Detection of network attacks using Graph Neural Networks

Final Master's Thesis

Master's degree in Advanced Telecommunication Technologies (MATT) and Master's degree in Cybersecurity (MCYBERS)

Author: Guillermo Cobo Arróniz
Supervisor: Pere Barlet-Ros

January 2023

Acknowledgements

Firstly, I would like to thank Marcos Postigo-Boix and Eva Rodríguez Luna, the directors of the Master's degree in Advanced Telecommunication Technologies (MATT) and the Master's degree in Cybersecurity (MCYBERS), for allowing me to pursue both studies simultaneously between September 2021 and January 2023.

Secondly, and without a doubt, I would like to thank Pere Barlet-Ros, my master's thesis supervisor, for having trusted me to work on such a novel and fascinating topic, the synergy between Cybersecurity and Artificial Intelligence.

Thirdly, I would like to thank the ETSETB secretariat, whose guidance and help were essential throughout the past year to make this combination of studies possible.

Finally, I would like to thank the UPC-ETSETB and the UPC-FIB, for the past years, including the bachelor's degree, as this journey has been life-changing.

Abstract

As the number of cyber-attacks targeting organisations has exponentially grown, so has the importance of securing computer networks. Network Intrusion Detection Systems (NIDS) play a crucial role in this critical task, as they monitor the incoming and outgoing traffic from a network to detect any potential threats.

To make these security systems more robust, a lot of research has been conducted on using advanced Machine Learning (ML) and Deep Learning (DL) techniques for network intrusion detection. Despite the publication of many proposals, commercial systems have failed to implement solutions which are mainly based on artificial intelligence.

The purpose of this master's thesis has been to study the potential of Graph Neural Networks (GNNs) in the detection of network attacks. GNNs are a branch of Neural Networks (NNs) that operate on graph-structured data, which implies a paradigm shift with respect to how traditional NNs analyse information.

GNNs have been applied to many domains of knowledge such as chemistry or physics, however, there is little research on the field of security and networking. Nevertheless, as network traffic can naturally be modelled as a graph, where there can be many types of nodes (devices) and edges (links), this novel branch of DL can bring huge value.

Table of Contents

List of Figures

List of Tables

List of Source Codes

1. Introduction

1.1 Structure of the memory

The main purpose of this subchapter is to elaborate on the main blocks that compose this memory.

In the first chapter, we primarily focus on what problem we were trying to solve with this master's thesis, the motivations that lead to the implementation of the project itself and the main objectives and goals that we aimed to obtain.

The second chapter is dedicated to introducing the domain of Intrusion Detection Systems (IDS). In particular, we will focus on Network Intrusion Detection Systems (NIDS), and how these play a key role when it comes to securing networks. Moreover, we will also talk about some limitations that NIDS currently present.

In the third chapter, we will introduce the topic of Graph Neural Networks (GNNs). The purpose of this chapter will be to provide the reader with some basic knowledge, if unfamiliar with the topic, to properly understand the following chapter, which will elaborate on the implementation of the actual project.

As mentioned above, in the fourth chapter, we will present the implementation of a GNN model for the detection of network attacks. We will talk about the architecture of the model, the data used for training and evaluation, the different experiments and hypotheses tested, and finally we will present the results obtained.

In the fifth chapter, we will present the conclusions of the project, and in the last chapter, we will elaborate on future research that can be done to continue this line of work.

1.2 Problem to solve

NIDS have evolved to play a key role when it comes to securing a network, despite these systems being generally very robust, they are still vulnerable to the new variants of malware developed. That is why, in the last few years, there have been many proposals for using advanced ML and DL techniques to detect network attacks.

Nevertheless, commercial systems have failed to implement artificial intelligence as their core detection engine, because most of the proposed implementations do not have reliable results in production environments. One of the main reasons behind this problem is that the developed models lack generalisation, meaning that when tested in unseen scenarios or different network setups, the prediction capabilities drastically drop. On top of that, current systems face a fundamental limitation, they analyse flows individually, which implies obviating the existing inter-dependencies between them.

1.3 Objectives and goals of the project

The main purpose of this project has been to study the potential of GNNs in network attack detection. GNNs are a novel family of traditional Neural NNs, which start from a different conceptual basis, they operate over graph-structured data.

Although GNNs have been researched in topics such as chemistry or physics, their application in the domain of cyber security is still under observation. That is why, one of the main contributions of this project has been to shed a light on the application of GNNs to network attack detection, to increase trust in their application and incentivise further research.

2. Intrusion Detection Systems (IDS)

2.1 Overview

When it comes to securing a network, traditionally, we talk about *access control*[1] mechanisms (which can be a policy in a firewall, credentials, authentication and authorization, or a rule in a router, for instance), however, most of these policies can not provide full security coverage of an entire computer network. Hence, to protect the systems of intrusions[2], we need more complex setups.

When we talk about an intrusion, we are talking about a set of actions which try to compromise the integrity, confidentiality and availability of a resource, which does not have to imply unauthorised access to a machine, but can also be the denial of a service.

The systems used to detect intrusions are called Intrusion Detection Systems (IDS) [1], which are either devices or software in charge of analysing multiple data points from different sources, to detect any malicious activities or threats. IDSs are not a new concept, as one of the first appearances dates from 1980.

These systems are in charge of monitoring the network traffic, among other things, to look for malicious or suspicious activities and generate alerts if necessary. Detected threats can be reported to a network administrator, for instance, or collected using a Security Information and Event Management (SIEM) system.

2.2 Types of Intrusion Detection Systems (IDS)

There are different types of IDS systems, which vary according to the setup where they are deployed. Those deployed in a particular host are known as *host-based*, whereas those deployed in a computer network are known as *network-based.* Both types have advantages and disadvantages, however, for the scope of this project, although we will introduce *host-based* systems, we will focus on the *network-based* type.

[1] https://www.incibe-cert.es/en/blog/basic-access-control

[2] https://www.sunnyvalley.io/docs/network-security-tutorials/what-is-network-intrusion

2.2.1 Host Intrusion Detection System (HIDS)

Host Intrusion Detection Systems (HIDS) run on hosts, instead of in an entire network. They inspect incoming and outgoing packets within a host and can generate alerts and reports if any suspicious or malicious activity is detected.

A HIDS can perform multiple security tasks, such as:

- Monitoring critical files.
- Notifying intrusions.
- Evaluating traffic.
- Threat intelligence (recognizing and eliminating malicious activities present within the system).

The main limitation concerning these types of systems is that the visibility they have is limited to a single host, which can be useful to maximise its security, but lacks a general overview of the entire network and the rest of the devices or hosts.

Moreover, as they are deployed within a host, they can consume resources from the machine, potentially impacting performance. On top of that, an attack will only be detected once it affects the host, jeopardising the security of the entire network.

2.2.2 Network Intrusion Detection System (NIDS)

NIDS are deployed within a network with the purpose to examine the traffic generated from all devices and machines in the network. They observe incoming and outgoing traffic and can use different techniques to find malicious traffic, such as comparing signatures from known attacks, looking for anomalies in the data, scanning ports, malformed packets, etc.

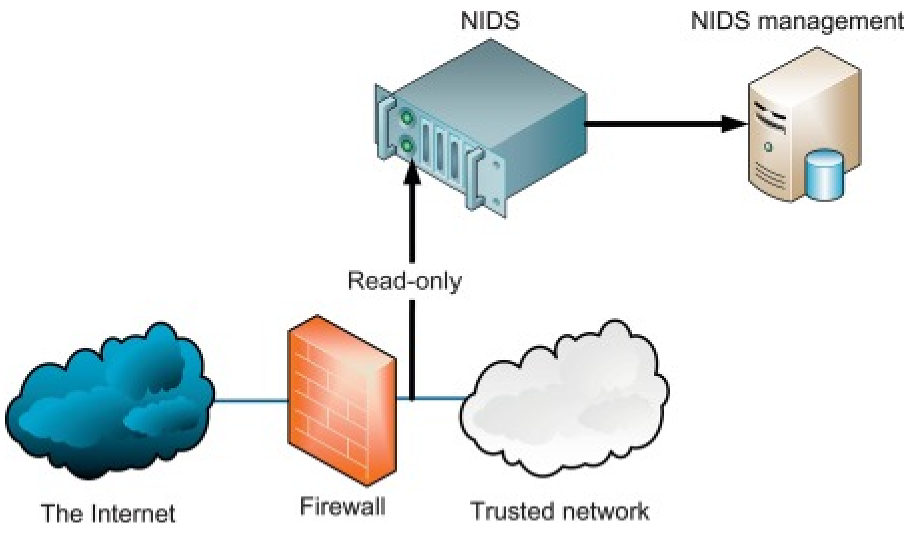

Figure 2.1: Network Intrusion Detection System (NIDS) deployment.

A NIDS works like a passive network monitoring device, as it does not interfere with the inspected traffic and does not ingest new packets into the network. As can be seen in *Figure 2.1* above, it is normally deployed within the trusted network (behind a firewall), and it basically captures the traffic and evaluates it.

The traffic that the NIDS uses as input can vary according to the network size and system type, however, in a scenario of a considerable network, it can use sampled *NetFlow*[3] records extracted from a router, for instance.

It is common to wonder what is the difference between a firewall and a NIDS, and even to question if they are both compatible or if we should choose one or another. In fact, they work together, and they can be understood as the "security guard" and "intruder alarm" in a physical security setup.

A firewall is in charge of blocking or restricting certain traffic incoming to the network, it can be considered a filtering device that often works on a set of predetermined rules. Whereas a NIDS, as already explained above, monitors the traffic that has already entered the network and inspects it to look for potential threats. Hence, these two types of security systems must work together, as they have different duties.

[3] https://es.wikipedia.org/wiki/Netflow

2.3 Detection methods

An IDS can leverage different detection methods to find threats or policy violations. There are mainly two types: *signature-based*, which works on the basis of looking for patterns of already known attacks, and *anomaly-based*, which relies on modelling the normal behaviour of the network and looking for events which differ from this normality.

2.3.1 Signature-based

As the name suggests, this detection technique is based on signatures[4] of known attacks, which means that the patterns found in the data are cross-checked against records of already-seen attacks. This method is similar to antivirus software, which leverages a database or record of past attacks to match any potential existing threats.

For instance, if certain IP addresses are blacklisted, a *signature-based* IDS will detect any traffic from these addresses and raise the corresponding alerts. Or if an attack matches a well-known pattern, such as a flooding attack, the IDS will recognize this known pattern and act accordingly.

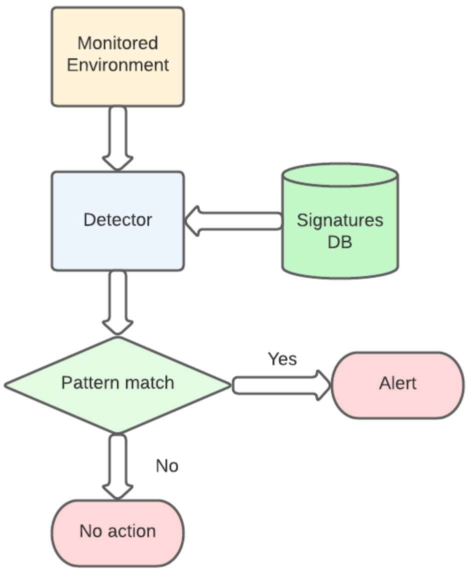

Figure 2.2: Signature-based detection method flowchart.

[4] https://encyclopedia.kaspersky.com/glossary/attack-signature/

Signature-based IDS have the main advantage that they are robust against known malware, however, it is clear to see its main limitation, they are vulnerable to zero-day attacks or any type of malware not recorded or known.

2.3.2 Anomaly-based

The fundamental basis of this detection method is to identify anomalous behaviours in the inspected traffic, which was introduced as a new detection method due to the rapid evolution of malware attacks. These systems mainly use ML techniques to create a trustful behaviour model of the network, and when something differs from the established model, it can be treated as an anomaly or a potential threat.

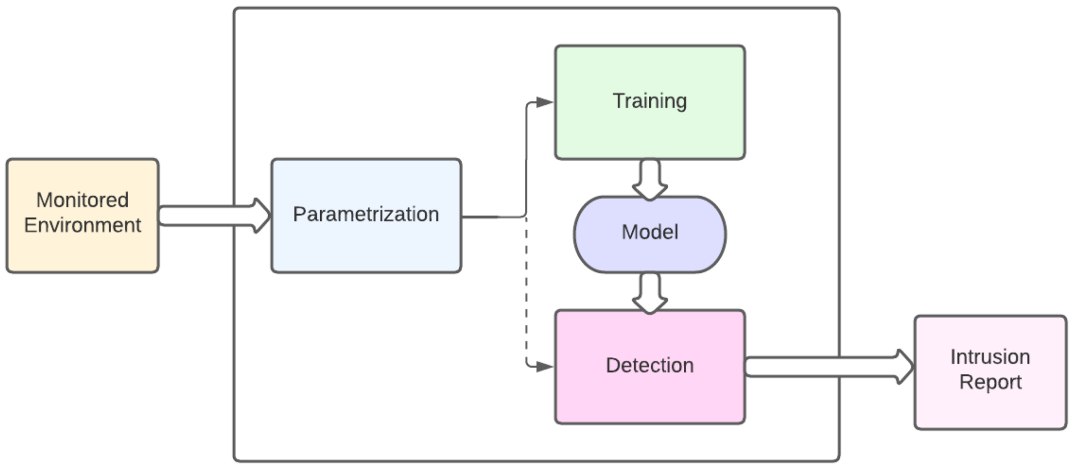

Figure 2.3: Anomaly-based detection method flowchart.

As can be seen in *Figure 2.3* above, they work like most ML models. Initially, the monitored environment is parametrized and data is collected to create training datasets. Once the ML model is built, it can be used on validation data to test its performance against unseen scenarios.

This process has to be iteratively repeated, as it is not enough to train a model once, we have to keep it updated by periodically feeding it with new training datasets. So, even if we achieve high performance with certain data, our model may not be as accurate when tested with totally different data, or may also become obsolete very fast.

Hence, *anomaly-based* models may prove to perform well in a test environment, but face problems when exposed to production data. This could mainly happen due to three different reasons:

- **Lack of generalisation**: these models lose prediction capabilities when exposed to new network scenarios and traffic.
- **Overfitting**: most proposals present very high classification results, but this is normally due to overfitting of the training datasets, which makes the model vulnerable to variations of attacks or new types of malware.
- **Features:** these models may be based on a certain set of features which are network dependent and can not be extrapolated to other setups and conditions.

The main challenge faced is to develop an *anomaly-based* system capable of performing well in any type of environment. To achieve this, we need to find a solution which tackles the three aforementioned limitations.

This means that we need to carefully select a set of features which can be applied in different network scenarios and build a model that does not create overfitting with the training data. That is why we proposed the use of GNNs, as these are based on certain theoretical concepts which could prove to solve these limitations (more on chapters 3 and 4).

3. Graph Neural Networks (GNNs)

3.1 Graph-structured data

Prior to talking about what are GNNs, it is essential to make a brief introduction to what are graphs[5] and why are these so important to represent many types of data.

A graph can be mathematically defined as $G=(V, E)$ where V = Vertex (or node) attributes and E = Edges (or link) attributes and directions. It is basically a structure that contains nodes and edges which interconnect these. Both nodes and edges can have features that describe them.

There are three main types of graphs: undirected, directed and weighted.

[5] https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

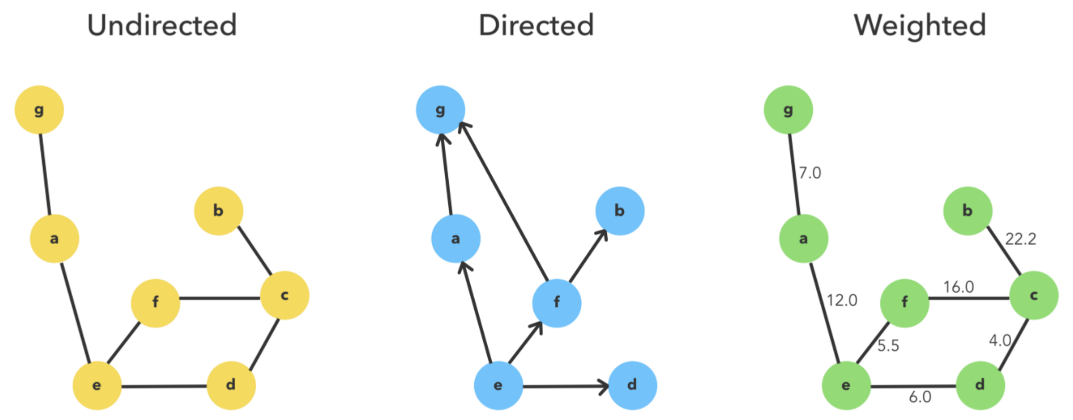

Figure 3.1: Types of graph structures.

It turns out that most of the data found in nature is better represented as a graph, rather than as a set of individual vectors containing features. For instance, social relationships, chemistry molecules, maps, traffic, etc. Hence, analysing certain types of data with this type of structure can bring immense value.

3.2 History and Introduction

GNNs [2] were introduced back in the year 2005, as a novel branch in the field of DL and NNs. One of the first appearances of this technology was presented in the paper *A new model for learning in graph domains* [3].

GNNs were born as a result of the necessity to apply NNs over graph-structured data. The prior existing approaches coped with graphical data by applying a pre-processing phase where they transformed the graphs into sets of flat vectors.

However, this way, the relevant topological information could be lost and the final results biassed by this preprocessing stage. Despite GNNs being old, they have started being popular in the past five years due to some advancements that have increased their capabilities and expressive power.

Recent publications have shown the potential that GNNs can have in certain domains, such as network optimization [4], network modelling [5] and traffic forecasting [6].

We can do multiple types of classifications within a graph:

- **Graph-level:** we can leverage a GNN to classify an entire graph.
- **Node-level:** we can leverage a GNN to classify only certain nodes of the graph.
- **Edge-level:** we can leverage a GNN to predict new edges within a graph.

3.3 Types of Graph Neural Networks (GNNs)

There are currently many types of GNNs, and despite the fact that they operate over similar principles, they are all different. Some of the most common types will be explained in the following subsections. It is important to state that, for the scope of this project we will especially focus on Message Passing Neural Networks (MPNNs).

3.3.1 Graph Convolutional Network (GCN)

Graph Convolutional Networks (GCNs) [7] operate under the same principle as convolution layers in Convolutional Neural Networks (CNNs)[6]. Convolution refers to multiplying the input neurons with a set of weights that are typically known as *filters* or *kernels*. These *filters* act as a sliding window across an entire image and enable the CNNs to learn some features from the neighbour cells.

GCNs operate similarly, but in this case, the model learns the features by inspecting the neighbour nodes. GCNs can also be classified into two different types: spatial and spectral graph convolutional networks.

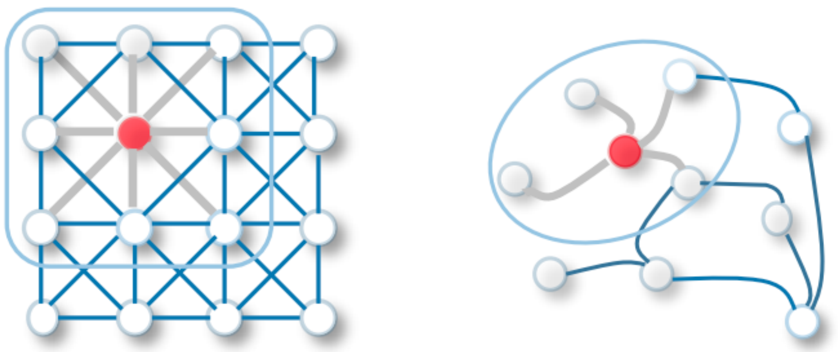

Figure 3.2: Applying convolution operations in a graph.

[6] https://en.wikipedia.org/wiki/Convolutional_neural_network

3.3.2 Graph Attention Network (GAT)

Graph Attention Networks (GATs) [8] are a neural network architecture that leverages masked self-attentional layers to address the drawbacks presented by graph convolutions or their approximations. A GAT enables specifying different weights to different nodes in a neighbourhood, by stacking layers in which the nodes are able to attend over their neighbour's features.

One of the crucial advantages that these present is that they don't require any kind of costly matrix operation, such as the inversion operation, or depend on knowing the structure of the graph upfront.

3.3.3 Message Passing Neural Networks (MPNNs)

We have mainly focused on MPNNs throughout this project, so we will provide a detailed explanation of how these work, so the reader can better understand the contents in chapter 4.

MPNNs were first presented in the year 2017, by J. Gilmer in the paper *Neural Message Passing for Quantum Chemistry* [9]. This effectively makes them a very novel technology, however, the last years have seen new domains of application for this type of GNNs, such as in the field of networks.

Its conceptual basis is simple: we start from a graph where there are relationships between the nodes (edges), and all of the nodes start with an initial *hidden state*. This *hidden state* is normally a vector of features (which can be padded if desired) that describes the state of the node.

$$h_i^0 = [x_0, ..., x_k, 0, 0, 0..., 0] \tag{1}$$

Then comes the core of the algorithm, which is known as the *message-passing phase*. During this phase, there will be *T* iterations throughout which the nodes will exchange *messages* with their corresponding neighbours. Let us explain what happens in each of these iterations.

An iteration is composed of three phases: exchanging the messages, aggregating the messages, and updating the hidden states of the nodes.

- Exchanging the messages

Nodes will "send" a message to all of their neighbours. The receivers of these messages will vary according to the direction of the edges. This means that in an undirected graph, messages will be exchanged in both directions of an edge, but in a directed graph, only in the way of the direction.

What is a message? In the simplest case, it can be the current hidden state of the sender, for instance. However, it can also be the result of applying a mathematical function to the hidden state of the receiver node, the hidden state of the sender node, and the features of the edge that connects both nodes.

$$M_t(h_v^t, h_w^t, e_{vw}) \tag{2}$$

After exchanging all of the messages, each node has received as many messages as neighbours it has. As can be seen in *Equation 2* above, each message contains information related to the one-to-one relationship of the nodes, as it combines their hidden states and edge features.

- Aggregation phase

Once each node has received all of the neighbour messages, it aggregates them. This can be done via simple mathematical operations, such as the sum, the mean or the average.

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \tag{3}$$

- Update phase

Finally, each node updates its hidden state by applying a mathematical function to the result of aggregating all messages and their current hidden state. This basically results in the new current state for the next iteration phase of the algorithm.

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \tag{4}$$

So, we can infer that, after one iteration, the hidden state of each node has been updated as a combination of the hidden states of all of its neighbours, effectively "accumulating" neighbouring information in each of the nodes. In one iteration, at most, a node will end up containing information about the nodes that are one step away (depth = 1). However, if we repeat this process *T* times, each node will end up containing information on the neighbours of its direct neighbours.

Why is this so powerful? Because depending on the graph structure, throughout the iterations, we will end up in a situation where nodes have information about their neighbours, hence their status will be determined by their position in the graph.

Finally, once the *message-passing phase* has finished, we will have to make a classification with the obtained structure, either classifying the entire graph or the individual nodes. This will be called the readout phase.

- Readout phase

After the *T* iterations, all nodes will contain information from their neighbours, so if we want to classify them, we can simply apply a traditional NN, which will be known as the *readout function*. In this phase, we will have the same situation as with traditional DL, as we will need to define the number of input neurons, output neurons, hidden states, etc.

$$\hat{y} = R(\{h_v^T | v \in G\}) \tag{5}$$

So, as the reader might be able to infer, the previous *message-passing phase* has enabled all hidden states of the nodes to contain information about their neighbours and the neighbours of their neighbours. Effectively conditioning their state depending on the states of others. This is very powerful, as the graph structure will directly affect the status of all nodes.

3.4 Development frameworks

There are a few existing frameworks for the development of GNN models. Although these could be implemented in C or C++ (as most of the core ML and DL libraries), the most common language for rapid modelling and design is Python.

For PyTorch[7] implementations, there is a framework known as PyG[8] (also known as *torch-geometric)* [10], which is very useful for starting to develop GNN models.

It provides a data structure called *Data* which enables the definition of a graph structure using torch tensors. In fact, these models use graphs as input data, but these graphs are represented as a set of float tensors.

For instance, to define a graph, we need:

- **Features tensor:** which has N x M dimension. Where N represents the number of nodes in the graph and M is the size of the tensor that represents the state of each node.
- **Edge index:** this is a tensor in the coordinate format (COO), that has dimension 2 x N. Where N represents the number of edges in the graph.

[7] https://pytorch.org/

[8] https://github.com/pyg-team/pytorch_geometric

As an illustrative example, the following graph would be represented as:

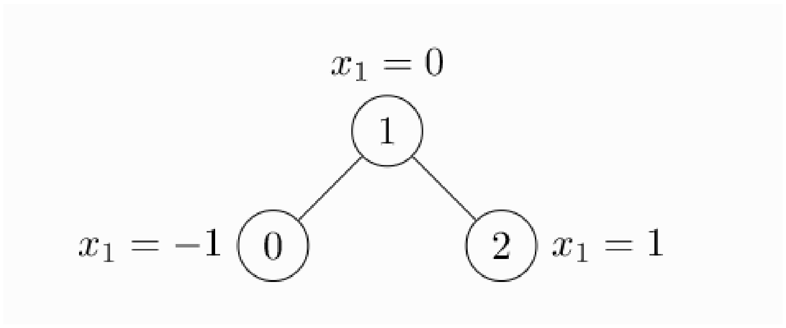

Figure 3.3: Undirected graph.

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1],
                           [1, 0],
                           [1, 2],
                           [2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index.t().contiguous())
>>> Data(edge_index=[2, 4], x=[3, 1])
```

Source code 3.1: Creating a Data (graph) object.

When developing this project we found some limitations with the PyG framework. Mainly due to the fact that for more custom implementations of the *message-passing* algorithm, it is limited and presents some errors. This framework is good for starting to learn how to work with graphs and how to structure the data to be fed into the model.

Custom implementations (without specific GNN frameworks), can be done both with PyTorch and Tensorflow[9] (as any NN model).

[9] https://www.tensorflow.org/?hl=es-419

4. Implementation of a GNN model to detect network attacks

The previous chapters contained contextual information regarding the two main topics involved in this project, Network Intrusion Detection Systems (NIDS) and Graph Neural Networks (GNNs).

Acquiring knowledge on these two topics was necessary for the actual development and implementation of a GNN model capable of detecting network attacks. Throughout the next sections, we will elaborate on the different components involved in the development stage.

This project has been based on the paper *Unveiling the potential of Graph Neural Networks for robust Intrusion Detection* [11]. However, our approach has been different, as we have proposed a new way of modelling the network traffic and the designing of the model (training, evaluation, features, etc.). This research has mainly focused on increasing the explainability of the technology, in order to incentivize further research.

4.1 Datasets

As with any ML or DL implementation, the training and validation datasets play a crucial role, if not the most important one. A model can only be as good as the data it trains on, as, even if the model is perfectly optimised if the data is not realistic or does not prepare the model for generalisation, it becomes useless.

In the field of security, it is particularly challenging to find public datasets which contain relevant data. Production (or "real" data datasets) can contain sensitive information, such as IP addresses, hence open-sourcing them can imply privacy risks. Some of them are transformed to mask some of the data fields, however, throughout the process, they can lose relevant information.

Moreover, most companies are not willing to share their own data, especially if it contains attack records that have affected them. For this reason, it becomes challenging to access production data, hence we need to leverage "lab" data. This basically relates to synthetic datasets that have been specifically built in controlled environments.

Although these datasets are a good starting basis, they still can not be compared to real data which can be way more complex and contain much more data points.

4.1.1 CIC-IDS 2017

One of the few available security datasets is called *CIC-IDS 2017* [12]. As its name suggests, this dataset was created in the year 2017, by the University of Brunswick (UNB) and the Canadian Institute of Cybersecurity (CIC). It contains benign traffic and up-to-date network attacks.

To give some context, in order to generate the benign traffic, they used a previously proposed *B-Profile system (Sharafaldin, et al. 2016)*[10] which profiles the abstract behaviour of human interactions and generates naturalistic background traffic.

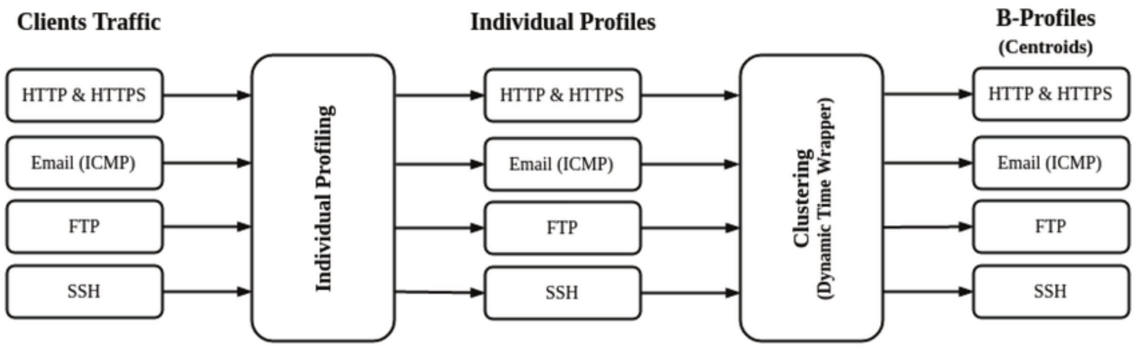

Figure 4.1: Benign profiling design.

In this particular case, the behaviour of twenty-five different users was abstracted based on the use of HTTP, HTTPS, FTP, SSH and email protocols. On the other hand, the attacks were generated using different tools, which can emulate PortScan attacks, Denials of Service, Distributed Denials of Service, etc.

In contrast to most datasets used for ML and DL purposes, network traffic needs to be processed prior to generating a set of CSV files which can be used to train a model.

Adding a little bit of networking background, we can capture the traffic that traverses a network and dump it into *pcap*[11] files. These files basically contain some core information that describes the packet captures (source IP, source Port, destination IP, destination Port, etc).

[10] https://www.researchgate.net/figure/Benign-profiling-design_fig2_318286637

[11] https://www.reviversoft.com/en/file-extensions/pcap

These files can become huge in size, and in their pure raw format are not very useful for an ML model. So they are processed to create flow records which can contain features on each flow, such as flow duration, packets sent, inter-arrival times, etc. A flow can be described as an identifier of the traffic exchanged between a source and a destination (considering the IP and the Port).

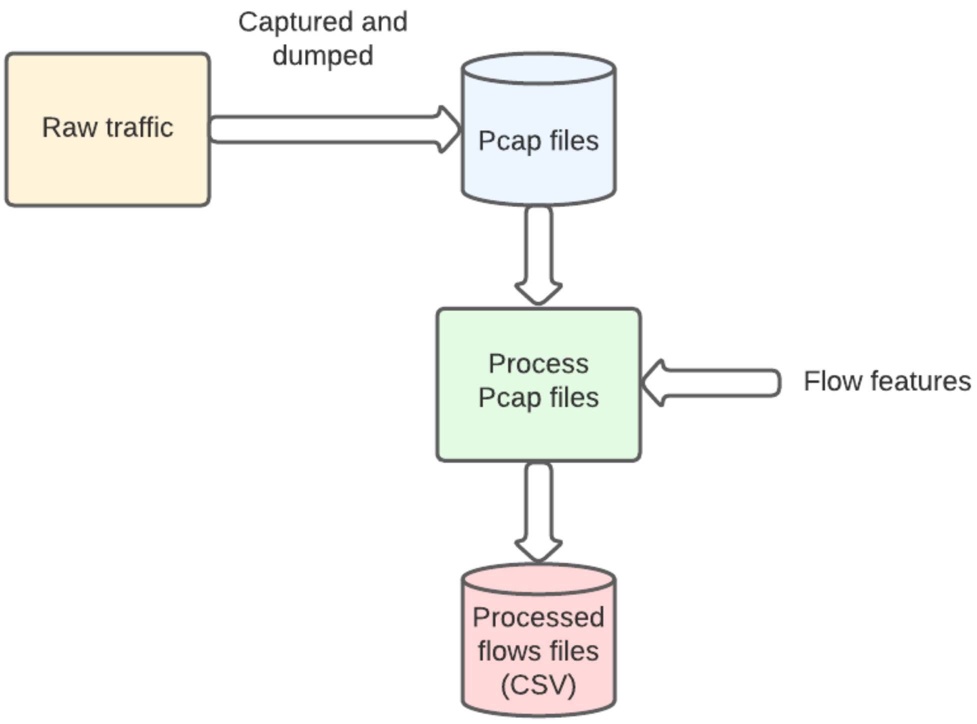

Figure 4.2: Flow diagram - Extracting flows from raw traffic.

For this dataset, the processor they used to extract the flow data is called *CIC-FlowMeter* [13] and was developed also by them. Why is it important to know all this information on how the data was generated? Basically because if we build a model with certain features and through certain training and validation datasets, we are going to need the same format always, hence in the case that we want to test our model with unseen data, it must have been generated through a processor that extracts the same exact features from the *pcap* files.

This dataset contains 80 network flow features from the generated network traffic.

4.2 Pre-processing stage

As with most ML and DL datasets, we need to apply certain pre-processing techniques to clean the data. In this particular case, we first removed flows which had negative or equal to zero flow durations, as these were invalid values, and we then proceeded to normalise the data.

4.3 GNN (MPNN) model

Our main purpose was to develop a GNN model (MPNN-based) capable of classifying network flows as benign or malign (or specifying a certain network attack). We refer to a network flow as a unique identifier of a connection between two machines and which can be described with the tuple *(Source IP, Source Port, Destination IP, Destination Port).*

We could have also decided to classify IP addresses or Ports, but taking into account that a certain machine might be simultaneously generating malign and benign traffic via different ports, we decided to take the other approach.

The overall idea was to be able to emulate the behaviour of an anomaly-based NIDS, i.e, being able to determine which flows are malicious based on certain network flow graphs.

4.3.1 Graph representations

As it can be inferred, the graph-structured data needs to be representative of the network events in order to provide value. This means that, even if we have a dataset, we need to design a graph representation which will be useful when classifying the network attacks against benign traffic.

The first approach was to build a graph structure with two different types of nodes:

- (IP, Port) nodes: where each unique tuple represents a node.
- Flow nodes: where each flow identifier between two (IP, Port) nodes tuples represents a node.

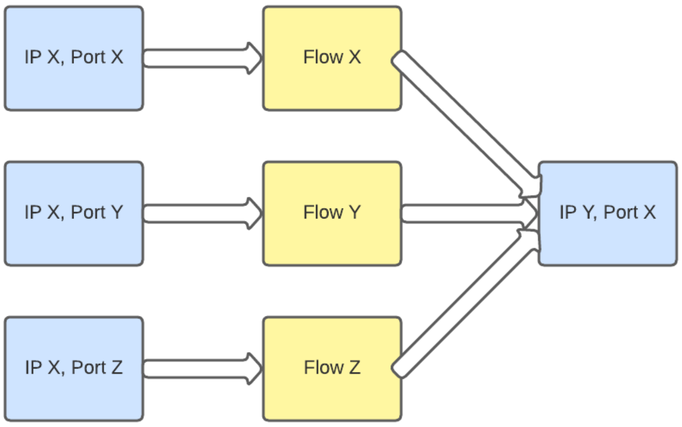

Figure 4.3: Graph representation with (IP, Port) and Flow nodes.

For instance, we could filter by one type of attack (such as a Distributed Denial of Service - DDoS), and visualise how this graph representation plays out.

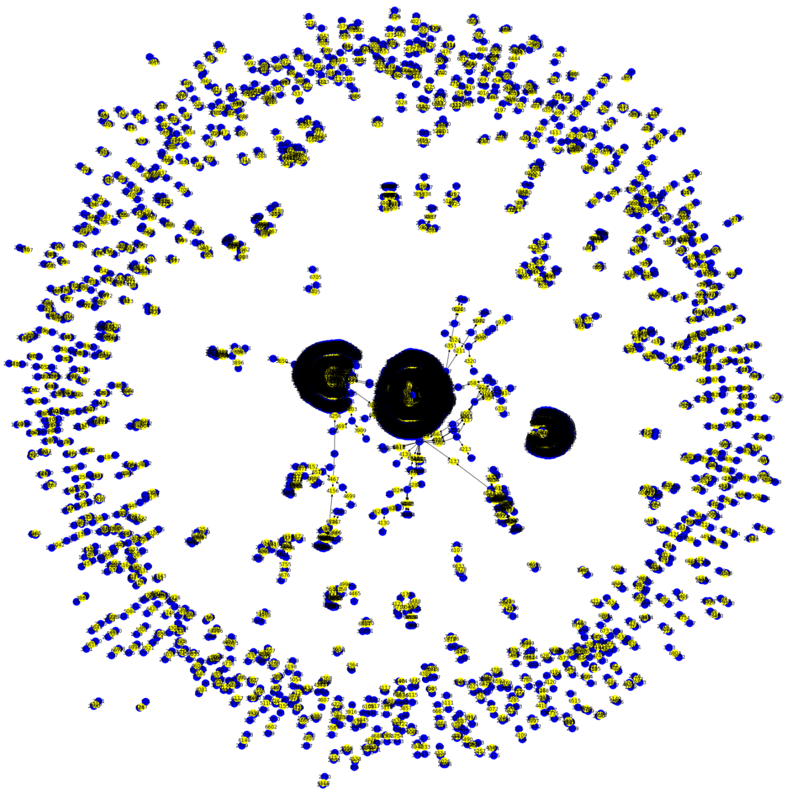

Figure 4.4: DDoS and Benign traffic graph representation - CIC-IDS 2017.

As can be seen from the previous capture, the graph representation of the dataset provides a more elaborate view of the traffic and the network attacks. We can clearly see a few clusters in the middle of the plot, which are directly related to the DDoS attacks, whereas there is a lot of unconnected traffic related to the benign behaviours.

In fact, we can actually zoom-in in on one of the clusters to see how this flooding attack appears to be structured:

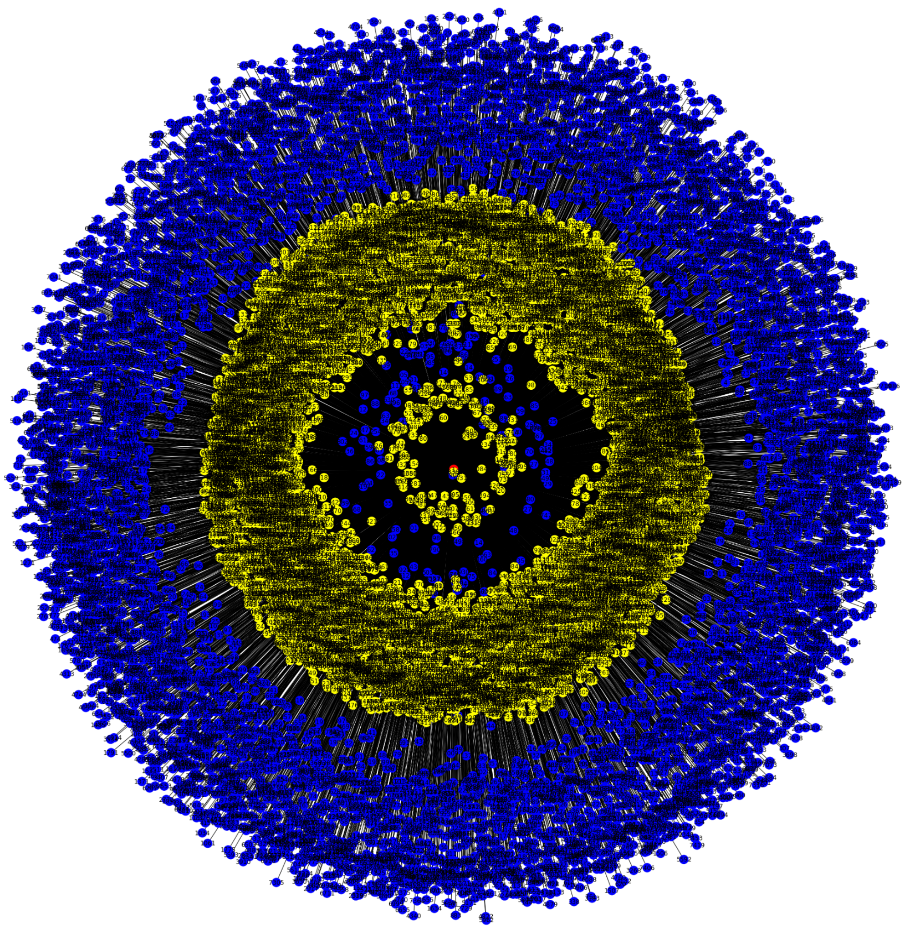

Figure 4.5: Flooding (DDoS) attack.

Although difficult to see, the red dot in the middle represents the victim (IP, Port 80), whereas all the other nodes represent (IP, Port) tuples targeting traffic to the HTTP port of the victim. By creating such a plot, we can already start to see the benefits of graphically analysing the network traffic, instead of analysing individually each network flow (as most AI models do).

Despite this illustration being representative, we found some limitations to it. For instance, we tried analysing the graphical differences between analysing a PortScan attack and a DDoS attack.

In theory, a PortScan attack should be represented as a lot of network flows targeting multiple ports in the same victim, whereas in a DoS or DDoS attack, we should see many network flows targeting a single port.

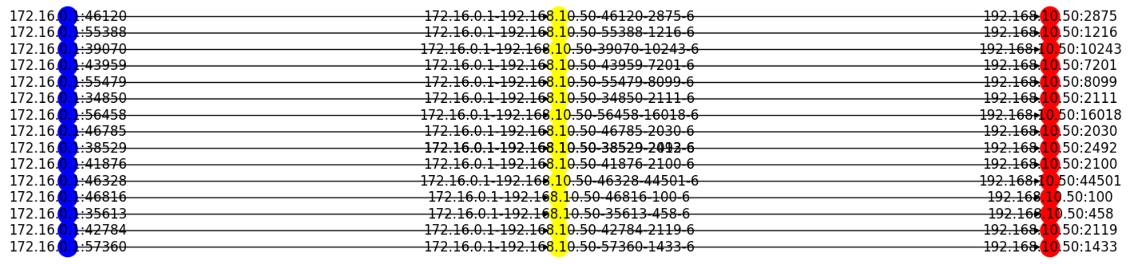

Figure 4.6: PortScan (reconnaissance) attack.

What we could see is that, indeed, in a PortScan attack, multiple ports from the same host were targeted, however, via this representation, there was no connection between them. Essentially it was like having many unrelated network flows. This is the opposite of what we are looking to achieve, as we aim to create representative graphs.

Another approach was to create two different types of nodes:

- IP addresses.
- Flow nodes.

However, the problem with this approach was that ports were not playing any role, hence representations of different attacks appeared to be the same.

That is why we came up with a novel approach, creating a graph representation that contained three different types of nodes:

- Ip addresses.
- (Ip, Port) tuples.
- Flow nodes.

This way, not only ports would be included in the graphs, but (IP, Port) nodes would be connected to the IP addresses nodes. Let us better illustrate how this looks:

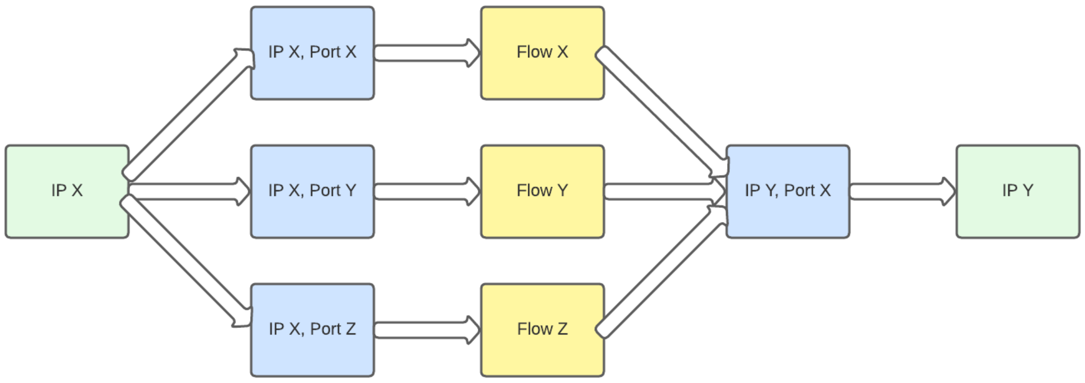

Figure 4.7: Graph representation with IP, (IP, Port) and flow nodes.

If we take a look at how this graph representation looks for two different attacks (PortScan and DDoS), we can see how it provides different graph structures that can be used in the GNN model (message passing implementation).

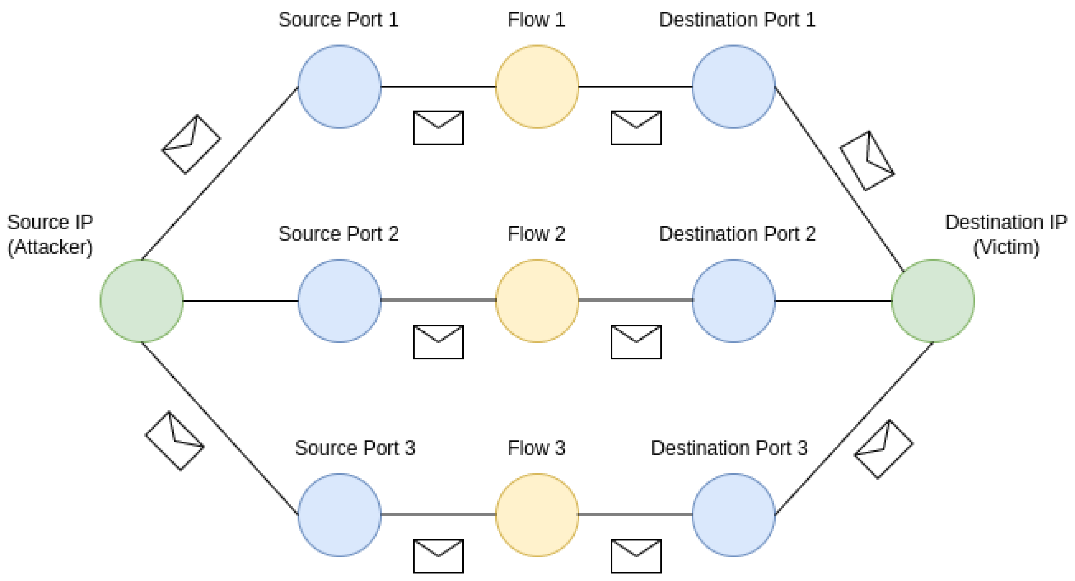

Figure 4.8: PortScan graph representation.

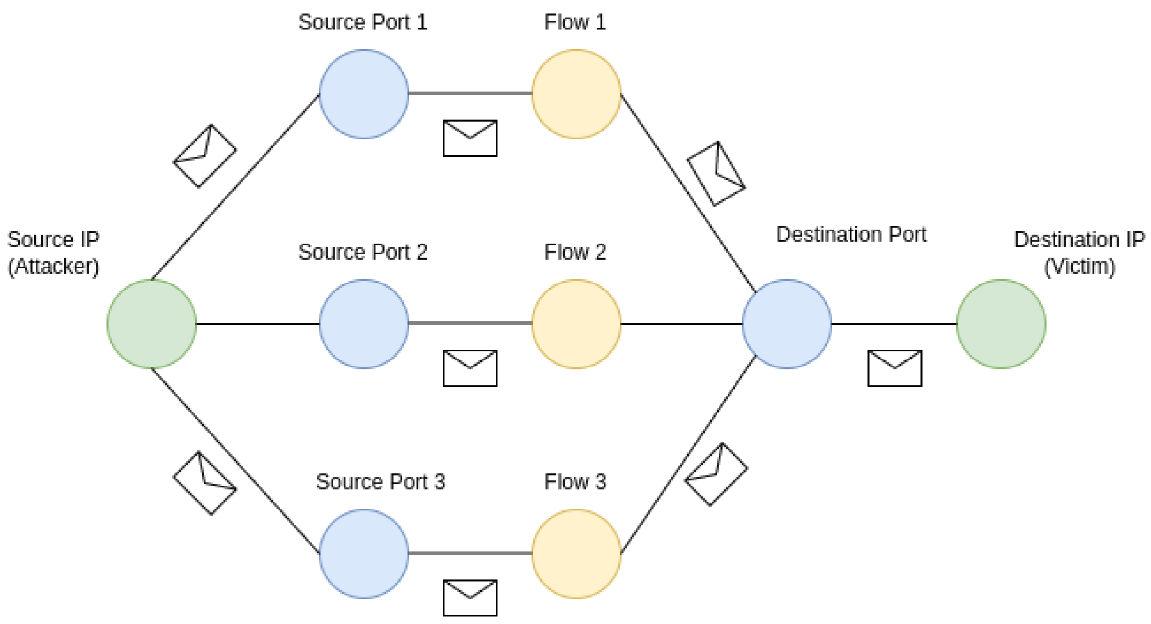

Figure 4.9: DDoS graph representation.

Figures 4.8 and *4.9* represent the theoretical structure that our graphs should have, however, we needed to translate these illustrations into code, so they could be used as input for training the model. For it, we used the *HeteroData*[12] class included in the PyG library.

As the reader may notice, the class name contains the "hetero" word. This means that our graphs are *heterogeneous*. A *heterogeneous* graph contains nodes of different types, whereas a *homogenous* graph contains nodes of the same type. In our case, it is clear to see that there are three different types of nodes.

To create the *HeteroData* object we first need to define the different types of nodes. In our case, we have:

- *host_ip_port*: which refers to the (IP, Port) nodes.
- *host_ip:* which refers to the IP nodes.
- *connection*: which refers to the network flow nodes.

For each of these node types, we can add the *features* tensor, which represents the hidden state of each of the nodes, and we can also add the *labels* tensor, which represents the labels that we add to the nodes. We only added labels to the *connection* nodes, as these were the ones we intended to classify.

[12] https://pytorch-geometric.readthedocs.io/en/latest/modules/data.html#torch_geometric.data.HeteroData

```
hetero_data = HeteroData()

# We add three different types of nodes
hetero_data["host_ip_port"].x = x_hosts_ip_port
hetero_data["host_ip"].x = x_hosts_ip
hetero_data["connection"].x = x_flows
hetero_data["connection"].y = y_flows
```

Source code 4.1: Adding nodes to an HeteroData object.

Once the node types have been defined, we need to specify the different edges the graph will have. According to the illustration of the graph representations previously attached, we have four different types of edges:

- IP node to (IP, Port) node.
- (IP, Port) node to connection node.
- Connection node to (IP, Port) node.
- (Ip, Port) node to IP node.

```
# We add four different types of edges
hetero_data["host_ip", "to",
            "host_ip_port"].edge_index = ip_to_port

hetero_data["host_ip_port",
            "to", "connection"].edge_index = port_to_flow

hetero_data["connection",
            "to", "host_ip_port"].edge_index = flow_to_port

hetero_data["host_ip_port",
            "to", "host_ip"].edge_index = port_to_ip
```

Source code 4.2: Adding edges to an HeteroData object.

4.3.2 Architecture

Our GNN model is a Message Passing Neural Network (MPNN), which, as explained before, is divided into two main blocks:

- *Iterations* phase, which contains in each iteration:
    - Message Function: Dense[13] Layer.
    - Aggregation Function: Mean.
    - Update Function: Gate Recurrent Neural Network[14].
- *Readout* phase:
    - Formed by a group of Dense Layers with a final Softmax activation[15].

```
self.message_func = nn.Sequential(
        nn.Dropout(self.dropout),
        nn.Linear(hidden_channels * 2, hidden_channels)
)
```

Source code 4.3: Message function.

```
self.update = nn.GRU(hidden_channels, hidden_channels)
```

Source code 4.4: Update Function.

```
self.readout = nn.Sequential(
        nn.Linear(hidden_channels, hidden_channels),
        nn.ReLU(),
        nn.Dropout(p=0.5),
        nn.Linear(hidden_channels, 64),
        nn.ReLU(),
        nn.Dropout(p=0.5),
```

[13] https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/

[14] https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

[15] https://en.wikipedia.org/wiki/Softmax_function

```
            nn.Linear(64, self.output_channels),
            nn.Softmax(dim=1)
    )
```

Source code 4.5: Readout Function.

Let us walk through the core of the model, the *message-passing* phase and the *readout* phase. As mentioned before, we have three different types of nodes in our graph. The connection nodes (network flows) are initialised with a set of features extracted from the dataset. Whereas the IP and (IP, Port) nodes are initialised with a tensor of all zeros.

As there are four different types of edges, there are going to be four different types of messages exchanged. Each of these edges contains a tensor in the COO format that maps all the neighbourhoods in that particular edge type.

In terms of code implementation, the message passing is implemented as follows:

- We parse the *edge_index* tensor that describes the edges between two different nodes.
- Through the dictionaries in the *HeteroData* object, we use the indexes that identify the nodes to create a concatenated features tensor.
- We apply the *message* and *aggregation* functions mentioned above to the tensor and leverage the *scatter module* to perform this process in one go.

The *scatter* module is a core component of the message-passing and aggregation steps, as it allows to perform all operations from within the same tensor.

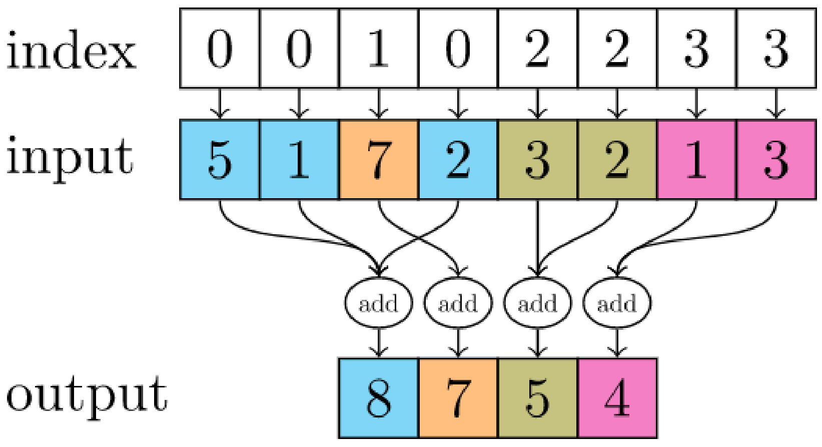

Figure 4.10: Scatter Function.

This illustration can help the reader better understand what is happening in the code. Let us say we have an *index* tensor, which contains the indexes of the message receiver nodes, and an *input* tensor which contains all the messages sent. By applying the scatter function we can effectively compute the aggregations of all messages targeting each receiver by simply operating through two tensors.

We can take a look at the actual implementation. Essentially we repeat the same operation as many times as the types of edges we have.

We first extract the indexes of the sender and receiver nodes from a certain edge type. We convert these two tensors into two tensors that contain the hidden states of the sender and receiver nodes. We then concatenate the tensors and apply the message function. Finally, we aggregate all of the messages via the de scatter module and update the hidden states of all the nodes.

```
for _ in range(T):
     # PART 1
     # Ip to Port
    ip_gather = h_ip[src_ip_to_port]
    port_gather = h_ip_port[dst_ip_to_port]
    nn_input = torch.cat((ip_gather,port_gather),dim=1).float()
    ip_to_port_message = self.message_func_ip(nn_input)
    ip_to_port_mean = scatter(ip_to_port_message,
                        dst_ip_to_port,dim=0,reduce="mean")

    # PART 2
    # Port to Ip
    port_gather = h_ip_port[src_port_to_ip]
    ip_gather = h_ip[dst_port_to_ip]
    nn_input = torch.cat((port_gather,ip_gather),dim=1).float()
    port_to_ip_message = self.message_func_ip(nn_input)
    port_to_ip_mean =scatter(port_to_ip_message,
                       dst_port_to_ip,dim=0, reduce="mean")

    # PART 3
    # Port to connection
    port_gather = h_ip_port[src_port_to_connection]
    connection_gather = h_conn[dst_port_to_connection]
    nn_input = torch.cat((port_gather, connection_gather),
                      dim=1).float()
    port_to_connection_message = self.message_func_ip(nn_input)
    port_to_connection_mean = scatter(
                          port_to_connection_message,
                               dst_port_to_connection, dim=0,
                               reduce="mean")

    # PART 4
    # Connection to port
    connection_gather = h_conn[src_connection_to_port]
    port_gather = h_ip_port[dst_connection_to_port]
    nn_input = torch.cat((connection_gather, port_gather),
                      dim=1).float()

    connection_to_port_message = self.message_func_ip(nn_input)
    connection_to_port_mean = scatter(
                                 connection_to_port_message,
                                 dst_connection_to_port, dim=0,
                                 reduce="mean")
    # PART 5
    # update nodes
    _, new_h_ip = self.ip_update(port_to_ip_mean.unsqueeze(
              0), h_ip.unsqueeze(0))  # (2, 128), (2, 128)
    h_ip = new_h_ip[0]
    _, new_h_conn = self.connection_update(
                 port_to_connection_mean.unsqueeze(0),
                 h_conn.unsqueeze(0))
    h_conn = new_h_conn[0]
    _, new_h_ip_port = self.ip_update(
              connection_to_port_mean.unsqueeze(0),
              h_ip_port.unsqueeze(0))
    h_ip_port = new_h_ip_port[0]


   _, new_h_ip_port = self.ip_update(
              ip_to_port_mean.unsqueeze(0),
              h_ip_port.unsqueeze(0))
   h_ip_port = new_h_ip_port[0]

return self.readout_nn(h_conn)
```

Source code 4.6: GNN model.

4.3.3 Hypotheses validations

Once the model was built, we focused on validating some hypotheses that had been pre-established:

- (H0) *The GNN model can work without features, solely based on the graph structures.*
- (H1) *The structure of the graph is relevant for the classification of the nodes.*
- (H2) *The GNN model is more resilient to attack variabilities than other traditional models.*

(H0) *The GNN model can work without features, solely based on the graph structures.*

We evaluated this hypothesis by initialising all nodes with a one's tensor and the model was not able to classify the network events. This basically led to the belief that graph structure is relevant when there is also features information to complement it.

(H1) *The structure of the graph is relevant for the classification of the nodes.*

To check this hypothesis we set up an experiment where we compared the performance of our MPNN model against a NN with the same structure as our readout function. Both models were trained with the same set of initial features.

The results of the experiment were:

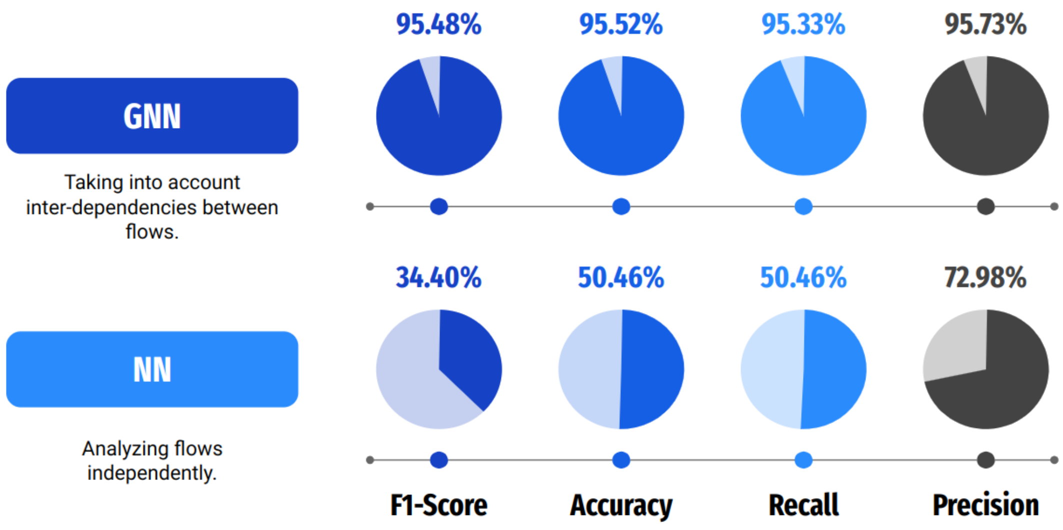

Figure 4.11: Classification metrics H1.

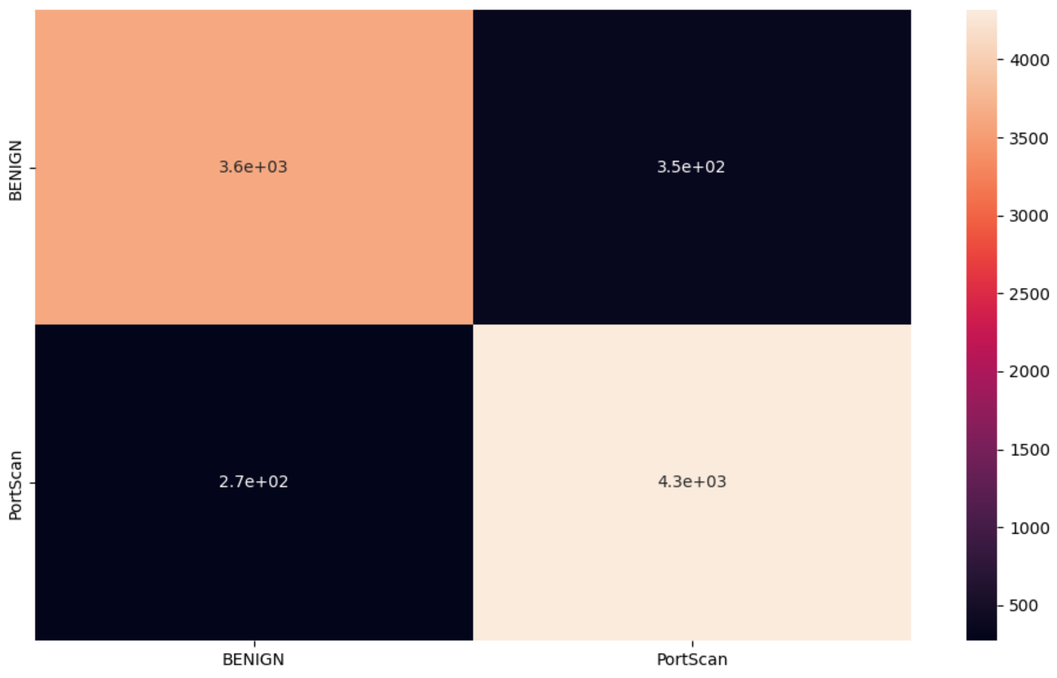

Figure 4.12: Confusion matrix - GNN.

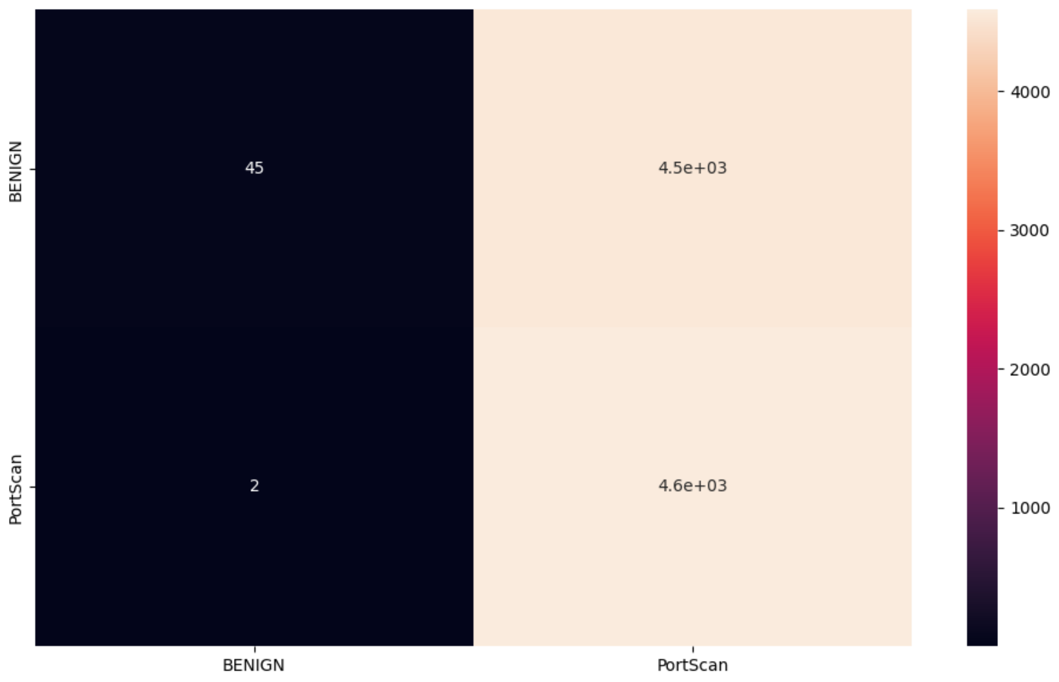

Figure 4.13: Confusion matrix - NN.

As can be seen, although using the same initial features, the GNN model, which takes into account inter-dependencies between flows, yielded better results compared to the NN. This basically means that the structure of the graph and the message-passing phase affected the classification results. If this were not relevant, the MPNN and the NN would have obtained similar results.

(H2) *The GNN model is more resilient to attack variabilities than other traditional models.*

The purpose of this experiment was to evaluate how resilient our GNN model was compared to other traditional ML and DL models when the evaluation datasets were slightly modified. We compared the GNN model with a NN [14], a Random Forest Classifier (RF) [15] and a Support Vector Machine (SVM) [16].

All models were trained with the same dataset and the evaluation dataset was synthetically altered to present some variations (trying to resemble some variations in the attack patterns, as if an attacker was trying to bypass the security systems).

Some of these variations were adding random overheads to the packet sizes and altering the flow durations. We initially trained the models with the same set of features to obtain good classification results in the training datasets:

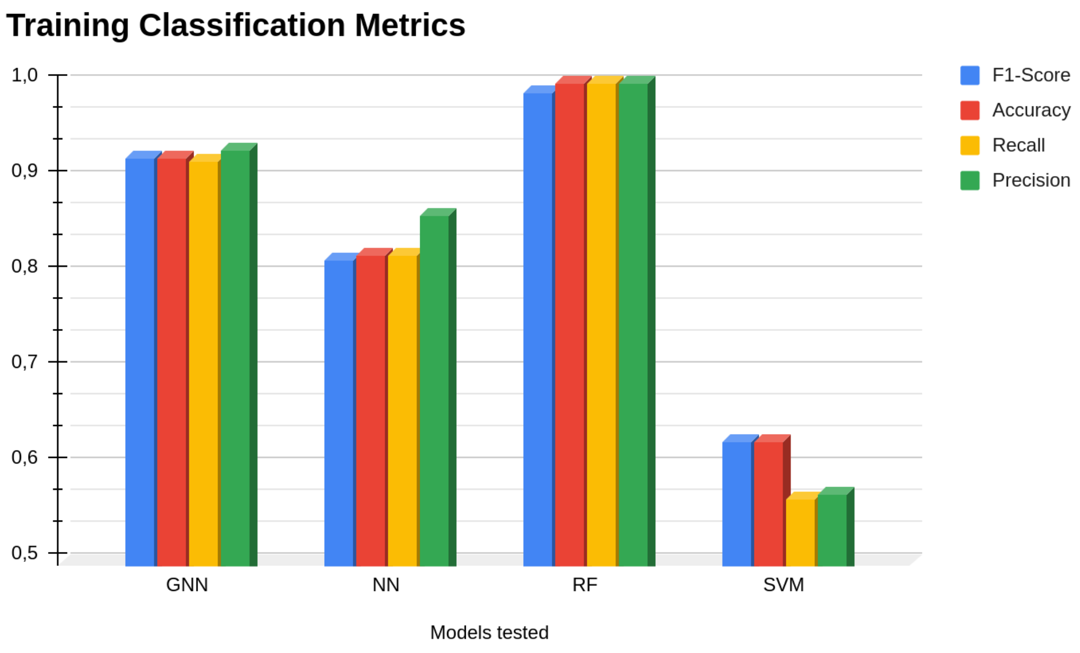

Figure 4.14: Training results - GNN, NN, RF and SVM.

The GNN, the NN and the RF model, obtained good classification results during the training phase, specially the RF model, which obtained values close to 0.99 in all the classification metrics. On the other hand, the SVM model presented very low performance.

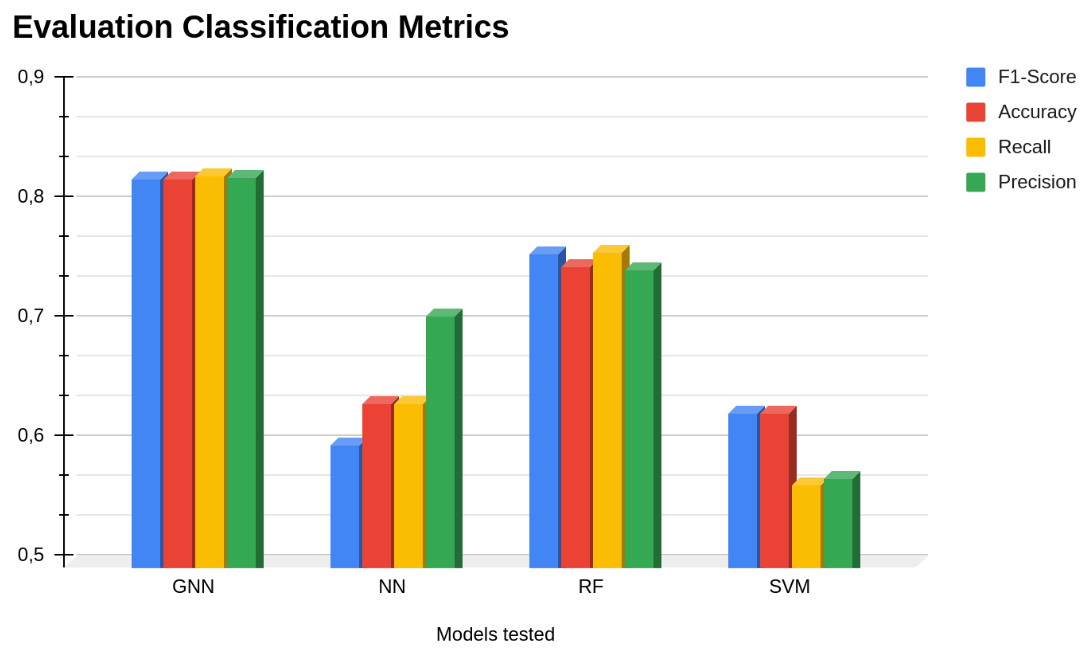

Figure 4.15: Evaluation results on tampered data.

As can be seen from the evaluation graph, the GNN model was the only one which maintained good performance after being tested with the tampered dataset. It is important to mention that the RF model presented a huge drop, as the evaluation metrics dropped to values around 0.74.

We can actually take a look at some of the confusion matrices resulting from this experiment, where, in comparison, it is clear to see that the GNN model was the only model capable of maintaining good classification results.

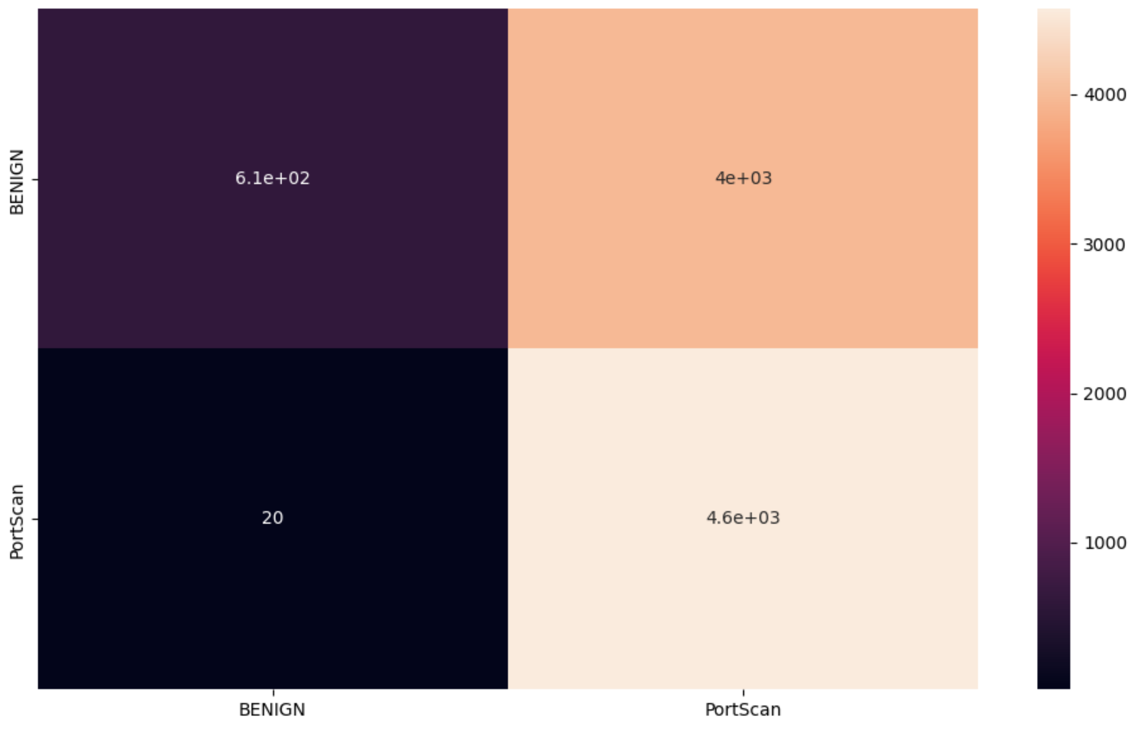

Figure 4.16: Confusion Matrix - NN.

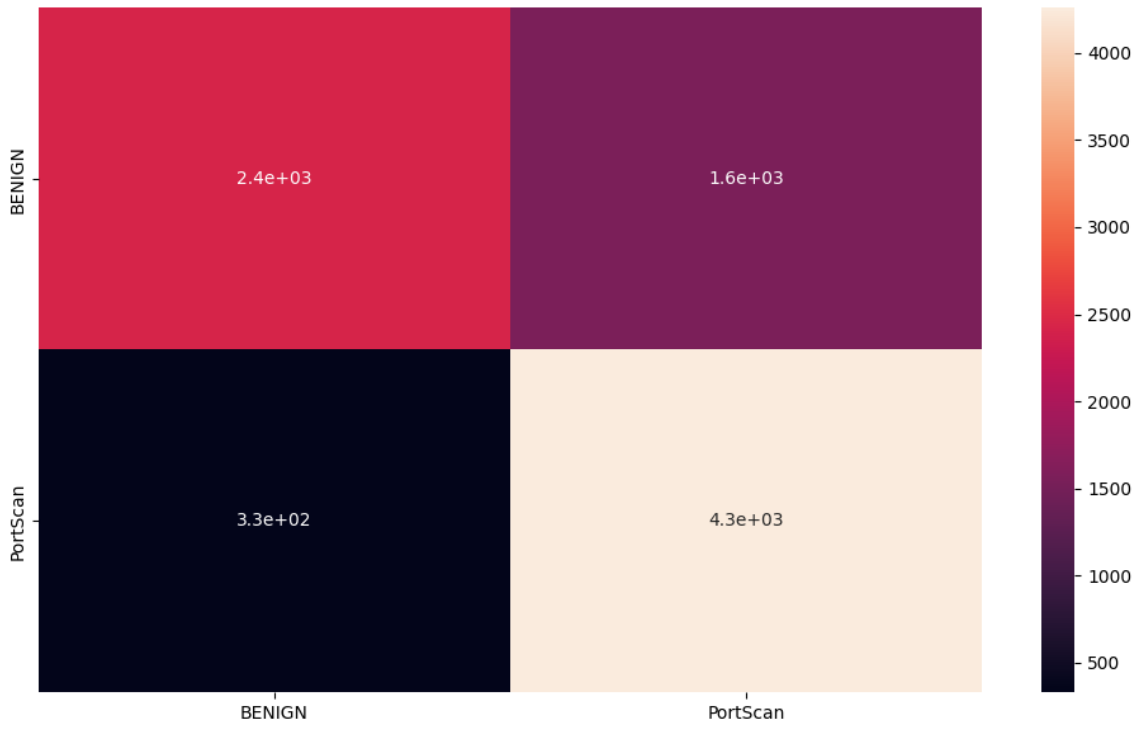

Figure 4.17: Confusion Matrix - GNN.

The results were very positive, as they were proof of the generalisation and resilience capabilities that can be achieved by using a graph-based model in comparison to other well-known models.

4.3.4 Final Features

Selecting the features was a crucial part of the process. If a model uses many features for classifying, it is very likely that it will eventually acquire good results on the training datasets, however, this may be due to overfitting, and not because of the resilience of the model.

Hence we decided to use a small number of features, for the following reasons:

- Avoid overfitting.
- Emphasising the relevance of the graph structures against the features used.
- Making the model generalizable to other network scenarios (features are meant to be as network-independent as possible).
- Using features that could be extracted through different flow processors (to ease integration with other traffic capture systems).

For our implementation of the GNN model, the selected features were:

Feature name	**Description**
Flow Duration	Duration of the flow in Microseconds.
Flow Packets / s	Number of flow packets per second.
Flow IAT Mean	Mean time between two packets sent in the flow.
Fwd IAT Mean	Mean time between two packets sent in the forward direction.
Bwd IAT Mean	Mean time between two packets sent in the backward direction.

Table 4.1: Features selected.

As mentioned before, the *CIC-FlowMeter* processor extracts around 80 features to describe each flow. So we could have loaded the model with as many features as possible. However, the intention of our project was not to present what final features should be selected, but rather unveiling the potential of Graph Neural Networks for intrusion detection.

4.4 Training and Evaluation

We have mainly focused on three types of network events:

- Benign traffic.
- PortScan attacks [17].
- DoS / DDoS attacks [18].

The main reason behind this selection is the fact that, PortScan attacks (which are part of the *reconnaissance* attacks), are graphically very distinguishable from the Denial of Service attacks (which are part of the *flooding* attacks). This way, we start from a scenario where the network events will mainly be identified by their overall structure and flow inter-dependencies, instead of solely based on the individual features that describe each flow.

A PortScan attack is mainly characterised by multiple network flows targeting multiple ports in a victim. Its purpose is to discover the status of the ports to potentially find some vulnerabilities. Attackers normally execute them by sending packets using different protocols, such as TCP, UDP or ICMP.

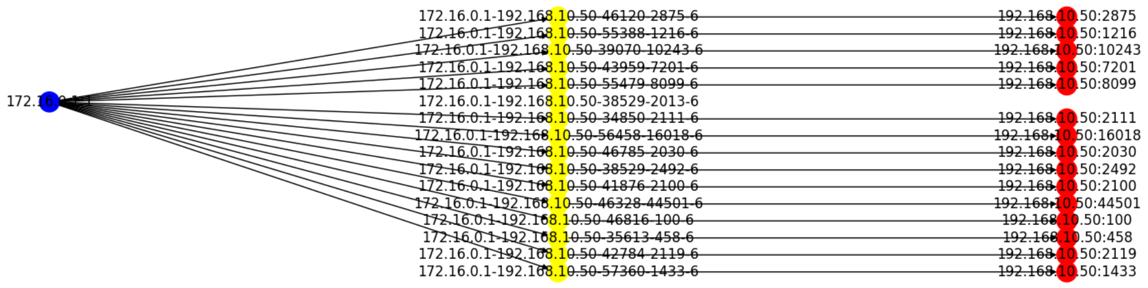

Figure 4.18: Network flows representing a PortScan attack.

On the other side, a Denial of Service (DoS) or Distributed Denial of Service (DDoS) attack aims at flooding a certain port that can be running an important service, such as port 80 - HTTP, to bring it down. Hence, its graph representation is significantly different from the PortScan, as this time, we can see all network flows directing traffic to a single port. In the case of a distributed attack, we would see the same behaviour, but coming from different sources.

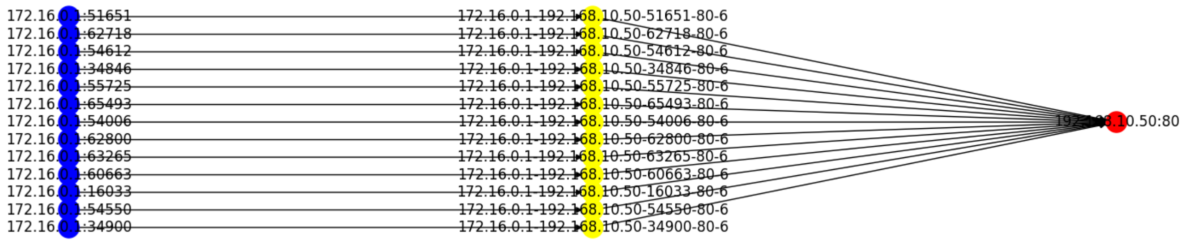

Figure 4.19: Network flows representing a DoS attack.

4.4.1 Batching traffic by the number of network flows

For the training and evaluation phase, we used the same dataset mentioned in section *4.1.1 CIC-IDS 2017*. However, prior to getting into the actual results, it is important to add some context. These datasets contain a huge number of flows (in the order of hundreds of thousands). The most intuitive approach would be to build an entire graph from all this data records, and use it as the input data for training.

However, despite this could be done, it is far from what a production NIDS would require. A NIDS is meant to detect network attacks, to do so it needs to analyse batches of collected network traffic data. This means that if it collects data once a day it will only be able to make assumptions about the status of the network once a day.

That is why NIDS normally collect data according to two different criteria:

- **Time Interval**: for instance, flows can be collected every minute, every hour, etc.
- **Number of flows**: batches can be created based upon receiving a certain amount of flows, such as 50, 100 or 1000 records.

Although both options are similar, we focused on the latter. Mainly because collecting by the number of flows allows the system to work with networks of different sizes.

A huge network could collect many more flows in the same time span as a small network, hence a model trained for collecting data every X seconds may not behave equally under different scenarios.

On the other hand, if a model is trained based upon the number of flows, it might take less in a bigger network to collect them than in a smaller network, but the model will behave similarly.

Hence, in the following sections, the results will be presented as a relation between the number of flows collected in each batch and the performance of the model. The main metrics that were analysed were:

- F1-Score.
- Accuracy.
- Precision.
- Recall.

For training the model, we used the *Adam* [19] optimizer, with learning rate *1e-3* and weight-decay *5e-4*a and we used the *Cross-Entropy Loss* [20] criterion for the backward loss propagation.

4.4.2 PortScan attack vs Benign Traffic

In this initial setup we aimed to test our model in a binary classification scenario, with benign traffic and PortScan attack records. Instead of creating a single graph with all the data, as mentioned before, we analysed the performance of the model according to the number of flows included in each batch. For training the model, we:

- Shuffled the dataset, so samples would be distributed throughout the multiple graphs.
- Splitted the dataset in chunks (varying according to the number of flows).
- Fed the model each of these chunks via a *DataLoader*[16] object.

The classification results in the training phase were:

[16] https://pytorch-geometric.readthedocs.io/en/latest/modules/loader.html#torch_geometric.loader.DataLoader

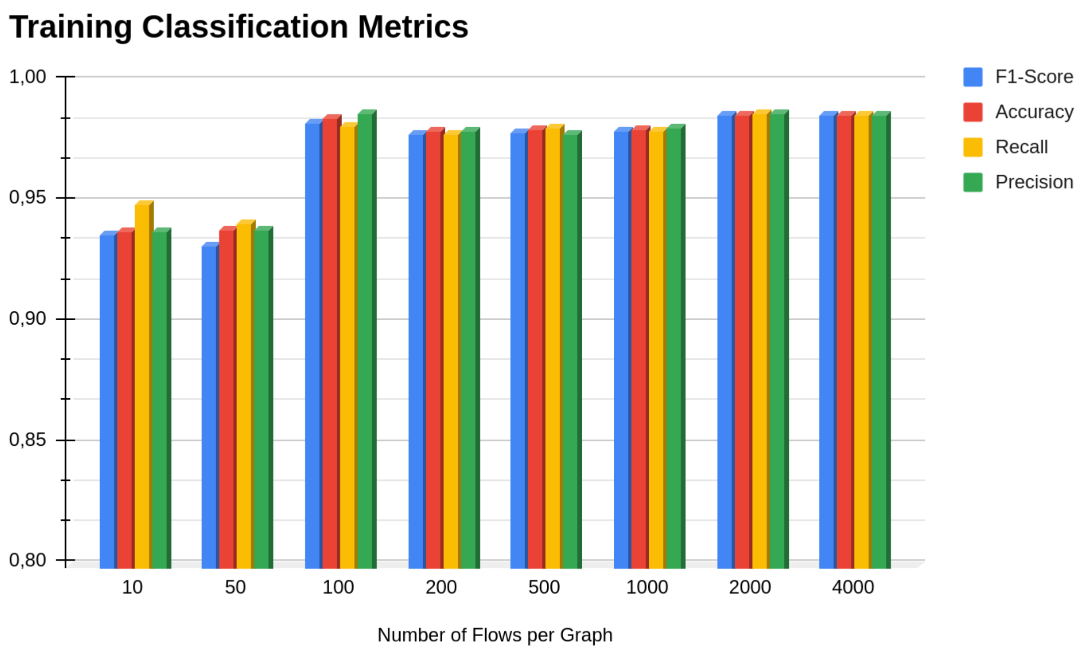

Figure 4.20: Training Classification Metrics related to Number of Flows per Graph.

As can be seen from the previous graph, the *Number of Flows per Graph* axis ranges from 10 up to 4000. For the smallest number of flows (10), the model still acquires values above 0.90 in all metrics.

The results are good for all the chunk sizes, however, the bigger the graph, the higher the classification metrics. For instance, the models that use graphs containing between 2000-4000 flows acquire values around 0.97 in all metrics.

This makes sense, as smaller graphs tend to be less representative of an attack, as they can be confused with benign traffic. Nevertheless, a performant model on smaller graphs is faster in intrusion detection, as it does not require that much information to make decisions.

The models presented very good classification results in the validation phase. In this case, it was the model trained with batches of 500 flows per graph the one that acquired the highest classification metrics.

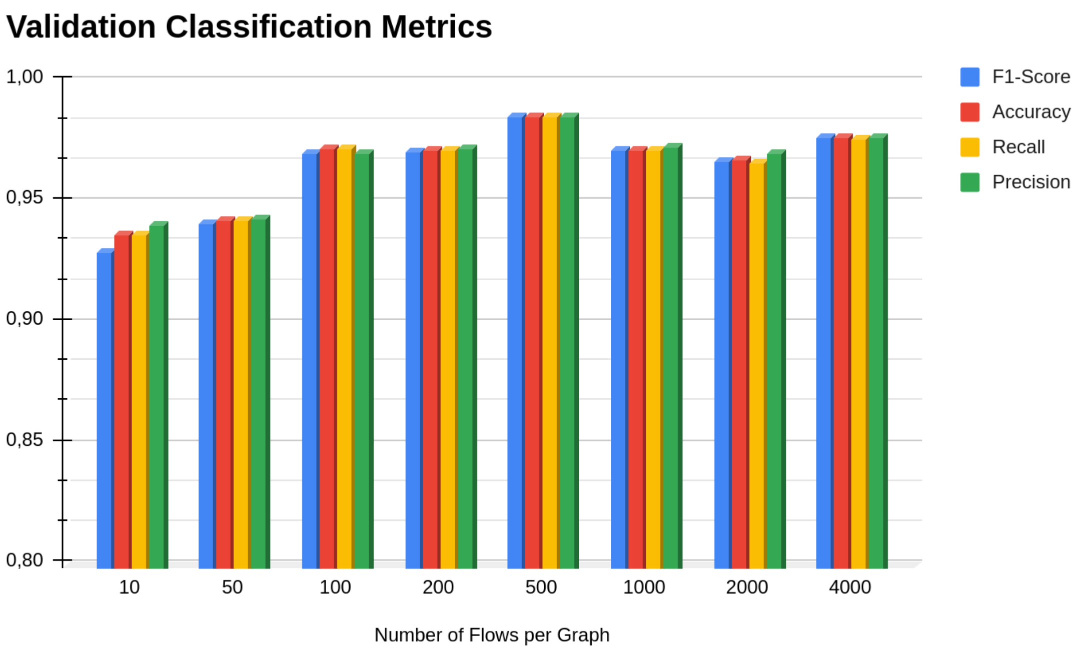

Figure 4.21: Evaluation Classification Metrics related to Number of Flows per Graph.

We can specifically take a look at the F1-Score results for training and validation, as this is a good representative metric of the performance of a model. As we can see from the graphs attached below, in this scenario, our model achieves very good classification results, despite using few features and varying the number of flows included in each graph.

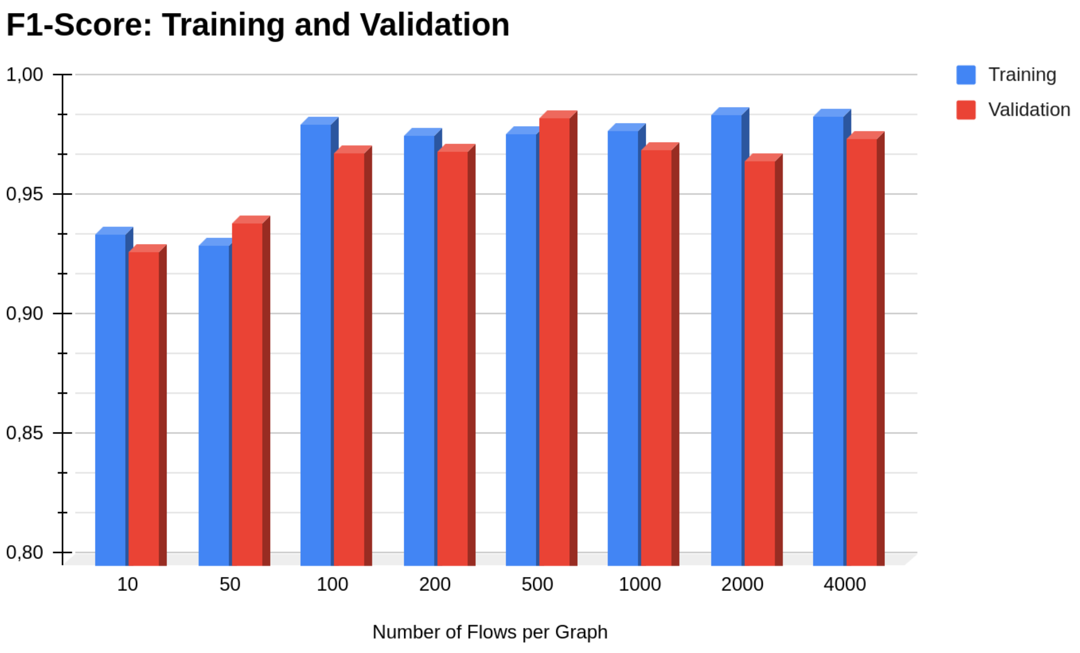

Figure 4.22: F1-Score results - Training and validation (bar chart).

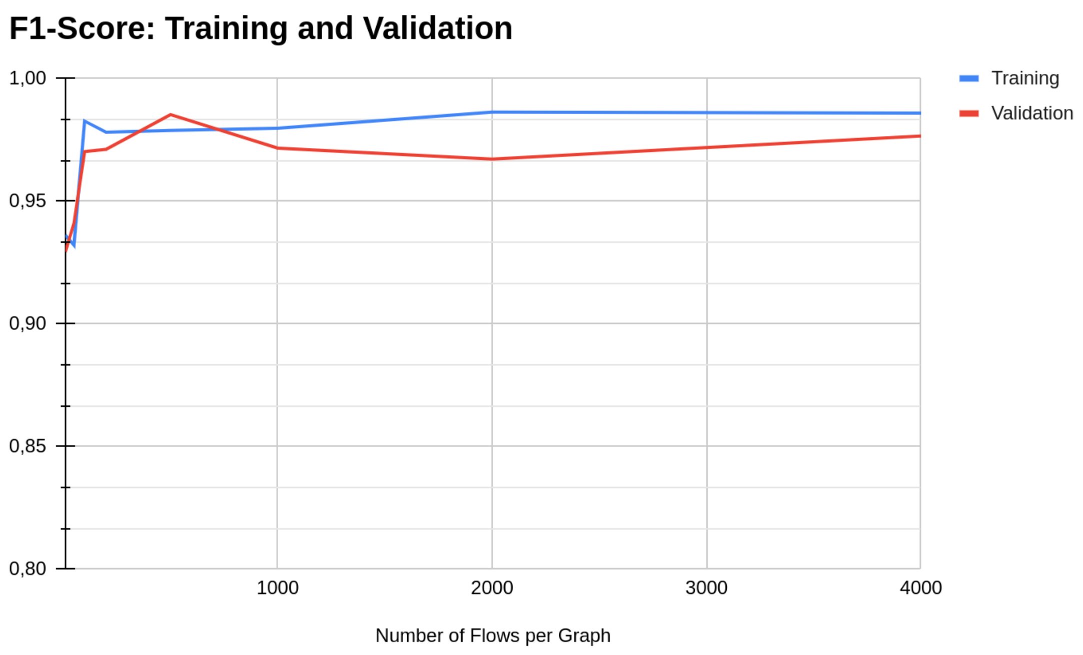

Figure 4.23: F1-Score results - Training and validation (tendency line).

We selected one of the models to create some illustrations of how the graphs are classified. This is a good way of better understanding how our GNN model works, as some of the explanations used in this field can be sometimes abstract.
The following graph represents a benign connection. The model will colour the flow nodes as green if the flow is benign or red if the flow is malign.

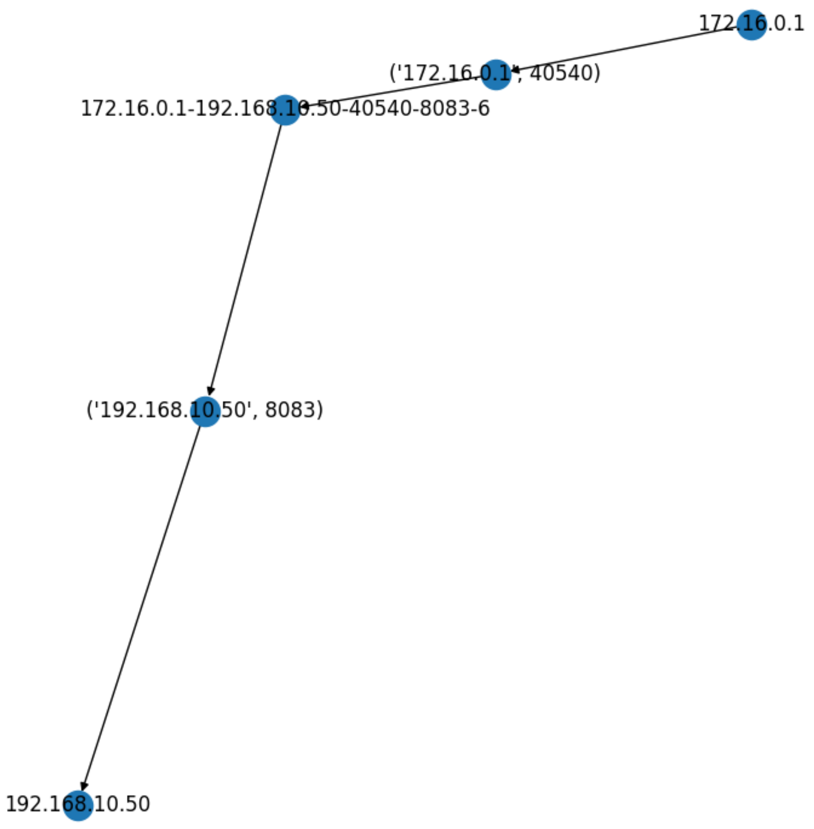

Figure 4.24: Benign traffic graph (raw).

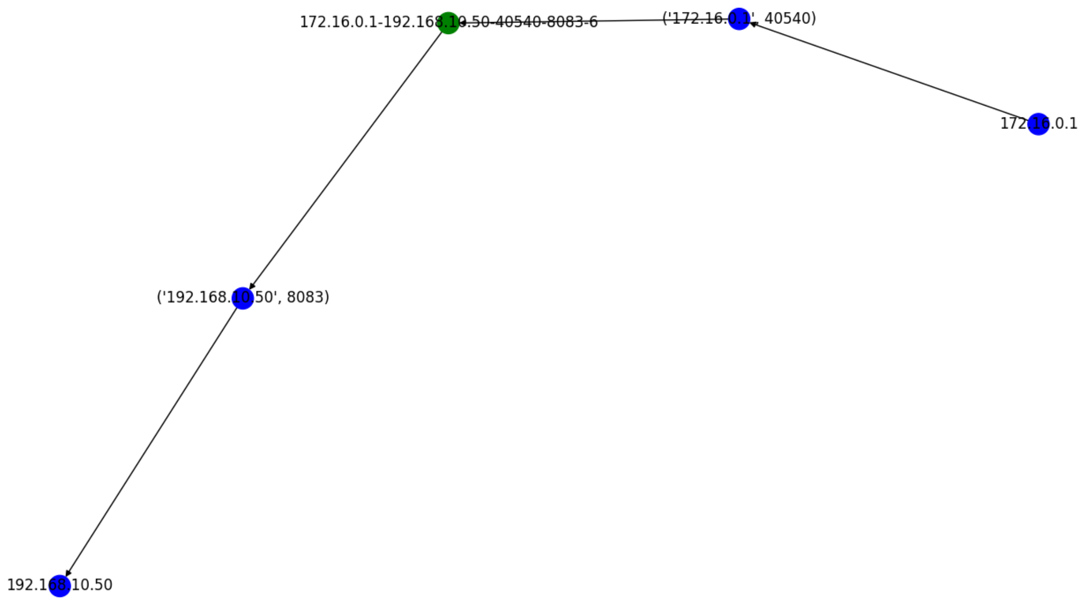

Figure 4.25: Benign traffic graph (classified).

As we can see, our model properly understands the flow in the graph as not malicious. In the case of a graph that represents a PortSan attack:

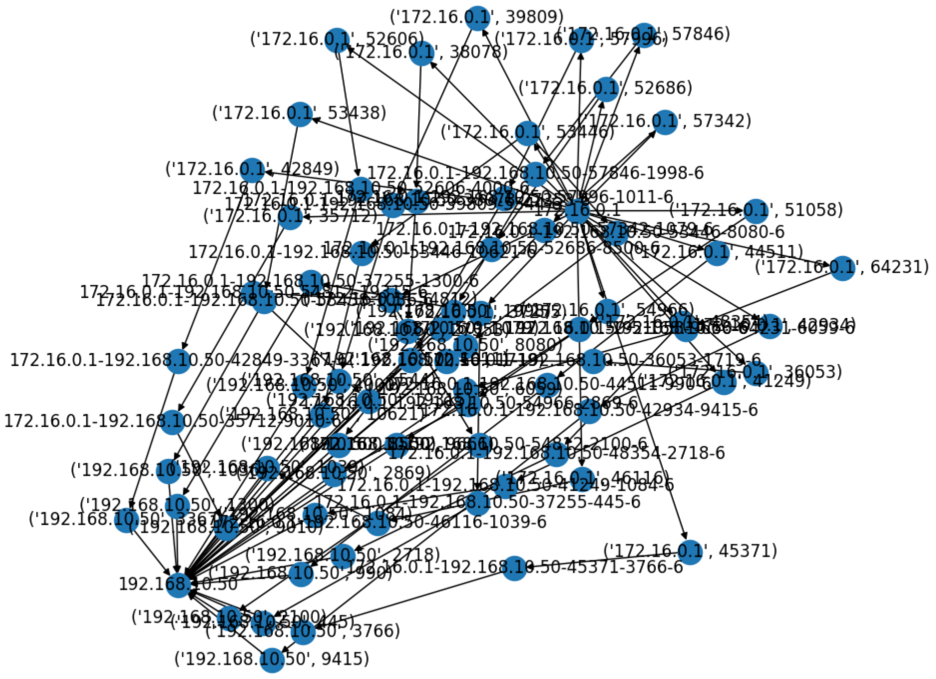

Figure 4.26: PortScan attack graph (raw).

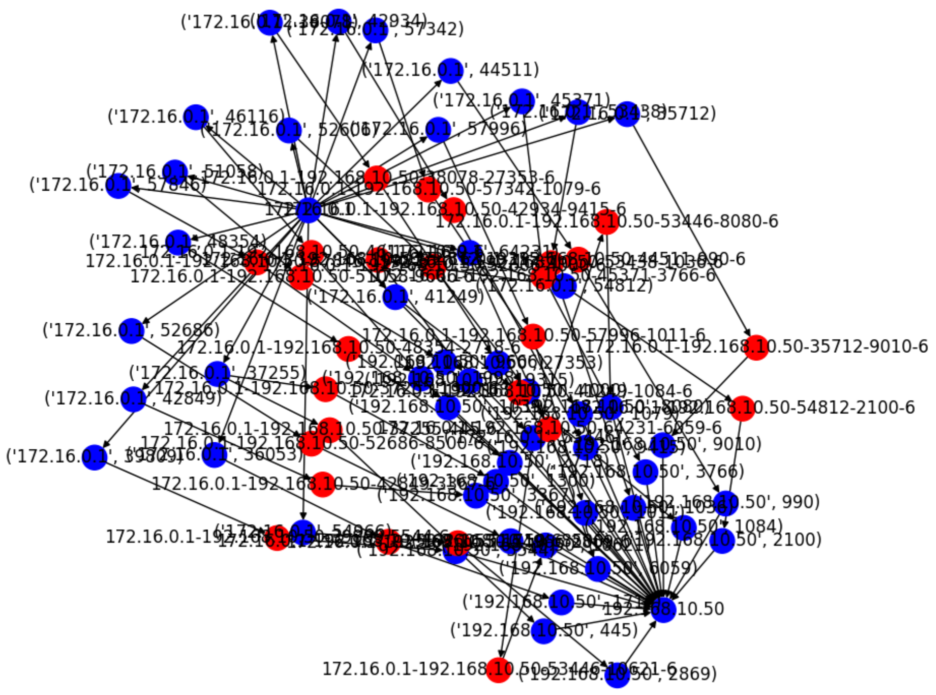

Figure 4.27: PortScan attack graph (classified).

Our model is able to classify all of the flow nodes properly in the graph, as it labels them as malign, and in fact these represent a PortScan attack.

These good results were initially expected, as the graph representations of a PortScan attack and benign traffic are very differentiable. Our next approach was to use some training data that contained other types of graphs which could complicate the classification results.

4.4.3 PortScan attack vs Not-PortScan traffic

In this case we created a dataset that contained three different types of network events, but classified binary:

- **PortScan** traffic.
- **Not-PortScan** traffic: which contained both benign traffic and DoS traffic.

The idea behind this approach was to add more complex graph structures in the training and validation datasets. The training results were positive, as even varying the number of flows included in each graph, we were able to achieve very high classification metrics.

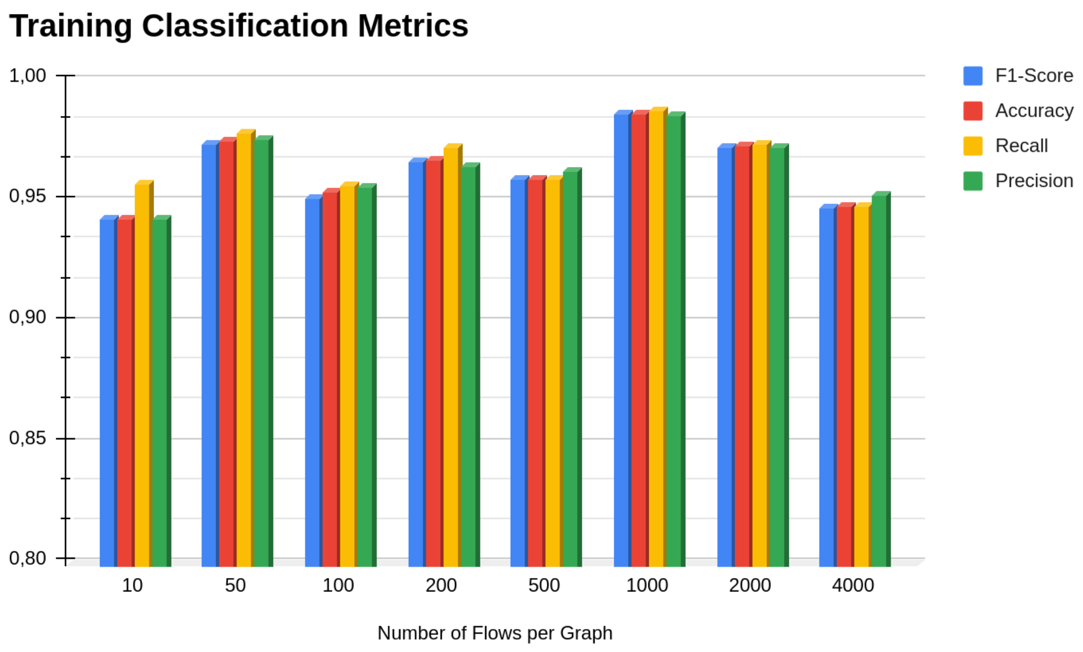

Figure 4.28: Training Classification Metrics related to Number of Flows per Graph.

In comparison with the previous experiment, the classification results dropped a little bit. The model trained with very few flows per graph dropped in F1-Score and Accuracy to values between 0.76-0.78. However, some of the models, such as the one using 2000 flows per graph, still acquired very good classification results, obtaining values superior to 0.90 in all metrics.

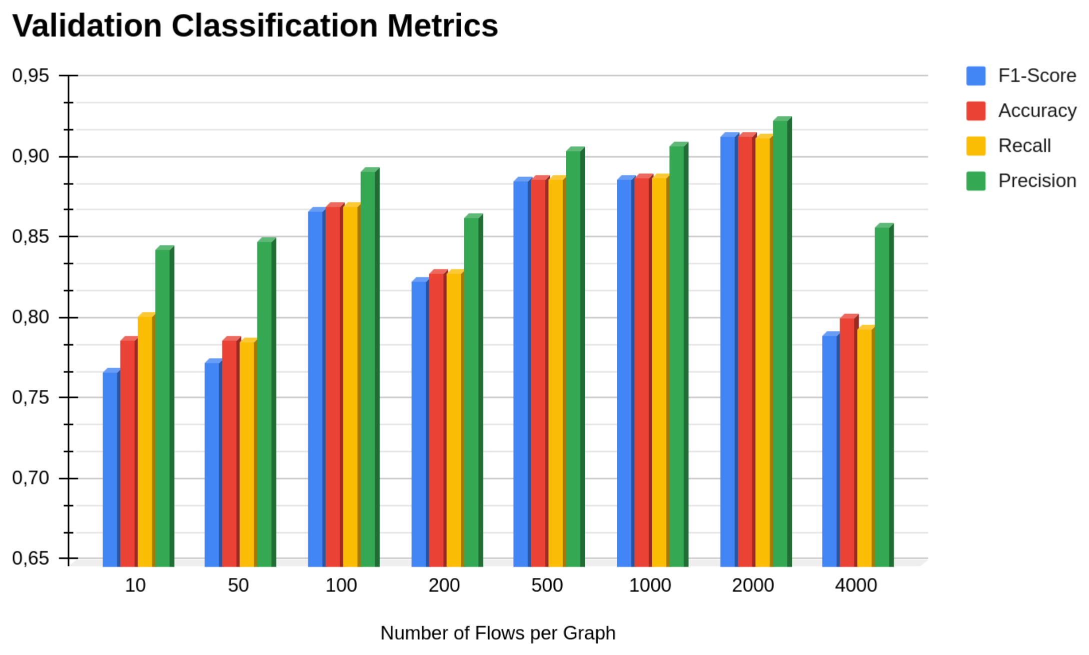

Figure 4.29: Validation Classification Metrics related to Number of Flows per Graph.

Surprisingly, the model that is trained with batches of 4000 flows, acquires similar results to the model trained with batches of 10 flows. This could be due to the increase in graph complexity that comes when using a huge number of flows representing multiple network events. In these scenarios, the graph differentiability between different network events can be drastically reduced.

We can specifically take a look at the F1-Score results for training and validation, as this is a good representative metric of the performance of a model. As we can see from the graphs attached below, the results in the training phase are this time clearly better than the validation phase.

It is clear to see that the models that use around 2000 flows per graph obtained better results than the other types of models. This could be due to multiple reasons, as for instance, it is expected that graphs with very few flows are not representative for complex network events. Nevertheless, this model still obtained very good results, over 0.90 for the F1-Score.

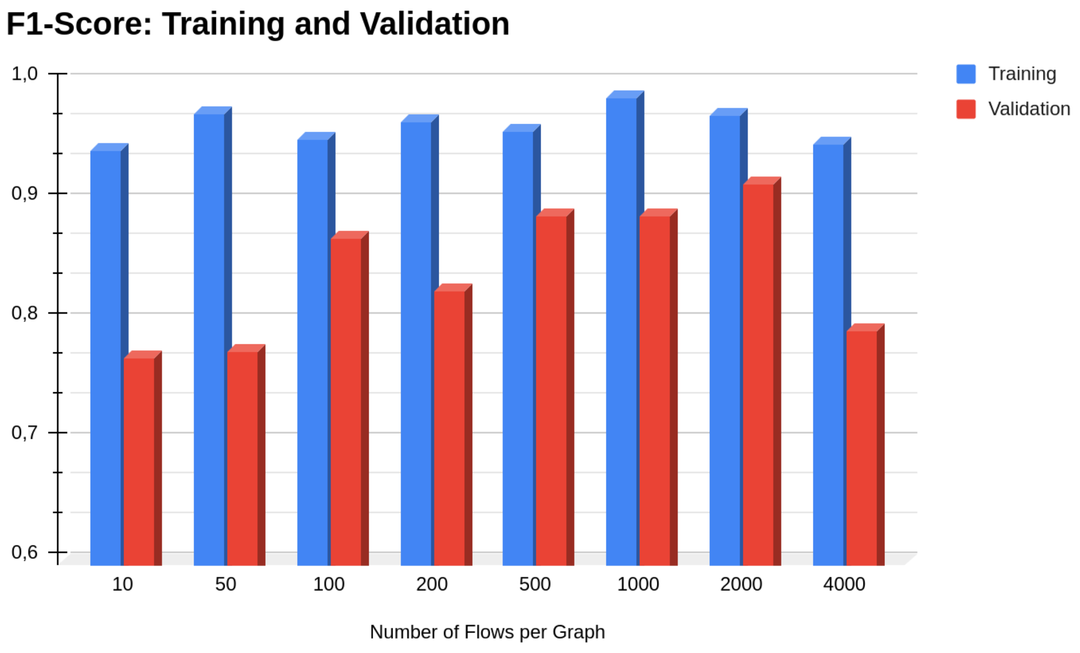

Figure 4.30: F1-Score results - Training and validation (bar chart).

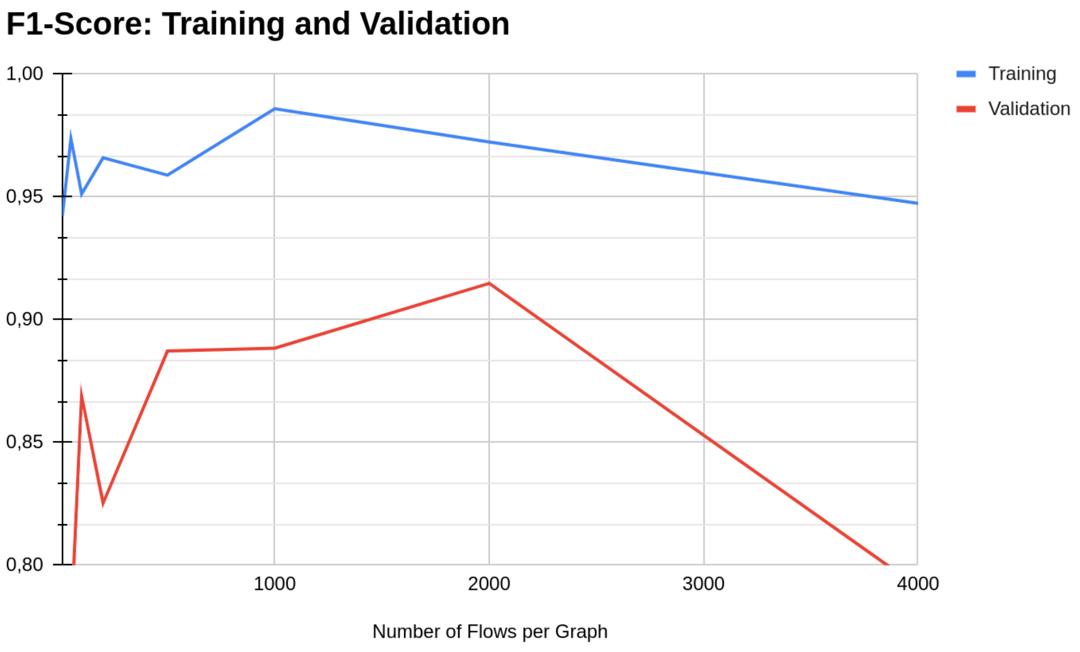

Figure 4.31: F1-Score results - Training and validation (tendency line).

As with the previous experiment, we selected one of the models to create some illustrations on how the model classifies the different types of traffic included in this dataset.

The following graph represents a mix of benign traffic and a PortScan attack.

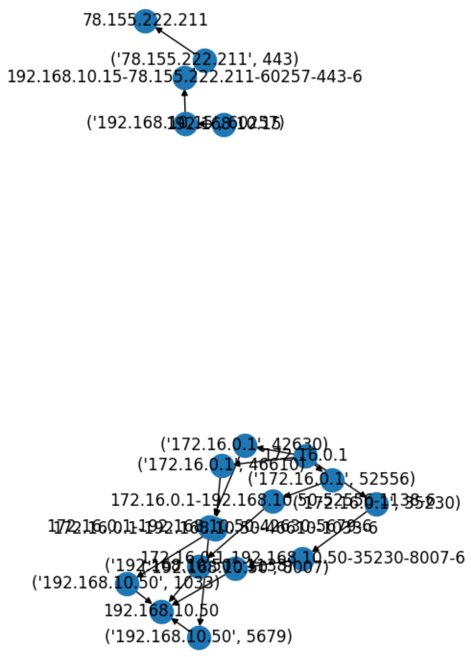

Figure 4.32: Graph including benign traffic and PortScan attack (raw).

Our model is able to properly distinguish both network events, as it correctly labels the Not-PortScan flow nodes (green) and the PortScan flow nodes (red).

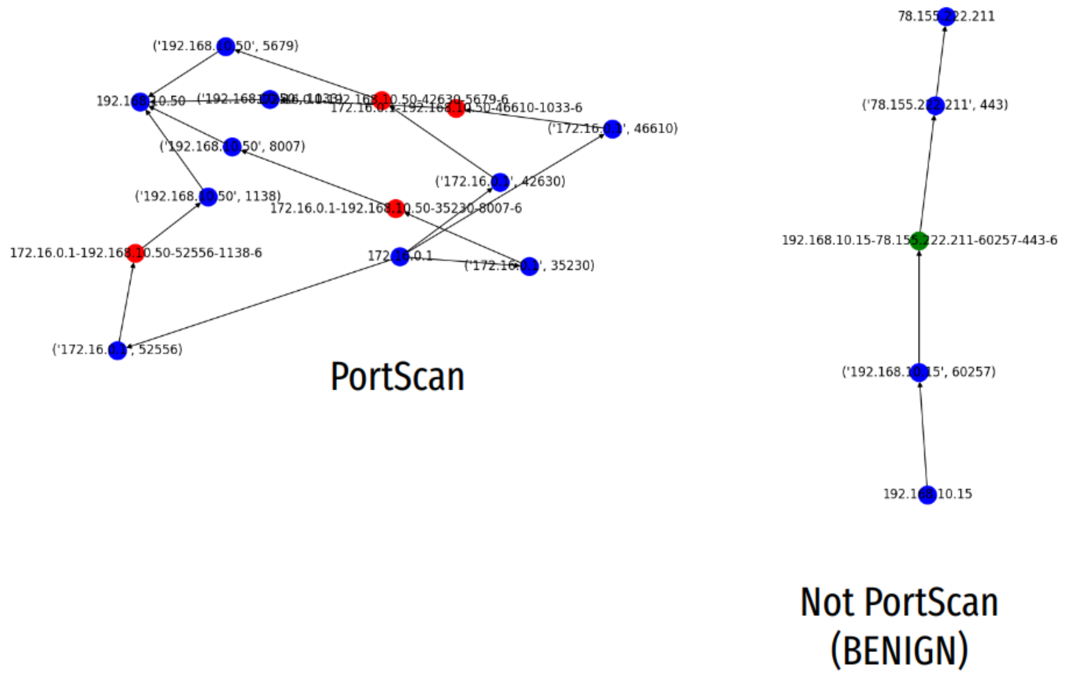

Figure 4.33: Graph including bening traffic and PortScan attack (classified).

In this case, the graph contains DDoS samples and PortScan samples:

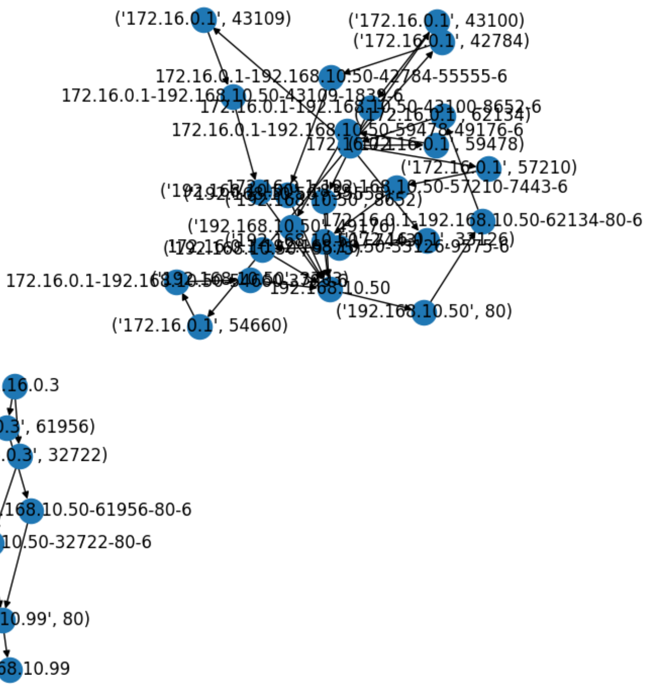

Figure 4.34: Graph including DDoS and PortScan samples (raw).

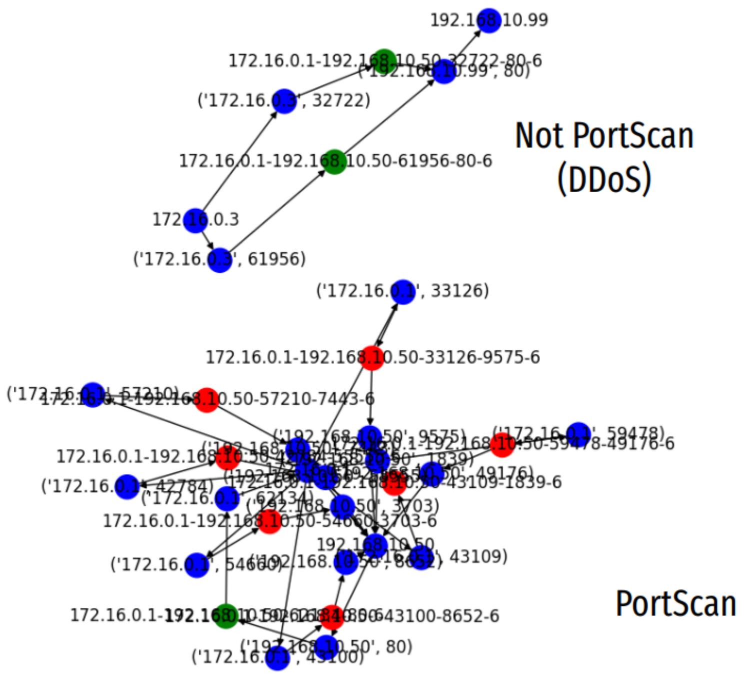

Figure 4.35: Graph including DDoS and PortScan samples (classified).

Our model is able to properly distinguish the two graph structures, despite the fact that one of the flow nodes is still misclassified. In this scenario it has been trained to label Not-PortScan traffic in green and PortScan samples in red.

We can still create a more complex graph, which contains all three types of network events: benign traffic, DDoS samples and PortScan samples.

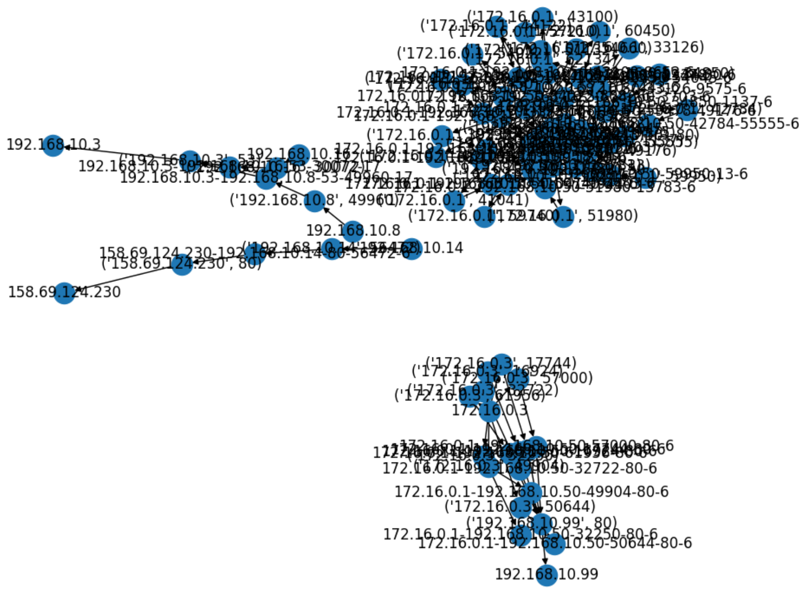

Figure 4.36: Graph including benign traffic, DDoS and PortScan samples (raw).

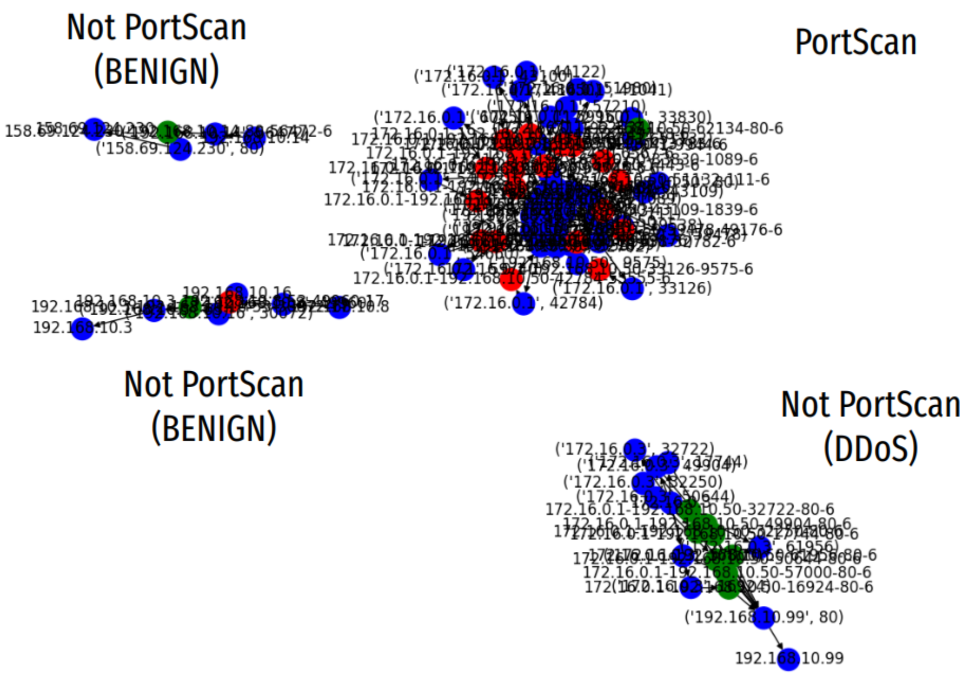

Figure 4.37: Graph including benign traffic, DDoS and PortScan samples (classified).

Again, our model is able to distinguish the different network events with complex and nested graph structures (despite some minor misclassifications). These illustrations are very helpful to reduce the abstraction that comes when evaluating a model and looking at the classification metrics and the bar charts.

Our model needed the graphs to be more representative this time, to achieve good classification results. It was expected that small graphs would not yield very high results, as the data used is more complex and graphs need to be insightful enough for the model to differentiate between the different network events.

4.4.4 PortScan, DDoS and Benign traffic

We also tested our model in a ternary classification scenario, where the three types of network events included were:

- PortScan attack.
- DDoS attack.
- Benign traffic.

The idea behind this experiment was to see the performance of the model when trying to specifically differentiate between different types of network attacks. In this case, and based upon the results obtained in the two previous experiments, we only tested the model for batch sizes including between 100 and 2000 flows.

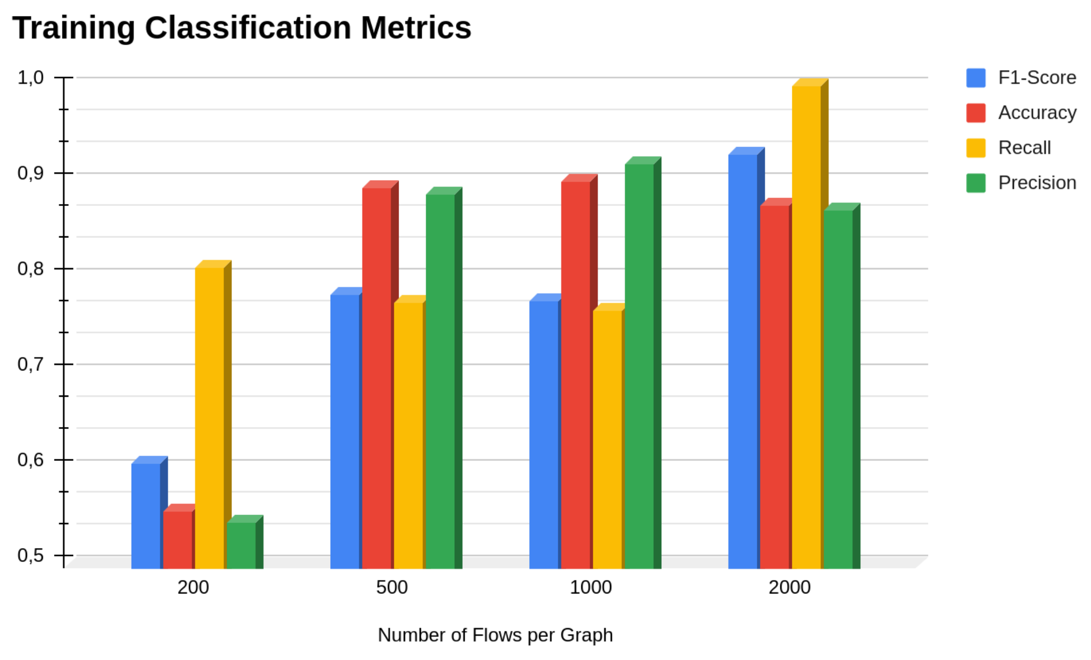

Figure 4.38: Training Classification Metrics related to Number of Flows per Graph.

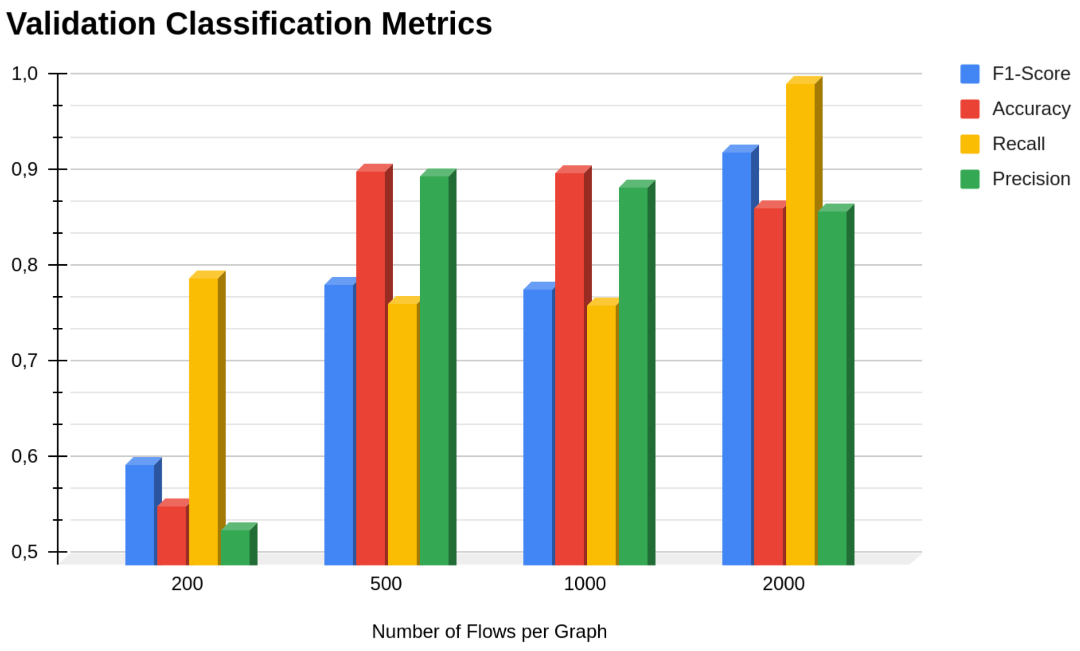

Figure 4.39: Validation Classification Metrics related to Number of Flows per Graph.

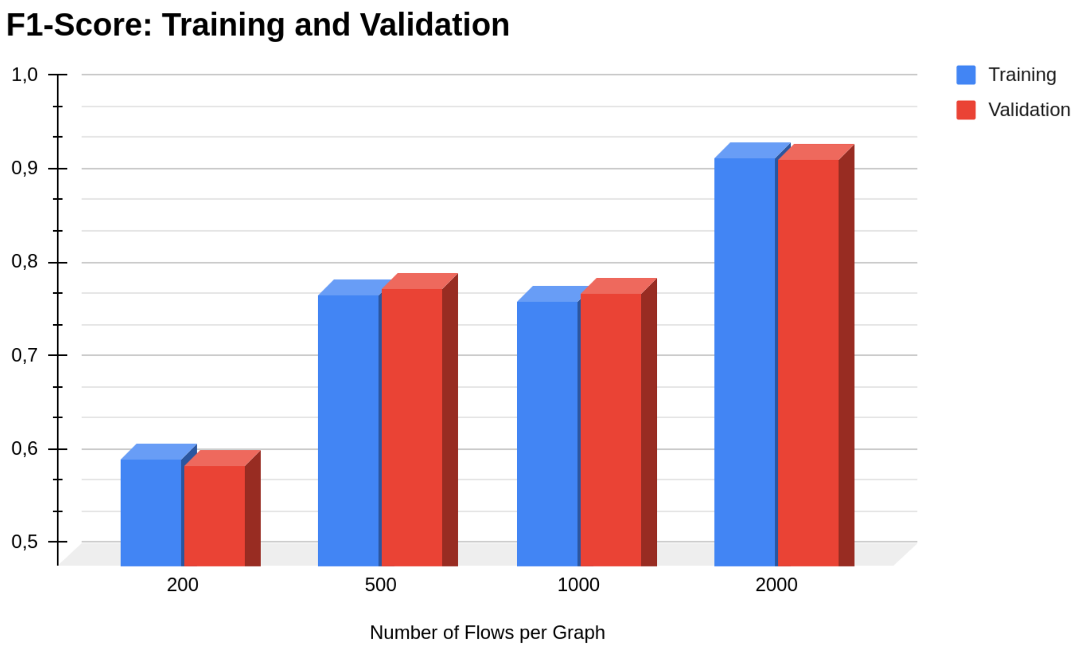

Figure 4.40: F1-Score results - Training and validation (bar chart).

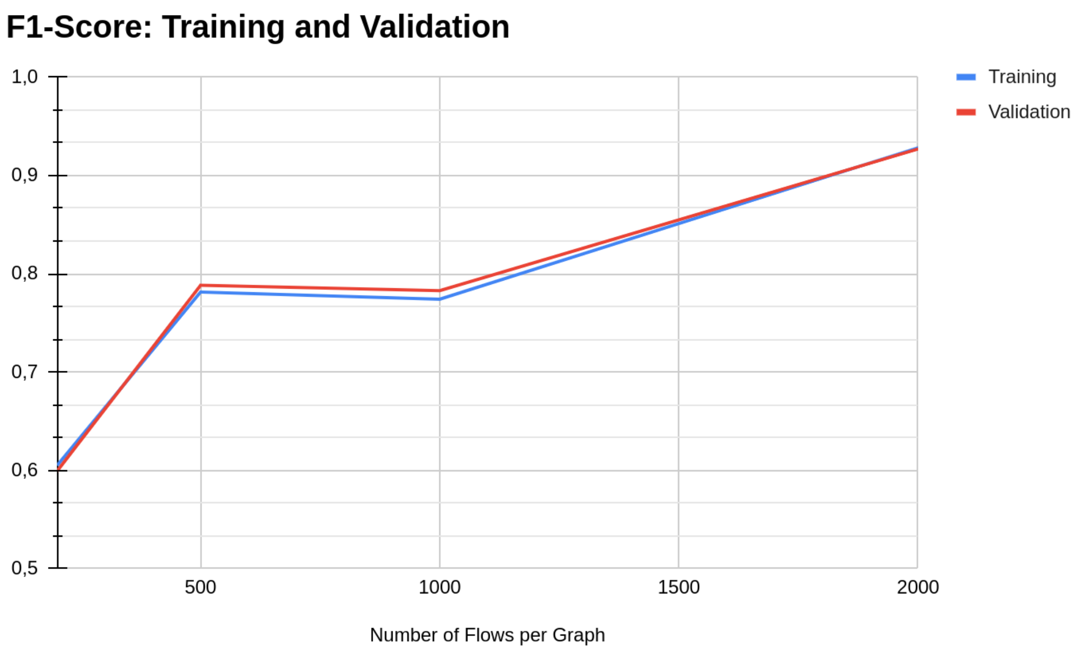

Figure 4.41: F1-Score results - Training and validation (tendency line).

As expected, we can see that the performance of the model directly increases with the number of flows included in each batch. We set up a scenario where there were more complex graph structures and three types of classes. Hence, the models which were trained with few flows, could not yield good classification results.

We can see an increase in the F1-Score when the number of flows included is higher than 500. This does not mean that models trained with batches containing fewer nodes can not perform properly. But it is true that, as we are using very few features from the original dataset, the graph structure will impact the classification.

If the graph structure did not alter the classification, the results would be the same for all the models, as it would be the same as applying a traditional Neural Network on each of the set of initial flow features.

On the other hand, we also do not want to use models based on a huge number of flows, as these might exponentially increase in graph complexity and hence reduce the performance of the model. These experiments have shown us that models based on approximately 1000-2000 network flows, achieve good results in most scenarios and without the need of using many network features.

4.4.5 Evaluation of a “Reconnaissance” attack

Our next experiment was to evaluate the created models with some attack records that belonged to a completely different dataset. To do so, we selected the *CIC-BoT-IoT*[17] dataset, which is a dataset that was created to train models for *Internet of Things*[18] applications.

The purpose of this test was to see the performance of the different models with data that had been generated in a completely different way. The main intuition behind this approach was that, despite there were going to be some potential changes in the features and their values, the attacks contained could have similar structures and resemble the ones used for training.

That is why we selected the Reconnaissance[19] attack, which can present a similar structure as the PortScan attack. For it, we created a dataset which contained a set of Reconnaissance attack samples and also some benign traffic.

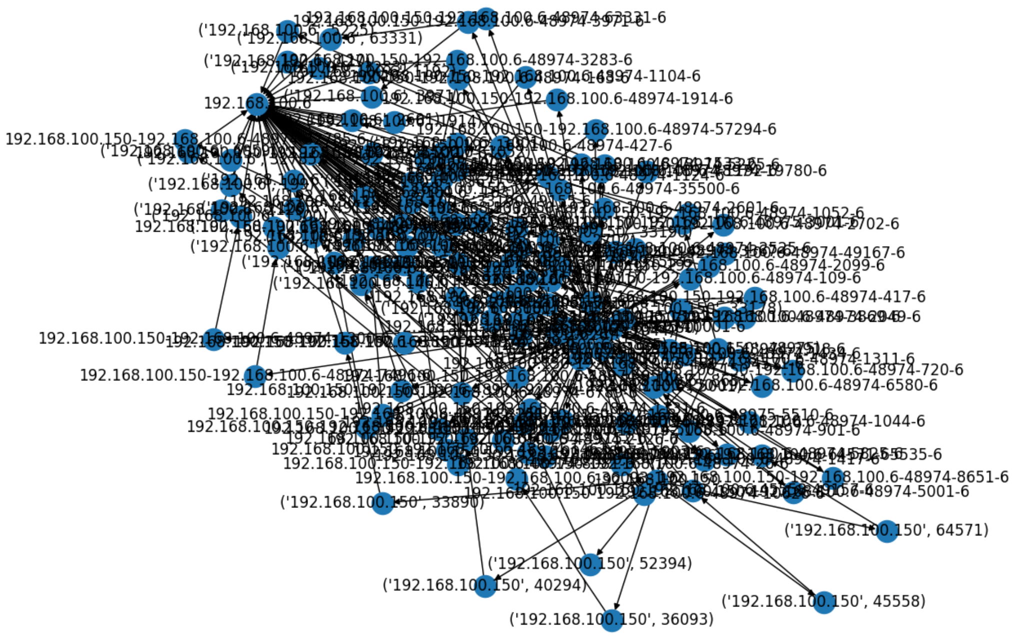

Figure 4.42: Reconnaissance attack graph.

[17] https://staff.itee.uq.edu.au/marius/NIDS_datasets/#RA14

[18] https://en.wikipedia.org/wiki/Internet_of_things

[19] https://www.blumira.com/glossary/reconnaissance/

As can be seen in the graph illustration, even if this dataset was generated in a different way, it still presents some similarities to the graphs that we used for training and evaluation.

We first tested the models that had been trained to perform a binary classification between PortScan samples and Benign traffic. Among the results obtained we selected the one that obtained higher classification metrics.

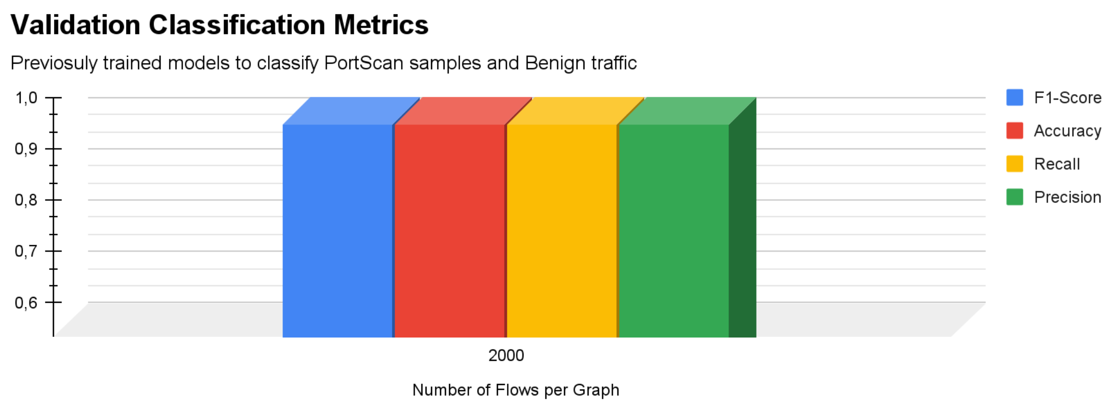

Figure 4.43: Validation Classification Metrics related to Number of Flows per Graph (on Reconnaissance attack samples).

As can be seen, the model that yielded the highest results was the one trained with 2000 flows per input graph. The results were very good if we take into account that the data being used for evaluation had been generated in a totally different way, hence our model was still maintaining predictive capabilities in an unseen scenario.

In fact, the model that was trained with batches of 2000 flows per graph, managed to perfectly classify all of the samples. We can take a look at its confusion matrix to further analyse the results.

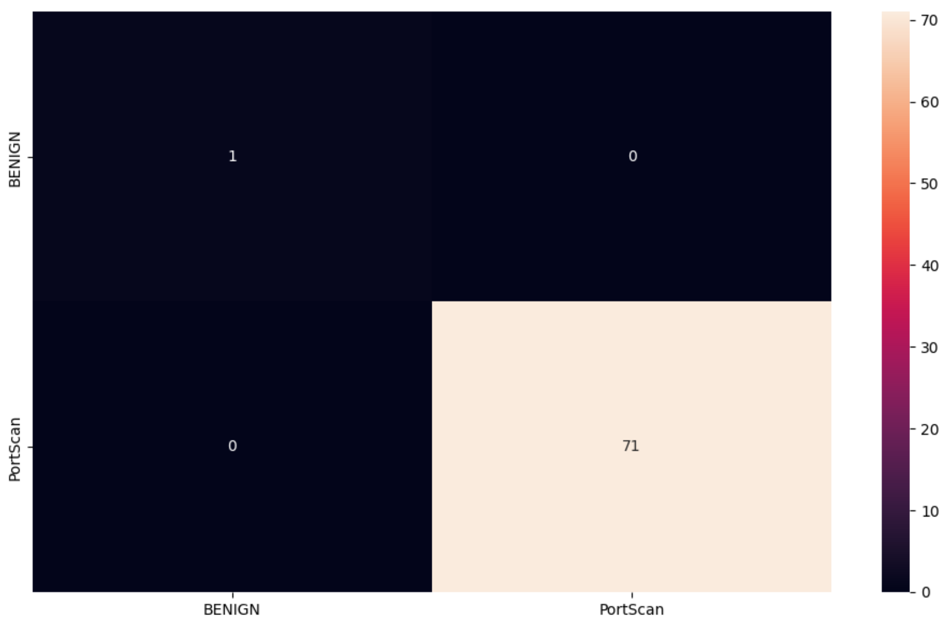

Figure 4.44: PortScan vs Benign model (2000 flows per graph) confusion matrix.

The model is able to perfectly distinguish among the samples in the attack cluster and the extra benign traffic that is contained. We can also look at the graphical prediction:

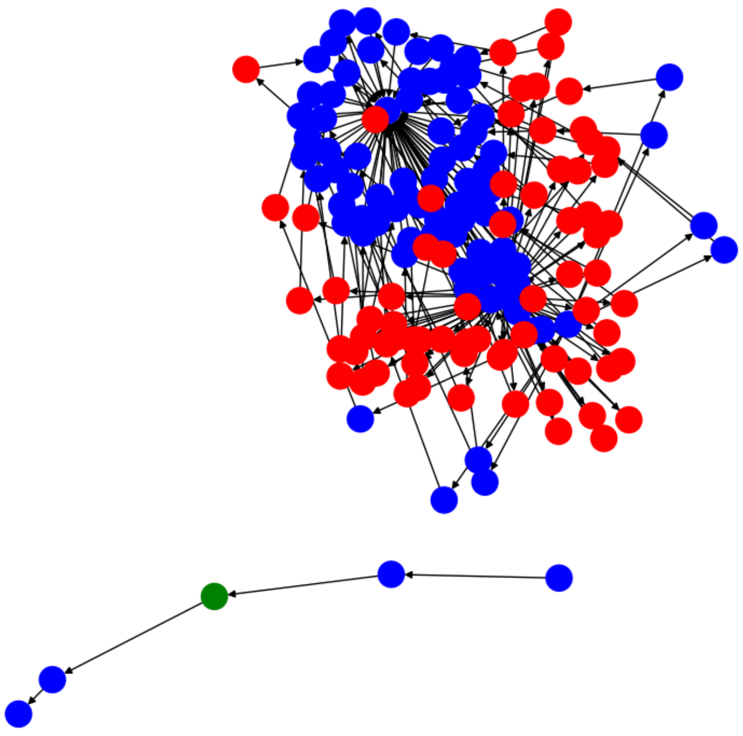

Figure 4.45: PortScan vs Benign model (2000 flows per graph) graphical prediction.

All the nodes in the cluster (Reconnaissance attack) are coloured in red, whereas the legitimate traffic is coloured in red.

We then tested the models that had been trained to perform a binary classification between PortScan samples and Not PortScan traffic. As we recall, the Not PortScan traffic contained a mix of benign and DDoS/DoS samples. Among the results obtained we selected the ones that obtained higher classification metrics.

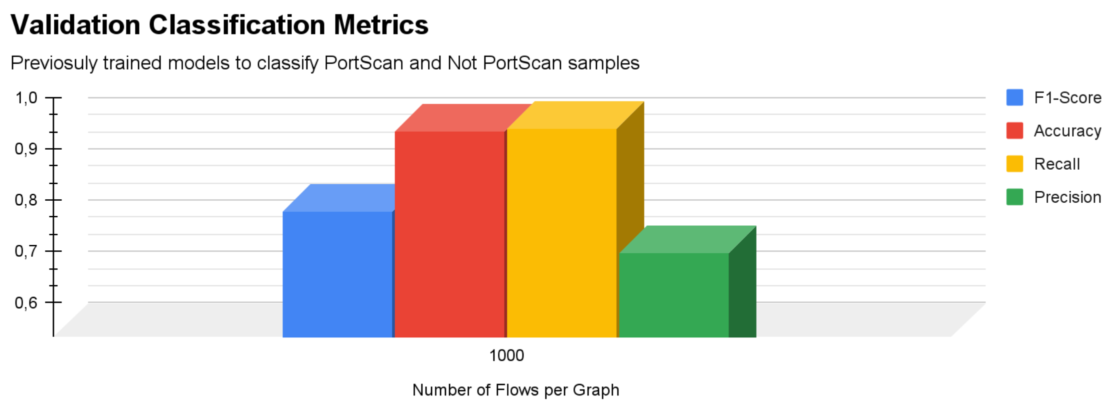

Figure 4.46: Validation Classification Metrics related to Number of Flows per Graph (on Reconnaissance attack samples).

The model that obtained the highest results was the one that had been trained with batches of 1000 flows per graph. It obtained an F1-Score of 0.82 and an accuracy of 0.98. If we inspect its confusion matrix we can see that most of the samples were properly classified, as only 1 was misclassified.

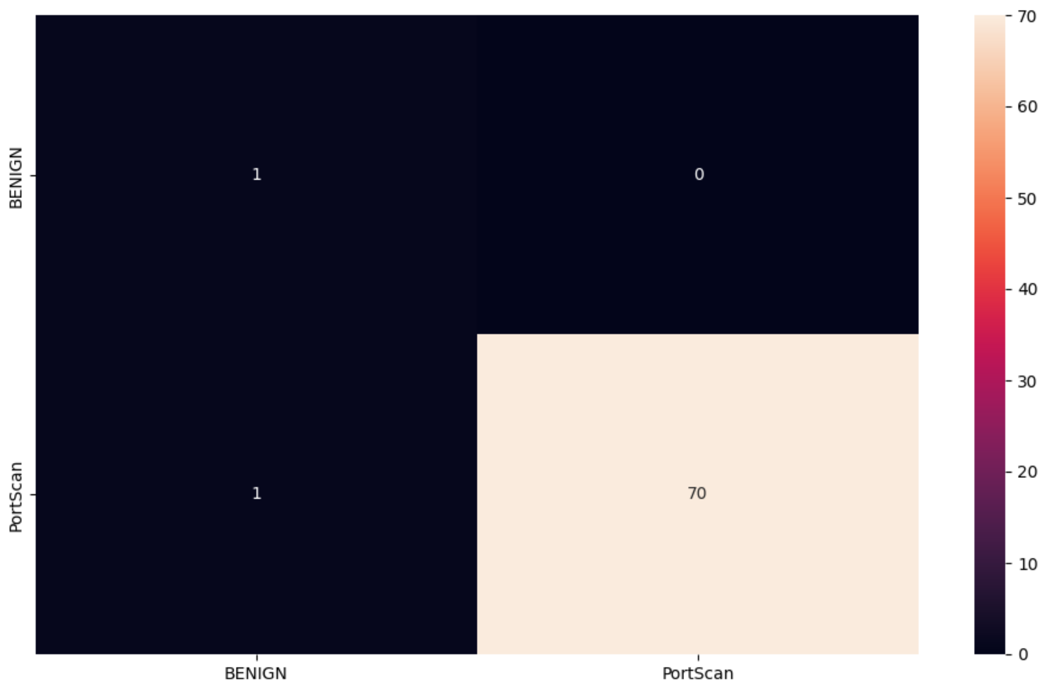

Figure 4.47: PortScan vs Not PortScan model (1000 flows per graph) confusion matrix.

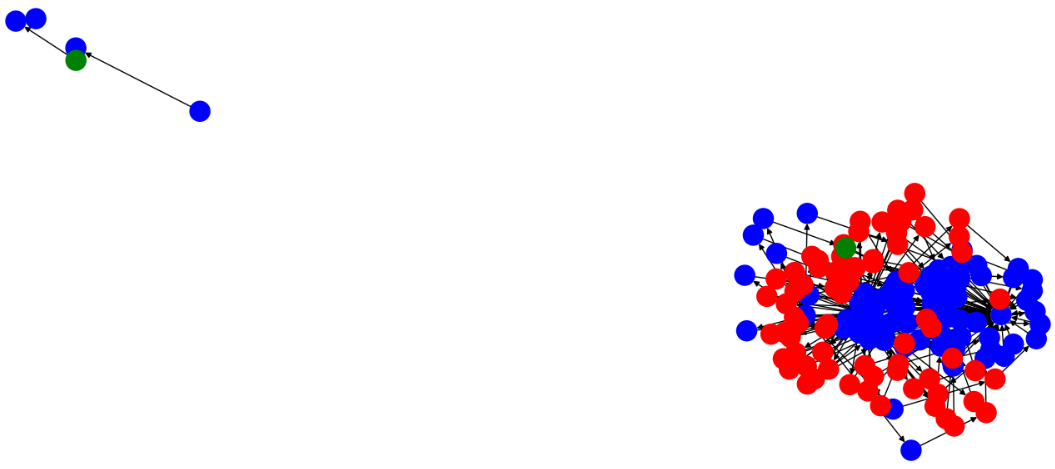

Figure 4.48: PortScan vs Not PortScan model (1000 flows per graph) graphical prediction.

For both cases, despite some minor misclassifications, the results are very good. We have to take into account that we are testing the trained models with fully unseen data. This led us to believe that the similarity in the attack structures that the PortScan and Reconnaissance samples favoured that some models could accurately classify the different samples.

4.4.6 An alternative approach: training with individual graphs

As mentioned at the beginning of this section, throughout the previous experiments, we trained the models according to different batch sizes of flows per graph. However, an alternative approach would be to train the model via feeding individual / isolated graphs.

This would mean that each training sample would be a single graph, which could either describe a benign connection or any sort of attack. The main difference would be that in the same graph there would not be different types of network events or traffic included.

We compared the results yielded by this alternative approach to the ones obtained in the previous subsections:

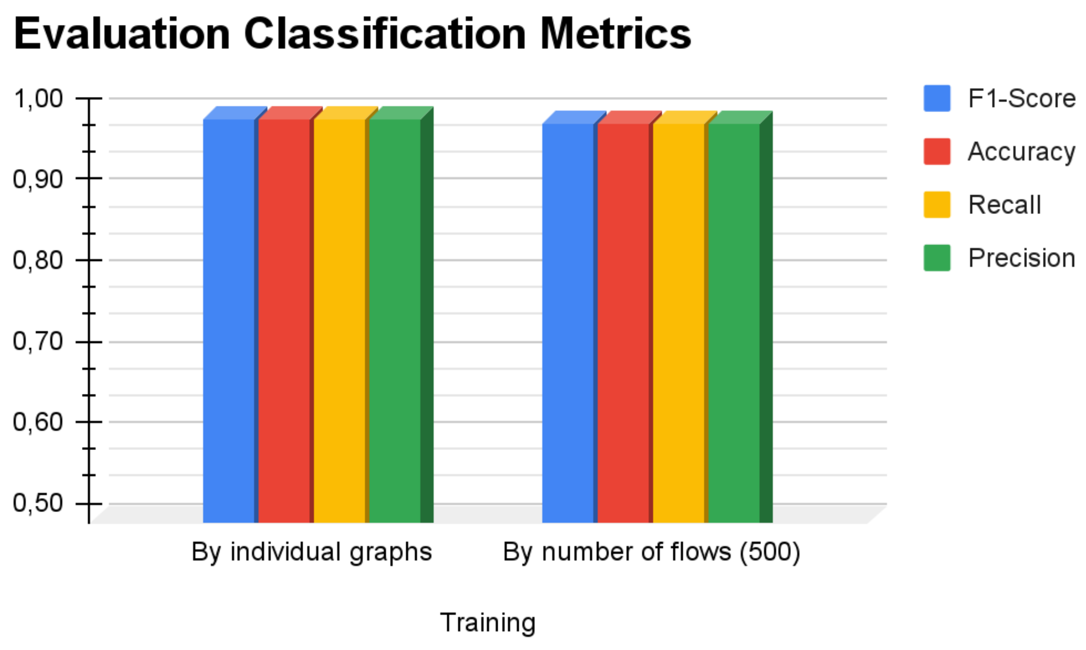

Figure 4.49: PortScan vs Benign classification - comparison of training types.

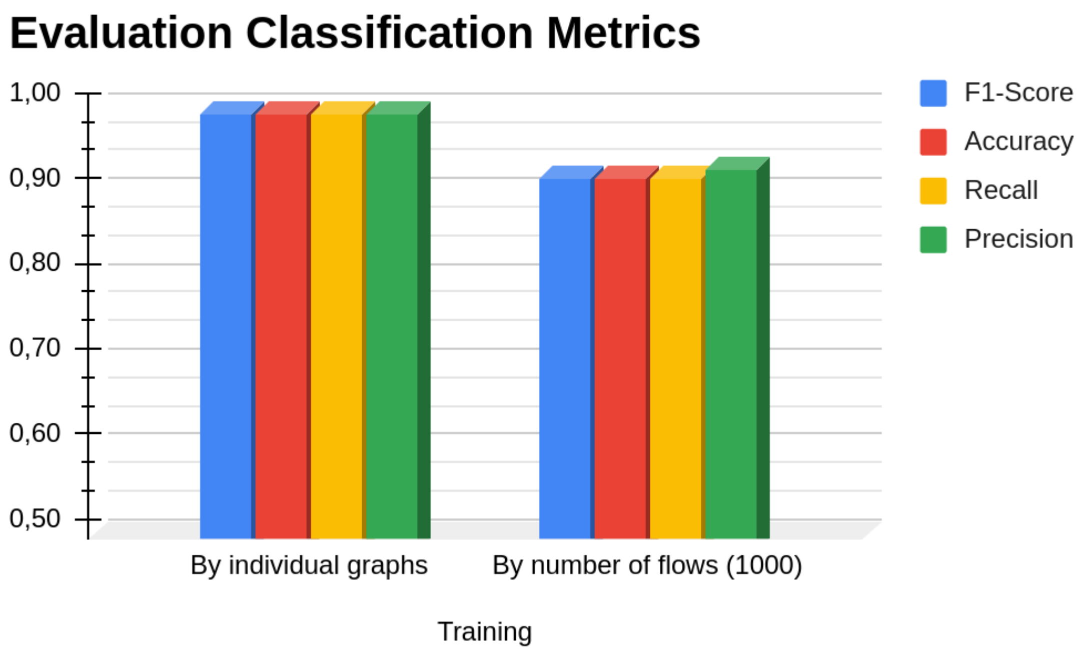

Figure 4.50: PortScan vs Not PortScan classification - comparison of training types.

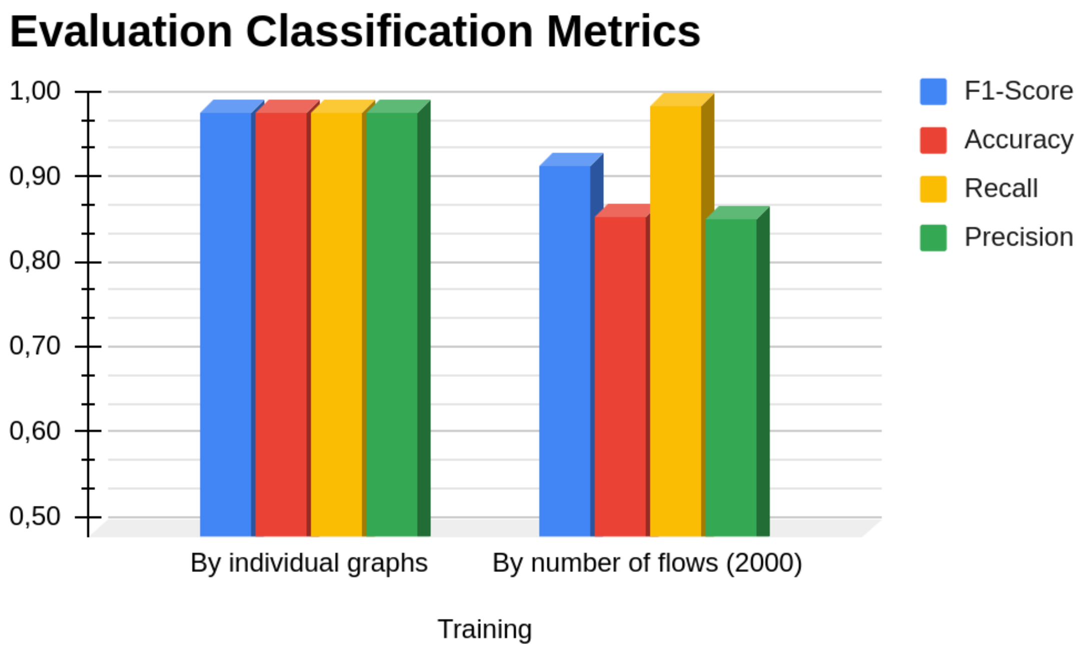

Figure 4.51: Benign, PortScan and DDoS classification - comparison of training types.

We compared the model trained with individual graphs with the best performing models from the previous experiments. What we can clearly see is that the former obtained the highest classification results in all cases. It obtained 0.99 in all evaluation metrics. These results prove that this alternative approach could yield higher results than the one analyzing by batches of flows.

We evaluated one of the models (*PortScan vs Benign*), trained with individual graphs, with the "Reconnaissance" attack samples, as we did in one of the previous subsections:

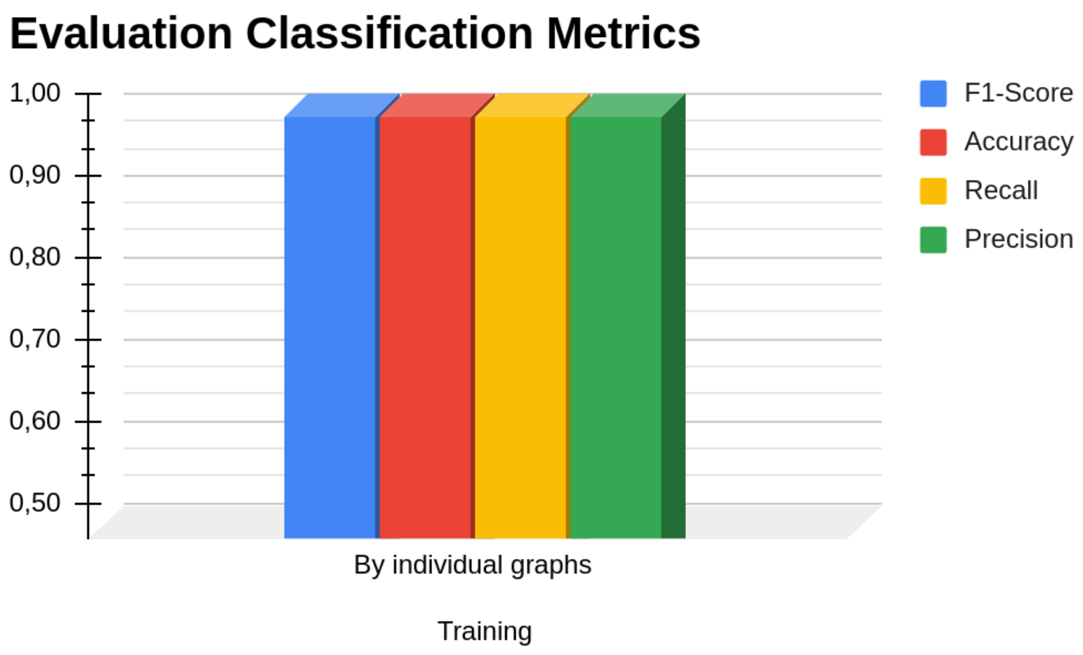

Figure 4.52: Evaluating the PortScan vs Benign model on the "Reconnaissance" attack samples.

The model was able to propely classify all samples and present very high performance. In fact, an ideal approach could be to combine both of the aforementioned methods. On the one hand, we would inspect the traffic by batches of the number of flows, and on the other hand, we would process the collected traffic batch to create individual graphs and feed them into the model.

In a production environment, we could find some problems to create these individual graphs, as benign and malign traffic can come from the same source, hence reducing the separability required to analyse each graph individually.

4.5 Source code

All the source code is stored in a GitHub repository, and each of the modules, files and functions developed has a description of its purpose and functionality. Nevertheless, we will elaborate on the main directories and files included.

The main folders of the source code are structured as follows:

Folder	**Location**	**Contents**
datasets	./	Contains the datasets used for training and validation.
eval	./datasets/	Contains the datasets used for validation.
train	./datasets/	Contains the datasets used for training.
src	./	Contains the main python modules involved in the development and implementation of the model.
runs	./src/	Can be integrated to store records to be presented in the *tensorboard* dashboard.
venv	./	Contains the virtual environment setup used for this project.

Table 4.2: Source code main folders.

The main files of the source code are structured as follows:

Folder	**Location**	**Contents**
README.md	./	Contains relevant information on how to set up the virtual environment and run the scripts.
requirements.txt	./	Contains the list of required pip packages to be installed.
config.ini	./src/	Contains the main variables used in the python modules: file names, batch sizes, sampling rates, etc.
data.py	./src/	Contains the set of functions to process

		the CSV datasets.
graph.py	*./src/*	Contains the set of functions to build the graphs.
graphical_prediciton.py	*./src/*	Contains a module to create visual plots of graph classifications.
metrics.py	*./src/*	Contains the set of functions to compute the main classification metrics.
model.dat	*./src/*	Will be generated once a new model is trained to hold the models' state dictionary.
model.py	*./src/*	Contains the GNN (MPNN) model.
nn.py	*./src/*	Contains the NN module used for comparing with the GNN.
rf.py	*./src/*	Contains the RF module used for comparing with the GNN.
svm.py	*./src/*	Contains the SVM module used for comparing with the GNN.
plot.py	*./src/*	Contains the set of functions used to plot graphs, loss curves, etc.
predict.py	*./src/*	Contains the module to validate the model.
train.py	*./src/*	Contains the module to train the model.

Table 4.3: Source code main files.

5. Conclusions

GNNs were created to work over graph-structured data, which essentially makes them ideal for certain domains. Networks and security are one of them, as network data and traffic can be naturally represented as a graph. Existing approaches analyse network flows individually, effectively obviating the fact that flows present inter-dependencies between them. That is why this novel approach can unlock and create tremendous value.

That being said, as a general conviction, we are going to state the most important conclusions obtained throughout the development of this research project:

1) The different network events, specially the network attacks, should be graphically distinguishable, as differentiability between graphs will favour the GNN model in the classification process. That is why, the dataset used should be previously deeply inspected to create a graph representation tailored to the contents of the data. In our case, we came up with a three-node representation that helped distinguish between PortScan, DDoS and benign traffic.

2) Through the different hypothesis-validation experiments, we came up with the conclusion that solely the graph structure is not enough to classify the network events, as we also need some relevant features information.

3) Our GNN model was able to obtain good classification results from a set of features that did not work for a NN. This worked as proof that the graph structure and the message-passing phase created extra value in comparison to analysing the flows individually.

4) The GNN has proven to be resilient against certain synthetic modifications of the validation datasets (resembling an attacker's behaviours to bypass the security systems). In comparison to three other models, such as a NN, a RF and a SVM, whose prediction capabilities dropped when exposed to the tampered data.

5) The network flow features should be carefully selected to favour generalisation and to avoid overfitting. This means that features that are network-dependent, may affect the model when evaluating it with unseen data. For instance, some features may depend on the capacities of network links, which is something that can drastically change between networks.

6) When analysing the traffic by batches of collected network flows, we have seen that if the batch contains very few flows, it will lack information and present low prediction capabilities if the scenario is complex, on the other hand, if the batch contains a huge number of flows, the graph complexity will heavily increase and also affect the performance. We have seen that batches of around 100 flows already start yielding very good results and are small enough to be rapidly processed (less detection delays).

7) To maximise the detection capabilities, the model should be based on a combination of two approaches, batching by number of network flows and analysing graphs individually. This approach would essentially bring two main advantages, firstly, by batching the traffic, an anomaly-based system would reduce the processing delays and provide early-detection, and secondly, by analysing graphs individually, the model would learn to better differentiate among different graph structures.

6. Future Work

GNNs have proven to be a very powerful alternative for network intrusion detection. However, we are still in the early stages of research. The main challenge faced by the industry is the availability of insightful security datasets that could be used for training. Despite the fact that there are a few datasets available, most of them can not be used to create graph representations of network events. However, some of them can still be synthetically modified to serve the purpose.

Starting from the basis of this research project, future research could continue examining how to implement an anomaly-based system that leverages a GNN model for network intrusion detection. We think that one of the main areas of focus would be creating graph representations that can present differentiability among multiple types of attacks. For the scope of this project, we focused mainly on benign traffic, DDoS and PortScan samples, but there are many other types of threats.

There is still a lot of work to do before implementing this technology in a production environment, but we believe that we have been able to shed a light on the potential that GNNs could have in the future of network security.

References

[1] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-YuanTong. Intrusion detection system: A comprehensive review. Journal of Network and Computer Applications, January, 2013.
https://www.sciencedirect.com/science/article/abs/pii/S1084804512001944

[2] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. An introduction to Graph Neural Networks. September 2, 2022.
https://distill.pub/2021/gnn-intro/

[3] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. IEEE International Joint Conference on Neural Networks, August 4, 2005.
https://ieeexplore.ieee.org/document/1555942/authors#authors

[4] Miquel Ferriol-Galmés, José Suárez-Varela, Jordi Paillissé, Xiang Shi, Shihan Xiao, Xiangle Cheng, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Building a Digital Twin for network optimization using Graph Neural Networks. Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 217, Issue C, November 9, 2022.
https://www.sciencedirect.com/science/article/pii/S1389128622003681

[5] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Unveiling the potential of Graph Neural Networks for network modelling and optimization in SDN. Proceedings of the ACM Symposium on SDN Research (SOSR), pp. 140-151, October 28, 2019.
https://arxiv.org/abs/1901.08113

[6] Weiwei Jiang, and Jiayun Luob. Graph neural network for traffic forecasting: A survey. Expert Systems with Applications Volume, vol. 207, November 30, 2022.
https://www.sciencedirect.com/science/article/abs/pii/S0957417422011654

[7] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. Computational Social Networks volume 6, Article number: 11, November 10, 2019.
https://computationalsocialnetworks.springeropen.com/articles/10.1186/s40649-019-0069-y?ref=https://githubhelp.com

[8] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. ICLR 2018, October 30, 2017.
https://arxiv.org/abs/1704.01212

[9] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. 34th International Conference on Machine Learning, PMLR 70:1263-1272, April 4, 2017.
https://arxiv.org/abs/1704.01212

[10] Matthias Fey, Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. ICLR 2019 (RLGM Workshop), March 6, 2019.
https://arxiv.org/abs/1903.02428

[11] David Pujol-Perich, Jose Suarez-Varela, Albert Cabellos-Aparicio, and Pere Barlet-Ros. Unveiling the potential of Graph Neural Networks for robust Intrusion Detection. ACM SIGMETRICS Performance Evaluation Review, 49(4): 111-117, March 2022.
https://dl.acm.org/doi/10.1145/3543146.3543171

[12] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Towards Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. 4th International Conference on Information Systems Security and Privacy, January, 2018.
https://www.scitepress.org/papers/2018/66398/66398.pdf

[13] Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of Tor Traffic using Time based Features. 3rd International Conference on Information Systems Security and Privacy, January, 2017.
https://pdfs.semanticscholar.org/d76f/32eb3af1a163c0fde624e9fc229671ca75b6.pdf

[14] Juergen Schmidhuber. Deep Learning in Neural Networks: An Overview. Technical Report IDSIA-03-14, April 30, 2014. https://arxiv.org/abs/1404.7828

[15] Aakash Parmar, Rakesh Katariya, and Vatsal Patel. A Review on Random Forest: An Ensemble Classifier. Part of the Lecture Notes on Data Engineering and Communications Technologies book series (LNDECT ,volume 26), December 21, 2018.
https://link.springer.com/chapter/10.1007/978-3-030-03146-6_86

[16] M.A. Hearst, S.T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. Support vector machines. Intelligent Systems and their Applications, IEEE 13 (4): 18 -28, July, 1998.
https://ieeexplore.ieee.org/document/708428

[17] Cynthia Bailey, Lee Chris Roedel, and Elena Silenok. Detection and Characterization of Port Scan Attacks. 2003.
https://cseweb.ucsd.edu/~clbailey/PortScans.pdf

[18] Zhang Chao-yang. DOS Attack Analysis and Study of New Measures to Prevent. International Conference on Intelligence Science and Information Engineering, August 20, 2011.
https://ieeexplore.ieee.org/document/5997473

[19] Diederik P. Kingma, and Jimmy Ba. Adam: A Method for Stochastic Optimization. International Conference for Learning Representations, San Diego, December 22, 2014.
https://arxiv.org/abs/1412.6980

[20] Tianyu Pang, Kun Xu, Yinpeng Dong, Chao Du, Ning Chen, and Jun Zhu. Rethinking Softmax Cross-Entropy Loss for Adversarial Robustness. ICLR 2020, May 25, 2019.
https://arxiv.org/abs/1905.10626

Glossary

NIDS - Network Intrusion Detection System
ML - Machine Learning
DL - Deep Learning
GNNs - Graph Neural Networks
NNs - Neural Networks
IDS - Intrusion Detection System
SIEM - Security Information and Event Management
HIDS - Host Intrusion Detection System
MPNN - Message Passing Neural Network
GCN - Graph Convolutional Network
GAT - Graph Attention Network
CNN - Convolutional Neural Network
COO - Coordinate Format
DDoS - Distributed Denial of Service
DoS - Denial of Service
RF - Random Forest
SVM - Support Vector Machine