Temporal Graph Neural Network-Based Real-Time Intrusion Detection for IoT Devices

Kaichun Yang
University of Illinois at Urbana-Champaign
ky27@illinois.edu

Abstract

The increasing popularity of IoT devices has facilitated our lives, and many different IoT devices have their own roles and functions in daily life, such as home smart cameras, speakers, door locks, etc. However, due to the hardware limitations of IoT devices, most of them lack built-in security measures, which makes them more vulnerable to attackers than the other network devices; And IoT devices from large companies like Amazon [1] and Xiaomi can monitor the received traffic and upload it to the cloud platform for analysis, but this approach cannot detect and report suspicious traffic in real time. To solve this problem, many studies have been conducted to identify traffic attacks based on network traffic characteristics. SVM [3] and neural networks are used to identify attacks against IoT. However, they all have certain flaws to varying degrees, such as the inability of SVM methods to effectively capture complex distributed attacks. Although the neural network method is effective and can cover most attacks, it relies on fine-tuning to correct the model, and its performance in judging sparse traffic is not optimistic. It requires collecting a large amount of traffic to form a graph of a certain scale before making accurate enough classification. Therefore, I combined the existing E-GraphSAGE [5] method with the characteristics of TGN to form E-GraphSAGE-T. According to my plan, this model not only has the same lightweight as E-GraphSAGE, but also, with the introduction of time encoding, can incorporate temporal change information as a member of attributes into learning, enabling the model to receive traffic of any scale and make judgments in real-time. I trained my model on the ToN_IoT_v3 [6] dataset and evaluated it on both the ToN_IoT_v3 and BoT_IoT_v3 datasets. The model showed good performance during the training phase but was not ideal during the testing phase. These issues may be due to the method I chose to incorporate time into the model, as well as computer performance problems that resulted in training a sufficiently complete model, with a maximum completion of only 20 epochs. However, this is still an attempt to introduce TGN related content such as time reasoning and memory mechanisms into real-time attack detection for IoT.

1 Introduction

Due to the hardware performance of IoT devices, most of them are unable to run other host-based security solutions, such as antivirus software or traffic analysis tools. There are also some solutions that collect traffic data and upload them to the cloud. Although such solutions can collect a large amount of traffic data and helpful for analyzing anomalies, they cannot prevent attacks in real time and are also susceptible to access policies such as network latency or interruptions. There are some methods for detecting IoT devices, such as using random forests or Bayesian classification methods to identify attacks in real time [2]. There are also methods to identify attacks through neural networks [5]. The method of using random forests for real-time detection is quite mature, while the second method using neural networks does not have the ability for real-time monitoring. The one thing neither of these methods has achieved is to incorporate temporal changes as part of the attribute into the learning process to provide the model with the ability to detect attacks based on temporal changes. In recent years, gateway level intrusion detection systems have become a reliable method to detect abnormal traffic. The gateway can serve as a centralized security checkpoint to monitor all traffic entering and exiting local network devices. However, such technology also poses certain challenges, firstly in terms of the performance of gateway devices themselves. Most home gateway devices do not have enough memory and storage space for large model, which means that the inference time and available memory of the model will be limited. Secondly, attacks are constantly evolving, and it will become difficult for well-trained static models to generalize with changes in attacks. Therefore, in order to deploy models that can be monitored in real-time and learn automatically on gateway devices, special design is required. GNN provides a framework to address these challenges, which is E-GraphSAGE, a GNN model specifically designed for detecting attacks against IoT devices. It takes address and destina-

tion as nodes, attaches traffic attributes to edges, and classifies edges to construct a static traffic graph and perform message passing between nodes to detect whether traffic is benign or malicious. Due to the lightweight characteristic of GraphSAGE, it can be deployed smoothly in gateways and perform inference without consuming a large amount of memory. However, E-GraphSAGE has some limitations for real-time monitoring environments. Firstly, E-GraphSAGE does not have real-time monitoring capabilities, which means that the original E-GraphSAGE can only rely on static traffic snapshots for learning and inference; Secondly, E-GraphSAGE does not incorporate temporal changes into the model, so attacks that vary over time cannot be well captured by E-GraphSAGE. In addition, when the traffic is sparse, that is, when the size of the graph is small, the accuracy of E-GraphSAGE will greatly fluctuate. This is related to the characteristics of GNN itself. The transmission mechanism of neural networks has already explained the information transmission defects and judgment errors caused by incomplete information in graph datasets that are too small. This has also been demonstrated in my experiments on the well performing E-GraphSAGE model. To address these issues, I designed E-GraphSAGE-T, which changes the SAGE layer of E-GraphSAGE so that the model is no longer trained solely on static graphs. Instead, it first stores the memory of the nodes generated in the previous forward and then learns new ones based on the previous changes. This gives the network the ability to accept updated edges, make judgments, and capture the behavior of nodes over time. Therefore, it is more suitable for real-time detection and inference on lightweight devices such as gateways. I conducted training and testing on two new datasets generated by NetFlow, BoT_IoT, and ToN_IoT: NF_ToN_IoT_v3 and NF-BoT_IoT_v3. The experimental results showed that combining time for inference can improve the accuracy and robustness of detection. However, due to the limited number of training epoch and parameter setting of the model, the current precision is poor. However, it still demonstrates the feasibility of incorporating time elements into GNN in protecting IoT security measures and provides a framework for edge's real-time anomaly detection methods. Code in: Temporal Graph Neural Network-Based Real-Time Intrusion Detection for IoT

2 Related Work

2.1 E-GraphSAGE

E-GraphSAGE [5] is built on the basis of GraphSAGE [4] and adopts an edge classification scheme to address the classification of IoT traffic. Each unique node is concatenation of address and port, and each edge represents the network traffic between two nodes. The task is to predict whether the edge is benign or malicious. E-GraphSAGE aggregates all traffic within a fixed observation window and constructs a static traffic graph. Nodes do not carry any features, only edges carry relevant features of the data flow, such as protocols, in/out packages, and so on. In addition, it applies a message passing network with two layers of GraphSAGE. In each layer, node embeddings are updated by aggregating messages from adjacent nodes, which are calculated based on the features of adjacent embeddings and edges. Finally, the model uses MLP predictor to perform edge classification through node embedding and edge features. The authors demonstrated through experiments using NetFlow based datasets NF_ToN_IoT and NF_BoT_IoT that E-GraphSAGE outperforms traditional machine learning in performing binary tasks for IoT traffic classification, such as using SVM or random forest. However, its static snapshot method cannot distinguish the traffic at different times within the window, thus losing the temporal variation and sequence information of events.

2.2 Temporal Graph Networks (TGN)

The Temporal Graph Network (TGN) [8] solves the problem of dynamic graph learning by integrating memory modules and time encoding into message passing. Unlike static graph methods, TGN views a graph as a series of time stamped events, where each event's update triggers an update to the relevant node state, allowing the network to capture changing times and topological patterns. TGN has four important components: memory module, message function, memory update, and embedding module. Each node has a hidden state, which is memory used to accumulate all information about past interactions. When an event is triggered, the memory of the node will be retrieved for subsequent calculations. Given an event (u, v, t, e), where u, v are nodes, t is the timestamp, and e are the edge features. TGN calculates the current message based on the current memories of u and v, as well as their edge features. The information will be aggregated and used to update the memories of nodes, ensuring that the latest information is used for subsequent event calculations. Node embedding will update these memories and use them for calculation in the convolutional layer. TGN leverages continuous time encoding to embed relative time information into features. This allows the model to distinguish between times that occur at different times and learn their specific variations. My work in this project is to draw inspiration from the memory and time encoding mechanism of TGN and combine it with the static snapshot method of E-GraphSAGE to quickly update memory and achieve real-time detecting.

2.3 NF_ToN/BoT_IoT Dataset

NF_ToN_IoT and NF_BoT_IoT [9] are NetFlow versions of the ToN_IoT and BoT_IoT datasets [7]. The dataset reconstructed for machine learning to detect IoT targeted attacks. They used nProbe to extract the attributes of ToN_IoT and BoT_IoT and integrated 14 features that are easy for machine learning to form NF_ToN_IoT and NF_BoT_IoT [9].

These two datasets have undergone multiple version updates, with the latest versions being NF_ToN_IoT_v3 and NF_BoT_IoT_v3 [6]. The third edition expanded the number of features and retained timestamps. The original E-GraphSAGE model used the first version, which only had 14 features. Because I used the core part of E-GraphSAGE, I still used these 14 features and kept the timestamp from the third version for time encoding and node memory.

3 Data Processing

I used NF_ToN_IoT_v3 dataset for training step. However, since the initial version of E-GraphSAGE only used the 14 features of NF_ToN_IoT, so keep the same 14 features from NF_ToN_IoT_v3 and add the new feature Flow_START_BY_MILLISECONDS from the third version as timestamps. Please refer to Table 1 for detailed information of the features. Because the dataset is too large, I only extracted 8% of the data for use. To ensure timestamp coherence of the data, I divided the entire dataset into 30 buckets and extracted a certain amount of data flow from each bucket.

I first concatenated each flow's source address and port for NF_ToN: I combined `IPV4_SRC_ADDR` with `L4_SRC_PORT`, and likewise combined `IPV4_DST_ADDR` with `L4_DST_PORT`. This produced unique `source` and `destination` values, after which I dropped the original four columns and replaced them with the two new `source` and `destination` columns. Next, I processed `FLOW_START_MILLISECONDS` (the timestamp) by normalizing it with `min_max` and storing the result. I also preserved another form of the timestamp by converting it from milliseconds to seconds and storing that. Then I split the data into training and test sets in a 7:3 ratio, placing all test data at the end of the dataset. After that, I applied an encoder to the three columns `TCP_FLAGS`, `L7_PROTO`, and `PROTOCOL`, and normalized all edge features storing them in a new column.

Through the above steps, I obtain the dataset required for training, and prediction uses the same format. I attempted to use Zeek to monitor ports and extract the features; except for `L7_PROTO` and `TCP_FLAGS`, all other features can be used directly as extracted by Zeek. For `L7_PROTO`, Zeek can only extract the service, whereas the NF_ToN_IoT authors used nProbe, which is based on the nDPI service list; I created a map to correspond one-to-one with it, thereby obtaining the same `L7_PROTO` numerical values as in the training data. However, note that Zeek cannot fully extract service-protocol content, so it cannot cover as many different services as nProbe; Zeek can only recognize common service names, and most entries are blank. Zeek extracts `TCP_FLAGS` as the initial letters of the control section, so the corresponding decimal can be computed directly. This ensures that data extracted by Zeek can also be fed directly into the model for use.

4 Model Design

E-GraphSAGE-T is a temporal extension of the static E-GraphSAGE model, designed to perform real-time intrusion detection on gateways devices. Each network flow arrives as a 4-tuple $(\mathrm{src}, \mathrm{dst}, \mathbf{e}, t)$, where $\mathrm{src}, \mathrm{dst} \in \{0, \ldots, N-1\}$ index communicating endpoints, $\mathbf{e} \in \mathbb{R}^F$ is a fixed-length feature vector summarizing protocol, L7_PROTO(service), IN/OUT Bytes/Packages TCP flags, and flow duration in milliseconds, and $t \in \mathbb{R}$ is the normalized start timestamp. To capture both structural and sequential patterns, I maintain a persistent memory embedding

$$M \;=\; \mathrm{mem.weight} \;\in\; \mathbb{R}^{N\times H},$$

initialized to zeros, which accumulates contextual information from all past flows incident on each node. This memory enables the model to adapt its predictions based on long-term behavioral trends, such as repeated scans from the same source or gradual payload increases over time.

At runtime, the system groups incoming flows into mini-batches of size B according to non-overlapping time windows (e.g., 1s for low-latency detection or 120s for higher throughput). For each batch, E-GraphSAGE-T executes the following pipeline:

1. **Time encoding.** Each scalar timestamp t is fed into a lightweight linear layer and then mapped through element-wise sin and cos activations, producing a $2T$-dimensional vector $\mathbf{t}_{\mathrm{enc}}$. This encoding captures both periodic and relative temporal information without fixed discretization.

2. **Memory lookup.** I fetch the full embedding table M for all N nodes; only rows corresponding to src and dst indices actively participate in subsequent computations.

3. **Message passing.** A minimal graph is constructed with edge indices $\mathrm{edge\_index} = [\mathrm{src}; \mathrm{dst}]$. I concatenate each flow's original features $\mathbf{e}$ with its time encoding $\mathbf{t}_{\mathrm{enc}}$ to form edge attributes. Two sequential GraphSAGE layers then propagate, aggregate, and update node states by mixing neighbor memories with these enriched edge attributes.

4. **Edge classification.** Updated hidden states $h_u, h_v \in \mathbb{R}^{B\times H}$ for each flow's endpoints are concatenated with $\mathbf{e}$ and $\mathbf{t}_{\mathrm{enc}}$, and passed through a two-layer MLP to produce logits for benign vs. malicious classification.

5. **Memory update.** Finally, I perform an in-place overwrite of the memory rows for all $\mathrm{src}, \mathrm{dst}$ indices with their newly computed hidden states, ensuring that subsequent batches "see" the most recent context.

This end-to-end design, combining continuous time encoding, persistent node memory, and lightweight message

Table 1: NF_ToN_IoT v1 Features

Feature	Description
FLOW_START_MILLISECONDS	Flow start timestamp in milliseconds
IPV4_SRC_ADDR	IPv4 source address
IPV4_DST_ADDR	IPv4 destination address
L4_SRC_PORT	IPv4 source port number
L4_DST_PORT	IPv4 destination port number
PROTOCOL	IP protocol identifier byte
TCP_FLAGS	Cumulative of all TCP flags
L7_PROTO	Layer 7 protocol (numeric)
IN_BYTES	Incoming number of bytes
OUT_BYTES	Outgoing number of bytes
IN_PKTS	Incoming number of packets
OUT_PKTS	Outgoing number of packets
FLOW_DURATION_MILLISECONDS	Flow duration in milliseconds

Algorithm 1 E-GraphSAGE-T

Require: Mini-batch of B flows: $\{(\text{src}_i, \text{dst}_i, e_i, t_i)\}_{i=1}^{B}$, memory table $M \in \mathbb{R}^{N\times H}$, model components $\{\text{TimeEncode}, \text{SAGELayer}_1, \text{SAGELayer}_2, \text{MLPPredictor}\}$

$\mathbf{t}_i \leftarrow \text{TimeEncode}(t_i)$ for $i = 1, \ldots, B$
$\mathbf{X} \leftarrow M$
edge_index $\leftarrow$ [src; dst] $\in \mathbb{N}^{2\times B}$
edge_attr$_i \leftarrow [e_i \,\|\, \mathbf{t}_i]$ for $i = 1, \ldots, B$
$\mathbf{H}^{(1)} \leftarrow \text{SAGELayer}_1(\mathbf{X}, \text{edge\_index}, \{\text{edge\_attr}_i\})$
$\mathbf{H}^{(1)} \leftarrow \text{Dropout}(\mathbf{H}^{(1)})$
$\mathbf{H}^{(2)} \leftarrow \text{SAGELayer}_2(\mathbf{H}^{(1)}, \text{edge\_index}, \{\text{edge\_attr}_i\})$
$\mathbf{H}^{(2)} \leftarrow \text{Dropout}(\mathbf{H}^{(2)})$
for $i = 1$ to B **do**
    $h_{u_i} \leftarrow \mathbf{H}^{(2)}[\text{src}_i]$
    $h_{v_i} \leftarrow \mathbf{H}^{(2)}[\text{dst}_i]$
    $\text{logits}_i \leftarrow \text{MLPPredictor}(h_{u_i}, h_{v_i}, e_i, \mathbf{t}_i)$
end for
all_idx $\leftarrow$ [src; dst]
all_state $\leftarrow [h_{u_1}, \ldots, h_{u_B}, h_{v_1}, \ldots, h_{v_B}]$
no-grad: M[all_idx] $\leftarrow$ all_state
return $\{\text{logits}_i\}_{i=1}^{B}$

passing, enables accurate, low-latency detection of evolving IoT attack on edge gateways.

4.1 Input Representation and Embedding Initialization

As mentioned in the data pre-processing part, before any neural computation, raw dataflow records are processed to produce the inputs required by the model. The source and destination address-port pairs are concatenated into strings:

$$\texttt{src\_key} = \texttt{IPV4\_SRC\_ADDR} : \texttt{L4\_SRC\_PORT},$$
$$\texttt{dst\_key} = \texttt{IPV4\_DST\_ADDR} : \texttt{L4\_DST\_PORT}.$$

and each unique key is mapped to a compact integer via a runtime-maintained dictionary `id_map`. This mapping grows as new endpoints appear and prunes stale entries not seen for a configurable duration to keep the total number of nodes N bounded. Simultaneously, the original feature vector $\mathbf{e}$ is assembled by selecting columns for protocol number, L7 protocol code, incoming/outgoing packet and byte counts, TCP flags, and flow duration, each normalized via Min–Max scaling. The flow start timestamp in milliseconds is also Min–Max normalized to $[0, 1]$ per window, yielding t. At deployment, I allocate an embedding layer:

```
self.mem = nn.Embedding(num_nodes, hidden_dim)
nn.init.zeros_(self.mem.weight)
```

which creates the persistent memory table $M \in \mathbb{R}^{N\times H}$. Zero initialization provides a cold-start memory with no prior bias.

4.2 Temporal Encoding Module

To embed continuous-time information without fixed discretization, I implement an encoder in the `TimeEncode` module. Each scalar timestamp $t \in [0, 1]$ is first projected to a T-dimensional vector $\mathbf{z} = W_t\, t + b_t$, where $W_t \in \mathbb{R}^{T\times 1}$ and $b_t \in \mathbb{R}^T$ are learned parameters. Then compute $\sin(\mathbf{z})$ and $\cos(\mathbf{z})$ element-wise and concatenate the results to form a $2T$-dimensional encoding:

$$\mathbf{t}_{\text{enc}} = [\sin(\mathbf{z}) \,\|\, \cos(\mathbf{z})] \in \mathbb{R}^{2T}.$$

This design captures both periodic and relative temporal patterns, while the learned projection parameters adapt to the time scales most relevant for IoT attack detection. In my implementation, I set $T = 16$, yielding a 32-dimensional vector that is concatenated with the original edge features $\mathbf{e}$.

4.3 Node Memory Embedding and Update

The embedding table `mem.weight` $\in \mathbb{R}^{N\times H}$ serves as a node memory, where each row corresponds to a node's hidden state (memory). During each forward pass, the model fetch the entire table into $X \in \mathbb{R}^{N\times H}$. Although only entries indexed by the current batch's *src* and *dst* are accessed in message passing, fetching all rows leverages contiguous GPU memory access and simplifies implementation. After two layers of message passing produce updated hidden states $h_u, h_v \in \mathbb{R}^{B\times H}$ for the batch's endpoints, immediately perform an in-place memory update without gradient changing context. This mechanism replaces each node's prior memory with its newly computed state.

4.4 Message-Passing Layers

I adapt the GraphSAGE basic idea to integrate both edge and time features into the message-passing process. On each directed edge $j \to i$, the neighbor's hidden state $x_j \in \mathbb{R}^H$ is concatenated with the corresponding edge attribute vector, then passed through a learnable linear transformation W_{msg} to produce a message vector $\mathbf{m}_{j\to i} \in \mathbb{R}^H$. For each node i, incoming messages are aggregated by calculating their mean:

$$\bar{\mathbf{m}}_i \;=\; \frac{1}{|N(i)|}\sum_{j\in N(i)} \mathbf{m}_{j\to i}.$$

The node's new hidden state is then computed by concatenating the aggregated message $\bar{\mathbf{m}}_i$ with its previous state x_i, applying a second linear transformation W_{apply}, and passing the result through a activation function:

$$x_i' \;=\; \text{ReLU}\Big(W_{\text{apply}}\,[x_i \,\|\, \bar{\mathbf{m}}_i]\Big).$$

A dropout layer with rate 0.2 is applied between the two GraphSAGE layers to reduce overfitting on batches. Since there are two layers, the model achieves information propagation across two hop neighborhoods, allowing it to capture more complex relational patterns that are indicative of coordinated or multi-stage intrusion behaviors.

4.5 Edge Classification

Once the two GraphSAGE layers have produced updated node embeddings, I extract for each flow its source embedding $h_u = [\text{src}]$ and its destination embedding $h_v = [\text{dst}]$. These two vectors are concatenated with the original edge feature vector $\mathbf{e} \in \mathbb{R}^F$ and the time encoding $\mathbf{t}_{\text{enc}} \in \mathbb{R}^{2T}$ to form a single combined representation of dimension $2H + F + 2T$. This representation is then passed through a lightweight, two-layer perceptron: the first linear projection reduces it to a hidden dimension and applies a ReLU activation, while the second linear projection produces two raw logits corresponding to the benign and malicious classes.

During training, I initialize a weighted cross-entropy loss using a manually defined class-weight tensor:

$$\mathcal{L} \;=\; \text{CrossEntropyLoss}(\mathbf{y}, \mathbf{t}; \mathbf{w}),$$

where $\mathbf{y} \in \mathbb{R}^2$ are the predicted logits, $\mathbf{t} \in \{0,1\}$ are the true labels, and $\mathbf{w}$ is the precomputed weight vector passed to the loss function. By calling 'loss.backward()', gradients from $\mathcal{L}$ flow through both the MLP and the preceding GNN layers, enabling end-to-end learning even in the presence of severe class imbalance.

4.6 Forward Pass

In each forward pass, begin with applying the sinusoidal time encoding to map raw timestamps into high-dimensional vectors. Next, I retrieve the current node memory table and assemble a mini-batch graph by pairing source and destination nodes and attaching the combined edge features and time embeddings. That graph is then passed through the two-layer message-passing network, which updates every node's hidden state. After message passing, immediately extract the updated embeddings for current batch's sources and destinations and write them back into the memory table in-place without gradient tracking to preserve new context for subsequent batches. Finally, I concatenate these refreshed embeddings with the original flow features and time encodings and feed the result into the classification to produce the final logits. Throughout this process, the entire transformation is differentiable, allowing gradients to flow end-to-end during training.

4.7 Training Configuration

I trained E-GraphSAGE-T using the Adam optimizer with an initial learning rate of 1×10^{-3} and a weight decay of 1×10^{-5}. Each epoch processes the entire set of time-windowed batches, and grouped into one hour intervals without shuffling the sequence in each window to preserve temporal order, but the order of windows is randomized each epoch. Due to compute constraints, training was run for only 20 epochs, at which point the validation loss had not yet fully converged. I evaluate performance after each epoch on a held-out validation split (30% of the data), tracking accuracy, precision, recall, and F1-score.

5 Evaluation

I partitioned the preprocessed NF_ToN_IoT_v3 and NF_BoT_IoT_v3 datasets into 7:3 training and test set, preserving the natural chronological order to avoid information leakage. All experiments were conducted on my own laptop with an NVIDIA RTX 4060 GPU and 16 GB of RAM. Flows were batched into non-overlapping one-second windows, and a manually specified, fixed class-weighted cross-entropy loss was used to mitigate severe class imbalance.

5.1 Training Performance

Over the course of 20 training epochs, the weighted cross-entropy loss on the training set declined steadily, from approximately 0.80 in the first epoch to around 0.28 by epoch 20. From the training accuracy Figure 1 shows, it rose from about 56% to just over 90%. These learning curves indicate that the model was hardly converging and training was halted. That means my model may not have had strong learning ability from the beginning, or it may have been unable to fit well due to some issues in the early stages.

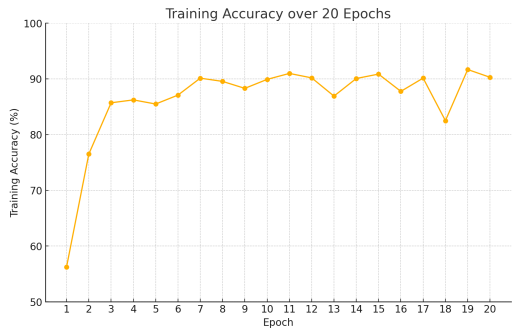

Figure 1: Training accuracy over 20 epochs

5.2 Test Evaluation

On the held-out test set, however, the model nearly failed to identify any malicious flows, predicting every example as benign. As a result, the precision and recall for the malicious class were both 0%, and the F1-score is undefined (conventionally set to 0). The overall test accuracy matched the fraction of benign samples, confirming that the model effectively learned the majority-class baseline rather than the intrusion detection task.

5.3 Evaluation of Results

The great difference between high training accuracy and zero precision on the test set highlights a severe overfitting to the benign majority. Despite existing the evidence of convergence in training, the model never learned to generalize to malicious traffic under the chosen loss, batching, and memory-update strategies. These outcomes underscore the need for more imbalance mitigation, longer training time, and refined memory mechanisms to achieve meaningful detection performance.

6 Conclusion and Conclusion

The final results were disappointing with very low precision. I believe this was mainly because my laptop's performance did not allow me to train for more epochs. As mentioned above, I had to stop at only twenty epochs while the model was still converging. Moreover, due to time constraints, I was unable to thoroughly refine the model or tune its hyperparameters, particularly the part where I replaced the original E-GraphSAGE memory with a TGN style time-varying memory. In the node embedding step, I simply overwrote the old information with the new; in the future I could try updating by combining the original and new information, thereby retaining past context while incorporating new signals. Additionally, the GraphSAGE propagation mechanism itself could be modified, for example, by adding more layers to ensure more effective information flow, which I would like to explore. I also only implemented a basic time encoding and memory update, without incorporating other specialized TGN techniques; adding components such as LSTMs could further improve the model's ability to capture temporal variations. I also believe that using fixed, non-overlapping time windows to batch data will also influence the model's temporal sensitivity. As noted in the training section, the first hour of traffic contained 190,000 flows that is more than one-eighth of the entire training set, yet the full dataset spanned four days, which shows an extremely uneven distribution. Although I normalized the timestamps, this skew could not be eliminated. The same issue affects NF_BoT_IoT: tens of thousands of flows often occur within the same second, followed by gaps of hours or even a full day, making any model's ability to learn time-based features nearly impossible. Even if the datasets themselves were well balanced, my windowing method was flawed: I did not experiment with overlapping windows to create more coherent temporal groupings, which could improve context continuity and detection of slow-speed attacks. In addition, I think that starting every prediction from no memory state and learning incrementally is not ideal. Without any initial node embeddings, the model's early predictions are very poor—just as E-GraphSAGE struggles on small graphs. Those initial, low-quality embeddings and memories then negatively affect all subsequent predictions and classifications, leading to generally low performance. Instead, one could first collect traffic for a period of time and train a standard E-GraphSAGE model to obtain its node embeddings. These embeddings and their memory table could then be used to initialize E-GraphSAGE-T, giving it some predictive capability from the start and stabilizing its later accuracy. Furthermore, periodically retraining static E-GraphSAGE and refreshing both embeddings and memory would maintain high memory fidelity over the long term.

Overall, the failure of my model reflects multiple mistakes and shortcomings in model selection, data processing, and the training process. This does not prove that the approach itself is invalid, with more testing on the hyperparameter tuning, more careful model selection, or longer training could yield much better results.

References

[1] AMAZON WEB SERVICES, INC. AWS IoT Device Defender. https://aws.amazon.com/cn/iot-device-defender/, 2025. Accessed: 2025-05-15.

[2] ASHRAF, J., RAZA, G. M., KIM, B.-S., WAHID, A., AND KIM, H.-Y. Making a real-time iot network intrusion-detection system (inids) using a realistic bot–iot dataset with multiple machine-learning classifiers. *Applied Sciences 15*, 4 (2025), 2043.

[3] ELNADY, N., ADEL, A., AND BADAWY, W. Intrusion detection for iot networks using svm algorithm. In *Proceedings of the 34th International Conference on Computer Theory and Applications (ICCTA)* (Alexandria, Egypt, Dec. 2024), IEEE.

[4] HAMILTON, W. L., YING, R., AND LESKOVEC, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems* (2017), vol. 30. Also available as arXiv:1706.02216 [cs.SI].

[5] 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 *Proceedings of the 2022 IEEE/IFIP Network Operations and Management Symposium (NOMS '22)* (2022). Also available as arXiv:2103.16329 [cs.NI].

[6] LUAY, M., LAYEGHY, S., HOSSEININOORBIN, S., SARHAN, M., MOUSTAFA, N., AND PORTMANN, M. Temporal analysis of netflow datasets for network intrusion detection systems, 2025. arXiv preprint arXiv:2503.04404.

[7] MOUSTAFA, N. A new distributed architecture for evaluating ai-based security systems at the edge: Network TON_IoT datasets. *Sustainable Cities and Society 68* (2021), 102994.

[8] ROSSI, E., CHAMBERLAIN, B., FRASCA, F., EYNARD, D., MONTI, F., AND BRONSTEIN, M. Temporal Graph Networks for Deep Learning on Dynamic Graphs. *arXiv preprint arXiv:2006.10637* (2020). v3, last revised 9 October 2020.

[9] SARHAN, M., LAYEGHY, S., MOUSTAFA, N., AND PORTMANN, M. Netflow datasets for machine learning-based network intrusion detection systems. In *Big Data Technologies and Applications (BDTA 2020, WiCON 2020)*, vol. 371 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer, 2021, pp. 117–135.