FEATHER embeddings, however, did show a decline in accuracy when removing flow/packet-level features. Overall, trained GNN embeddings seem to outperform embeddings derived from the FEATHER algorithm. We hypothesize this difference can be attributed to the GNN's ability to learn from each packet's payload as well as its relationship with other packets in the graph. Since the FEATHER embeddings were created only using the edge weights between packets, it doesn't have the ability to learn from the payload itself. Despite this, FEATHER embeddings still shows comparable results, possibly meaning it could generalize better to novel data.

5. CONCLUSIONS

In conclusion, our experimental results demonstrated the effectiveness of augmenting traditional flow and packet-level features with embeddings derived from graph representation learning methods. Both embeddings we experimented with showed promise in adding relevant context awareness to a network intrusion detection system, making it more accurate and more reliable for detecting attacks. We also demonstrated that graph representation learning can add context at both the packet and flow levels.

Limitations of our methodology arise when dealing with zero-day exploits that may not be a part of our

training dataset. Since we are learning at least in part on the payload itself, new exploits whose payloads have not been seen by the model during training are likely to be misclassified. Our methodology for creating FEATHER embeddings, however, only relies on the edge weights between packets leading us to believe it will generalize better to novel data.

Future work should focus on expanding these findings to other NIDS datasets, such as UNSW-NB15, and comparing results to see how well this methodology generalizes to other NIDS data. Further work can also test the susceptibilities of the models to novel or out-of-distribution input data, by withholding a particular attack type during training and testing if it is still classified as malicious. Using a multi-class classification model can also be explored to identify not only if a packet is malicious but also what attack category it belongs to.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. Army Combat Capabilities Development Command (DEVCOM) Army Research Laboratory under Support Agreement No. USMA21050, the U.S. Army DEVCOM C5ISR Center under Support Agreement No. USMA21056, and the U.S. Air Force Research Lab under Support Agreement No. USMA2226. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the U.S. Military Academy, U.S. Army, U.S. Department of Defense, or U.S. Government.

REFERENCES

[1] Maxwell, P., Alhajjar, E., and Bastian, N. D., "Intelligent feature engineering for cybersecurity," in [*2019 IEEE International Conference on Big Data (Big Data)*], 5005–5011, IEEE (2019).

[2] Lucia, M. J. D., Maxwell, P. E., Bastian, N. D., Swami, A., Jalaian, B., and Leslie, N., "Machine learning raw network traffic detection," in [*Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III*], 117460V, International Society for Optics and Photonics, SPIE (2021).

[3] Zhang, H., Wu, C. Q., Gao, S., Wang, Z., Xu, Y., and Liu, Y., "An effective deep learning based scheme for network intrusion detection," in [*2018 24th International Conference on Pattern Recognition (ICPR)*], 682–687, IEEE (2018).

[4] Bierbrauer, D. A., Chang, A., Kritzer, W., and Bastian, N. D., "Cybersecurity anomaly detection in adversarial environments," *Proceedings of the AAAI Fall 2021 Symposium on AI in Government and Public Sector. arXiv:2105.06742.* (2021).

[5] Lazarevic, A., Ertoz, L., Kumar, V., Ozgur, A., and Srivastava, J., "A comparative study of anomaly detection schemes in network intrusion detection," in [*Proceedings of the 2003 SIAM international conference on data mining*], 25–36, SIAM (2003).

[6] Lin, Z.-Z., Pike, T. D., Bailey, M. M., and Bastian, N. D., "A hypergraph-based machine learning ensemble network intrusion detection system," *arXiv preprint arXiv:2211.03933* (2022).

[7] Iliofotou, M., Pappu, P., Faloutsos, M., Mitzenmacher, M., Singh, S., and Varghese, G., "Network monitoring using traffic dispersion graphs (tdgs)," in [*Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*], 315–320 (2007).

[8] Zola, F., Segurola-Gil, L., Bruse, J., Galar, M., and Orduna-Urrutia, R., "Network traffic analysis through node behaviour classification: a graph-based approach with temporal dissection and data-level preprocessing," *Computers & Security* **115**, 102632 (2022).

[9] Xiao, Q., Liu, J., Wang, Q., Jiang, Z., Wang, X., and Yao, Y., "Towards network anomaly detection using graph embedding," in [*Computational Science – ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part IV*], 156–169, Springer-Verlag, Berlin, Heidelberg (2020).

[10] Lopez-Martin, M., Carro, B., Arribas, J. I., and Sanchez-Esguevillas, A., "Network intrusion detection with a novel hierarchy of distances between embeddings of hash ip addresses," *Knowledge-Based Systems* **219**, 106887 (2021).

[11] Lo, W. W., Layeghy, S., Sarhan, M., Gallagher, M., and Portmann, M., "E-graphsage: A graph neural network based intrusion detection system for iot," in [*NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*], 1–9 (2022).

[12] Roy, A., Cobb, A., Bastian, N. D., Jalaian, B., and Jha, S., "Runtime monitoring of deep neural networks using top-down context models inspired by predictive processing and dual process theory," in [*AAAI Spring Symposium Designing Artificial Intelligence for Open Worlds*], (2022).

[13] Acharya, M., Roy, A., Koneripalli, K., Jha, S., Kanan, C., and Divakaran, A., "Detecting out-of-context objects using graph context reasoning network," *IJCAI* (2022).

[14] "Intrusion Detection Evaluation Dataset (CIC-IDS2017)," (2017). University of New Brunswick https://www.unb.ca/cic/datasets/ids-2017.html.

[15] Farrukh, Y. A., Khan, I., Wali, S., Bierbrauer, D., Pavlik, J. A., and Bastian, N. D., "Payload-byte: A tool for extracting and labeling packet capture files of modern network intrusion detection datasets," in [*2022 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*], 58–67 (2022).

[16] Sharafaldin, I., Lashkari, A. H., and Ghorbani, A. A., "Toward generating a new intrusion detection dataset and intrusion traffic characterization.," *4th International Conference on Information Systems Security and Privacy (ICISSP)* **1**, 108–116 (2018).

[17] Hamilton, W., [*Graph Representation Learning*], Synthesis lectures on artificial intelligence and machine learning, Morgan & Claypool Publishers (2020).

[18] Xu, K., Hu, W., Leskovec, J., and Jegelka, S., "How powerful are graph neural networks?," in [*International Conference on Learning Representations*], (2019).

[19] Simonovsky, M. and Komodakis, N., "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in [*Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*], (July 2017).

[20] Rozemberczki, B. and Sarkar, R., "Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models," in [*Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*], 1325–1334, ACM (2020).

[21] Kudo, Y., Rossi, R. A., and Vert, J.-P., "Karate club: An api oriented open-source python framework for graph mining." https://github.com/benedekrozemberczki/karateclub (2021). Accessed: 2023-04-03.

[22] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al., "Scikit-learn: Machine learning in python," *the Journal of machine Learning research* **12**, 2825–2830 (2011).

[23] Chen, T. and Guestrin, C., "Xgboost: A scalable tree boosting system," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* , 785–794 (2016).

[24] Gama, F. and Kipf, T., "Spektral: A python library for graph deep learning," *Journal of Machine Learning Research* **22**, 1–6 (2021).

[25] Hagberg, A., Swart, P., and S Chult, D., "NetworkX: A python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks," *ACM Transactions on Mathematical Software (TOMS)* **41**(4), 1–7 (2014).