Evaluating Cybersecurity in ICS Environments Using Graph Neural Networks (GNNs)

Zach Bovaird

July 14, 2024

Abstract

In Industrial Control Systems (ICS) environments, robust cybersecurity architecture is critical. Building upon previous work on graphs and hypergraphs, this paper introduces Graph Neural Networks (GNNs), focusing specifically on Graph Convolutional Networks (GCNs). GCNs, along with other GNN subtypes such as Graph Attention Networks (GATs) and Global-Local Integration Networks (GLINs), are compared to provide a clear understanding of their applications in ICS cybersecurity. This paper guides professionals on implementing GCNs for network topology analysis and segmentation, leveraging data from tools such as GrassMarlin and Zeek/ELK stack, aiming to enhance security based on the IEC 62443 standard. This paper provides the results and comparison of running the same data in GNNs vs GCNs, with the all necessary files and code availabale for at https://github.com/zbovaird/Graphs-and-GNNs/tree/main/GNN

1 Introduction

The cybersecurity threat landscape in ICS environments is rapidly evolving, driven by increasingly sophisticated cyberattacks. Traditional security measures often fall short, necessitating the development of new tools. This paper presents GNNs as a cost-effective approach to enhance cybersecurity architecture, focusing on GCNs due to their simplicity and efficiency. By leveraging GNNs, organizations can improve their detection and mitigation of potential threats without significant additional costs.

2 Tools Used

2.1 Zeek/ELK Stack

- **Zeek**: A network analysis framework that provides extensive logging capabilities for network traffic. It helps in monitoring network activity and identifying anomalies in real-time.

- **ELK Stack**: Comprises Elasticsearch, Logstash, and Kibana. Elasticsearch stores and indexes data, Logstash processes and transforms it, and Kibana visualizes the data for analysis. This stack allows for real-time data ingestion, storage, and analysis.

2.2 GrassMarlin

- **Description**: An open-source tool for network discovery and mapping in ICS environments. It passively collects metadata from network traffic to create a detailed map of the network, including device information and communication patterns.
- **Usage**: Provides a source of metadata that can be used to build and update graph representations of the network.

2.3 NetworkX

- **Description**: A Python library for the creation, manipulation, and study of complex networks of nodes and edges.
- **Usage**: Used to build and manage the graph representations of the network. It supports various graph algorithms that are essential for network analysis.

2.4 PyTorch and PyTorch Geometric

- **PyTorch**: A deep learning framework that provides tools for building and training neural networks.
- **PyTorch Geometric**: An extension library for PyTorch, designed specifically for deep learning on irregularly structured data like graphs. It simplifies the implementation of GNNs and provides pre-built components for various GNN models.
- **Usage**: These tools are used to implement and train GNN models, including GCNs, for network analysis and anomaly detection.

3 Graph Neural Networks (GNNs)

GNNs are machine learning models designed to operate on graph-structured data. They leverage the connections between nodes to enhance learning and prediction tasks. A key concept in GNNs is the use of feature vectors.

3.1 Feature Vectors

A feature vector is a set of numerical values that represent the attributes or characteristics of a node in the graph. In the context of ICS, a feature vector for a device (node) could include various attributes such as its type, the volume of traffic it handles, the protocols it uses, and its connectivity pattern.

3.1.1 Importance of Feature Vectors

- **Representation**: Feature vectors provide a numerical representation of each node's characteristics, making it possible to process this information using machine learning algorithms.
- **Context**: By considering the feature vectors of neighboring nodes, GNNs can capture the local context and relationships within the graph, which is crucial for understanding the network's structure and behavior.
- **Flexibility**: Feature vectors can be tailored to include any relevant attributes, allowing GNNs to be applied to a wide range of problems.

3.1.2 Attributes to Include in Feature Vectors

When analyzing an OT/ICS network with GrassMarlin, the feature vector for each device can include the following attributes:

- **Device Type**: Type of device, such as PLC, RTU, sensor, HMI, etc.
- **IP Address**: The IP address of the device.
- **MAC Address**: The MAC address of the device.
- **Role**: The role or function of the device in the network.
- **Vendor Information**: Information about the device manufacturer and model.
- **Protocol Usage**: The communication protocols used by the device (e.g., Modbus, Profibus, DNP3).
- **Traffic Volume**: The amount of data sent and received by the device.
- **Connection Count**: The number of connections the device has with other devices in the network.
- **Connection Types**: The types of connections (e.g., direct, indirect) the device has.
- **Operating System**: The operating system running on the device, if applicable.
- **Software Versions**: The versions of software or firmware installed on the device.

- **Device Status**: The current status of the device (e.g., active, inactive, faulty).
- **Time-Related Attributes**: Timestamps of the latest communication or configuration changes.

3.2 Message Passing

Message passing is a key operation in GNNs, where nodes in the graph exchange information with their neighbors. Each node sends its current state (or features) to its neighbors and receives the states of its neighbors in return. This process allows nodes to update their states based on the information received from their local neighborhood.

Formally, for a node v at layer l, the message passing operation can be described as:

$$m_v^{(l)} = AGGREGATE\left(\{h_u^{(l)}, \forall u \in \mathcal{N}(v)\}\right)$$

where $h_u^{(l)}$ is the state of neighbor node u at layer l, $\mathcal{N}(v)$ is the set of neighbors of v, and $AGGREGATE$ is a function that aggregates the received messages.

3.3 Message Aggregation

Message aggregation combines the information received from neighboring nodes to update the state of the node. Different GNN models use different aggregation functions, such as summation, mean, or maximum.

The state update for node v at layer l can be described as:

$$h_v^{(l+1)} = UPDATE\left(h_v^{(l)}, m_v^{(l)}\right)$$

where $h_v^{(l+1)}$ is the updated state of node v at layer $l+1$, and $UPDATE$ is a function that combines the node's current state with the aggregated messages.

3.4 Backpropagation in GNNs

Backpropagation in GNNs involves computing the gradient of the loss with respect to the parameters of the model. Unlike regular neural networks, where the gradient is computed for each layer independently, GNNs require the gradients to be propagated through the message passing and aggregation steps. This involves computing the gradients of the aggregation function and ensuring that the message passing steps are differentiable.

3.5 Graph Convolutional Networks (GCNs)

GCNs extend the concept of convolution from grid-like data to graph data. In a GCN, each node in the graph updates its state by combining its own features with those of its neighbors. This process allows the network to learn patterns that depend on the graph structure.

Here's a simplified way to understand it:

1. **Initial Features**: Each node has some initial features. Think of these as the starting information about each device in the network.
2. **Combining Information**: Each node looks at its neighbors and combines their features with its own. This is done in a way that weights each neighbor's contribution.
3. **Update State**: The combined information is used to update the node's features.

In mathematical terms, this can be written as:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

Where:

- $\mathbf{H}^{(l)}$ represents the features of all nodes at layer l.
- $\hat{\mathbf{A}}$ is the adjacency matrix of the graph with added self-loops (connections of nodes to themselves).
- $\hat{\mathbf{D}}$ is a diagonal matrix that helps normalize the adjacency matrix.
- $\mathbf{W}^{(l)}$ is a matrix of learnable weights at layer l.
- σ is an activation function, like ReLU, that adds non-linearity to the model.

Backpropagation in GCNs involves calculating gradients through this process of combining and updating features, which is more complex than in regular neural networks due to the graph structure and normalization steps.

3.6 Graph Attention Networks (GATs)

GATs introduce attention mechanisms to weigh the importance of different neighbors during aggregation, allowing the model to focus on more relevant parts of the graph. This weighting process enables GATs to handle graphs with heterogeneous nodes and varying importance of connections.

3.7 Global-Local Integration Networks (GLINs)

GLINs combine global and local information in the graph, integrating high-level network structure with detailed local interactions.

3.7.1 Components

- **Preprocessor**: Transforms raw data into initial vectors for message passing.
- **Encoder**: Learns node embeddings using methods like GCN, GAT, and Graph-SAGE.
- **Pooling Module**: Aggregates node embeddings to create global representations.
- **Integration Module**: Combines global and local embeddings through concatenation or fusion.
- **Decoder**: Maps integrated embeddings to output labels using fully connected layers.

The components are described in the paper by Wang et al. (2022) on global-local integration for GNN-based anomalous device state detection in ICS.

3.8 Differences from Regular Neural Networks

Regular neural networks operate on fixed-size, grid-like data and do not account for the relational information between data points. GNNs, through feature vectors, message passing, aggregation, and specialized backpropagation, explicitly incorporate the graph structure into the learning process, making them suitable for tasks where relationships and interactions between entities are crucial.

4 Implementation and Comparison of GNN and GCN Models

4.1 Data Preparation and NetworkX Graph Creation

Nodes and edges were loaded from CSV files representing an OT/ICS network into a NetworkX graph. The nodes represented devices, while edges represented communication links between devices.

4.2 Converting NetworkX Graph to PyTorch Geometric Data Format

The NetworkX graph was converted to a PyTorch Geometric `Data` object with dummy features and labels for each node.

4.3 GNN Model Implementation and Results

4.3.1 GNN Model Definition and Training

A Graph Neural Network (GNN) model was defined and trained using PyTorch Geometric.

4.3.2 GNN Training Output

The GNN model was trained for 100 epochs with the following loss reduction:

```
Epoch 0, Loss: 0.6931473016738892
Epoch 10, Loss: 0.598453164100647
Epoch 20, Loss: 0.5151417851448059
Epoch 30, Loss: 0.44355952739715576
Epoch 40, Loss: 0.38309967517852783
Epoch 50, Loss: 0.33254337310791016
Epoch 60, Loss: 0.2904347777366638
Epoch 70, Loss: 0.25534459948539734
Epoch 80, Loss: 0.22600263357162476
Epoch 90, Loss: 0.2013387680053711
```

Training completed in 0.37 seconds.

4.4 GCN Model Implementation and Results

4.4.1 GCN Model Definition and Training

A Graph Convolutional Network (GCN) model was defined and trained using PyTorch Geometric.

4.4.2 GCN Training Output

The GCN model was trained for 100 epochs with the following loss reduction:

```
Epoch 0, Loss: 1.1390044689178467
Epoch 10, Loss: 0.6408043503761292
Epoch 20, Loss: 0.38536536693573
Epoch 30, Loss: 0.2381220906972885
Epoch 40, Loss: 0.13158735632896423
Epoch 50, Loss: 0.07088468223810196
Epoch 60, Loss: 0.04071911796927452
Epoch 70, Loss: 0.025904536247253418
Epoch 80, Loss: 0.018121762201189995
Epoch 90, Loss: 0.013625713996589184
```

Training completed in 0.17 seconds.

4.5 Key Findings from GCN Model Implementation

- **Efficiency**: The GCN model trained very quickly, completing 100 epochs in just 0.17 seconds.
- **Loss Reduction**: The loss consistently decreased, indicating effective learning by the model.

4.6 Comparison Between GNN and GCN Models

4.6.1 General GNN Model

- **Flexibility**: Can use various aggregation and update functions.
- **Complexity**: More complex in terms of customization and implementation.
- **Use Cases**: Suitable for a broad range of applications, including those with non-structural data.

4.6.2 GCN Model

- **Simplicity**: Uses a fixed aggregation method (normalized adjacency matrix) which simplifies implementation.
- **Efficiency**: Less computationally intensive, suitable for real-time applications.
- **Effectiveness**: Capable of capturing the graph structure efficiently, making it ideal for network topology analysis.

4.7 Why Start with GCNs?

- **Ease of Implementation**: GCNs are straightforward to implement using tools like PyTorch Geometric. This makes them accessible to professionals who may not have extensive experience with machine learning.
- **Efficiency**: GCNs are computationally efficient, allowing for quick training and inference, which is crucial in real-time cybersecurity applications.
- **Effectiveness**: GCNs effectively capture the structure of the network, which is essential for identifying critical nodes and potential vulnerabilities in ICS environments.

4.8 Practical Benefits for OT/ICS Cybersecurity

- **Network Analysis**: GCNs can analyze the network topology, identifying key devices and communication patterns.
- **Anomaly Detection**: By learning the typical behavior of the network, GCNs can help detect anomalies and potential threats.
- **Segmentation Improvement**: Using the insights from GCNs, organizations can improve network segmentation, isolating critical nodes and reducing attack surfaces.

4.9 Real-Time Data Pipeline

Set up a pipeline for near-real-time data acquisition and analysis:

1. **Data Stream**: Continuously collect and preprocess data using Zeek/ELK stack.
2. **Graph Update**: Periodically update the graph representation with new data.
3. **Model Inference**: Run the GCN model to detect anomalies and analyze network topology.

5 Network Segmentation and Security (IEC 62443)

Utilize the GCN to evaluate and improve network segmentation based on the IEC 62443 standard:

- **Identify Critical Nodes**: Use the GCN to detect nodes that are central to network communication and may pose security risks if compromised.
- **Enhance Segmentation**: Adjust network segmentation to isolate critical nodes and reduce the risk of spreading attacks.
- **Generate Alerts**: Implement a system to generate alerts when anomalies are detected, integrating with existing security infrastructure.

6 Conclusion

Starting with GCNs offers a practical and effective approach to incorporating GNNs into ICS cybersecurity. By leveraging tools like GrassMarlin and Zeek/ELK stack, and focusing on real-time analysis and network segmentation, organizations can enhance their security posture with minimal additional cost and complexity. Future work can explore more advanced GNN models like GATs and GLINs to further improve detection capabilities and resilience.

This paper serves as a guide for ICS cybersecurity professionals to understand and implement GNNs, providing a stepping stone towards more sophisticated network analysis and protection strategies.

7 References

1. Kipf, T. N., & Welling, M. (2016). "Semi-Supervised Classification with Graph Convolutional Networks." arXiv preprint arXiv:1609.02907.
2. Velickovic, P., et al. (2017). "Graph Attention Networks." arXiv preprint arXiv:1710.10903.

3. Hamilton, W. L., Ying, R., & Leskovec, J. (2017). "Inductive Representation Learning on Large Graphs." Advances in Neural Information Processing Systems.

4. GrassMarlin User Guide. National Security Agency (NSA).

5. Zeek Network Security Monitor Documentation.

6. IEC 62443-3-3:2013: "System security requirements and security levels."

7. Wang, K., et al. (2022). "Global-local integration for GNN-based anomalous device state detection in ICS." Expert Systems with Applications.