An Intrusion Detection System using Graph Neural Networks

Tsvetomir Hristov

Delft University of Technology

An Intrusion Detection System using Graph Neural Networks

by

Tsvetomir Hristov

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 28, 2025 at 11:00.

Supervisor:	Prof. Dr. G. Smaragdakis
Group:	Cybersecurity group
Faculty:	Faculty of Electrical Engineering, Mathematics, and Computer Science
Student number:	4834119
Project duration:	November, 2024 – August, 2025
Graduation Committee:	Prof. Dr. G. Smaragdakis (TU Delft)
	Dr. H. Griffioen (TU Delft)
	Dr. M. Khosla (TU Delft)
	Manos Leontaris (Fox-IT)
Company Supervisors:	Annelies Heek (Fox-IT)
	Manos Leontaris (Fox-IT)

Cover: Generated by Sora
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TUDelft

Preface

I was wondering how I should start my preface, so I will start this with a mere "Thank you", to the reader. The pages that you will go through in the next hour or so will resemble a 9-month long period of hard work, dedication, and fun. Hope you enjoy and I hope that this thesis fuels your curiosity and interest into AI-powered cybersecurity.

To the reader

If there is anything I would like you to get from this thesis, it is these two things:

“A problem well-stated is a problem half-solved”

Working on this project for 9 months was challenging, but fulfilling. It all started one late night, when I was a Security Operations Center (SOC) analyst, being angry at the amount of alerts that we have to handle. By then I knew what the problem was, I have been tackling it for the past couple of weeks every shift, and then I decided that there has to be a solution. Next thing I know, I threw myself head first, with no knowledge in AI or GNNs, trying to figure out how to tackle the problem. The first month was hard, it was the first time I am installing `PyTorch`, the second time I am working with `Pandas`. Getting used to Python was a whole challenge by itself, but nonetheless, with a clear goal, it always seemed like the path lights up in front of you one step at a time, so just take the leap and see where it gets you! Sure, you will get lost, you will be frustrated, but, at the end of the day, "Just Keep Swimming" [1].

“Know your data”

In one of the many instances when I was lost and had no idea what to do, I went to a professor, who is from the DSAIT (Data Science and Artificial Intelligence Technology) for ideas. It wasn’t a long meeting, however, there was only one question from her which I wrote down “Do you know what’s happening in your data"? From then on, every time the AI was doing something shady, my first step was to see whether we are looking at the same data. I am sorry if this is obvious for people with AI background, but for me, a software engineer, this was a major revalation.

Now, this is it with the preface, feel free to skip the rest if you’re not family or friends. Enjoy reading!

Family & Friends

I would like to apologize, but since not everyone can read English in my family, the next section will be in Bulgarian.

Мамо, тате, како, Ванко, искам да Ви благодаря с цялото си сърце за усилията, подкрепата и доверието. Радвам се, че през цялото си обучение (22 години) винаги знаех, че моето семейство ще е зад мен, ще ме подкрепя, и ще бъде моята опора във всяко начинание. Вие ми дадохте най-важният подарък в моят живот: правилно обучение (и вкъщи, и в училище)!

I would like to thank all my friends throughout the years for the continuous support and sacrifices. Every single one of you made my years in university a little more bearable and I am happy I got to share those years with you!

To my girlfriend, I thank you for all the love, support, and dedication. Every day, when I was going home, frustrated for my thesis, that the results aren’t as expected, or the method does not work, I was at least happy that I can get home and share my feelings with you! I am happy, that, because of you, I was able to find strength every day to get up and tackle the most challenging project I have ever worked on.

[1]https://www.youtube.com/watch?v=y9FGsJ3PYVw

Mentors

Lastly, I would like to thank my mentors for this research. George, I am extremely grateful to be your student: every meeting I would go in lost and nervous, and would come out with brand new ideas and paths to explore. I admire that you have dedicated your life to research and I am hoping that this research has met your expectations. I would like to thank you for the opportunity, the guidance, and all the resources you have provided, as the thesis would not have been possible without your support!

For my mentors at Fox-IT (Annelies Heek and Manos Leontaris), thank you for your support and understanding! Alongside you, I fought the greatest “battle" in my professional career: the battle with the NCC Legal department. And, even though we lost and this has not been mentioned even once in this whole thesis, I am kind of satisfied with how this whole research project ended! I am happy to have such understanding managers, who always had my back and were ready to support me in everything!

Tsvetomir Hristov
Delft, August 2025

Abstract

Cybersecurity attacks are increasingly sophisticated, while traditional, rule-based intrusion detection systems (IDS) remain prone to high false alert rates. This research explores temporal graph learning for network intrusion detection, introducing a framework that combines temporal graph construction with Graph Attention Networks and recurrent modeling (GATv2 + LSTM). We evaluate on the LANL [19] authentication logs and Zeek logs from the University of West Florida (UWF) [6, 5, 10].

On LANL, our models (Try1/Try2) outperform state-of-the-art baselines for temporal link prediction, achieving high precision and robustness: Accuracy ≈ 0.994, F1 ≈ 0.993, AUC ≈ 0.993–0.998, AP ≈ 0.999. On Zeek data, edge prediction is sensitive to how malicious activity is distributed over time: a simple "Day" shuffling that preserves the temporal structure while also spreading the clusters of attack activity, yields large gains (e.g., Accuracy ≈ 0.969, AUC ≈ 0.996, F1 ≈ 0.959, AP ≈ 0.995), whereas random shuffling harms temporal dependencies and performance.

Extending to edge classification (benign vs. malicious) reveals a key limitation: despite high accuracy, AUC and AP remain low due to a tendency to label nearly all edges as benign under class imbalance and temporal clustering, producing many false negatives. We test mitigation strategies (dropout, alternative loss formulations with confidence weighting), which provide a small increase in stability but do not fully resolve the issue.

With our results, we find that the proposed temporal graph method is a strong fit for anomaly detection via edge prediction: robust across datasets, resilient to imbalance, and practically applicable. In contrast, edge classification currently lacks reliability for production without improved data balancing, graph construction, and training.

Contents

List of Figures

1 Introduction

In today's hyperconnected world, cybersecurity has emerged as one of the most pressing concerns across all sectors of society. As individuals, businesses, and governments increasingly rely on digital platforms for communication, commerce, healthcare, and infrastructure, the attack surface for cybercriminals has grown dramatically. The consequences of these attacks have evolved from mere data loss or financial damage to tangible threats to human safety. A stark example came in 2020, when a ransomware attack on a hospital in Germany reportedly led to the first known fatality linked to a cyberattack [17]. This tragic incident highlighted the potentially life-threatening impact of cyber attacks on critical systems.

The financial toll of cybercrime is equally staggering. Global losses from cyberattacks are projected to reach trillions of dollars annually [29], affecting organizations of every size. Beyond monetary damage, successful breaches erode public trust, disrupt essential services, and compromise sensitive information. Despite growing awareness and increased investment in cybersecurity solutions, many organizations continue to struggle with timely threat detection and effective response.

One of the major shortcomings of current cybersecurity systems is their reliance on static, rule-based approaches that are unable to keep pace with the rapidly evolving tactics of modern threat actors. Many intrusion detection systems (IDS) flood security analysts with thousands of alerts daily, the majority of which turn out to be false positives. This phenomenon, known as alert fatigue [38], is still overlooked in the industry [4], even though it has been studied extensively in research [16].

Although the cybersecurity market has grown extensively [29], the way organizations do cybersecurity has hardly changed since the last decade. The techniques which are considered "novel": canaries, Identity Access Management (IAM), Machine Learning (ML), have already been discussed in the research space for more than 10 years. The first proposal of caray tokens [41] is already 11 years old, the first federal IAM adoption plan for the USA is from 2009 [15], and machine learning has been leaving and rejoining the picture ever since the 1990s [34, 42].

Complementing the challenge is the use of artificial intelligence and machine learning by cybercriminals themselves [28]. These technologies enable adversaries to launch more sophisticated, adaptive, and targeted attacks. Ironically, while attackers are embracing AI to improve their effectiveness, many defenders have yet to fully integrate AI into their security strategies. There is a clear and urgent need to shift from reactive, manual processes to intelligent, adaptive systems that can identify threats with greater accuracy and speed [24].

In this context, AI-driven intrusion detection systems represent a promising frontier in cybersecurity [24]. By leveraging machine learning algorithms to detect anomalies and patterns indicative of malicious behavior, these systems offer the potential to drastically reduce false positives, enhance threat detection, and automate response mechanisms. Moreover, AI can continuously learn and adapt to new threats, making defenses more resilient over time [39].

1.1. Problem statement

Cybersecurity is a critical concern in today's digital world, where increasing reliance on online systems has expanded the attack surface for cybercriminals. Cyberattacks now pose not just financial risks [29], but also threats to human life, as illustrated by a fatal ransomware attack on a German hospital in 2020 [17]. Despite growing awareness and investment, current security systems often rely on outdated, rule-based methods that are overwhelmed by false alerts [40], contributing to alert fatigue and slow response times.

While attackers are increasingly using AI and machine learning (ML) to enhance their methods [28], many organizations lag behind in adopting these technologies for defense [24]. There is an urgent need for intelligent, adaptive cybersecurity solutions. AI-driven intrusion detection systems (IDS) offer a promising path forward by improving threat detection accuracy, reducing false positives, and enabling automated responses [39].

1.2. Research questions

This research aims to test potential applications of AI in effort to improve current network intrusion detection and prevention systems. In order to solve this question, the following 3 subquestions will be answered in this paper.

Q1: Can AI detection be improved

Notable research has already been conducted on the use of Graph Neural Networks (GNNs) to improve security monitoring. Our research investigates further improvements which could be implemented to advance existing state-of-the-art models.

Q2: Can graph representation improve the detection

There exists a gap in research on how graphs should be constructed from event data. Our research aims to investigate whether changes in the graph construction process could improve the detection of malicious events.

Q3: Can AI detection be used alongside Zeek

Although Graph Neural Networks have shown huge potential for network intrusion detection, they are yet to be used to analyze Zeek logs.

1.3. Contributions

The key contributions of this research are as follows.

- 3 PyTorch Lightning Modules have been implemented, allowing for easy reproducibility [1]
- All PyTorch Lightning DataModules and Models follow a modular design, allowing for quick and easy adjustment for further research [2]
- All models have been meticulously tested with multiple different hyperparameters, loss functions, metrics, and datasets
- 4 datasets have been implemented as a PyTorch Lightning DataModule, allowing for an automated downloading and loading of the data
- Our results show that the model is extremely accurate when tackling edge prediction (and anomaly detection)
- Results show that our model lacks robustness when tackling edge classification, however, a nice classifier can be achieved if the model is well trained and tuned

[1]The code is publicly available at `https://github.com/Elkozel/MSc-Thesis`

[2]For further information on how to reuse the models, please see the project on GitHub or contact the researchers directly

1.4. Organization

This thesis is organized as follows: Chapter 2 provides background on intrusion detection systems, network monitoring, temporal graphs, and graph-based learning techniques. Chapter 3 reviews related work in the field of AI-driven intrusion detection. Chapter 4 describes the methodology and model architecture used in this study.It also outlines the datasets and data processing pipeline. Chapter 5 details the implementation of the models. Chapter 6 presents experimental results and performance metrics. Chapter 7 discusses findings, limitations, and potential for real-world deployment. Chapter 8 concludes with a summary and calls for future work.

2 Background

The section will introduce key topics for this research, such as network monitoring, Intrusion Detection Systems (IDS), and lateral movement. Since the method uses Graph Neural Networks (GNNs), the core mechanisms for graph storage and convolution are also explained.

2.1. Network Security

This section aims to introduce vital network security background, critical for the understanding of this work.

Network Monitoring

Network monitoring is the practice of systematic collection and analysis of all traffic, passing trough a computer network. This practice can serve various purposes, including enhancing security, controlling access, or merely logging traffic. The process can be boiled down to three steps.

Firstly, the network must be configured to permit certain devices to receive a copy of all traffic that is routed through it. Typically, this configuration occurs on networking devices, where designated ports are configured as “monitoring" ports, receiving a copy of all traffic which the device manages. The duplicated data is then sent to specialized devices called “sensors". These sensors are running tools, which “capture" the network traffic, analyze it, and store metadata that summarizes the most critical information. Finally, the analysis generates alerts that are logged alongside the general metadata logs.

Intrusion Detection and Prevention Systems

Intrusion Detection and Prevention Systems (IDPSs) represent a wide range of varied systems that work together to form the backbone/core mechanism of modern cyber security [27]. These systems play a crucial role in supervision, identification and prevention of possible malicious activities within a computer network. The functionality of these systems extends from the extraction of valuable data from computers or the network, the efficient storage of that data, its analysis, and the generation of alerts based on the findings.

The storage options could vary greatly, depending on the system used to capture and analyze the traffic. Systems like Suricata [33] focus on real-time inspection and generate simple, structured logs, primarily highlighting alerts and basic protocol information. It’s effective for detection but limited in historical depth and context. Zeek [34], by contrast, provides a richer view of network activity by logging detailed metadata across many protocols. Instead of just flagging events, it records connection-level summaries and application-layer transactions, making it better suited for threat hunting and forensic analysis. Some systems can even be configured to store full packet captures (PCAPs), preserving complete traffic data for a more in-depth review, although this is resource-demanding and typically designated for specific use cases.

Early IDPSs were dependent on human-crafted rules to identify malicious activities. One of the most widely used signature-based IDS is Suricata [33]. With the emergence of machine learning and the recent surge in artificial intelligence, the adoption of machine learning-based detection has significantly increased. The primary tool used for machine learning is Zeek (previously named Bro[34]). These two tools show an important tradeoff in IDS and IPS. The inherently human aspect of rule-based systems facilitates the creation of explainable and easily interpretable rules, which can be finely tuned for specific malicious incidents. Consequently, signature-based systems tend to exhibit higher accuracy; however, these rules have poor performance on previously unseen malicious activities. Conversely, machine learning systems operate without the need for predefined rules; they focus on identifying behavioral anomalies that may indicate malicious activity. This approach often results in a higher false-positive rate and less explainable detection. The increased volume of alerts, coupled with a significant number of false positives, contributes to alert fatigue [38], leading to serious consequences [4, 40].

Over the years, both industry and academia have tried to combine the two methods and eliminate their shortcomings. Automated rule writing (combining the readability of human-written rules and rule generation based on anomalies from behavioral-based systems) has been heavily researched, however, these rules still require manual checks and are not meant to be fully autonomous [7].

Overall, this system allows for the effective capture, analysis, and storage of traffic, while also allowing additional tools to use the metadata, enhancing their functionality without the need of changing the underlying infrastructure.

Zeek

Given that our research primarily focuses on Zeek logs, we want to provide a deeper explanation to how they are generated and logged. Zeek operates on an event-driven architecture where network traffic is analyzed and broken into a stream of events, such as `connection_established`, `dns_request`, or `http_reply`. These events are queued in internal event queues, which are processed by scripts, written in Zeek's scripting language. This design enables users to create event handlers that can manage these events in a flexible manner by either logging them, triggering alerts, or maintaining state across connections. For example, the default `conn.zeek` script handles connection-related events and writes structured session data to the `conn.log` log file. As we can see in Listing 1, in addition to basic connection information: time, duration, source, and destination, the system also records information regarding the size, possible service, history, and other usefull data. This capability enables the system's functionality to be enhanced by external systems, which can use this data for their own purposes. Zeek also allows for the addition of more scripts, enabling the capture, decoding, and storage of more complex or proprietary protocols.

Lateral Movement

Lateral movement is a fundamental concept in cybersecurity, focusing on the actions taken by attackers after an initial breach. It involves techniques used to navigate a network in order to obtain broader access, elevate privileges, and achieve objectives such as data exfiltration or system disruption.

Given there is often a gap between the computers/users attackers initially gain access and the computers/users they target, lateral movement is often used to bridge the gap, allowing attackers to gain access to critical computers/users.

Lateral movement often starts with reconnaissance to understand the network, identify valuable targets, and find ways to escalate privileges or access sensitive information. A notable aspect of this is its use of legitimate administrative tools and protocols, which helps attackers avoid detection by traditional security systems.

The ability to move laterally increases the impact of a security breach, allowing attackers to compromise multiple systems and access more assets. Thus, understanding and detecting lateral movement is crucial for modern threat detection and incident response.

2.2. Graph Neural Networks (GNNs)

This section contains vital context for the data and AI methods used in our approach.

```
{
  "ts": 1591367999.305988,
  "uid": "CMdzit1AMNsmfAIiQc",
  "id.orig_h": "192.168.4.76",
  "id.orig_p": 36844,
  "id.resp_h": "192.168.4.1",
  "id.resp_p": 53,
  "proto": "udp",
  "service": "dns",
  "duration": 0.06685185432434082,
  "orig_bytes": 62,
  "resp_bytes": 141,
  "conn_state": "SF",
  "missed_bytes": 0,
  "history": "Dd",
  "orig_pkts": 2,
  "orig_ip_bytes": 118,
  "resp_pkts": 2,
  "resp_ip_bytes": 197,
  "ip_proto": 17
}
```

Listing 1: Example of an event in the zeek connection logs file

Temporal Graph Data

Graph data refers to any data, represented in a graph structure. A graph is typically defined as $G = (V, E)$, where V is a set of nodes (or vertices) and $E \subseteq V \times V$ is a set of edges connecting the nodes. Each node $v \in V$ may be associated with a feature vector $\mathbf{x}_v$, and edges may optionally carry weights or additional features.

Unlike static graphs, which represent a fixed set of nodes and edges, temporal graphs allow for the topology, edge attributes, or node features to change over time. This is especially effective for encoding events, as each event can be represented as a change in the graph. This dynamic nature makes temporal graphs highly relevant for modeling complex real-world systems such as social networks, communication logs, financial transactions, and biological processes.

A temporal graph can be represented as a sequence of graph snapshots $\{G_t = (V_t, E_t)\}_{t=1}^{T}$, or as a set of temporal edges $E = \{(u, v, t)\}$, where each edge is annotated with a timestamp t indicating when the event occurred.

Temporal graphs can be categorized based on the way time is presented:

- **Discrete-time graphs**: The graph is represented at fixed time intervals (e.g. snapshots).
- **Continuous-time graphs**: Interactions are represented as a stream of events, each containing a timestamp.

The added temporal dimention allows for a new set of problems to be tackled by Temporal Graph Neural Networks (TGGNs), such as:

- **Temporal link prediction**: Predicting future interactions based on historical data.
- **Temporal node classification**: Assigning labels to nodes based on their past behavior.
- **Temporal community detection**: Discovering groups of nodes that persist or evolve over time.

Thus, temporal graphs present a simple solution, which can encapsulate complex time-sensitive data. Their usage is essential for link prediction and classification.

Graph Convolution

Graph convolution is a fundamental operation in Graph Neural Networks (GNNs). It extends the concept of convolution from traditional Euclidean domains, such as images and sequences, to non-Euclidean data structures represented as graphs. This advancement enables the processing of complex relational data where the underlying structure is a list of nodes and edges (aka a graph). This allows for GNNs to learn meaningful representations of nodes (or entire graphs) by aggregating and transforming information from each node's local neighborhood.

Graph convolutional operations are often categorized into four main approaches: spectral, spatial, attention, and sampling based methods [14, 9]. Spectral methods define graph convolution in the frequency domain, spatial methods act directly in the node domain, attention methods build on that by learning the importance of each neighbor, and sampling methods sample nodes randomly to perform convolution.

In these approaches, the representation of a node is updated by applying a function to the features of its neighbors. Since this paper explores only spatial and attention methods, the update rule only for these methods will be described. Formally, the update rule for a node can be expressed as:

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

where $e_{j,i}$ denotes the edge weight from the source node j to the target node i (default: 1.0), $\mathcal{N}(i)$ denotes the neighbors of node i, x_i denotes the features of node i, and Θ represents the trainable feature weights.

Graph convolution enables powerful modeling of data in domains such as social networks, molecular chemistry, recommendation systems, and knowledge graphs. By capturing both local structure and feature information, GNNs have demonstrated state-of-the-art performance in tasks such as node classification, link prediction, and graph classification.

In conclusion, graph convolution represents a vital method that bridges deep learning and graph theory. Its capacity to generalize convolutional operations to arbitrary graph structures has opened new opportunities for research and application in machine learning on graph data.

Evaluation Metrics

Evaluating the performance of machine learning models requires reliable and interpretable metrics. Different metrics capture different aspects of classification quality, such as overall correctness, the balance between false positives and false negatives, or the ability to rank positive samples above negative ones. In this section, we briefly introduce four commonly used evaluation metrics: Accuracy, F1 score, Area Under the ROC Curve (AUC), and Average Precision (AP).

Accuracy. Accuracy is one of the most widely used evaluation metrics in classification tasks. It is defined as the proportion of correctly classified instances among all instances in the dataset.

$$\text{Accuracy} = \frac{1}{N} \sum_{i}^{N} 1(y_i = \hat{y}_i)$$

While accuracy provides an intuitive measure of overall performance, it can be misleading in scenarios with imbalanced datasets, where a classifier may achieve high accuracy simply by predicting the majority class.

F1 Score. The F1 score is a metric that combines precision and recall into a single measure by computing their harmonic mean. Precision quantifies the proportion of predicted positive instances that are truly positive, while recall measures the proportion of actual positives that are correctly identified.

$$F_1 = 2\frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

The F1 score is particularly useful when both false positives and false negatives carry significant cost, as it balances the trade-off between precision and recall.

AUC. The Area Under the Receiver Operating Characteristic Curve (AUC-ROC) evaluates the performance of a classifier across different decision thresholds. The ROC curve plots the true positive rate against the false positive rate, and the AUC measures the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative one. AUC values range from 0.5, indicating random guessing, to 1.0, indicating perfect classification. This makes AUC a robust metric for assessing ranking quality regardless of threshold selection.

Average Precision (AP). The Average Precision (AP) score summarizes the precision–recall curve by computing the weighted mean of precision values at different levels of recall.

$$AP = \sum_n (R_n - R_{n-1}) P_n$$

Unlike AUC, which is based on the ROC curve, AP is more informative in highly imbalanced datasets, where the distribution of positive and negative samples is skewed. By considering both precision and recall across multiple thresholds, AP provides a comprehensive measure of the model's ability to detect positive instances under challenging conditions.

3
Related work

Intrusion Detection Systems (IDS) are a critical component of cybersecurity infrastructure, and recent advances in deep learning have led to more intelligent, adaptive models. A particularly promising direction is the use of Graph Neural Networks (GNNs), which model entities and their interactions in the network as structured graphs, capturing rich relational patterns. This is especially relevant in enterprise networks, where relationships between users, hosts, and services often reveal latent attack behaviors.

Several recent surveys [2, 44] show how the IDS systems have evolved and the use representing the raw data into a graph structure helps improve the overall score. The surveys show the wide variety of methods which can be used for intrusion detection, such as graph convolutional networks (GCN), Graph Attention Networks (GAT), and GraphSAGE, each of which contributes different advantages. The survey [44] focuses more deeply into GNN-based IDS approaches, using both static and dynamic graphs.

Multiple other methods has been used to tackle the problem as well. One notable mention is using autoencoders for unsupervised learning annomaly detection [3, 13, 21]. These methods have proved to be very reliable, with `EdgeTorrent` [21] having a perfect accuracy, and do not require labels for their training, which provides a huge advantage. Nonetheless, although annomaly detection is highly effective, our goal is to provide a method which can accurately and robustly classify attacks, which autoencoders are not suited for.

Several papers [20, 36] have tackled the problem by boiling it down to link prediction. Euler [20] has taken the more classical approach, encoding the raw authentication events as a homogeneous discrete-time graph. The framework also allows for modular switching of the components that the algorithm uses (the GNN, the RNN) and aims to find the best combination for the task at hand. On the other side, HetGLM [36] aims to expand the approach by using heterogeneous graphs, encapsulating different node and edge types, proving to be more precise and effective.

Although we consider the advancements in the Euler framework [20] a major advancement in the usage of AI for lateral movement detection, we believe that the most important advancement of the framework is its distributed nature. The framework is designed to be distributed over multiple hosts, allowing the problem to be scaled horizontally. This shows that the framework can be used for real-time intrusion detection.

Boiling down the problem to link prediction is very usefull, as it also allows the use of techniques from various disciplines. For instance, in finance, detection of fraudulent transactions with the help of neural networks (including GNNs) has been extensively researched [8, 43]. This highlights areas where the benefits of GNNs for cybersecurity are understudied, one notable example is the implementation of Competitive Graph Neural Networks (CGNNs) for anomaly detection, which has demonstrated to be robust and reliable in financial fraud detection [43].

Looking at the datasets used, we focus on two recent datasets used in the cybersecurity research sphere.

The first dataset is LANL [19], being used by both Euler [20] and HetGLM [36]. Both methods encode data from the LANL dataset as a graph and train a model to predict edges, thus detecting possible anomalies.

The other group of datasets used are the three datasets produced by the University of West Florida [6, 5, 10]. The datasets have been used by two ML-based models: one classifying edges using Multi-class Support Vector Machines [23], the other detecting anomalies using a GCN-Autoencoder [37].

In summary, although the area has recieved a lot of research over the years, the large amount of possible methods which could be used show that this area is still not researched well enough and still requires research. Our research aims to fill one of the gaps and make it feasable for IDS systems to be integrated within a live SOC environment and improve their performance by exploring ways to encode raw data into grahps.

Paper	**Data structure**	**Model structure**	**Purpose**
Los Alamos National Laboratory (LANL) Dataset			
Euler [20]	Homogeneous Discrete Temporal Graph	GNN → RNN → Decoder	Link prediction
HetGLM [36]	Heterogeneous Discrete Temporal Graph	FCN/GNN → Autoencoder	Link prediction
University of West Florida Datasets			
MC-SVM [23]	Vector	Multi-Class Support Vector Machines	Edge classification
GCN-Autoencoder [37]	Homogeneous Discrete Temporal Graph	GCN →Autoencoder	Link prediction

Table 3.1: Related models

4 Methodology

This chapter outlines the methodology used to design, implement, and evaluate a graph-based machine learning framework for detecting and classifying malicious network activity. The primary goal is to model the dynamic behavior of computer network events using temporal graphs and apply a combination of Graph Neural Networks (GNNs) and Recurrent Neural Networks (RNNs) to forecast and classify suspicious behavior, as shown in Figure 4.1. The system is implemented using the PyTorch Lightning framework, which provides a well-organized and modular environment for experimentation and testing.

4.1. Model architecture

The architecture of the proposed model follows a layered design, composed of a GNN encoder, a temporal RNN layer, and a final decoder for link prediction and classification. The graph encoder processes each network snapshot independently, extracting latent node features that capture local structural information such as communication patterns and service usage. Different architectures for link prediction have been researched and compared on numerous metrics [35], showing that further improvements could be done on this layered approach, however, we chose to stick with the “basic" GNN layer, followed by an RNN layer and then a decoder to decode the features obtained by the RNN. This is done with the idea that a more "basic" approach will be more suited for basic testing and improvements, as, for example, the complexity of DGNNs could increase the workload.

Additionally, the architecture of the model and the approach followed closely the approach from Euler [20], as we hope that our approach will also be able to be run in the distributed nature that Euler was run into, enjoying the same benefits that the Euler framework has, which are going to be vital for real-time or scalable intrusion detection in production environments. Unfortunately, as we aim to explore how the method can be improved to have higher accuracy and be more robust, the method has not been adapted to be run in a distributed environment. However, we believe that our implementation could easily be adapted to run in a distributed environment, due to the use of the Pytorch Lightning library [11].

Inspired by the distributed Euler framework for temporal link prediction, our design also considers potential for horizontal scaling. While this thesis does not implement a distributed variant, the

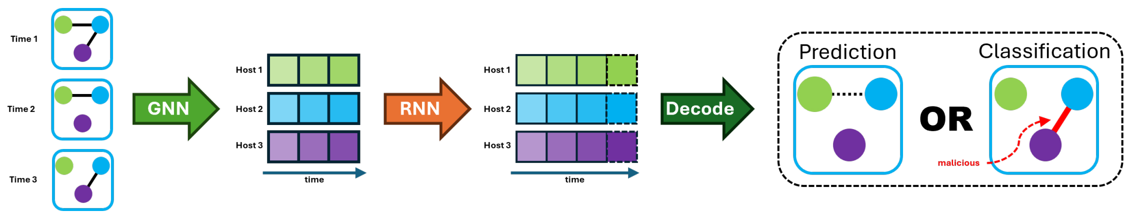


Figure 4.1: Explanation of the model’s workings

modularity of the architecture ensures compatibility with future extensions for real-time or scalable intrusion detection in production environments.

To model temporal dependencies across network states, the node embeddings from the GNN are passed through a Recurrent Neural Network. Initially, a GRU (Gated Recurrent Unit) was used to predict the embeddings of future time windows. However, subsequent experiments demonstrated better performance with LSTM (Long Short-Term Memory) networks, particularly in capturing long-range temporal dependencies and reducing prediction noise. The RNN receives a sliding window of past graph embeddings and produces predictions for the next time step, effectively enabling temporal forecasting of network activity.

The decoder component consists of two distinct heads: a bilinear layer for binary link prediction and a multi-layer perceptron (MLP) for edge classification. The bilinear decoder estimates the probability of an edge (i.e., network connection) existing between two nodes in the predicted graph, while the MLP determines whether that edge is malicious. This dual-task setup enables the model to not only anticipate connections but also assess their threat level. Additionally, the edge predictions are also given as input to the edge classifier to make use of the prediction in the classification stage.

Model training is supervised and uses a combination of positive and negative samples generated from each temporal graph snapshot. For each time step, a batch of graphs is processed by the GNN, then encoded as a time series matrix and fed into the RNN. Predictions from the decoder are compared against ground truth labels, and losses from link prediction and classification are computed using binary cross-entropy. These are combined using a tunable weighting parameter α to balance both objectives. Multiple metrics, such as Accuracy, Average Precision, F1, and AUROC, are computed per batch using TorchMetrics, and the Lightning training framework handles gradient updates, validation steps, and logging.

The model was developed through an iterative process involving three prototypes: `Try0`, a baseline GCN model with no temporal layer; `Try1`, which added a GRU for time modeling; and `Try2`, the final version using GATv2Conv, LSTM, and improved normalization and regularization techniques. Each version was tested using the same datasets, allowing for comparative performance assessment and progressive refinement.

In summary, this methodology presents a structured pipeline that combines structural and temporal modeling of network events. By leveraging the strengths of GNNs and RNNs within a unified framework, the system aims to enhance detection capabilities for sophisticated threats such as lateral movement and advanced persistent attacks.

4.2. Datasets

For our research, two data sources are used. The first source is the Los Angels National Laboratory, which has produced three datasets. Our research has used the "Comprehensive, Multi-Source Cyber-Security Events" dataset [18], containing data obtained over 58 days of networking events, authentications, DNS requests, and processes. The data can be labeled using the list of known activities performed by the red team. The second source is the department of Computer Science in the University of West Florida. The department has produced four datasets: UWF-ZeekData22, UWF-ZeekData22Fall, UWF-ZeekData24, UWF-ZeekData24Fall, each containing labeled zeek connection logs from different collection periods [6]. In essence, this section introduces the data with which we worked with in our research, how did we create a graph from that data and why some decisions were made.

Each separate dataset is represented as a *LightningDataModule* class from the Pytorch Lightning library [11]. This provides for an easy preparation and integration of the dataset with the models we have produced in our research. It is important to note that the datasets have a specific naming convention within our implementation. Every *LightningDataModule* is named after the dataset it serves with added suffixes. The 'L' suffix indicates that the dataset will be stored locally, 'H' indicates that the dataset produces heterogeneous data, and 'R' indicates that the data is stored remote.

All *LightningDataModule* classes in our repository follow a simple pipeline shown in Figure 4.2. The first stage of the pipeline is data download, where the code attempts to download the needed files from the original website. The function also has a "guard" statement to skip downloading a file if the file is

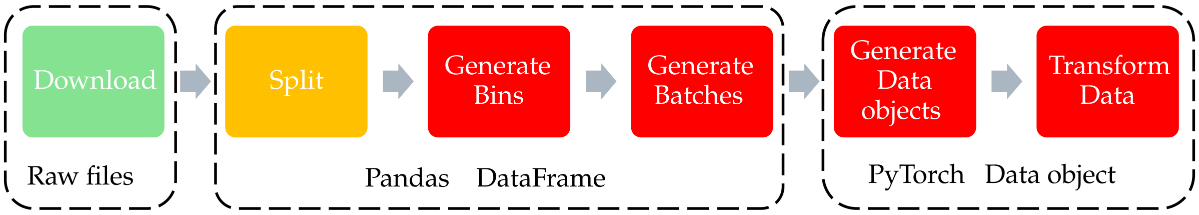


Figure 4.2: Pipeline used for data loading in *LightningDataModule* classes

already present. This is the only stage which is called in the *preparation* stage (indicated in the image in green)

The next stage generates the splits for the data. Since the UWF-ZeekData22Zeek dataset consists of data gathered from multiple sessions, this creates “holes" (hours, where 0 events are stored). This is not particularly effective, as it could “clutter" the dataset with hours of empty data being fed to the neural networks. Thus, to avoid this time waste, our splitting algorithm is run for each file. The algorithm starts by extracting the time range of the file (the min and max time). Once the time range is known, the bin ranges are calculated by flooring them to the closest valid bin, as shown in Equation (4.1). For example, if the event is at time 34 seconds and the bin size is 20 seconds, the event will be assigned to bin 20. Once the minimum and maximum bins are determined, the amount of bins and batches which can be generated from this file are calculated, as shown in Equation (4.3).

$$B_{start} = \left\lfloor \frac{T_{start}}{S_{bin}} \right\rfloor * S_{bin} \tag{4.1}$$

$$B_{end} = \left\lfloor \frac{T_{end}}{S_{bin}} \right\rfloor * S_{bin} \tag{4.2}$$

$$Bin_{range} = \frac{Bin_{end} - Bin_{start}}{Size_{bin}} \tag{4.3}$$

$$Batch_{range} = \left\lceil \frac{Bin_{range}}{Size_{batch}} \right\rceil \tag{4.4}$$

One the amount of batches from the file is known, a batch mask is generated using a split of (60%, 25%, 15%) for training, validation, and testing respectively. Once the batch mask is done, it is saved within the class for later use. For a quick reference, as simplified diagram of the whole process is presented in Section 4.2. The figure shows the process for 20 seconds per bin, 5 bins per batch, and a uniform split of batches. This is the only stage which is called in the *setup* stage (indicated in the image in yellow)

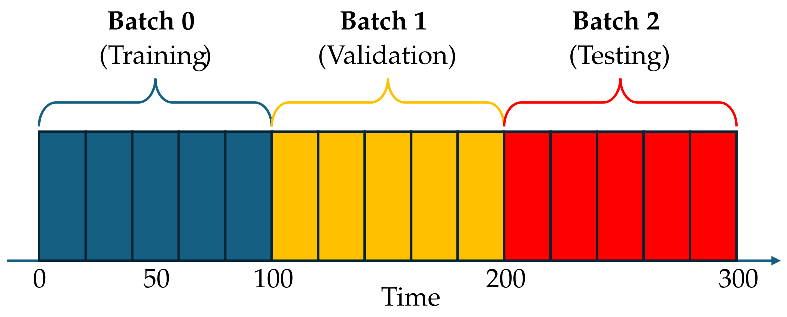


Figure 4.3: The process of generating batches

The rest of the steps are performed once the data is requested from the *DataModule*. The process starts with the *Generate Bins* stage. The bin of each event is calculated with the formula specified in Equation (4.1), after which the data is grouped into bins. Each bin is then passed to the *Generate Batches* stage, where multiple bins are batched and passed to the *Generate Data objects* stage. Each bin is first transformed into a *Data* object from the Pytorch Geometric library [12]. Each batch is then created using the batching functionallity provided by the Pytorch Geometric library.

It is important to note that the transformation of pandas DataFrames to Data objects require the generation of a host map, containing all previously seen hosts, and a servicemap, containing all previously seen unique combinations of a host and a service. This is done dynamically in the *Generate Bins* stage, where a special keymap is used to store unique values per column. Unique values are stored in an ordered set, where each unique value has a unique index assgned, based on the order in which they were included into the set. This simple mechanism is presented in Listing 2. One important note is that certain columns are not counted in the keymap, such as the time column, as it would introduce heavy performance impact without any gains. Other fields are also excluded, such as the source and destination hosts, as they need to be combined together rather than creating an individual map for each column.

```
self.keyword_map = {col: OrderedSet([]) for col in columns}

for bin in bins:
    (...)
    for col in self.keyword_map:
        self.keyword_map[col].update(bin[col].unique())
```

Listing 2: Keymap generation for each column

As a last step, once the Data objects are created and batched, all predefined transformations are applied in the *Transform Data* stage.

Most of the stages in the pipeline are implemented as generator functions, which aims to reduce the memory footprint of the *DataModule* and allow for the computation to be “distributed" along multiple requests, rather than done upfront.

4.2.1. Los Alamos National Laboratory

The Los Alamos National Laboratory has released an annonymized dataset, named “Comprehensive, Multi-Source Cyber-Security Events" [18]. For simplicity, since this is the only dataset that this research uses, this dataset will be refered to as LANL. The LANL dataset is divided into 5 files based on the type of data collected: the dataset contains authentication logs, network flows, DNS requests, processes ran on hosts, and a list of redteam activities. The events have been collected in the timerange of 58 consecutive days from 5 distinct sources within the corporate internal computer network of Los Alamos National Laboratory.

Since two of the related works [20, 36] already use only authentication logs, our research also focuses only on computer authentication logs. The authentication logs consist of 1,648,275,307 events in total for 12,425 users, and 17,684 computers.

To download the dataset, a special link from the website of the Los Alamos National Laboratory is required, which is why this dataset cannot be downloaded automatically. Furthermore, only the necessary files (auth.txt.gz and redteam.txt.gz) are downloaded by the *DataModule* class.

One notable change about this dataset is that the module only loads the first 14 days from the dataset (although this behavior can be overwritten by providing the to_time argument to the *DataModule*). This is done as the researchers from the Euler paper [20] indicated that there might be issues with the data capture after that period. Due to this, we also decided to exclude any events after day 14.

Dataset	Num. Events	First Event	Length	Unique Hosts
LANL	1648275307	-	58 days	17684
LANL (first 14 days)	239471459	-	14 days	14658
UWF-ZeekData22	18562468	2021-12-17 07:00	65 days	392
UWF-ZeekData22Fall	700340	2021-12-17 13:00	310 days	122
UWF-ZeekData24	1916757	2024-02-28 04:00	252 days	357

Table 4.1: General dataset statistics

4.2.2. University of West Florida

The university of West Florida has published 3 datasets, obtained from multiple student excises performed within a sandboxed environment [6, 5, 10].

Each dataset is divided into multiple parquet files, based on the time the events were captured. The datasets also have differences in size, length, and amount of hosts involved, as shown in Table 4.1.

To further visualize our data violin plots are used in Figure 4.4 to show the distribution of malicious and benign edges across all datasets.

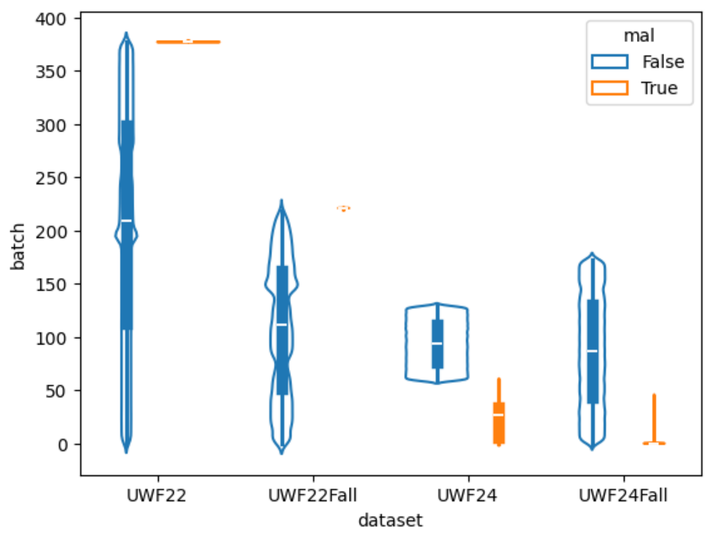


Figure 4.4: UWF datasets malicious edge distribution

From the figures, we can see that the datasets have a major drawback for the training of our model: the malicious and benign events are separated and a small subset of all bins contain both benign and malicious events in the 20/80 (malicious/benign) split stated by the creators of this dataset. This is a problem for the training of our model, since, as we attempt to encapsulate temporal data from previous edges, a lot of that temporal information will be extracted from edges of the same class. For example, if we sample a malicious event from the `UFW22Zeek` dataset, almost all edges included as temporal information will also be malicious. This is visualized in Figure 4.5a, where we can see that the majority of malicious edges are present in the last couple of batches. However, this does not reflect a real-world scenario, where the majority of the traffic is legitimate. To tackle this issue, we try to shuffle the dataset in a way we can obtain the desired mix.

Shuffling

Our first attempt to shuffle the data was in the form of random shuffling. We wanted to preserve some short-term temporal information, which is why instead of shuffling per event, we decided to shuffle data in bins as showin in Equation (4.5). The algorithm firstly bins each event based on the bin length specified. It also records the time difference between the bin and the time of the event.

$$\text{BinID} = \left\lfloor \frac{Time_{event}}{Size_{bin}} \right\rfloor \tag{4.5}$$

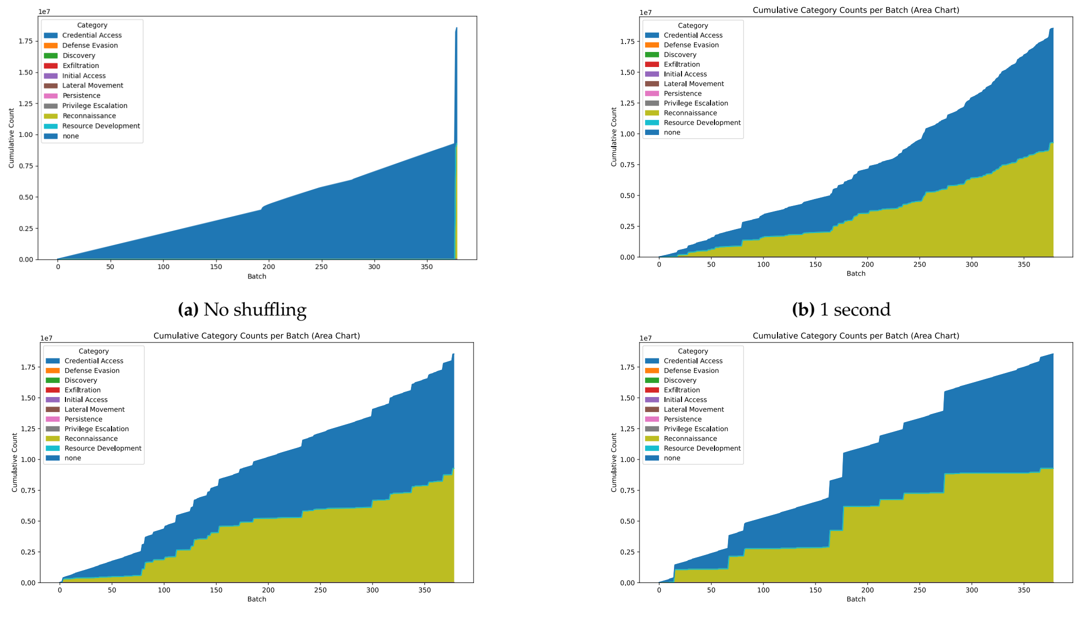
$$\text{Offset} = \text{Time} - (\text{Bin} * Size_{bin}) \tag{4.6}$$

Once the bins are constructed, each bin is randomly assigned a new id and the time of the event is derrived by the bin id and the offset as shown in Equation (4.7). This allows for the dataset to be fully reshuffled without requiring further adjustments to the tools in the loading pipeline.

$$\text{BinTime} = \text{BinID} * Size_{bin} \tag{4.7}$$

$$\text{Time}_{event} = \text{BinTime} + \text{Offset} \tag{4.8}$$

This method of shuffling proves to be very effective, as even large bin sizes can shuffle the data effectively, as shown in Figure 4.5d, however, it destroys a lot of the temporal information. This is a huge downside, as our model needs to learn and use this temporal information during training. Thus, we need another way to shuffle malicious and benign events, which does not destroy the temporal information.



(a) No shuffling

(b) 1 second

(c) 10 seconds

(d) 400 seconds

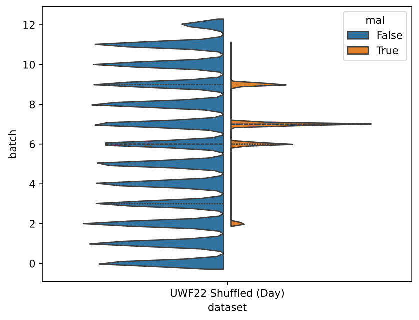
Figure 4.5: Comparison of random shuffling with different bin sizes

Our next shuffling technique is closely related to the way the dataset was created. Since the datasets have been collected across multiple days, we can merge all days together, effectively collapsing the dataset to a single day. We hope that this simulates better how a real-life network looks like, where malicious and benign connections happen in the same timeframe.

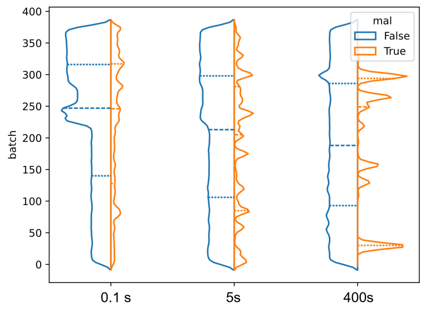
The shuffling is done in a simmilar way as the random shuffling, however the events are binned based on the day they occur. The offset of the event is also calculated in seconds elapsed since midnight.

It is important to note that this method relies on there being no temporal relationships between events across multiple days, as these relationships will effectively be lost. However, this is only possible 10 minutes around midnight, as this is the RNN window, which is why we decided to ignore this limitation.

Another limitation is that, since the events are not shuffled throughout the day, the data still has a huge variance between the edge count and incosistency as to how benign and malicious events are distributed. Figure 4.6 shows this drawback, compared to the random shuffling, where events are distributed more evenly across batches.



(a) Day shuffling



(b) Random shuffling

Figure 4.6: Comparison between "random" and "day" shuffling

Data representation

Zeek connection events are logged within the `conn.log` file represented in a JSON format, as shown in Listing 1. Each event is processed and represented as an edge within the graph. The source and destination nodes are fetched from the hostmap using the *id.orig_h* and *id.resp_h* fields respectively. A directed edge is then added with the following data from the dataset:

- conn_status
- src_port_zeek
- dest_port_zeek
- orig_bytes
- orig_pkts
- resp_bytes
- resp_pkts
- missed_bytes
- local_orig
- local_resp
- duration
- proto$*$
- service$*$
- conn_state$*$

Data marked with an asterisk ($*$) is embedded using a pre-generated mapping function, where each string value is represented as a number.

Service and hostmaps

The datasets from the University of West Florida were also further enriched. One of the processes used was an enrichment of the host information, which is handed to the neural network. Since the IPs are not annonimized, extra information obtained from the IP is added. Additionally, since the each event includes a duration, the duration is also used to create new events.

As usual, the normal pipeline creates a hostmap and a service map containing all hosts and services obseved thus far. Before these maps are included inside the Data object, each of them is enriched. The host map is enriched by adding the following information for each individual IP address:

- IP address type: v4 or v6
- Whether the IP address is public or private
- Whether the IP address is a multicast, broadcast, or unicast

It is important to note that there are a couple of shortcomings to this method. Firstly, although this process could further be improved by using outside resources, this was not implemented in our *DataModule*. Additionally, certain hosts could be represented more than once in the hostmap. For example, if Dual Stacking [30] is used, the host will be represented by both its IPv4 and IPv6 host addresses, as there is no reliable way to determine the two addresses used by the same host.

The servicemap is enriched with the following data:

- Is the port known

This enrichment is also not enriched by any outside resource, which could prove to be a big limiting factor for the neural network to properly identify similar services and learn their behavior.

Accounting for duration

Another special transformation is performed to allow for the duration of a connection to also be encoded inside the graph. To explain why this algorithm is needed, we would like to present the following simple scenario:

```
{ "ts": 16, "source": 1, "dest": 2, "duration": 1 }
{ "ts": 16, "source": 1, "dest": 3, "duration": 16 }
{ "ts": 18, "source": 2, "dest": 3, "duration": 2 }
```

In the scenario we have three events in the same time snapshot $[15, 30)$, however, we can see that one of the events “spills" trough multiple bins. To adjust for this, the algorithm expands the event by adding an extra record and creates an extra column named $conn\_status$:

```
{ "ts": 0, "source": 1, "dest": 3, "duration": 16, "conn\_status": "started" }
{ "ts": 16, "source": 1, "dest": 2, "duration": 1, "conn\_status": "closing" }
{ "ts": 16, "source": 1, "dest": 3, "duration": 16, "conn\_status": "closing" }
{ "ts": 18, "source": 2, "dest": 3, "duration": 2, "conn\_status": "closing" }
```

Listing 3 shows how “spilled" events are explanded. Firstly, the start and end times are computed for the event. After that, a range is calculated with the time range with a step equal to the bin size: $[ts_{start}, ts_{end}, size_{bin})$. This range shows the time values of all spilled events: if the range is $[1, 8, 3)$, then the resulting range will be $[1, 4, 7]$, indicating that the event should start in bin 0 with a timestamp of 1, then it will also be added to bin 3 with a timestamp of 4 and then it will appear in bin 6 with a timestamp of 7. As you might have noticed, we now have two events in bin 6, one with a timestamp of 7 and the original event at time 8. Due to this reason, our algorithm removes the last event from the range and replaces it with the original one.

```python
def expand(x):
    duration = x["duration"]
    end_ts = x["ts"]
    start_ts = float(end_ts - duration)

    # Otherwise, determine how many adjustments will need to be made
    all_ts = np.arange(start_ts, end_ts, self.bin_size)[:-1] # cut the last one
                                                #(we already have the original event)
    all_ts = np.append(all_ts, end_ts) # add the original event

    open_conn_states = ["open" for _ in range(all_ts.size - 2)]
    all_conn_states = ["started"] + open_conn_states + ["closing"]

    return pd.Series([all_ts, all_conn_states], index=['ts', 'conn_status'])
```

Listing 3: Expand function

The connection states for each event are also generated and put into the $conn\_status$ column. The first connection state for each event which "spills" is *"started"*. All other events, except the original one, get the connection state of *"open"*, while the original event has the connection state of *"closed"*.

Once the two arrays are created (one for timestamps and one for connection states), the built in *explode* function in Pandas is used, where new events will have identical data, except for their timestamp and connection status.

Using this algorithm our research aims to better represent connections, improving the performance and accuracy of the model.

5 Implementation

As our research relies heavily on the correct implementation of our proposed model, this section introduces all details, decisions and challenges to our Python implementation.

5.1. Models

For this research, multiple models, summarized in Table 5.1, are developed through an iterative process. Although our research mainly focuses on the latest iterations, we believe it is beneficial for all models developed to be introduced.

Try0

`Try0` is an experimental model, created without an RNN in order to assess how much it helps with the prediction. The model consists of 3 GCNConv [22] layers, mixed with 2 RELU activation layers in-between. Embeddings produced by the GNN are fed directly to the decoder stage, where the decoder tries to predict and classify the edges.

Try1

The `Try1` model is designed with simplicity in mind. The model is meant to be as simple as possible, but still containing all the necessary parts to properly predict and classify the edges. Similarly to `Try0`, the model has the same GNN structure, however the embeddings are then run trough a GRU layer, used to predict the embeddings in the next time window. The decoding process uses an MLP netowrk with two layers and an RELU activation function for both prediction and classification.

Try2

`Try2` is created in an iterative process, where changes to `Try1` are made and then tested. Multiple GNN layers were tested: `SimpleConv`, `GCNConv`, `GATConv`, `GATv2Conv`. Each test lasted one epoch and were assessed on the accuracy of the link prediction. Our testing showed that the `GATv2Conv` layer produced the best results. The GNN structure was also improved with the addition of Layer Normalization and support for dropout. The same iterative process also showed that an `LSTM` layer works better to encapsulate and predict the communication patterns in the data.

Lastly, a `Bilinear` and a `MLP` was tested for decoders, however, no significant differences were observed. Thus, for the decoders, we decided that a simple `Bilinear` decoder will be used for link prediction and an `MLP` for link classification. While researching on the best decoder to use, we stumbled on two ideas: a simple, predictable, decoder, which would require more “learning" from the layers above, or a complex one, which would try to capture some data and help the prediction.

The results from the edge prediction are also passed as input to the edge classification decoder, to allow for better accuracy.

Model		Layers	Trainable parameters	Estimated parameter size
Try0	GNN	3x GCNConv	6.7 K	0.027 MB
	RNN	-		
	Decoder	MLP (Prediction) Linear (Classification)		
Try1	GNN	3x GCNConv	5.2 K	0.021 MB
	RNN	1x GRU		
	Decoder	MLP (Prediction) Linear (Classification)		
Try2	GNN	3x GATv2	39.3 K	0.157 MB
	RNN	5x LSTM		
	Decoder	MLP (Prediction) Linear (Classification)		

Table 5.1: Summary table of all models

5.2. Pipeline

Each model is implemented as a `LightningModule`, provided by the Pytorch Lightning library [11]. Lightning is a wrapper around PyTorch, which does a lot of the “heavy lifting", which is usually associated with Pytorch. Firstly, once the module is wrapped, it can be easily used and integrated into any project, which uses the same library. Additionally, modules could be viewed as self-container, meaning that all the logic of the module can be enclosed within a single file, making the transfer and reuse really easy.

The structure of a `LightningModule` is shown in Listing 4. The module starts with an `__init__()` method, which defines and initializes the core components of the model architecture, such as layers, submodules, or other model-specific parameters required for the forward computation. The `forward()` method specifies the data flow through the model during inference or evaluation. It defines how input data is transformed as it passes through the model's layers to produce output. The `training_step()` method defines the operations that occur during a single iteration of the training loop. It receives a batch of data, performs the forward pass, computes the loss, and optionally logs metrics. Lastly, the `configure_optimizers()` method sets up the optimization strategy for training. It returns one or more optimizer instances that will be used by the training loop.

```python
class ModuleStructure(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def forward(self, inputs, target):
        return self.model(inputs, target)

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.model.parameters(), lr=0.1)
```

Listing 4: General structure of a Lightning Module class

The way our module works is presented with the pipeline in Figure 5.1. The pipeline outlines 6 steps used in the training loop of our model.

Firstly, a batch is loaded from the dataset. This step is performed automatically by the PyTorch Lightning

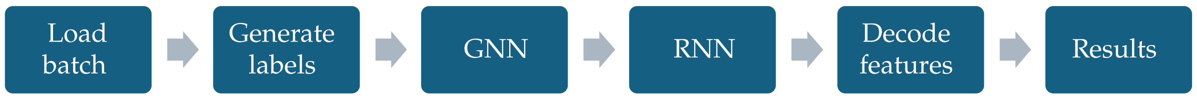


Figure 5.1: Module pipeline

`Training` class, which also calls the `training_step()` method with the obtained batch.

The method then generates edge pairs to be used during the `Decode features` stage. The edge pairs are divided into two groups: positive and negative. Positive edges are edges which should occur in the predicted graph, while negative edges are edges, which should not be present. Once these edge pairs are obtained, a label mask is also created to keep track of positive and negative edges. The label mask is used in the `Results` stage for calculation of the acuracy and loss.

The next step of the pipeline invoves running the graphs trough the GNN. Due to the way that graph objects are batched together, a full batch can be run trough the GNN, speeding up the process without impacting accuracy.

The model works on batches of i timesteps ($[g_0, g_1, ..., g_i]$), each timestep being fed trough the graph neaural network, producing i feature vectors ($[fg_0, fg_2, ..., fg_i]$). The feature vectors are then concatenated into a matrix M with size (t, h, i), where t is the amount of time steps, h represents the amount of nodes (hosts) in the connection graph, and i represents the amount of features generated per node.

Once the features are obtained, the data is put trough an RNN using a sliding window. For example, given a window of 30, the RNN will be fed all features starting from timestamp 0 till timestamp 29 ($x = [M_{0,:,:}, M_{1,:,:}, ..., M_{29,:,:}]$) with the labels extracted from timestamp 30 ($y = M_{30,:,:}$). The x values are then ran trough an RNN, with the hidden state after all data has been processed treated as a prediction y'.

Once all windows are processed, each prediction is then passed on to the `Decode features` stage. The stage also uses the edge pairs, which were created in the `Generate labels` stage. Two separate decoders are used, one for predicting whether an edge exists, another for classification of edges. Given the features, a decoder is used to generate predictions for edges in the edge pairs. This results in edge predictions and classification for each edge.

Lastly, the edge predictions and classifications are compared to the true labels in the `Results` stage.

The structure of our models (see Listing 5) introduces one new method named `run_trough_batch()`. This is done in order to make the class more readable and maintainable. As there are no differences in how batches are handled in all stages.

5.2.1. Speedup

Although the forementioned pipeline shows the way our model computes predictions, it is important to note that the implementation of the pipeline is highly inefficient, due to the large amount of for loops. The high overview of the pipeline shown in Listing 6 could be used best to describe the issue. On first glance, we can see two for loops on lines 1 and 4, which can slow down the computation. There are also multiple operations which are performed per sliding window, such as the negative edge sampling (line 8), label creation (line 15) and loss calculation (line 24).

```python
class ModuleStructure(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()

    def forward(self, inputs, target):
        pass

    def run_trough_batch(self, batch: Batch, stage: Literal['train', 'val', 'test']):
        pass

    def training_step(self, batch, batch_idx):
        pass

    def validation_step(self, batch: Batch, batch_idx):
        pass

    def test_step(self, batch: Batch, batch_idx):
        pass

    def configure_optimizers(self):
        pass
```

Listing 5: General structure of Modules in this research

```python
    features = [self.gnn(data) for data in batch.to_data_list()]
    features = torch.stack(features)

    for i in range(num_windows):

        # (...)

        # Positive and negative edge sampling
        negative_edges = negative_sampling(
            edge_index=y_edge_index,
            num_nodes=y_nodes.size(dim=0),
            num_neg_samples=max(positive_edges.size(1),
            self.negative_edge_sampling_min)
        )
        # (...)

        labels = torch.cat([
            torch.ones(positive_edges.size(1)),
            torch.zeros(negative_edges.size(1))
        ]).to(self.device)

        # Grab scores
        link_pred, link_class = self(x_features, edge_pairs)

        # Calculate the loss for link prediction
        loss = F.binary_cross_entropy_with_logits(link_pred, labels.float())

        # (...)

        total_loss += loss

    avg_loss = total_loss / num_windows
```

Listing 6: A simplified view of unoptimized `run_trough_batch` method

To optimize this process, we decided to use the functionallity of how the PyTorch Geometric library batches graphs in a single batch. This involves concatenating node features, edge indices, and other attributes while adding a batch vector that tracks graph membership for each node. Edge indices are automatically offset to maintain correct graph connectivity after the nodes are reindexed. This allows for the node features of all the batched graphs, negative edge sampling and labels to be calculated per batch, instead of in a for loop.

Once the batch goes trough the GNN, it needs to be reformatted for the RNN. Firstly, a loop is used to create all windows, stacking them together, which produces a matrix of shape $[Batch, Window, Node, Feature]$. To adapt the shape to the shape expected by the RNN: $[Batch_R, Sequence_R, Feature_R]$, the tensor is reshaped into $[Batch * Node, Window, Feature]$, after which the data is run trough the RNN and reshaped back.

Since this is a vital step of our speedup process, we would like to explain it further. Ultimately, our goal for each node feature is to see how it evolves trough time, so, if we assume that we have a single node with 3 sliding windows, we can batch them together, resulting in a batch with 3 sequences, each representing one window.

As an example, if we consider a sliding window of 3 and 5 features: $[f_0, f_1, f_2, f_3, f_4]$, we can batch all sliding windows, calculating all of them in parallel:

$$\text{Batch} = \begin{bmatrix} f_0 & f_1 & f_2 \\ f_1 & f_2 & f_3 \\ f_2 & f_3 & f_4 \end{bmatrix} \tag{5.1}$$

We can then extend the system to two hosts A and B: $[A_{0...4}, B_{0...4}]$, batching all features as follows:

$$\text{Batch} = \begin{bmatrix} A_0 & A_1 & A_2 \\ A_1 & A_2 & A_3 \\ A_2 & A_3 & A_4 \\ B_0 & B_1 & B_2 \\ B_1 & B_2 & B_3 \\ B_2 & B_3 & B_4 \end{bmatrix} \tag{5.2}$$

This produces the aforementioned shape $[Batch * Node, Window, Feature]$, which can then be run trough the RNN to produce a result, where the last feature represents the resulting prediction:

$$\text{Batch} = \begin{bmatrix} A_0 & A_1 & A_2 & \hat{A_3} \\ A_1 & A_2 & A_3 & \hat{A_4} \\ A_2 & A_3 & A_4 & \hat{A_5} \\ B_0 & B_1 & B_2 & \hat{B_3} \\ B_1 & B_2 & B_3 & \hat{B_4} \\ B_2 & B_3 & B_4 & \hat{B_5} \end{bmatrix} \tag{5.3}$$

This allows for all windows to be batched together and calculated in parallel, significantly speeding up the RNN computation. Once completed, the resulting predictions are reshaped back to their original form, so that the decoding process and the loss calculation can be performed on the whole batch.

5.3. Training

The training of the model is performed using a helper class, implemented in the PyTorch Lightning library. The class allows for easy training of a model, as shown in Listing 7, while also handling a lot of the manual tasks. The `Trainer` class automatically fetches batches from the dataloader and calls the `training_step()` function from the model.

During the training process, multiple metrics about the model's performance are logged using the classes provided by the Torchmetrics library [31].

First, the accuracy of the predictions are calculated for each window in the batch. The method is shown by Equation (5.4), where y_i represents the expected labels and $\hat{y}_i$ represents the actual predictions. For

```
# model
autoencoder = LitAutoEncoder(Encoder(), Decoder())

# train model
trainer = L.Trainer()
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

Listing 7: Trainer code

floating point values, the threshold was set to 0.5. The calculated accuracy is then averaged across all windows, as show in Equation (5.4) to produce the accuracy value for the whole batch.

$$\text{Accuracy} = \frac{1}{N}\sum_{i}^{N} 1(y_i = \hat{y}_i) \tag{5.4}$$

$$\text{Accuracy}_{batch} = \frac{1}{S}\sum_{i}^{S} Accuracy(S) \tag{5.5}$$

The AUROC metric is also calculated with methods implemented in the Torchmetrics library, which automatically handles averaging the result per batch.

Lastly, the loss from the edge predictions and classifications is calculated separately using cross-entropy [26]. The two losses are then combined with the help of an alpha value (α), as shown in Equation (5.6). The alpha value is used to indicate the importance of both problems and which one should be optimized more.

$$\mathcal{L} = \mathcal{L}_{prediction} + \alpha * \mathcal{L}_{classificaion} \tag{5.6}$$

In an attempt to improve the acuracy of the model, two other loss functions were used, named “confidence" and “v2". The confidence loss, shown in Equation (5.7), tries to integrate the confidence of the model’s link prediction in the calculate for the link classification loss, so that a correct classification would be awarded more if the model correctly predicted that an edge should exist.

$$\mathcal{L}_{\text{conf}} = \frac{1}{N}\sum_{i=1}^{N} p_i \cdot \mathcal{L}_{\text{class}} \tag{5.7}$$

where p_i is the probability of edge i existing and N is the total amount of edges in the batch.

Additionally, the “v2" loss, shown by Equation (5.8), attempts to focus the model’s training for link prediction in the absence of malicious edges while increasing the penalty for classification when there are malicious edges. This is implemented through a dynamic alpha variable determined by the number of malicious edges. Additionally, the variable is constrained to a minimum threshold of 0.2, attempting to allow for the model to learn basic edge classification.

$$\alpha = \max\left(\frac{M}{N}, 0.2\right) \tag{5.8}$$

$$\mathcal{L}_{\text{v2}} = (1-\alpha)\,\mathcal{L}_{\text{pred}} \;+\; \alpha\,\mathcal{L}_{\text{class}} \tag{5.9}$$

where M is the amount of malicious edges and N is the amount of edges in the batch.

The backpropagation of the losses is automatically handled by the Lightning library and will not be explained in detail in this paper.

5.4. Experimental setting

To evaluate our proposed improvements, multiple tests were performed in sequential order on both CPU-only and GPU-accelerated environments. This allows for a proper assessment of the algorithm's speed and accuracy.

The experiment used two hosts. A virtualized server inside TU Delft infrastructure was used for the CPU-only host, while the Delft AI Cluster (DAIC) [1] provided a GPU-accelerated environment. The CPU-only host contains 12 virtualized Intel(R) Xeon(R) Platinum 8358P CPU @ 2.60GHz cores, 64GB RAM memory, and 1.5 TB SSD storage. The DAIC environment has a wide variety of host configurations, with 137 GPUs spread accross 42 compute nodes. Additionally, DAIC has 24 additional compute nodes, which do not have GPUs, however, our definition file states a requirement of at least 1 GPU, which means that those 24 hosts will not be used to run our experiments.

As both environment are virtualized, performance interference could cause inconsistency in the timing of the process, however, since our research does not assess the models on how fast they can be trained, these inconsistencies can be ignored.

To support maximum reproducibility of the result, the repository of the project also supplies a `.def` file, which can be used to create a container image for Apptainer [25]. The image depends on a Docker image by the Nvidia NGC [32], which comes with all PyTorch and PyTorch Geometric preinstalled. This eases the image creation process, as the definition file only needs to specify how to install the project. The full definition file can be observed on Github and in Appendix A. The created image is then used to run the experiments on the DAIC cluster.

6 Results

The following chapter presents the results we have obtained from our experiments. In this chapter we present the benefits of having an RNN layer, our model's performance in edge prediction (anomaly detection) and edge classification.

6.1. The benefits of an RNN

Our initial experiment is relatively straightforward: to evaluate the benefits of having an Recurrent Neural Network (RNN) for edge prediction. The premise is that we can skip the RNN for prediction and use the decoder to predict the next edges. The experiment will show the importance of the RNN layer and it's benefits.

Our initial experiment was relatively straightforward, aimed at evaluating the advantages of incorporating a Recurrent Neural Network (RNN) within the system. The premise is that we can bypass the RNN for prediction purposes and instead utilize a decoder to forecast the subsequent edges. Table 6.1 demonstrates the benefit of the RNN layer and its associated benefits.

	Try0	**Try1**
Dataset	**UWF-ZeekData22**	**UWF-ZeekData22**
Accuracy	0.96964	0.99419
AUC	0.97804	0.99867
F1	0.96903	0.99418
AP	0.97881	0.99890

Table 6.1: `Try0` compared to `Try1`

The initial indication from the results is the increase in the AP score by 2%, showing that the RNN layer helps increase the recall. Since the F1 and AUC scores integrate both recall and precision, it is natural to see a rise in those scores as well. Finally, considering the elevated levels of recall and precision, along with a 3% improvement in accuracy, we can conclude that an RNN greatly helps with the edge prediction performance of the model.

6.2. Edge prediction

Edge prediction is critical for anomaly detection in cybersecurity. To evaluate our models (`Try1` and `Try2`), we conducted experiments comparing them to the state-of-the-art `Euler` model using an established benchmark dataset, shown in Section 6.2. Analyzing the results, we see that both models we have proposed outperform `Euler`.

The proposed model in `Euler` achieves a 84% true positive rate, with 0.6% false positives. The high AUC score is consistent with the high amount of true positives and the low false positive rate, however, the F1 and AP scores indicate that the model has poor precision. `HetGNN` shows a nice improvement with a 93% accuracy and a 89% F1 score. Compared to Try1 and Try2, we can see that the two models achieve a significantly higher precision and recall, reflected both by the AP and F1 scores.

Model		Accuracy	TPR	FPR	F1	AUC	AP
Euler	mean	-	0.84786	0.00611	0.00979	0.98996	0.00782
	stderr	-	0.02954	0.00011	0.00029	0.00102	0.01685
HetGNN	mean	0.9325	-	-	0.8928	-	-
Try1	mean	0.99419	0.99108	0.00504	0.99336	0.99338	0.99880
Try2	mean	0.99328	0.99061	0.00558	0.99275	0.99828	0.99846

Table 6.2: Performance of Try1 and Try2 compared to Euler and HetGNN

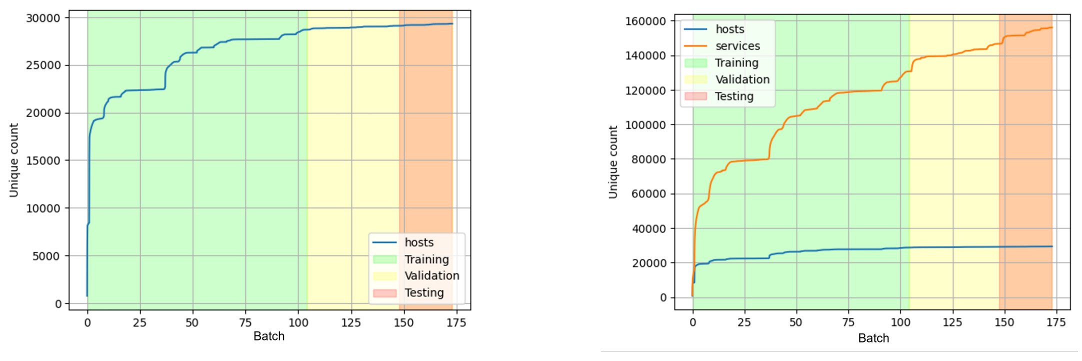
Comparing our model to HetGNN is not as straightforward, as the two models are fundamentally different, which is why we can only compare on the basis of results obtained. However, since Euler and our model share similar architectures, we can trace the differences in results to multiple contributing factors. Firstly, the Euler framework is made with speed in mind, which is why it uses larger data bins and smaller layers to achieve significantly shorter training times. In contrast, our models are constructed with a focus on accuracy, leading to training epochs that may extend for several hours. Secondly, since the two models both prepare the data on their own and construct the graphs independently, this suggests that variations in implementation could provide our models with a competitive advantage. The Euler paper also indicates a challenge related to labeling anomalies within the LANL dataset, further allowing for potential differences in implementation. Lastly, the Euler paper does not define a specific RNN sliding window, implying that there may be no window at all, with each prediction based solely on a single timestamp, effectively a sliding window of 1. This is significantly lower than the sliding window of 30, which our models use. In conclusion, the results perfectly illustrate the tradeoff that a smaller time bin and a larger sliding window, as well as the constraints imposed by the dataset's design.

An important distinction is the slight improvement in the AUC metric, which Try2 has over Try1. Looking at the differences between the two models, we believe that this improvement could be the result of factors: the size of the layers and the change in the GNN. Firstly, Try2 has a slightly bigger RNN layer, with the LSTM containing 5 layers, compared to the 1 GRU layer which Try1 has. Another factor is that Try2 has GATv2 GNNs, which are modified to use the edge attributes, while Try1 has GCNConv GNNs, only allowing for edge weights. This comparison also shows an interesting dynamic between the model size and the accuracy of the model, as we also see the smaller Try1 outperforming the bigger Try2 in the F1 score and the amount of false positives.

As my professor often cites: "(There are) Lies, damned lies, and statistics", which prompted us to delve deeper into the reasons behind the high scores. The first observation we made was the huge imbalance between the two classes (614 malicious to 239,470,861 benign events). Such an imbalance can help the model with achieving high accuracy scores, however, it does not help with the AUC and F1 metrics. To further explain the high scores there, we need to look at the amount of new hosts and services the model encounters during its testing stage. Figure 6.1a shows that nearly all hosts were observed during the training phase (as evidenced by the sharp increase in the line). This leaves the validation and testing stage with a large amount of hosts, that have already been included in the training dataset.

In simple terms, this means that the testing happens on previously observed hosts, which does not show how well the model has captured the fundamental behavioral patterns of malicious activities, but rather only the behavioral patterns of known hosts. Although, this is not considered good for model evaluation, we would like to point out that this is not entirely the case for computer network monitoring, as many attacks exploit assets that have been present within the monitored network for a considerable duration. For example, if we consider a successfull phishing attack, a lot of the times the attack starts from a machine, which has already spent a significant amount of time on the newtowrk before the phighing attack.Therefore, to summarize, while it would be highly beneficial to know how well the model performed with previously unseen data, the lack of unseen devices is not necessarily a bad thing in the domain of cybersecurity.

To demonstrate that the same model is applicable to real-world Zeek data, our model is also tested against three public Zeek datasets: UWF-ZeekData22 [6], UWF-ZeekData22Fall [5], and UWF-ZeekData24



(a) Only Hosts

(b) Hosts and Services

Figure 6.1: Unique hosts and services encountered in the testing stage of the LANL dataset

[10]. Unlike LANL, these datasets are evenly distributed, containing 50% benign and 50% malicious events. The results, shown in Table 6.3, reveal that the model is having difficulty in accurately predicting the network events recorded in the datasets.

	Try1			**Try2**		
Dataset	**Zeek22**	**Zeek22Fall**	**Zeek24**	**Zeek22**	**Zeek22Fall**	**Zeek24**
Accuracy	0.50941	0.58974	0.99330	0.50503	0.58974	0.88281
AUC	0.85263	0.75339	0.23237	0.98564	0.23533	0.14183
F1	0.01246	0.00000	0.22877	0.00000	0.00000	0.00000
AP	0.86246	0.77749	0.23248	0.98617	0.41024	0.14924

Table 6.3: Edge prediction results of `Try1` and `Try2` when trained on the datasets from the University of West Florida

A number of "tricks" were applied on the data in an effort to increase the prediction score of the model. Firstly, the `expand(df)` function (described in detail in Section 4.2.2) was used to expand events, which have duration spanning multiple bins. The results in Table 6.4 show no improvement in the overall performance. One possible reason could be the increase in edge count, resulting in more misclassified malicious edges. Unfortunately, we are unable to explain why this transformation does produce improvements, as it is our hypothesis that it should make the data more transparent and ease the prediction.

Model	**Try2**	
Dataset	**UWF-ZeekData22**	**UWF-ZeekData22 (Extended)**
Accuracy	0.99391	0.99005
AUC	0.00128	0.00051
F1	0.99391	0.99005
AP	0.07910	0.07799

Table 6.4: Edge prediction performand on the UWF-ZeekData22 Dataset with and without `expand(df)`

Furthermore, the shuffling algorithms described in Section 4.2.2 were used to enhance performance. The results, shown in Table 6.5, show similar results as with the accounting for duration, where the classification is not improved with random shuffling. This is to be expected, as random shuffling greatly impacts the temporal dependencies of each event. Breaking this temporal chain of events interacts with the ability of the RNN to capture the behavior of each host, resulting in poor performance.

Contrary to random shuffling, the `Day` shuffling seems to significantly improve the predictions of the algorithm, scoring a solid accuracy, high precision and recall. This could be explained with the distribution of the malicious edges, as the malicious events are no longer clustered at the end of the data (as shown in Figure 4.5a), but scattered across the day (visualized in Figure 4.6). This allows for the model to better “learn" and distinguish malicious behaviors entangled in between legitimate ones.

In summary, our evaluation demonstrates that both proposed models, `Try1` and `Try2`, outperform the

Model	Try2				GCN-Autoencoder
Shuffling	**Day**	**Random (0.1s bins)**	**Random (1s bins)**	**Random (10s bins)**	None
Accuracy	0.96868	0.25153	0.38135	0.33951	0.9970
AUC	0.99607	0.16061	0.10613	0.07909	-
F1	0.95890	0.25154	0.38135	0.33945	0.9997
AP	0.99480	0.70842	0.47823	0.37844	-

Table 6.5: Edge prediction performance with different shuffling for Try2

Euler and HetGLM frameworks on the LANL dataset, achieving higher precision and overall balanced performance. This improvement can be the result of design choices, such as prioritizing accuracy over training speed, the use of larger sliding windows, and graph construction methods that better encapsulate structural information.

However, when applied to more balanced, real-world UWF-Zeek datasets, there is a significant decline in performance, highlighting the critical role of dataset characteristics and temporal dependencies in edge prediction. In particular, we found that Day shuffling, which redistributes malicious events throughout the data set while maintaining daily temporal patterns, significantly improved accuracy, precision and recall. This indicates that the clustering of malicious events in the raw data may negatively impact model generalization and performance. These results show that our models are able to adapt, "learn", and acurately predict both events from the LANL and the UWF-Zeek datasets.

6.3. Edge classification

Edge classification is a crucial step forward in graph-based anomaly detection, as it moves beyond simple edge prediction, potentially reducing false positives significantly. While traditional edge prediction primarily serves as an anomaly detector to flag malicious activities, edge classification distinguishes explicitly between benign and malicious edges, allowing for more precise detection and fewer incorrect alerts.

Based on the results we initially obtained, we proceeded to upgrade our model to edge classification. However, upon evaluating the upgraded model, we encountered a notable challenge: despite achieving high accuracy, our edge classification approach had low F1 and AUC scores, as shown in Table 6.6.

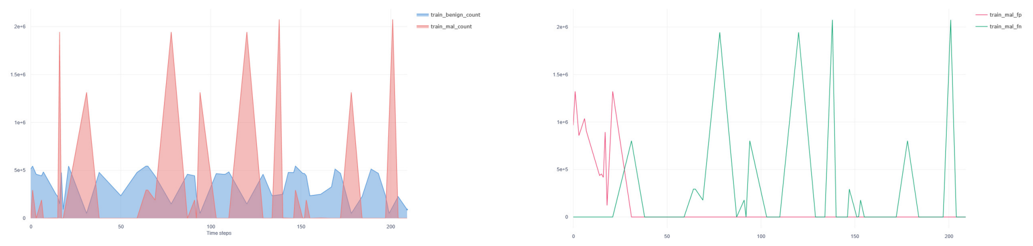
Model	Try1			Try2			MC-SVM
Dataset	**Zeek22**	**Zeek22Fall**	**Zeek24**	**Zeek22**	**Zeek22Fall**	**Zeek24**	**Zeek22**
Accuracy	0.93224	0.64816	0.79042	0.99391	0.08333	0.61718	0.8284
AUC	0.00160	0.01887	0.23586	0.00128	0.01610	0.23718	-
F1	0.93224	0.64816	0.79042	0.99391	0.08333	0.61718	0.7823
AP	0.08040	0.19124	0.16547	0.07910	0.17923	0.11964	-

Table 6.6: Edge classification results of Try1 and Try2 when trained on the datasets from the University of West Florida

The findings show an interesting pattern, where both models classify edges with accuracy above 60% for UWF-ZeekData22 and UWF-ZeekData24, however, performs poorly with the UWF-ZeekData22Fall dataset. This is interesting, as all 3 datasets are collected using the same infrastructure and have similar characteristics. The results further show a nice high accuracy for edge classification, however, the low AP and AUC metric indicate a high amount of misclassified events.

In comparison to the edge prediction scores, classification scores are not as robust. This robustness is needed since the high amount of misclassified events could lead to a high amount of alerts, leading to alert fatugue, and misclassifications of events could lead to actual attacks being missed. Thus, it is imporatnt that a model used in production must be more robust and not lead to high amount of false positives or negatives.

To further investigate the reason for the low AUC score, we plotted the predictions of the model against the true labels of the edges. The problem is visually presented in the confusion matrix in Figure 6.4a, where our model misclassifies all malicious edges as benign. The same relationship is shown in Figure 6.2: Figure 6.2a shows the labels of each edge used in the training stage, while Figure 6.2b shows the mistakes that the model made while trying to correctly classify all edges.



(a) True labels of edges **(b)** Incorrectly predicted edges

Figure 6.2: `UWF-ZeekData22` edge classification predictions against true labels

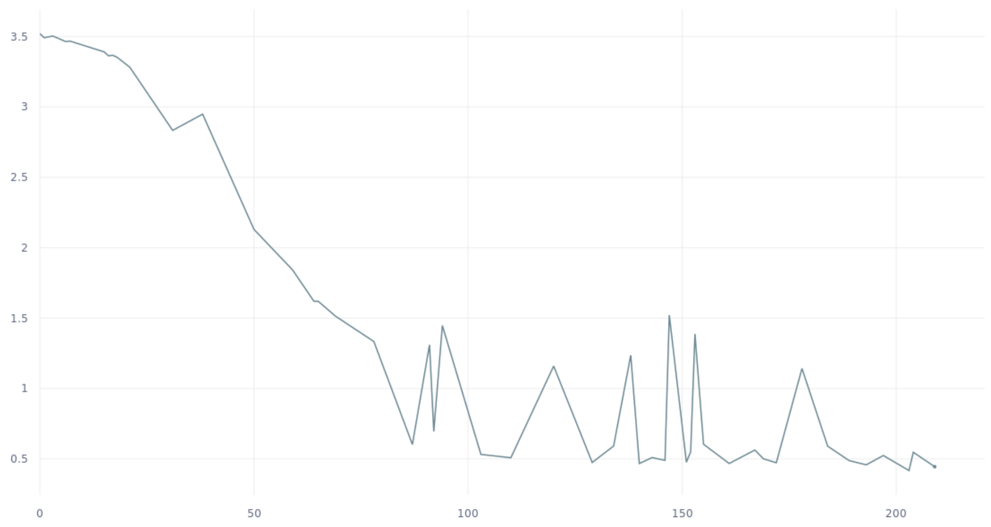


Figure 6.3: `UWF-ZeekData22` loss

As we can see in the figures, there is a visible correlation between the malicious edges and the incorrectly classified edges, indicating that the model's classification mostly classifies everything as benign, ignoring malicious edges. This ignorance is also penalized with the loss function, as shown in Figure 6.3, however, the backpropagation does not seem to fix the issue effectively.

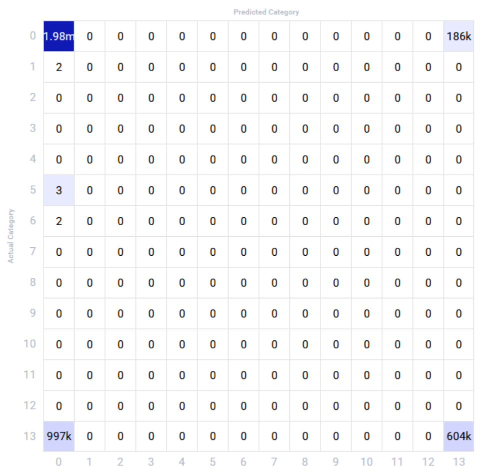
One of our hypotheses was that the model does not have enough batches to account for the classification error, as the malicious edges are predominantly found within a few batches throughout the entire epoch (as shown in Figure 4.4). This leads to an imbalance, resulting in a high amount of benign edges inside the test batch, which in turn artificially elevates our accuracy.

In an effort to mitigate this issue, we experimented with different shuffle tactics to penalize the model troughout the whole learning process. Differently shuffled datasets were tested with both `Try1` and `Try2` models, evaluating their performance on multiple metrics, shown in Table 6.7.

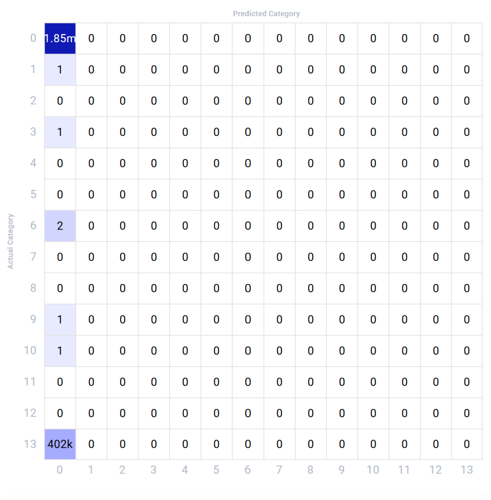
	Try2			
Shuffling	**Day**	**Random (0.1s bins)**	**Random (1s bins)**	**Random (10s bins)**
Accuracy	0.94749	0.25153	0.38135	0.33951
AUC	0.02640	0.16061	0.10613	0.07909
F1	0.94749	0.25154	0.38135	0.33945
AP	0.24819	0.70842	0.47823	0.37844

Table 6.7: Edge classification performance with different shuffling algorithms

Due to reasons highlighted in Section 6.2, the process of random shuffling appears to have limited effectiveness with edge classification. Regrettably, `Day` shuffling does not seem to impact classification as compared to its impact on edge prediction, however we can still see a small increase across all metrics. This is probably due to the increased imbalance caused by the `Day` shuffling, as shown in Figure 6.4b.



(a) Confusion matrix without shuffling



(b) Confusion matrix with `Day` shuffling

Figure 6.4: `Try2` confusion matricies when trained on `UWF-ZeekData22`

The confusion matrices show the high amount of misclassified events, however, it also shows a relationship between dataset imbalance and temporal dependencies. Unfortunately, since our model requires data to be sampled in a continuous period, it is extremely difficult for the dataset to be properly split into a balanced set of malicious and benign edges.

To further evaluate the robustness of the model, we tested different dropout rates and loss functions. Table 6.8 reports the performance of the edge classification task under varying dropout rates.

	Dropout 0.05	**Dropout 0.1**	**Dropout 0.2**	**Dropout 0.4**
Accuracy	0.99220	0.99496	0.99250	0.99622
AUC	0.00119	0.00078	0.00042	0.00107
F1	0.99220	0.99496	0.99250	0.99622
AP	0.07989	0.07716	0.07606	0.07840

Table 6.8: Edge classification performance with different dropout rates

Overall, the results show that moderate dropout values improve generalization while avoiding severe degradation of performance. In particular, a dropout rate of 0.4 achieved the highest accuracy and F1 score, with only a small reduction in average precision (AP).

In addition, different loss functions were explored to assess their influence on model performance as shown in Table 6.9. The `Normal` loss combines the standard link prediction and classification terms using an alpha value. The `Confidence` variant incorporates the predicted edge probability into the class loss, effectively weighting samples by the model's confidence. The `v2` formulation adjusts the contribution of prediction and classification losses based on the proportion of positive edges in the batch.

The results indicate that the `Confidence` loss achieves slightly higher accuracy and F1 score compared to the Normal baseline, suggesting that weighting by prediction confidence can improve classification reliability. The v2 loss, however, leads to a significant drop in both accuracy and F1 score, while maintaining a comparable AP. This suggests that dynamically weighting loss terms by the fraction of malicious edges may destabilize training. Overall, the `Confidence` loss approach appears to be the most effective among the tested formulations.

Metric	Normal	Confidence	v2
Accuracy	0.99391	0.99434	0.00563
AUC	0.00128	0.00031	0.00069
F1	0.99391	0.99434	0.00564
AP	0.07910	0.07537	0.07916

Table 6.9: Edge classification performance with different loss functions

In conclusion, we investigated edge classification as an extension of edge prediction for anomaly detection. While edge classification has the potential to reduce false positives by explicitly distinguishing between benign and malicious edges, our experiments revealed several challenges. Despite achieving consistently high accuracy across datasets, the models often suffered from low AUC and AP scores, indicating a tendency to misclassify malicious edges as benign. This imbalance is further highlighted by the distribution of malicious edges across batches, where all malicious edges are clustered at the end of the timeline.

Attempts to address these shortcomings through data shuffling, dropout regularization, and alternative loss functions produced mixed results. Shuffling showed limited impact, dropout offered modest stability gains, and the `Confidence` loss slightly improved reliability. Overall, edge classification proved less robust than edge prediction, with its misclassification tendencies making it unsuitable for production use without further improvements. Addressing imbalance and refining model architectures remain key directions for future work.

7 Discussion

This thesis aims to explore the potential of Graph Neural Networks (GNNs) for intrusion detection, focusing on edge prediction for anomaly detection and edge classification for full intrusion detection. The experiments provided interesting insights into both the promise and the challenges of applying AI models in cybersecurity.

7.1. Key insights

One of the most striking outcomes was the relative ease and effectiveness of edge prediction. The chosen architecture demonstrated robust anomaly detection capabilities, achieving high AUC and AP scores across datasets. This highlights the ability of temporal GNN-based methods for recognizing structural patterns in network traffic and detecting anomalies even when the dataset is imbalanced.

In contrast, edge classification proved considerably more difficult. Although it provided the theoretical benefit of clearly differentiating between benign and malicious edges, our models consistently struggled to properly classify malicious edges. Despite achieving high accuracy, they tended to classify nearly all edges as benign, resulting in poor AUC and AP scores and high rates of false negatives. This revealed both the sensitivity of classification to dataset imbalance and the challenges associated with maintaining precise control over the model's learning process.

A key takeaway is the importance of knowing the data. Although the datasets from the University of West Florida are balanced, closer inspection revealed that malicious edges were heavily segregated from the benign edges. This clustering compromised classification by artificially inflating the accuracy metric while obscuring poor performance in classifying other classes. More representative and specialized datasets are therefore essential for improving edge classification approaches.

7.2. Limitations

Several limitations were identified during our research. First, the datasets used, pose inherent challenges. The LANL dataset, while widely adopted, is highly unbalanced and provides only limited labeling, making it difficult to assess whether strong results might not be the result of the high unbalance and implementation differences. The `UWF-ZeekData` datasets, though more balanced, contained malicious edges clustered in ways that complicated the splitting of the dataset and limited the model's ability to train on malicious edges.

Another limitation concerns the graph construction process. Some potentially valuable information, such as the full history of each connection, was excluded due to the complexity of encoding it into the graph structure. The absence of this information likely limited the model's performance.

Scalability also remains a concern. While the models produced good results in controlled experiments, their performance on large-scale, real-world network traffic could not be tested. The computational complexity of GNN-based models raises open questions about whether such approaches can be deployed in real-time intrusion detection systems without sacrificing performance.

Finally, controlling and tuning the classification models proved particularly challenging. Despite attempts to adjust hyperparameters, loss functions, and regularization strategies, the models often defaulted to labeling all edges as benign. This suggests that more advanced training techniques, such as better sampling, weighted loss functions, or ensemble methods will be necessary to achieve reliable classification performance.

Our findings identify an interesting trade-off between robustness and interpretability. Edge prediction provides stable and reliable anomaly detection, but does not explicitly identify malicious edges. Edge classification, while able to provide more interpretable detection in theory, currently lacks the reliability to be used in production environments.

Lastly, we would like to stress on the importance of datasets in AI research. Obtaining robust outcomes from flawed or unrepresentative datasets poses the danger of creating a false sense of progress, undermining the practical value of the research. For AI-driven intrusion detection systems to mature, greater emphasis must be placed on creating and sharing realistic, well-labeled datasets that capture the complexity of real-world network traffic.

In summary, this research demonstrates that graph-based anomaly detection, particularly edge prediction, holds strong promise for intrusion detection. At the same time, it highlights the significant challenges of edge classification and the risks of relying on limited datasets. Progress in this field will depend not only on refining model architectures but also on addressing dataset quality and relevance, to ensure that methods remain effective in real-world environments.

8 Conclusion

This thesis investigated the use of Graph Neural Networks (GNNs) for improving intrusion detection, with a focus on edge prediction and edge classification as anomaly detection strategies. Motivated by the shortcomings of traditional intrusion detection systems, particularly high false positive rates and poor adaptability, we developed a temporal graph-based framework that integrates GNNs with recurrent layers. The framework was evaluated on multiple datasets, including the LANL authentication logs and UWF Zeek logs, to assess its ability to detect malicious activity.

8.1. Summary

Our results show that temporal graph modeling with edge prediction can produce robust anomaly detection performance, while edge classification presents more significant challenges. Through extensive experiments, including tests of data shuffling strategies, dropout regularization, and loss function variants, we identified the strengths and limitations of each approach.

The research questions posed at the start of this thesis can now be answered as follows:

RQ1. Can AI detection be improved?
Yes. Edge prediction proved effective across datasets, achieving higher AUC and AP scores than classification. The models consistently captured anomalies despite dataset imbalance and temporal clustering of malicious edges, showing resilience and reliability for practical use. Edge classification achieved high accuracy but underperformed in AUC and AP, largely due to misclassifying malicious edges as benign. This was caused by imbalanced data distributions, where malicious edges were concentrated in a few batches. Compared to prediction, classification was less robust and carried a greater risk of missing attacks. In summary, edge prediction's key advantage is robustness in detecting anomalies with fewer false positives. Edge classification provides explicit labeling, but in practice, its misclassifications, alert fatigue, and sensitivity to imbalance undermine its utility.

RQ2. Can graph representation improve the detection?
Yes, graph representation improved anomaly detection compared to traditional vector-based approaches. By modeling entities and their interactions as temporal graphs, the framework captured structural and temporal dependencies that would otherwise be lost. This was evident in the performance gains of edge prediction, which benefited from temporal graph encoding to detect anomalies more reliably. However, the experiments also revealed that graph construction decisions, such as handling imbalance, temporal continuity, and enrichment of host and service information, directly impacted model performance. Thus, while graph representation enables stronger detection, its effectiveness depends heavily on how the underlying data is structured and preprocessed.

RQ3. Can AI detection be used alongside Zeek?
Maybe. This thesis demonstrated, to the best of our knowledge, one of the first applications of GNN-based anomaly detection on Zeek logs. The results confirm that AI-driven graph methods can

be integrated with Zeek data, providing an additional analytical layer beyond traditional rule-based detection. However, performance varied across the different UWF Zeek datasets, with some suffering from severe imbalance and clustering of malicious edges. This indicates that while AI detection can complement Zeek, further refinement of data preprocessing and balancing techniques is needed before reliable deployment in production environments.

In summary, this thesis shows that while edge classification has conceptual appeal, edge prediction currently provides the more reliable basis for anomaly detection in practice. Future research should prioritize methods to counter dataset imbalance, refine classification loss functions, and explore architectures capable of better distinguishing rare malicious edges. Only then can classification approach the robustness and reliability already demonstrated by prediction.

8.2. Future work

While the proposed framework demonstrates promising results, it remains in an early stage of development, with considerable room for improvement. The current architecture could benefit a lot from data enrichment, different graph creation processes, and improvements in the GNN and RNN layer. In this chapter, we outline several directions for future research and development, aimed at improving the system's accuracy, usability, and real-world applicability. These improvements span from better data enrichment and advanced graph structures to more expressive neural network architectures and proactive threat mitigation strategies.

The system leaves considerable room for improvement for enriching of the input data. While computer communication is typically dictated by some form of protocol, in our system the model is left to learn the behavior on its own. Nevertheless, given that we have the documentation for much of the open communication, we can encode this knowledge, thereby removing the need for the AI to independently learn the communication protocols. This approach may also allow for different protocols to be easily integrated into the system, provided their documentation is accurately encoded for the model to understand. Ultimately, this could allow for the detection of more complex behavior, as the model will be tasked with learning behavioral deviations, rather than simple anomalies.

Data can also be enriched by external sources, such as VirusTotal, AnyRun, Censys, etc. This could bring the huge knowledge base of Open Source Inteligence (OSINT) to be used by the model to improve detections and reduce the number of false positives. However, this would also require extensive research on the reliability of the data, as unreliable information can negatively impact the outcomes.

Further research can be conducted on the graph structure to encode more specific information. For example, the current graphs we use do not represent the relationships between users/services and computers as bipartite. However, such an encoding could help the model learn specific behaviors associated with particular services. The same principle can be applied to hypergraphs, which could show complex relationships between multiple nodes across different node types.

The embedding process derived from the graphs can also be improved trough the usage of Continuous GNNs or Dynamic GNNs, both of which outperform typical GNNs [35]. We believe that adapting the system for such GNNs could increase the accuracy and better capture the temporal "tendencies" of the underlying traffic.

Looking at the other layers of the model, the inherent characteristics of the RNN can allow for predictions to be made for several time steps into the future. This ability could be utilized to anticipate and mitigate complex attacks, such as detecting and preventing access from certain accounts to critical systems immediately upon detection of the enumeration on those accounts. Implementing such a reactive system could represent a significant stepping stone for security frameworks, where AI can recognize "motifs", predict the attack chain and prevent the attack before it has been performed. It is crucial to emphasize that additional research is needed in this area, as such a system system also carries the risk of disrupting and disabling crucial business logic in the event of a false positive.

Another potential improvement for the RNN layer, is its ability to perform bidirectional detection. The concept involves creating a detection model similar to the one in the Euler [20] paper capable of identifying missing edges. However, this can further be improved into a fully operational post-exploitation model, to be used in forensic analysis or Security Operations Center (SOC) hunting sessions.

Since our model already allows for edge prediction and classification, we believe it would not be a huge leap forward extending the RNN to analyze both past and future events. This modification will allow the model to use a bidirectional RNN to analyze historical traffic data, spot annomalies, and recognize attacks post-exploitation.

Lastly, we would like to tackle the challenges faced in this research involving the datasets from the University of West Florida [6, 10, 5]. Although we consider these datasets as a significant advancement for research into real-world Intrusion Detection Systems (IDS), we still believe that a dataset containing a consistent mix of both benign and malicious traffic would provide a more accurate depiction of the real-world.

In conclusion, although the existing framework provides a solid basis for AI-based intrusion detection, its full potential is yet to be realized. The proposed directions for improvement, ranging from data enrichment and advanced graph modeling, to predictive and retrospective detection, highlight both the flexibility and the potential usability of this system. Further research and experimentation are crucial to convert these theoretical advances into solutions tested in the real world, ultimately leading to more robust, adaptive, and intelligent IDS systems.

References

[1] Delft AI Cluster (DAIC). *The Delft AI Cluster (DAIC), RRID:SCR_025091*. 2024. DOI: 10.4233/rrid:scr_025091. URL: https://doc.daic.tudelft.nl/.

[2] Oluwadamilare Harazeem Abdulganiyu, Taha Ait Tchakoucht, and Yakub Kayode Saheed. "A systematic literature review for network intrusion detection system (IDS)". In: *Int. J. Inf. Sec.* 22.5 (2023), pp. 1125–1162. DOI: 10.1007/S10207-023-00682-2. URL: https://doi.org/10.1007/s10207-023-00682-2.

[3] Ahmad Ahmad, Aleksandr Kovalenko, and Ilya Makarov. "Anomaly Detection Using Graph-Based Autoencoder with Graph Structure Learning Layer". In: *2024 IEEE 6th International Symposium on Logistics and Industrial Informatics (LINDI)*. 2024, pp. 89–94. DOI: 10.1109/LINDI63813.2024.10820392.

[4] Atlassian. *Understanding and fighting alert fatigue | Atlassian*. 2025. URL: https://www.atlassian.com/incident-management/on-call/alert-fatigue.

[5] Sikha S. Bagui et al. "Introducing the UWF-ZeekDataFall22 Dataset to Classify Attack Tactics from Zeek Conn Logs Using Spark's Machine Learning in a Big Data Framework". In: *Electronics* 12.24 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12245039. URL: https://www.mdpi.com/2079-9292/12/24/5039.

[6] Sikha S. Bagui et al. "Introducing UWF-ZeekData22: A Comprehensive Network Traffic Dataset Based on the MITRE ATT&CK Framework". In: *Data* 8.1 (2023). ISSN: 2306-5729. DOI: 10.3390/data8010018. URL: https://www.mdpi.com/2306-5729/8/1/18.

[7] Brian T. Carr. "Automating Suricata Rule-Writing". English. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-09-15. PhD thesis. Utica College, 2021, p. 56. ISBN: 9798538151660. URL: https://www.proquest.com/dissertations-theses/automating-suricata-rule-writing/docview/2572564940/se-2.

[8] Dawei Cheng et al. "Graph neural networks for financial fraud detection: a review". In: *Frontiers Comput. Sci.* 19.9 (2025), p. 199609. DOI: 10.1007/S11704-024-40474-Y. URL: https://doi.org/10.1007/s11704-024-40474-y.

[9] Hong Cheng, Xifeng Yan, and Jiawei Han. "Mining Graph Patterns". In: *Frequent Pattern Mining*. Ed. by Charu C. Aggarwal and Jiawei Han. Cham: Springer International Publishing, 2014, pp. 307–338. ISBN: 978-3-319-07821-2. DOI: 10.1007/978-3-319-07821-2_13. URL: https://doi.org/10.1007/978-3-319-07821-2_13.

[10] Marshall Elam et al. "Introducing UWF-ZeekData24: An Enterprise MITRE ATT&CK Labeled Network Attack Traffic Dataset for Machine Learning/AI". In: *Data* 10.5 (2025). ISSN: 2306-5729. DOI: 10.3390/data10050059. URL: https://www.mdpi.com/2306-5729/10/5/59.

[11] William Falcon and The PyTorch Lightning team. *PyTorch Lightning*. Version 1.4. Mar. 2019. DOI: 10.5281/zenodo.3828935. URL: https://github.com/Lightning-AI/lightning.

[12] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[13] Fouzi Harrou et al. "Exploiting Autoencoder-Based Anomaly Detection to Enhance Cybersecurity in Power Grids". In: *Future Internet* 16.6 (2024), p. 184. DOI: 10.3390/FI16060184. URL: https://doi.org/10.3390/fi16060184.

[14] Negar Heidari, Lukas Hedegaard, and Alexandros Iosifidis. "Chapter 4 - Graph convolutional networks". In: *Deep Learning for Robot Perception and Cognition*. Ed. by Alexandros Iosifidis and Anastasios Tefas. Academic Press, 2022, pp. 71–99. ISBN: 978-0-323-85787-1. DOI: https://doi.org/10.1016/B978-0-32-385787-1.00009-9. URL: https://www.sciencedirect.com/science/article/pii/B9780323857871000099.

[15] Vincent C Hu et al. "Guide to attribute based access control (abac) definition and considerations (draft)". In: *NIST special publication* 800.162 (2013), pp. 1–54.

[16] Sandra L. Kane-Gill et al. "Technologic Distractions (Part 1): Summary of Approaches to Manage Alert Quantity With Intent to Reduce Alert Fatigue and Suggestions for Alert Fatigue Metrics". In: *Critical Care Medicine* 45.9 (2017). ISSN: 1530-0293. URL: https://journals.lww.com/ccmjournal/fulltext/2017/09000/technologic_distractions__part_1___summary_of.8.aspx.

[17] Sharon Kauffman. *World's first cyberattack death: Northdoor comment*. Jan. 2023. URL: https://www.northdoor.co.uk/insight/news/worlds-first-cyberattack-death/.

[18] Alexander D. Kent. *Comprehensive, Multi-Source Cyber-Security Events*. Los Alamos National Laboratory. 2015. DOI: 10.17021/1179829.

[19] Alexander D. Kent. "Cybersecurity Data Sources for Dynamic Network Research". In: *Dynamic Networks in Cybersecurity*. Imperial College Press, June 2015.

[20] Isaiah J. King and H. Howie Huang. "Euler: Detecting Network Lateral Movement via Scalable Temporal Graph Link Prediction". In: *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL: https://www.ndss-symposium.org/ndss-paper/auto-draft-227/.

[21] Isaiah J. King et al. "EdgeTorrent: Real-time Temporal Graph Representations for Intrusion Detection". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*. ACM, 2023, pp. 77–91. DOI: 10.1145/3607199.3607201. URL: https://doi.org/10.1145/3607199.3607201.

[22] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[23] Rocio Krebs et al. "Applying Multi-CLASS Support Vector Machines: One-vs.-One vs. One-vs.-All on the UWF-ZeekDataFall22 Dataset". In: *Electronics* 13.19 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13193916. URL: https://www.mdpi.com/2079-9292/13/19/3916.

[24] Ambuj Kumar. *AI SOC vs Traditional SOC: How to Power Your Cybersecurity Strategy?* June 2025. URL: https://simbian.ai/blog/ai-soc-tools-vs-traditional-soc-cybersecurity.

[25] Gregory M. Kurtzer et al. *hpcng/singularity: Singularity 3.7.3*. Zenodo, Apr. 2021. DOI: 10.5281/ZENODO.1310023. URL: https://zenodo.org/record/1310023.

[26] Anqi Mao, Mehryar Mohri, and Yutao Zhong. "Cross-Entropy Loss Functions: Theoretical Analysis and Applications". In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 23803–23828. URL: https://proceedings.mlr.press/v202/mao23b.html.

[27] Michal Markevych and Maurice Dawson. "A Review of Enhancing Intrusion Detection Systems for Cybersecurity Using Artificial Intelligence (AI)". In: *International conference KNOWLEDGE-BASED ORGANIZATION* 29 (July 2023), p. 2023. DOI: 10.2478/kbo-2023-0072.

[28] Bryson Medlock. *The dark side: How threat actors are using AI | ConnectWise*. Nov. 2024. URL: https://www.connectwise.com/blog/the-dark-side-how-threat-actors-are-using-ai.

[29] Steve Morgan. *Cybercrime to cost the world 10.5 trillion annually by 2025*. Nov. 2024. URL: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/.

[30] Juniper Networks. *Understanding dual stacking of IPv4 and IPv6 unicast addresses*. Dec. 2024. URL: https://www.juniper.net/documentation/us/en/software/junos/is-is/topics/concept/ipv6-dual-stack-understanding.html.

[31] Nicki Skafte Detlefsen et al. *TorchMetrics - Measuring Reproducibility in PyTorch*. Feb. 2022. DOI: 10.21105/joss.04101. URL: https://github.com/Lightning-AI/torchmetrics.

[32] Nvidia. *GPU-optimized AI, Machine Learning, & HPC Software: NVIDIA NGC*. 2025. URL: https://catalog.ngc.nvidia.com/.

[33] OISF. *Suricata*. `https://suricata.io/`. An open source Network Intrusion Detection System. 2024. URL: `https://suricata.io/`.

[34] Vern Paxson. "Bro: A System for Detecting Network Intruders in Real-Time". In: *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. Ed. by Aviel D. Rubin. USENIX Association, 1998. URL: `https://www.usenix.org/conference/7th-usenix-security-symposium/bro-system-detecting-network-intruders-real-time`.

[35] Joakim Skarding et al. "Corrections to "A Robust Comparative Analysis of Graph Neural Networks on Dynamic Link Prediction"". In: *IEEE Access* 12 (2024), pp. 6912–6913. DOI: `10.1109/ACCESS.2023.3336229`. URL: `https://doi.org/10.1109/ACCESS.2023.3336229`.

[36] Xiaoqing Sun and Jiahai Yang. "HetGLM: Lateral Movement Detection by Discovering Anomalous Links with Heterogeneous Graph Neural Network". In: *IEEE International Performance, Computing, and Communications Conference, IPCCC 2022, Austin, TX, USA, November 11-13, 2022*. IEEE, 2022, pp. 404–411. DOI: `10.1109/IPCCC55026.2022.9894347`. URL: `https://doi.org/10.1109/IPCCC55026.2022.9894347`.

[37] Xiaoqing Sun and Jiahai Yang. "HetGLM: Lateral Movement Detection by Discovering Anomalous Links with Heterogeneous Graph Neural Network". In: *IEEE International Performance, Computing, and Communications Conference, IPCCC 2022, Austin, TX, USA, November 11-13, 2022*. IEEE, 2022, pp. 404–411. DOI: `10.1109/IPCCC55026.2022.9894347`. URL: `https://doi.org/10.1109/IPCCC55026.2022.9894347`.

[38] Shahroz Tariq et al. "Alert Fatigue in Security Operations Centres: Research Challenges and Opportunities". In: *ACM Comput. Surv.* 57.9 (Apr. 2025). ISSN: 0360-0300. DOI: `10.1145/3723158`. URL: `https://doi.org/10.1145/3723158`.

[39] Conifers team. *AI-Powered SOC: The Definitive Guide for 2025*. July 2025. URL: `https://www.conifers.ai/blog/ai-powered-soc`.

[40] Cybereason Team. *The silent epidemic: Uncovering the dangers of alert fatigue and how to overcome it*. 2025. URL: `https://www.cybereason.com/blog/the-silent-epidemic-uncovering-the-dangers-of-alert-fatigue-and-how-to-overcome-it`.

[41] Nikos Virvilis, Bart Vanautgaerden, and Oscar Serrano Serrano. "Changing the game: The art of deceiving sophisticated attackers". In: *2014 6th International Conference On Cyber Conflict (CyCon 2014)*. 2014, pp. 87–97. DOI: `10.1109/CYCON.2014.6916397`.

[42] Isaac Wiafe et al. "Artificial Intelligence for Cybersecurity: A Systematic Mapping of Literature". In: *IEEE Access* 8 (2020), pp. 146598–146612. DOI: `10.1109/ACCESS.2020.3013145`. URL: `https://doi.org/10.1109/ACCESS.2020.3013145`.

[43] Ge Zhang et al. "eFraudCom: An E-commerce Fraud Detection System via Competitive Graph Neural Networks". In: *ACM Trans. Inf. Syst.* 40.3 (2022), 47:1–47:29. DOI: `10.1145/3474379`. URL: `https://doi.org/10.1145/3474379`.

[44] Meihui Zhong et al. "A survey on graph neural networks for intrusion detection systems: Methods, trends and challenges". In: *Comput. Secur.* 141 (2024), p. 103821. DOI: `10.1016/J.COSE.2024.103821`. URL: `https://doi.org/10.1016/j.cose.2024.103821`.

A

Apptainer definition file

```
Bootstrap: docker
From: nvcr.io/nvidia/pyg:24.07-py3

%files
    ./noether /usr/src/noether
    ./requirements.txt /usr/src/noether/requirements.txt

%post
    # Update APT
    apt-get update && apt-get upgrade -y

    # Update pip
    python -m pip install --upgrade pip

    # Install project requirements
    pip install -r /usr/src/noether/requirements.txt

%runscript
    python3 /usr/src/noether/run.py "$@"
```

Listing 8: Definition file for the Apptainer image

B

Experiment Command List

To improve the reproductivity of our research, we present the commands used for each experiment. Unless specified, the default arguments specified in Table B.1 are used.

Argument	**Description**	**Default value**
–model	Model to use	try2
–dataset	Dataset to use	UWF22
–bin-size	The size of bins used during training	10
–batch-size	The size of batches used during training	450
–dropout	The dropout rate used	0.0
–account-for-duration	Whether extra records should be added to account for connections bigger than the bin size	False
–link-prediction-only	Whether to only concentrate on link prediction	False
–shuffle	Whether the dataset should be shuffled	False
–shuffle-type	The type of shuffling to be applied	random
–shuffle-bin-size	The size of the bins used during the shuffling process	0.1
–shuffle-every-time	Whether to reshuffle the dataset if it has already been shuffled previously	True
–max-epochs	Maximum number of training epochs	10
–devices	The amount of devices used by the Trainer	auto
–accelerator	The accelerator used for training	auto
–num-workers	The amount of workers used by the dataloader	0
–dataset-folder	Folder to store all datasets	/data/datasets
–trainer-folder	Root folder for the Torch Lightning trainer	/data

Table B.1: Default arguments table

Thus, the commands used to produce the results are outlined in Listing 9.

```
# Try2 tests with shuffling
submit.sbatch --accelerator gpu --model try2 --dataset UWF22 --shuffle
--shuffle-type day --bin-size 5 --batch-size 100
submit.sbatch --accelerator gpu --model try2 --dataset UWF22 --shuffle
--shuffle-bin-size 0.1 --batch-size 100
submit.sbatch --accelerator gpu --model try2 --dataset UWF22 --shuffle
--shuffle-bin-size 1 --batch-size 100
submit.sbatch --accelerator gpu --model try2 --dataset UWF22 --shuffle
--shuffle-bin-size 10 --batch-size 100
submit.sbatch --accelerator gpu --model try2 --dataset UWF22 --shuffle
--shuffle-bin-size 400 --batch-size 100

# Try0 tests without shuffling
submit.sbatch --accelerator gpu --model try0 --dataset UWF22 --no-shuffle

# Try1 tests without shuffling
submit.sbatch --accelerator gpu --model try1 --dataset UWF22 --no-shuffle
--batch-size 100
submit.sbatch --accelerator gpu --model try1 --dataset UWF22Fall
--no-shuffle
submit.sbatch --accelerator gpu --model try1 --dataset UWF24 --no-shuffle

# Try2 tests without shuffling
submit.sbatch --accelerator gpu --model try2 --dataset UWF22 --no-shuffle
--batch-size 100
submit.sbatch --accelerator gpu --model try2 --dataset UWF22Fall
--no-shuffle
submit.sbatch --accelerator gpu --model try2 --dataset UWF24 --no-shuffle
```

Listing 9: Commands used for testing