UCLouvain

epl

École polytechnique de Louvain

Breaking the Unknown: Deep Learning Strategies for Zero-Day Attacks Detection

Author: **Mohammad Hassan ZAREIE**
Supervisor: **Ramin SADRE**
Readers: **Gorby KABASELE NDONDA, Charles-Henry Bertrand VAN OUYTSEL**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Acknowledgments

I would like to extend my warmest thanks to my friend Quentin Langlois for his invaluable support during this thesis, especially in the fields of artificial intelligence and deep learning. I am also immensely grateful to OpenAI's ChatGPT for its essential role in proofreading and refining the manuscript of this thesis. The prompt used for these corrections can be found in Appendix 8.3.

Contents

1 Introduction **1**

2 Background **3**

2.1 Network Flow Definition 3
2.2 Deep Learning 3
2.2.1 Neuron 4
2.2.2 Layers 4
2.2.3 Architectures 7
2.2.4 Categorical Cross-Entropy Loss 8
2.2.5 Generalized End-to-End Loss 9
2.2.6 Training Process 13
2.3 Clustering Method: Mini-Batch K-Means 14
2.4 Distance-Based Classifiers 15
2.4.1 K-Nearest Neighbors 15
2.4.2 Nearest Centroid 16
2.5 Evaluation Metrics 16
2.5.1 Confusion Matrix and Derived Metrics 17
2.5.2 Balanced accuracy 18

3 Related Work **20**

4 Baseline to Breakthrough: Zero-Day Detection Models **23**

4.1 Method's Genesis 23
4.2 Baseline Models: Simple Classifiers 25
4.3 Embedding Creation Models 27
4.3.1 From Classifier to Embedding Model 28
4.3.2 Model Trained with GE2E Loss 28
4.4 Zero-Day Detection Models 29
4.4.1 First Attempt: Embedding + Classification 29
4.4.2 Second Attempt: Clustering + Embedding + Classification . 30

5 Experiments **32**
5.1 Dataset . . . 32
5.1.1 Presentation . . . 32
5.1.2 Attack Scenarios . . . 33
5.1.3 Labels Distribution . . . 35
5.1.4 Pre-processing . . . 36
5.2 Models Training . . . 39
5.2.1 Deep Learning-based Classifier . . . 40
5.2.2 Clustering Model . . . 40
5.2.3 Embedding Creation Models . . . 41
5.2.4 Distance-based Models . . . 43
5.3 Test Scenarios . . . 44
5.3.1 Zero-Day Attack Simulation . . . 44
5.3.2 Clustering: Number of Benign Clusters . . . 46
5.3.3 K-NN: Number of Inputs per Label . . . 47
5.3.4 K-NN: Value of K . . . 48

6 Results **49**
6.1 Baseline Models . . . 50
6.2 First Type of Zero-Day Detection Models . . . 52
6.3 Second Type of Zero-Day Detection Models . . . 53
6.4 From Models to the Final NIDS . . . 55
6.5 Accuracy per Label . . . 57

7 Conclusion **60**

8 Appendix **62**
8.1 Dataset's Features Description . . . 62
8.2 Test Environment . . . 66
8.3 Manuscript Correction Prompt . . . 66

Chapter 1

Introduction

As the expansion of computer networks continues at an exponential rate and internet usage becomes an integral part of daily life, the number and frequency of cyber-attacks are also surging. This trend poses significant concerns for businesses and internet users alike, as detailed by Saxena et al. [1]. A *Network Intrusion Detection System* (NIDS) is one of the most prevalent and effective security devices used to protect computer networks. However, cyber-attacks are becoming increasingly sophisticated and stealthy, with malware authors employing various evasion techniques to avoid detection by NIDS.

According to the 2019 Internet Security Threat Report (ISTR) by Symantec [2], in 2018, one in ten URLs on the internet was found to be malicious. Additionally, 4,818 websites were compromised by Formjacking code, an attack targeting credit card details on e-commerce checkout web pages. The 2017 ISTR [3] reported over three billion *zero-day attacks* in 2016. Yang Guo [4] defines a zero-day attack as a novel cyber-attack, unknown to the public and cybersecurity community, hence the term "zero-day." It exploits vulnerabilities that are either undisclosed publicly or uses new attacking tactics to evade existing detection tools. Furthermore, the IBM *Cost of Data Breach Report* [5] states that the global average cost of an enterprise's data breach in 2023 was 4.45 million US dollars, representing a 15 percent increase over the last three years. This steep cost is attributed to the fact that only one-third of companies identified data breaches through their own security teams. All these statistics highlight the critical necessity of improving the efficiency and effectiveness of existing NIDS.

Saxena et al. [1] define an intrusion as any unauthorized activity that could potentially damage a computer system. This includes any attack threatening the confidentiality, integrity, or availability of information and services. A NIDS can be either a software or hardware system that passively monitors network traffic

to identify intrusions and alert network administrators. In some instances, it may also automatically counteract the intrusion. There are two primary types of NIDS: *Signature-based Network Intrusion Detection System* (SNIDS) and *Anomaly-based Network Intrusion Detection System* (ANIDS). SNIDS, also known as *Knowledge-based Detection System* or *Misuse Detection*, relies on a database of known intrusions and employs pattern matching to identify threats. Consequently, SNIDS are highly effective in detecting known attacks, resulting in fewer false positives compared to ANIDS. However, their major drawback is the inability to recognize new, previously unseen intrusions, including zero-day attacks. In contrast, ANIDS utilizes machine learning methods to establish a baseline of normal network behavior and detects deviations from this normal behavior. This capability enables ANIDS to detect zero-day attacks, although they generally incur a higher rate of false positives compared to SNIDS. Khraisat et al. [6] point out an additional advantage of ANIDS: it is challenging for attackers to discern what constitutes normal behavior without setting off an alarm.

In this work, I concentrate on ANIDS and propose a novel deep learning model explicitly crafted to detect zero-day attacks. This architecture comprises two models: the first converts netflows into vector representations, known as embeddings, which aim to capture the semantic essence of the flow. The second model then classifies these embeddings, determining whether the original flow is an attack. The rationale for creating these embeddings is that the model tasked with generating them should produce distinct embeddings, dissimilar to those of benign flows, even for previously unseen network attacks. Consequently, the classification model should be able to more effectively identify the embeddings of zero-day attacks as attacks. The efficacy of my models is assessed using the CSE-CIC-IDS2018 dataset [7]. I chose this dataset as it is the most recent one released by the *Canadian Institute for Cybersecurity*, designed specifically for intrusion detection, and covers a diverse range of attack types, rather than being limited to just one.

The rest of this thesis is organized as follows: Chapter 2 provides all the necessary background concepts required to understand the subsequent content. Chapter 3 reviews various papers that also address the detection of zero-day attacks. In Chapter 4, I present the techniques developed during my research. Chapter 5 details the experimental setup and testing scenarios. Chapter 6 presents all the results obtained from these experiments. Finally, Chapter 7 concludes this work.

Chapter 2

Background

In this chapter, I will cover all the concepts necessary to understand the remainder of this work. Firstly, I will define what constitutes a network flow. Subsequently, I will delve into various machine learning concepts, beginning with deep learning. Following this, I will explore a clustering method, then present two distance-based classifiers. The final section will discuss the evaluation metrics I have employed to assess the performance of the different models introduced in this work.

2.1 Network Flow Definition

A network flow is defined as the aggregation of a sequence of network packets transmitted between two devices over a network, representing the end-to-end communication between these devices. These packets are characterized by shared attributes such as source and destination IP addresses, source and destination ports, and the protocol type (e.g., TCP or UDP). There are various methods to define a netflow from a packet sequence. The specific definition used in this work will be detailed in subsection 5.1.1. Throughout this thesis, I will frequently use the term *netflow* to refer to a network flow.

2.2 Deep Learning

This section is primarily based on the book by John D. Kelleher [8]. Throughout this section, I frequently refer to (trainable) parameters and hyper-parameters. The distinction between them lies in the fact that hyper-parameters are set before the training process of the model and are not optimized during it. In contrast, parameters are typically initialized randomly at the beginning of the training, and their final values are learned by the end of the training. Hyper-parameters generally

influence the architecture or the training of the model. The process of testing and finding the optimal values for them is referred to as *Tuning*.

2.2.1 Neuron

The most fundamental building block of deep learning is the neuron, also known as the *deep learning perceptron*. A deep learning neuron is designed to simulate the behavior of a biological neuron. Figure 1 illustrates one such neuron. It takes an input vector $\mathbf{x}$, multiplies each of its dimensions x_j by a corresponding weight w_j (this is equivalent to computing the dot product of $\mathbf{x}$ and a weight vector $\mathbf{w}$), adds a bias b to this weighted sum, and then passes it through a **non-linear**[1] *activation function*. Therefore, if $\mathbf{x}$ has n dimensions, a neuron will have $(n+1)$ weights/trainable parameters.

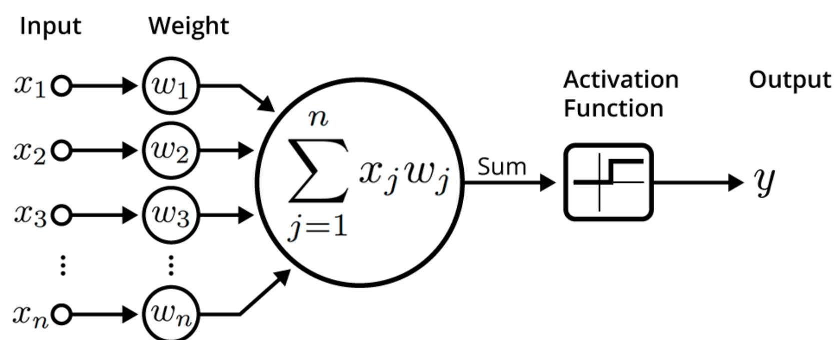

Figure 1: Illustration of an artificial neuron.
Image taken from the website "Becoming Human", at this link: `https://becominghuman.ai/artificial-neuron-networks-basics-introduction-to-neural-networks-3082f1dcca8c`.

2.2.2 Layers

A deep learning layer is essentially a function with one or more inputs and one or more outputs. Some layers have trainable parameters, while others do not. In the following, I will discuss all the layers that I have utilized in this work.

[1] This is the key distinction between the traditional perceptron and the deep learning perceptron.

Fully Connected

A fully connected layer, also known as a *Dense layer*, *Perceptron layer*, or *Linear layer*, computes the following function:

$$f(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + \mathbf{b} \tag{2.1}$$
$$\mathbf{x} \in \mathbb{R}^n,\ f(\mathbf{x}) \in \mathbb{R}^m,\ \mathbf{W} \in \mathbb{R}^{n \times m},\ \mathbf{b} \in \mathbb{R}^m$$

Here, $\cdot$ is the symbol for the dot product, $\mathbf{x}$ represents the input of the layer, $f(\mathbf{x})$ is the output, $\mathbf{W}$ is the *weights matrix*, and $\mathbf{b}$ is the *bias vector*. $\mathbf{W}$ and $\mathbf{b}$ are the layer's trainable parameters, specifically, each scalar within them.
A fully connected layer is essentially a stack of m neurons[2], where m is the sole hyper-parameter of the layer. Its name derives from the fact that each of the m stacked neurons receives $\mathbf{x}$ as input, i.e., $\mathbf{x}$ is fully connected to all the neurons. Consequently, the layer possesses $m * (n + 1)$ trainable parameters, corresponding to the total number of elements in $\mathbf{W}$ and $\mathbf{b}$.
Throughout this thesis, I will use the term *dense* to refer to this layer and *nb_neurons* to denote its hyper-parameter m.

Dropout

A dropout layer computes the following function:

$$f(\mathbf{x}) = \mathbf{x} \odot \mathbf{v} \tag{2.2}$$
$$\mathbf{x} \in \mathbb{R}^n,\ f(\mathbf{x}) \in \mathbb{R}^n,\ \mathbf{v} \in \{0, 1\}^n$$

Here, $\mathbf{v}$ is a random boolean vector and $\odot$ is the symbol for element-wise multiplication, also known as *Hadamard's product*. This operation multiplies each dimension of both vectors, thereby returning a vector of the same length as the inputs. Performing element-wise multiplication between $\mathbf{x}$ and a random boolean vector effectively means randomly setting some dimensions of $\mathbf{x}$ to zero. The number of zeros in $\mathbf{v}$, i.e., the number of zeroed dimensions in $\mathbf{x}$, is the sole hyper-parameter of the dropout layer. This hyper-parameter will be referred to as *drop_rate* throughout this thesis.
Dropout layers are typically connected directly to dense layers and are utilized to prevent a deep learning model from overfitting[3].
By incorporating dropout, we randomly alter the input of the dense layers. Hence, the same input vector $\mathbf{x}$ can be mapped to different output vectors $f(\mathbf{x})$, ensuring

[2] or m deep learning perceptrons, hence the term *Perceptron layer*.

[3] Overfitting is the phenomenon where a model learns the training data too well, to the point where it cannot generalize to other data, such as test data.

that the dense layers almost never see the same input, which helps prevent overfitting.
An important aspect of the dropout layer is that it is only active during the training process. Otherwise, it functions as the identity function, i.e., it returns its input: $f(\mathbf{x}) = \mathbf{x}$.
Lastly, it is important to note that the dropout layer does not have any trainable parameters.

Activation

Activation layers introduce non-linearities into networks and are crucial for training. These layers enable neural networks to discover complex patterns and relationships in data. They are typically applied to the output of neurons, as illustrated in Figure 1. There are many different activation functions available, chosen based on the task the neural network is learning. These functions are generally denoted by the Greek letter σ. In this work, I utilize two distinct activation functions:

1. **Softmax**

$$\sigma(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{C} e^{x_i}} \tag{2.3}$$
$$\mathbf{x} \in \mathbb{R}^C,\ \sigma(\mathbf{x}) \in [0,1]^C,\ x_i \in \mathbb{R}$$

   The softmax activation function is employed in *multi-class classification problems*, where there are more than two potential outcomes, termed *classes*. The number of classes in the problem is denoted as C. This function is typically applied to the output of the final dense layer in an architecture. This layer should have a length equal to the number of classes in the problem (the C in the formula) so that each dimension of the layer is associated with one class. Softmax transforms each dimension of its input vector $\mathbf{x}$ into probabilities, allowing the neural network to generate a distinct probability for each class. This indicates the likelihood that the network's input (different from $\mathbf{x}$) belongs to a particular class. To determine the class to which the network's input corresponds, the class with the highest softmax probability is predicted.

2. **Rectified Linear Unit (ReLU)**

$$\sigma(\mathbf{x}) = \max(\mathbf{x}, 0) \tag{2.4}$$
$$\mathbf{x} \in \mathbb{R}^n,\ \sigma(\mathbf{x}) \in \mathbb{R}^n$$

ReLU is a simple non-linear function used as an activation due to its reduced computational demand compared to softmax. It is primarily employed as the activation function for *hidden* dense layers, i.e., those that are not the final layer of the architecture.

2.2.3 Architectures

In deep learning, an architecture is a sequence of layers, where each layer processes the output of the preceding one. Therefore, an architecture can be viewed as a composition of several functions. One could consider an entire deep learning architecture as a single layer with one or more inputs and one or more outputs.
In my work, I use only one deep learning architecture: the *Multi-Layer Perceptron* (MLP). An MLP consists of a series of deep learning layer blocks. Each block comprises a perceptron layer (hence its name), followed by an activation layer and a dropout layer. The number of blocks in the sequence is a hyper-parameter of the architecture, which I refer to as nb_layers throughout this thesis. Since an MLP is made up of dense layers, it also inherits of its hyper-parameter $nb_neurons$. Figure 2 illustrates an instance of an MLP with the following hyper-parameters: $nb_layers = 1$, $nb_neurons = 64$, accepting a 30-dimensional input vector and Figure 3 presents a summary of the architecture.

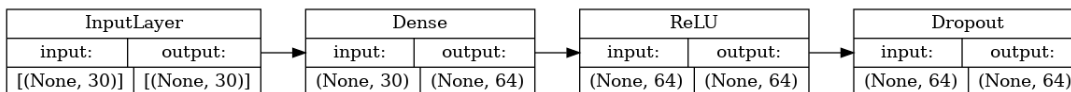

Figure 2: An MLP with the following hyper-parameters: $nb_layers = 1$, $nb_neurons = 64$ that takes as input a 30-dimensional vector.

Both figures, as well as the architecture, have been generated using the *TensorFlow* Python Package [9]. In the figures, we observe that each layer's input and output are matrices, not vectors, with the first dimension being None. This is because, in deep learning, computations are typically executed in *batches* instead of one sample at a time, to reduce computation time. A batch is a set of samples, and its size is known as the *batch size.* Therefore, the None in the figures represents the batch size. The batch size is usually not explicitly specified but is instead inferred by TensorFlow during the model training process. This is why all batch sizes are None in the figures.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 30)]              0

 dense (Dense)               (None, 64)                1984

 re_lu (ReLU)                (None, 64)                0

 dropout (Dropout)           (None, 64)                0

=================================================================
Total params: 1984 (7.75 KB)
Trainable params: 1984 (7.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 3: Summary of the MLP shown on Figure 2.

2.2.4 Categorical Cross-Entropy Loss

A loss function, also known as a *cost function*, measures the difference between a model's output (also known as its *prediction*) and the target value it should have predicted. It acts as a guide, directing the model on how to improve its outputs. Like activation functions, there are many different loss functions, each suited to the specific task the neural network is intended to learn. In this thesis, I have used two different loss functions: the *Categorical Cross-Entropy* loss and the *Generalized End-to-End* (GE2E) loss. The GE2E loss will be discussed in the next section, so let's focus here on the categorical cross-entropy loss.

The categorical cross-entropy loss is employed in multi-class classification problems, much like the softmax activation function, and this is not a coincidence. This loss takes as inputs the probability that the predicted sample belongs to class i, denoted as $\hat{y}_i$, and a binary indicator of whether the target sample belongs to class i, referred to as y_i. In practice, these inputs are vectors of length equal to the number of classes in the problem, denoted as C. Hence, the first input vector is $\hat{\mathbf{y}}$, where each dimension indicates the likelihood that the model's input belongs to a particular class, which corresponds exactly to the output of the softmax activation function. The second input vector is $\mathbf{y}$, where:

$$\mathbf{y} = [y_1,\ y_2,\ \dots\ , y_C]^4 \tag{2.5}$$

$$y_i = \begin{cases} 1 & \text{if the sample belongs to class } i \\ 0 & \text{otherwise} \end{cases}$$

The formula for the categorical cross-entropy loss is:

$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{2.6}$$

$$H(\mathbf{y}, \hat{\mathbf{y}}) \in \mathbb{R},\ \mathbf{y} \in \{0,1\}^C,\ \hat{\mathbf{y}} \in [0,1]^C,\ y_i \in \{0,1\},\ \hat{y}_i \in [0,1]$$

To compute the loss for the entire training dataset, rather than just between one prediction and one sample, the arithmetic mean of the loss across all samples in the training data is taken:

$$CCE = \frac{1}{N} \sum_{i=1}^{N} H(\mathbf{y}_i, \hat{\mathbf{y}}_i) \tag{2.7}$$

Where N represents the number of samples in the training data.

2.2.5 Generalized End-to-End Loss

The GE2E loss function, developed by Li Wan et al. at Google [10], was originally created to address the *Speaker Verification problem* [11]. This loss function is designed to generate *embedding vectors*, more precisely to guide a model to produce high-quality embeddings. First, let's define what an embedding is. Then, I will provide intuition behind the GE2E loss, followed by a more formal and mathematical definition. Finally, we will explore how the GE2E loss can be utilized to create embeddings.

What is an Embedding?

As described by Shabnam Mokhtarani [12], an embedding is a dense vector that represents objects and their relationships. The space formed by these vectors, known as the *embedding space*, quantifies the semantic similarity between different types of objects. For example, if two objects are similar, their embedding vectors should be close to each other in the embedding space. This concept originates from the natural language processing field with word embeddings. In this context, words with similar meanings are expected to have very close embeddings, whereas words with opposite meanings should have embeddings that are far away from each other. Using embeddings instead of raw input in a neural network can significantly enhance the model's ability to learn complex patterns in the data. However, creating and learning to generate these embeddings can be computationally demanding.

[4] This method of representing labels is known as *One-Hot Encoding*.

Intuition behind the GE2E Loss

As mentioned in subsection 2.2.3, processing data in batches is a common practice in deep learning. For the GE2E loss, using batches is not an optimization but a requirement. The GE2E loss processes data in batches, with each batch containing M samples from N different labels, where $N > 1$, i.e., each batch must contain at least two different labels. Consequently, each batch contains $(M * N)$ samples, meaning $batch_size = M * N$. The GE2E loss comes in two variants: the *Softmax GE2E loss* and the *Contrast GE2E loss*. In this thesis, I have exclusively used the softmax version, thus, I will not discuss the contrast version. For brevity, I will refer to the softmax GE2E loss simply as the GE2E loss throughout this thesis. Let's now examine how the GE2E loss functions.
The GE2E loss executes two main steps:

1. **For each label in the batch:** It computes its average embedding, known as the *label centroid*.

2. **For each embedding in the batch:** It pushes the embedding towards its own label centroid while simultaneously pulling it away from the centroids of all other labels.

Figure 4 graphically depicts these steps. In the figure, $\mathbf{e}_{ji}$ represents the embedding vector of the ith sample of the jth label, and $\mathbf{c}_j$, $\mathbf{c}_k$, $\mathbf{c}_{k'}$ are the label centroids of the jth, kth, and k'th labels, respectively. The figure demonstrates how $\mathbf{e}_{ji}$ is pushed towards its own label centroid $\mathbf{c}_j$, while being pulled away from all other label centroids within the batch, $\mathbf{c}_k$ and $\mathbf{c}_{k'}$.

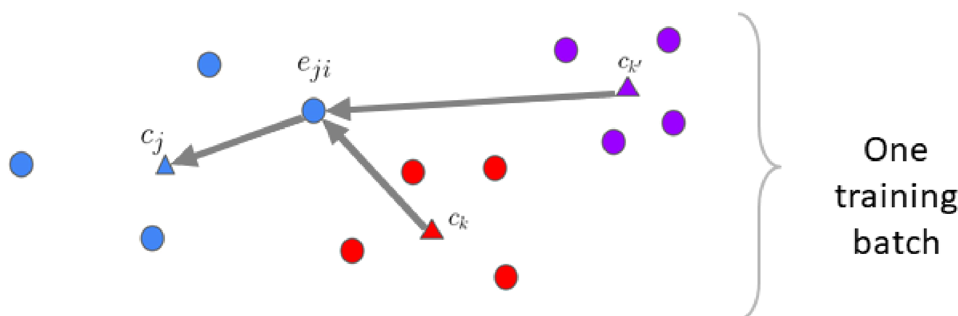

Figure 4: The GE2E loss pushes the embedding towards the centroid of its true label and away from the centroids of all other labels.
Image taken from the original GE2E loss paper [10].

Formal Definition of the GE2E Loss

Now that we have an intuition of how the GE2E loss works, let's now define the loss more formally. In my thesis, I have used an implementation of the GE2E loss that differs from the original one presented by Li Wan et al. [10]. This alternative implementation, that I have found on the *Yui-MHCP*'s GitHub repository [13], maintains the intuition of the original loss but employs a different formula. It involves using the categorical cross-entropy loss $H(\mathbf{y}, \hat{\mathbf{y}})$, as defined in subsection 2.2.4, but with specific arguments. Here is the formula for the GE2E loss for one batch containing M embeddings from N different labels:

$$GE2E = \frac{1}{N * M} \sum_{j=1}^{N} \sum_{i=1}^{M} L(\mathbf{e}_{ji},\ \mathbf{y}_j) \tag{2.8}$$

$$\tag{2.9}$$

Similar to the categorical cross-entropy loss, this is equal to the arithmetic mean of the loss applied to all embeddings in the batch. Let's define the loss for an individual embedding:

$$L(\mathbf{e}_{ji},\ \mathbf{y}_j) = H\left[softmax[\mathbf{s}(\mathbf{e}_{ji})],\ \mathbf{y}_j\right] \tag{2.10}$$

$$\mathbf{s}(\mathbf{e}_{ji}) = [similarity_metric(\mathbf{e}_{ji}, \mathbf{c}_1),\ \dots, similarity_metric(\mathbf{e}_{ji}, \mathbf{c}_N)] \tag{2.11}$$

$$\mathbf{c}_k = \frac{1}{M} \sum_{k=1}^{M} \mathbf{e}_{km} \tag{2.12}$$

$$\mathbf{y}_j = [y_1,\ y_2,\ \dots, y_N] \tag{2.13}$$

$$y_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

In this formula, $\mathbf{s}(\mathbf{e}_{ji})$ denotes the *similarity vector* associated with the embedding vector of the ith sample of the jth label $\mathbf{e}_{ji}$. $\mathbf{y}_j$ represents the one-hot encoding of label j, as defined in Equation 2.5, with one important difference: the length of the vector is not equal to the total number of labels but rather the number of labels in the batch, which is N. This is a sort of one-hot encoding that is local to the batch. $\mathbf{c}_k$ is the label centroid of the kth label. Each dimension k of the similarity vector quantifies the similarity between $\mathbf{e}_{ji}$ and $\mathbf{c}_k$. Applying a softmax function to the similarity vector allows for the handling of any similarity metric, ensuring that the maximum similarity is one and the minimum is zero.

Similarity is a core component of the GE2E. In the previous section, I only talked about distances, but in practice, the GE2E loss operates on a more general concept: similarity. The premise is that the more semantically identical two objects are,

the higher their similarity. Hence, a distance metric can also serve as a similarity metric. To be exact, we have to take the opposite or inverse of the distance metric because the closer or more similar two objects are, the less the distance between them. This is the opposite of the premise of similarity. The similarity metric is a hyper-parameter of the GE2E loss, allowing one to choose any metric of their choice.

The GE2E loss operates on batches. Therefore, the similarity vector of an embedding becomes the *similarity matrix* of the batch. Let us define the ***softmax similarity matrix***, denoted by **SS**:

$$\mathbf{SS} = \begin{bmatrix} softmax[\mathbf{s}(\mathbf{e}_{11})] \\ softmax[\mathbf{s}(\mathbf{e}_{21})] \\ ... \\ softmax[\mathbf{s}(\mathbf{e}_{NM})] \end{bmatrix}$$

From this matrix, we can mathematically formulate the second step of the GE2E loss, as explain in the previous section, in the following manner:

$$\forall j, k \ 1 \leq j, k \leq N, \ \forall i \ 1 \leq i \leq M$$

$$\mathbf{SS}_{ji,k} = 1, \ k = j \tag{2.14}$$

$$\mathbf{SS}_{ji,k} = 0, \ \forall k \neq j \tag{2.15}$$

Equation 2.14 compels the embedding $\mathbf{e}_{ji}$ to be completely similar to its own label centroid $\mathbf{c}_j$, while Equation 2.15 ensures $\mathbf{e}_{ji}$ is completely dissimilar to all other label centroids, $\mathbf{c}_k$ and $\mathbf{c}_{k'}$ in Figure 4. This means we ideally want a softmax similarity matrix that equals the following matrix:

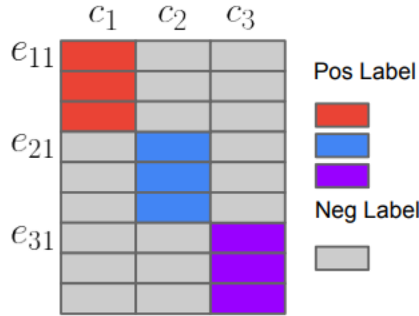

Figure 5: The ideal softmax similarity matrix that should be produced by a model trained with the GE2E loss.
Image taken from the original GE2E loss paper [10].

Using the GE2E Loss to Create Embeddings

Now, let's explore how to utilize the GE2E loss for creating embeddings. For this purpose, an appropriate architecture is required. For example, we can use the MLP described in subsection 2.2.3. The MLP takes a sample as input, and after being trained with the GE2E loss, it will output the corresponding embedding.

2.2.6 Training Process

The fundamental property of a loss function is that a lower value indicates a better model. Consequently, training a model essentially becomes an optimization problem, more specifically a minimization problem. In deep learning, this minimization is performed using the *Back-propagation algorithm* [14], which calculates the gradient of the architecture, particularly the gradient of each function comprising the architecture. This is complemented by the *Gradient Descent algorithm* [15][5], which applies these gradients to the trainable parameters to minimize the loss. We refer to such an algorithm as an *optimizer*. Both the optimizer and the loss function are hyper-parameters of the training process.

The back-propagation algorithm and the gradient descent algorithm are generally applied multiple times on a deep learning model until a stopping criterion is met. Each application of these algorithms is commonly referred to as an *epoch*. For my work, I decided to train my models until they ceased to learn, which is formally recognized when the validation loss (i.e., the loss calculated on the validation set) no longer decreases. The validation set is a subset of the dataset used to evaluate the model's performance during the training process. Practically, I implemented this by defining two more hyper-parameters for my training process:

1. **Threshold**: This refers to the minimum difference, between the best validation loss (i.e., the lowest value for the validation loss) and the validation loss of the current epoch, needed to consider that the model has made progress and learned something new.

2. **Patience**: This denotes the maximum number of epochs without improvement in the validation loss. If the number of epochs without improvement in the validation loss exceeds patience, the training process is halted.

[5] In practice, more advanced variants of this algorithm, such as *Adam* [16], are now commonly used

2.3 Clustering Method: Mini-Batch K-Means

In this section, I explore the only clustering method used in my work: the *Mini-Batch K-Means algorithm* (MBK-Means), developed by David Sculley [17]. This method is a variation of the classic *K-Means algorithm*, tailored to scale with very large datasets. Like the K-Means algorithm, its only hyper-parameter is K, which represents the number of clusters to be formed, i.e., the number of centroids to be generated. Unlike K-Means, which processes the entire dataset in each iteration, MBK-Means operates on batches. This approach significantly reduces the maximum amount of memory used in each iteration. Specifically, in each iteration, K-Means computes distances between all samples in the dataset, necessitating a spatial complexity of $\Theta(n) = n^2$, where n is the number of samples. MBK-Means also computes these distances but with a reduced n, where $n = batch_size$. This difference is why MBK-Means is capable of handling very large datasets.
One might think that the quality of the clusters produced by MBK-Means is inferior to that of K-Means. However, in practice, the difference in quality is often minimal, as demonstrated in the reference paper [17] and illustrated in Figure 6.

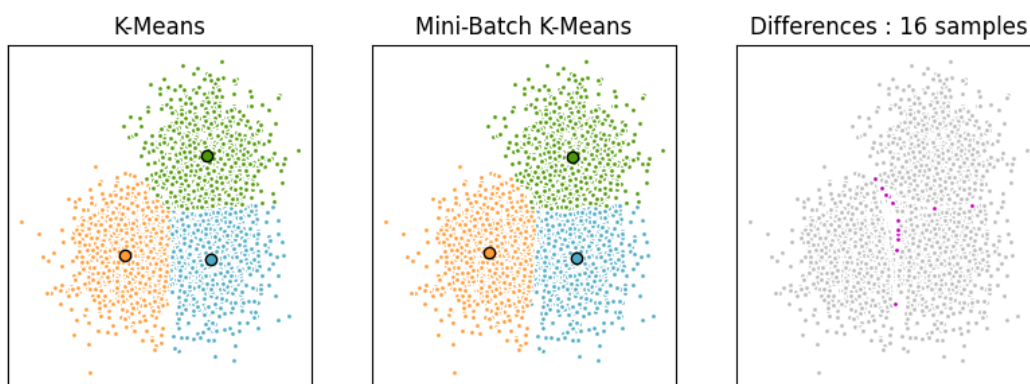

Figure 6: Comparison of the K-Means and Mini-Batch K-Means algorithms, including the differences in their outputs.
These illustrations were generated using a tutorial from the Scikit-Learn website: `https://scikit-learn.org/stable/auto_examples/cluster/plot_mini_batch_kmeans.html#sphx-glr-auto-examples-cluster-plot-mini-batch-kmeans-py`.

2.4 Distance-Based Classifiers

In this section, I will explore two distance-based classifiers that I have used throughout this thesis. First, I will discuss the K-Nearest Neighbors (K-NN) classifier, followed by a variation of K-NN known as the Nearest Centroid (NC) classifier.

2.4.1 K-Nearest Neighbors

Developed by Evelyn Fix et al. [18], the K-NN is a straightforward classifier. Its main advantage lies in its simplicity and its training which involves no computation. In fact, training a K-NN classifier consists merely of selecting a set of points. However, the logic and computation are concentrated in the prediction phase, which can be computationally intensive and require a significant amount of memory. Predicting the label y of a test point $\mathbf{x}$, based on the training points $\mathbf{X}$ and their labels $\mathbf{y}$, involves two steps:

1. **Distance Computation:**
   The distances between $\mathbf{x}$ and all training points $\mathbf{X}$ are computed. This step can be very computationally intensive and require a substantial amount of memory if the number of training points is very large.

2. **Majority Vote:**
   After computing all distances, the K nearest points to $\mathbf{x}$ and their labels are selected. The label that appears most frequently among these points will be assigned to y.

This classifier necessitates two design choices:

1. **Training Points:** Determining the quantity and which points are included in $\mathbf{X}$. This is a balance between the quality of the model and the computation time/memory required.

2. **Choice of K:** Deciding how many nearest points are considered to perform the majority vote.

The K-NN classifier has two main drawbacks:

1. It does not scale with large datasets, and the choice of selected samples is not trivial, as it varies depending on the problem.

2. Selecting an appropriate value for K may also be non-trivial and dependent on the problem.

2.4.2 Nearest Centroid

The NC classifier is a specialized form of the K-NN classifier, arising from specific design choices about the K-NN classifier. These choices are as follows:

1. **Training Points:** All points in the dataset are considered, but they are not directly used in the K-NN. Instead, the centroid of each label is computed (i.e., the mean of each label), and these centroids are used in the K-NN. As a result, the number of points in the NC classifier equals the number of labels in the dataset.

2. **Choice of K:** K is set to one.

The name "nearest centroid" stems from these design choices.
I decided to also use this classifier in my thesis as it addresses the two main drawbacks of the K-NN classifier:

1. **Scalability with Large Datasets:** The NC classifier is effective for large datasets since the number of training points is small (equal to the number of labels).

2. **Simplified K Selection:** There is no need to choose a value for K as it is fixed at one.

2.5 Evaluation Metrics

In this section, I detail the evaluation metrics used throughout this work to assess the performance of various models I have tested. Evaluation metrics, as the name suggests, are tools for evaluating models. Similar to loss functions, metric functions take the model's output and the target value it should have predicted as inputs, and then return a scalar quantifying the difference between the two. The choice of metrics depends on factors like the model type, input type, problem type, and the desired method of measuring difference.
In the context of NIDS, the objective is to determine whether a given netflow is an attack or a benign flow. Consequently, all my NIDS models perform binary predictions, meaning they return either true or false depending if the flow was considered as an attack or not, according to model, respectively. This places us in the realm of *binary (class) classification problems*. Thus, I will focus here on the "binary versions" of the evaluation metrics I have utilized.
Throughout this thesis, I have employed three different evaluation metrics: the *Balanced Accuracy*, the *True Positive Rate* (TPR), and the *False Alarm Rate* (FAR), also known as the *False Positive Rate*. Interestingly, all these three metrics

can be computed using another metric: the *Confusion Matrix* (CM). Below, I first explain the CM, then show how to use it to calculate the TPR, FAR, and balanced accuracy.

2.5.1 Confusion Matrix and Derived Metrics

In its binary version, the CM is a table with four entries:

Figure 7: Binary Confusion Matrix.
Image take from the website "Towards Data Science", at this link: `https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826`.

Let's examine each of these entries:

- **True Positive (TP):** The model correctly predicted the input flow as an attack.
- **False Positive (FP):** The model incorrectly predicted the input flow as an attack while it was a benign flow.
- **False Negative (FN):** The model incorrectly predicted the input flow as a benign flow while it was an attack.
- **True Negative (TN):** The model correctly predicted the input flow as a benign flow.

From these, we can compute the TPR and FAR. Let's see how and what these two metrics measure:

- **TPR:** Also known as *Sensitivity* or *Recall*, measures the proportion of **attacks** correctly identified by the NIDS.
  Ideally, NIDS should have a high TPR, close to 1.

- **FAR:** Measures the proportion of **benign flows** incorrectly identified **as attacks** by the NIDS.
  Ideally, NIDS should have a low FAR, close to 0.

Now, let's look at the formulas for both metrics:

$$TPR = \frac{TP}{TP + FN} \tag{2.16}$$

$$FAR = \frac{FP}{FP + TN} \tag{2.17}$$

2.5.2 Balanced accuracy

Balanced accuracy is a metric specially designed to deal with unbalanced datasets. In the context of NIDS, this means datasets that contain much more benign flows than attack flows. Balanced accuracy computes the accuracy of each class individually and returns the arithmetic mean of all computed accuracies. By doing this, each class's accuracy has the same impact, regardless of the number of samples. Firstly, here is the formula for standard accuracy:

$$Accuracy = \frac{\text{number of flows correctly predicted}}{\text{total number of flows}} \tag{2.18}$$

Then, here is the formula for the binary version of balanced accuracy:

$$Balanced\ Accuracy = \frac{\text{Benign Accuracy} + \text{Attack Accuracy}}{2} \tag{2.19}$$

$$Balanced\ Accuracy = \frac{1}{2} \cdot \frac{TP}{TP + FN} + \frac{1}{2} \cdot \frac{TN}{TN + FP} \tag{2.20}$$

We can also compute balanced accuracy using TPR and FAR, as follow:

$$\frac{TN}{TN + FP} = \frac{\mathbf{FP} + TN - \mathbf{FP}}{FP + TN} = 1 - \frac{FP}{FP + TN} = 1 - FAR$$

$$Balanced\ Accuracy = \frac{1}{2} TPR + \frac{1}{2} (1 - FAR) \tag{2.21}$$

From Equation 2.21, it can be concluded that the balanced accuracy is the arithmetic mean of the TPR and the complement of the FAR. The latter is also known as the *True Negative Rate*, which refers to the proportion of benign flows correctly identified by the NIDS.

Balanced accuracy v.s. Accuracy

The problem with metrics that do not take class imbalance into account is that they are heavily biased by the majority class. To demonstrate this, let's compare balanced accuracy with standard accuracy. Imagine a highly unbalanced dataset with 950 benign flows and only 50 attack flows, and a very dumb model that always predicts a flow as benign ($def\ predict(flow):\ return\ "Benign"$).
This very bad classifier will show high accuracy since it always predicts "Benign", the predominant label in the dataset:

$$Accuracy \ = \ \frac{950}{950 + 50} \ = \ 95\ \%$$

In contrast, the same model will show low balanced accuracy:

$$Balanced\ Accuracy \ = \ \frac{1}{2}\,\frac{950}{950} \ + \ \frac{1}{2}\,\frac{0}{50} \ = \ 50\ \%$$

Chapter 3

Related Work

The detection of zero-day attacks is a critical and much-discussed issue in network security. Accordingly, extensive efforts have been focused on improving NIDS to effectively recognize these attacks. In this chapter, I will review three papers focusing on the detection of zero-day attacks.

I will begin with a paper by Alefiya Hussain et al. [19], which proposes a NIDS that combines supervised and unsupervised learning techniques. Supervised learning involves training a model on labeled data, typically used for tasks like classification, while unsupervised learning employs unlabeled data, generally for clustering and discovering intrinsic relationships within the data. The NIDS proposed by Alefiya Hussain et al. consists of two main components:

1. **Offline IDS**, based on supervised learning, designed to detect and classify known attacks.

2. **Online IDS**, based on unsupervised learning, aimed at detecting benign netflows or unknown attacks.

According to the authors, the advantage of this IDS is its ability to achieve an accuracy of 80% in detecting unknown attacks. However, the authors also acknowledge a significant limitation: the low accuracy in detecting benign netflows, leading to a high rate of false negatives. In contrast, the NIDS I propose in this thesis does not suffer from this problem and demonstrates strong performance in identifying benign flows, as will be detailed in the results chapter (cf. chapter 6).

The second paper, written by Hanan Hindy et al. [20], introduces an auto-encoder based NIDS aimed at minimizing false negatives while maintaining a high true positive rate. An auto-encoder, typically employed in unsupervised learning, is composed of two parts:

1. **An encoder** which compresses input data into a lower-dimensional vector.
2. **A decoder** which reconstructs the original data from the encoder's output vector.

In their study, Hanan Hindy et al. developed an auto-encoder trained exclusively on benign netflows, aiming to create a model proficient in recognizing normal network activity. When predicting whether an unknown netflow is an attack, the model calculates the difference between the original netflow and the decoder's output. If this difference exceeds a predetermined threshold, the netflow is classified as an attack. The primary strength of this approach is its training, which only includes benign flows, thus ensuring that the model is really designed to detect zero-day attacks.
However, I see a notable limitation in their proposed NIDS, particularly in the setting of the threshold value. The authors acknowledge the importance of this threshold as a key hyper-parameter in classifying a netflow as an attack. According to me, while this threshold may work well on their test dataset, it could perform poorly on other datasets or in real-world scenarios. My approach diverges from this by not depending on such a threshold, which I believe could lead to more reliable and versatile performance across different datasets and practical applications.

The final paper I wish to discuss was written by Mahdi Soltani et al. [21], presenting a four-phase framework that integrates deep learning, clustering, and open set recognition methods for zero-day attack detection. Open set recognition is a concept in machine learning where the model is trained to classify not only known labels but also unknown ones it has never seen during training. These models do not make the assumption that all classes they will predict are known and included during the training phase, as traditional models do. The four phases of Mahdi Soltani et al.'s framework are as follows:

1. **Open Set Recognition:** This phase is crucial for identifying both known attacks and new types of netflows, which could be either benign or zero-day attacks.
2. **Clustering:** In this step, new types of netflows are collected and similar netflows are clustered for future analysis.
3. **Supervised Labeling:** Here, an expert categorizes the clusters of previously unidentified netflows. The main advantage of this framework is the ability to label entire clusters rather than individual netflows. Each cluster is classified into one of four subcategories: known malicious netflows, new attacks, unseen benign netflows, or temporary anomaly netflows.

4. **Updating the Model:** The final step involves updating the deep learning model with these new labels. This update allows the model to detect zero-day attacks.

I chose to delve more into this paper than the others because the framework it proposes closely aligns with the concept I am exploring in my thesis. However, there is a notable difference: my approach does not require the clustering phase. I aim to develop a model that by design forms a new cluster for unknown attacks. Thus, the third and fourth steps of their framework, while potentially beneficial for enhancing zero-day attack detection, are optional in my model. My method can autonomously detect such attacks without these two steps.

Chapter 4

Baseline to Breakthrough: Zero-Day Detection Models

In this chapter, I will explain how I progressed from simple netflow classifiers to models capable of classifying unseen zero-day attacks. This chapter is organized as a kind of timeline, representing my progress throughout my thesis. I begin by discussing the origins of my idea, then delve into simple netflow classifiers, and afterwards, I detail the models that I have designed to create embeddings from netflow. To conclude this chapter, I will discuss my two attempts at creating a NIDS that can fully detect zero-day attacks.

4.1 Method's Genesis

The objective of my thesis is to develop a model capable of identifying zero-day attacks. To achieve this, I aimed to create embeddings from netflow and then perform the classification on these embeddings, instead of simply training a model to classify netflow directly. As we have seen in section 2.2.5, an embedding is a vector that captures the semantic meaning of an object. In our case, it should capture the essence of a netflow. Therefore, if I succeed in creating a robust model for embedding creation, i.e., a model that truly learns to represent netflows, it should also effectively represent netflows it has never seen, specifically attack flows. Then, I can conduct the classification based on these embeddings, which encapsulate the meaning of the netflow. This was my primary idea.

Another important aspect I considered was the classification of these embeddings. If I have a good embedding creation model, the embeddings of unseen attacks should form a distinct cluster, separate from other known attack clusters. Thus, at some point, we might decide to label and add this new cluster to the

classifier's data, thereby making it more efficient at recognizing this new type of attack. Consequently, I was looking for a classifier that, on one hand, takes into account the notion of distance/clustering, and on the other hand, is easy to retrain, meaning it is straightforward and time-efficient to integrate new data for prediction. One classifier that came to mind, meeting these requirements, is the *K-Nearest Neighbors* (K-NN) classifier. As explained in subsection 2.4.1, K-NN is a distance-based classifier that requires no computation during its training phase. This is why I decided to use K-NN classifiers for classification of netflow's embeddings. In subsection 2.4.2, we saw that the *Nearest Centroid* (NC) classifier is a specific configuration of the K-NN classifier, designed to address its main drawbacks. Therefore, I also decided to test this classifier and compare which of the two performs better on netflows and netflow's embeddings.

In a nutshell, Figure 8 depicts the conventional method for classifying netflows, while Figure 9 illustrates the novel approach I aim to explore in this thesis. Each box represented in both figures corresponds to a topic covered in the following sections: the simple classifier is addressed in section 4.2, the embedding models are discussed in section 4.3, the distance-based classifier has already been presented in section 2.4, and finally, my original concept, encapsulated in Figure 9, will be detailed in section 4.4.

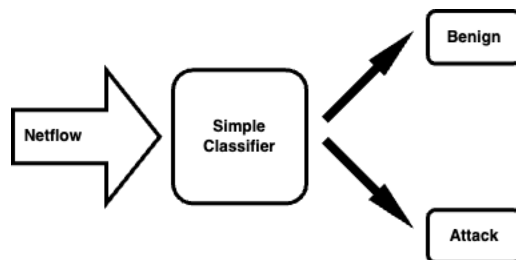

Figure 8: Conventional method for classifying netflows.
This image was created using the online tool *DrawIO*.

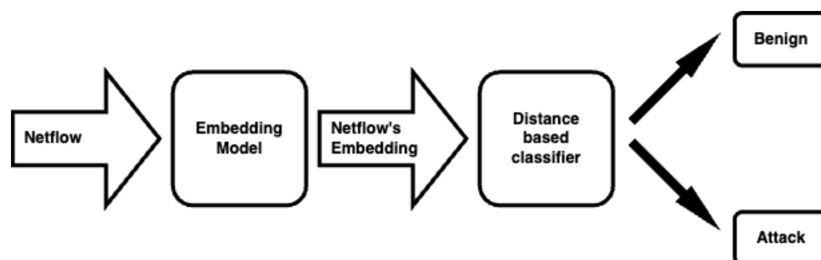

Figure 9: My novel approach for classifying netflows.
This image was created using the online tool *DrawIO*.

4.2 Baseline Models: Simple Classifiers

In this section, I define all the simple models I used to classify netflows. I initially created these models at the beginning of my thesis, primarily to test my dataset, ensuring that the pre-processing was adequate and the data were of good quality. Subsequently, I decided to retain these models as baselines to compare against the more elaborate models I developed later, thereby measuring the performance gains.

I utilized two types of simple classifiers: deep learning-based classifiers and distance-based classifiers. I have already explained how the distance-based K-NN and NC classifiers work in section 2.4. Therefore, in this section, I will focus solely on the deep learning-based classifier. This model is composed of a MLP layer (cf. subsection 2.2.3) followed by a *classification layer*. I will refer to this model as *MLP classifier* throughout the rest of the thesis. Let us first define what a classification layer is before delving into the model.

Classification Layer

As the name suggests, a classification layer is used for classification purposes. This layer's only hyper-parameter is the number of different classes, referred to as $nb_classes$. It consists of a dense layer with $nb_neurons = nb_classes$, followed by a softmax activation layer. As explained in section 2.2.2, this setup allows us to output a probability for each class, indicating the likelihood that the input belongs to each respective class. Figure 10 shows the architecture of a classification layer with $nb_classes = 15$.

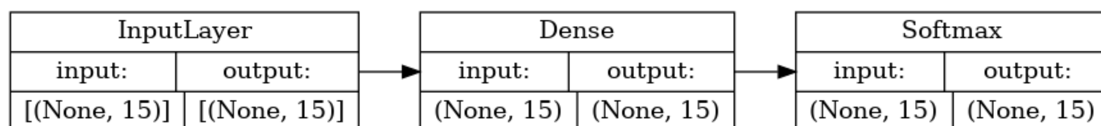

Figure 10: A classification layer with $nb_classes = 15$.

Deep Learning based Classifier

The simple deep learning-based classifier consists of an MLP layer followed by a classification layer. It takes as input a netflow with 78 features and outputs a 15-dimensional probability vector. In other words, the model is designed to distinguish between benign flows and 14 types of attacks. The MLP is configured with the following hyper-parameters: $nb_layers = 3$, $nb_neurons = 64$, and the classification layer is set with: $nb_classes = 15$. Figure 12 illustrates its architecture. We can clearly observe the three blocks in blue, each comprising a Dense layer, a ReLU layer, a Dropout layer, and the classification layer in green at the end. To convert the outputted probability vector into a label, we predict the label with the highest probability. For example, if $\mathbf{x}$ is an input netflow and $\hat{y}$ is the prediction, we have:

$$\hat{y} = \arg\max[model.predict(\mathbf{x})] \qquad (4.1)$$
$$\mathbf{x} \in \mathbb{R}^{78}, \ \hat{y} \in \{1, ..., 15\}$$

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 78)]	0
dense (Dense)	(None, 64)	5056
re_lu (ReLU)	(None, 64)	0
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 64)	4160
re_lu_1 (ReLU)	(None, 64)	0
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 64)	4160
re_lu_2 (ReLU)	(None, 64)	0
dropout_2 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 15)	975
softmax (Softmax)	(None, 15)	0

Total params: 14351 (56.06 KB)
Trainable params: 14351 (56.06 KB)
Non-trainable params: 0 (0.00 Byte)

Figure 11: Summary of the architecture shown in Figure 12.

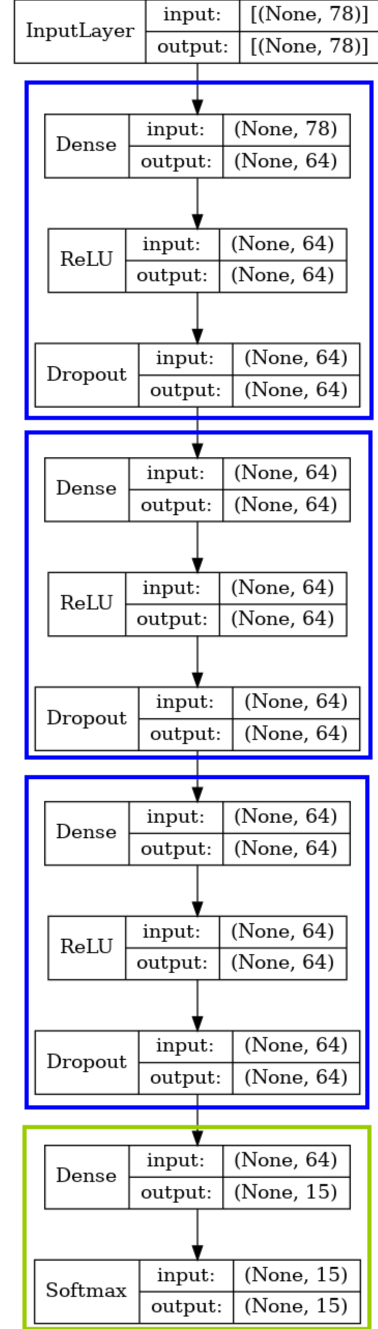

Figure 12: Architecture of the MLP classifier.

4.3 Embedding Creation Models

In this section, I will present the models I designed to create embeddings from netflows. I will start with a simple method based on the MLP classifier defined in the previous section. Then, I will introduce a more elaborate model trained with the GE2E loss, as presented in subsection 2.2.5. Interestingly, both models share the same architecture, differing only in their training approach. Figure 14 illustrates this architecture, and Figure 13 provides a summary of the architecture.

In this thesis, I really wanted to evaluate whether the GE2E loss is an effective option for creating embeddings from netflows. The decision to use the same architecture for both models was intentional, allowing me to accurately measure the impact of the GE2E loss on embedding creation, compared to a simpler method. Utilizing different architectures might have introduced performance variations attributable to the architecture rather than the loss itself.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 78)]	0
dense (Dense)	(None, 64)	5056
re_lu (ReLU)	(None, 64)	0
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 64)	4160
re_lu_1 (ReLU)	(None, 64)	0
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 64)	4160
re_lu_2 (ReLU)	(None, 64)	0

Total params: 13376 (52.25 KB)
Trainable params: 13376 (52.25 KB)
Non-trainable params: 0 (0.00 Byte)

Figure 13: Summary of the architecture shown in Figure 14.

4.3.1 From Classifier to Embedding Model

My first embedding creation model is essentially a repurposed version of the MLP classifier from section 4.2. After training the simple MLP classifier, I removed the classification layer and the last dropout layer from its architecture. This results in the architecture depicted in Figure 14. Consequently, the output of the model becomes the output of the last ReLU layer, which is a 64-dimensional vector initially trained for classifying netflows. Therefore, this vector captures the semantic essence of a netflow, which clearly corresponds to the definition of an embedding as stated in section 2.2.5. I will refer to this model as *MLP Embedding Model* throughout the rest of the thesis.

4.3.2 Model Trained with GE2E Loss

My second embedding creation model involves training the exact same architecture as the previous model (Figure 14) but using the GE2E loss defined in subsection 2.2.5. In this setup, the output of the last ReLU layer becomes our embedding and therefore, the input for the loss function, i.e., the $\mathbf{e}_{ji}$ in Equation 2.10. I will refer to this model as *GE2E Embedding Model* for the remainder of the thesis.

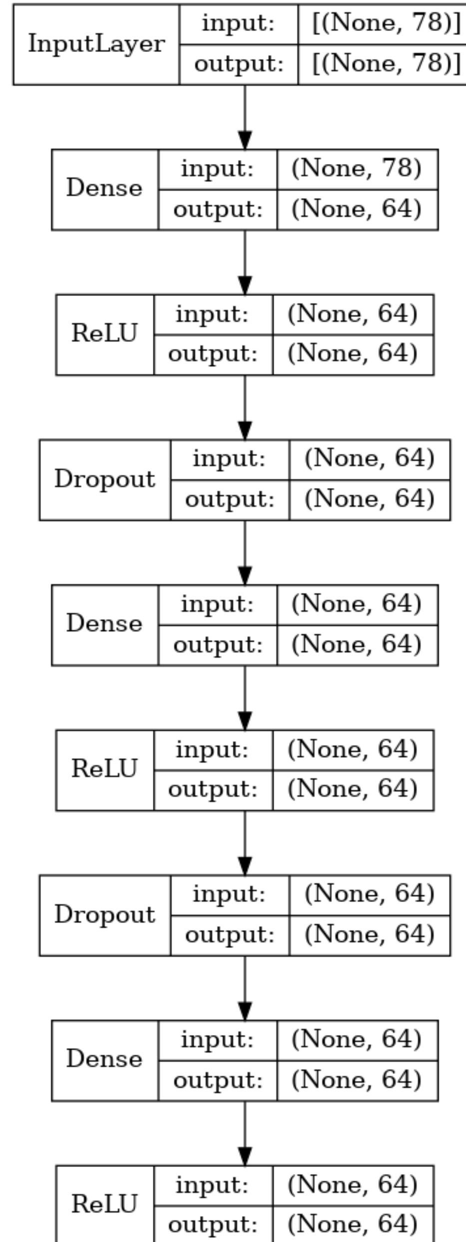

Figure 14: Architecture of the embedding creation models.

4.4 Zero-Day Detection Models

In this section, I will present the models that I have specifically designed for detecting zero-day attacks. I will refer to these types of models as zero-day detection models. During my thesis, I developed two types of zero-day models. In the following subsections, I will present each model type in the order in which I created them.

4.4.1 First Attempt: Embedding + Classification

The first model type I developed was a direct implementation of the concept I discussed in section 4.1, as depicted in Figure 9. Initially, I trained a model to create embeddings from netflows using one of the methods presented in section 4.3. Subsequently, I converted my training netflows into training netflow embeddings. Finally, I trained a classifier on these netflow embeddings using one of the distance-based classifiers presented in section 2.4. Figure 15 depicts the complete pipeline of my first type of zero-day detection model. This pipeline is identical for both the training and prediction phases.

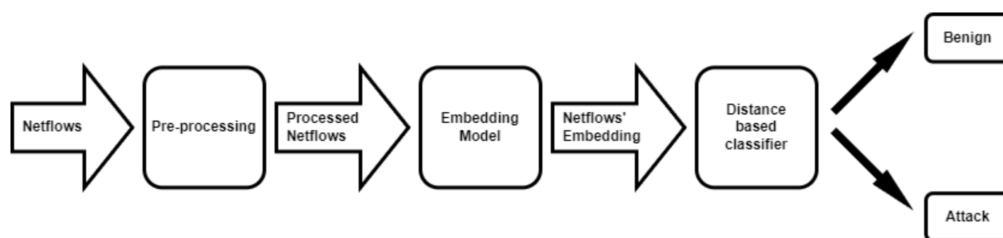

Figure 15: Training and prediction pipeline of my first type of NIDS designed to detect zero-day attacks.
This image was created using the online tool *DrawIO*.

This first model type demonstrated good performance with the MLP Embedding Model, but this was not the case for the GE2E Embedding Model. The primary objective of this thesis was to assess the effectiveness of the GE2E loss for creating netflow embeddings and, based on these embeddings, to develop a NIDS capable of detecting zero-day attacks. Consequently, I aimed to develop a zero-day detection model that efficiently works with embeddings produced by the GE2E loss. A considerable portion of my thesis was dedicated to enhancing this model to function effectively with the GE2E Embedding Model. However, all attempts to enhance its performance were unsuccessful. At this stage, my focus was primarily on improving netflow classification, without a specific emphasis on zero-day attack detection.

4.4.2 Second Attempt: Clustering + Embedding + Classification

While analyzing the label distribution in my dataset to understand the ineffectiveness of my initial zero-day detection model when combined with the GE2E Embedding Model, I discovered a significant imbalance: there were considerably more benign flows than attack flows. With this discovery and knowing how the GE2E loss functions in creating embeddings (cf. section 2.2.5), a new idea emerged. I decided to cluster benign flows, creating multiple labels that represent them instead of one.

The primary motivation behind my decision to cluster benign flows stems from the inherent nature of the GE2E loss, which is designed to group netflows with similar semantics. This characteristic is beneficial when applied to attack labels, as netflows within a single attack label generally share a common semantic, essentially describing the same type of attack. However, the scenario is quite different for the benign label. The benign label encompasses a diverse range of netflows, each representing a unique internet usage pattern, and consequently, possessing distinct semantics. This diversity within a single benign label poses a significant challenge for the GE2E loss, making it difficult for the loss to effectively learn and represent benign flows. By clustering benign flows into distinct sub-clusters, I aimed to overcome this issue. Ideally, each sub-cluster would represent a unique category of benign flow, effectively encapsulating a specific type of internet usage. Such an arrangement would simplify the GE2E loss's task, enabling it to understand and group each sub-category of benign flows more effectively and efficiently.

Clustering the benign flows significantly improved the performance of my model. A more detailed analysis comparing the performance of my first and second zero-day detection models will be presented in chapter 6. This approach not only enhanced the efficiency of my zero-day detection model in classifying netflows but also enabled it to detect unseen attacks. This marked the beginning of success for my approach, leading to the development of my second type and working zero-day detection model.

Both the prediction and training pipelines of my second type of zero-day detection model differ from those of my first type. For the training pipeline, the differences arise from the additional benign flows clustering phase and the post-processing phase. In the prediction pipeline, the difference is solely due to the post-processing phase. This phase involves regrouping the various sub-labels representing benign flows into a single one. Figure 16 depicts the training pipeline of my second type of zero-day detection model, while Figure 17 shows its prediction pipeline.

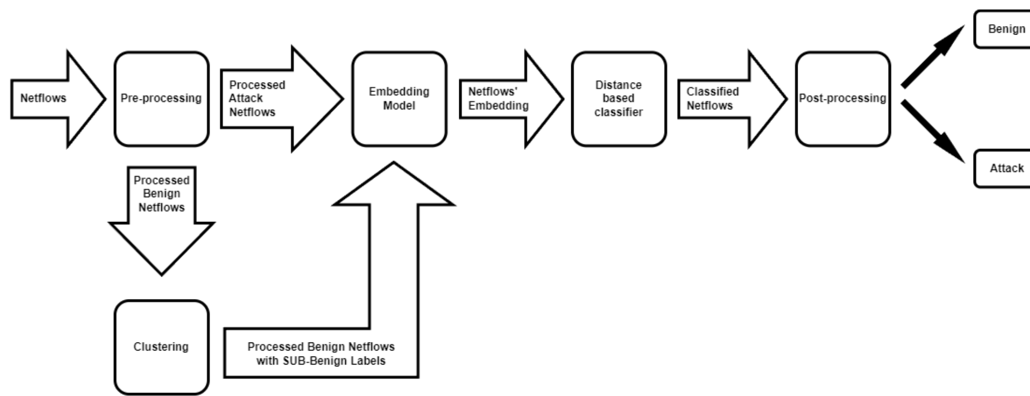

Figure 16: Training pipeline of my second type of NIDS designed to detect zero-day attacks.
This image was created using the online tool *DrawIO*.

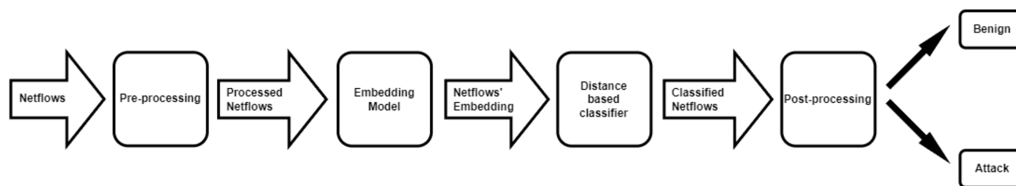

Figure 17: Prediction pipeline of my second type of NIDS designed to detect zero-day attacks.
This image was created using the online tool *DrawIO*.

Chapter 5

Experiments

In this chapter, I will go into detail about how I set up everything before testing and comparing my different models, which will be the subject of the next chapter. I will start by presenting the dataset that I have chosen for my thesis. Then, I will delve into how I have trained my different models. Finally, I will discuss the different test scenarios that I have performed before evaluating my models.

5.1 Dataset

Let's start with the dataset that I have used in this work. I will begin by presenting it, then move on to discuss the different attack scenarios used to generate the dataset. After that, I will provide the distribution of the labels inside the dataset. I will conclude by presenting the pre-processing that I have applied to the dataset before it was ready for use.

5.1.1 Presentation

For this work, I have selected the *CSE-CIC-IDS2018* dataset [7], a collaborative effort between the *Communications Security Establishment* (CSE) and the *Canadian Institute for Cybersecurity* (CIC). The CSE, as described on the Canadian Government's website [22], is Canada's national cryptologic agency, tasked with providing information technology security and foreign signals intelligence to the Government of Canada. The CIC, housed at the University of New Brunswick in Fredericton, is a multidisciplinary training, research, and development unit [23].

My choice of this dataset was motivated by its status as the most recent offering from the CIC for intrusion detection, not limited solely to *Denial of Service* attacks. This dataset encompasses captured network traffic along with the netflows

extracted from it. Each netflow is detailed with 80 features. These netflows were extracted from the captured traffic using the *CICFlowMeter-V3* tool developed by the CIC [24]. According to its web page, CICFlowMeter-V3, a tool written in Java, generates bidirectional netflows. It determines the forward (source to destination) and backward (destination to source) directions based on the first packet. TCP netflows are terminated upon *TCP teardown*, a process that concludes a TCP connection through a four-step handshake, involving the exchange of FIN and ACK packets. UDP netflows are terminated after a timeout period, specifically following 600 seconds of inactivity. The tool is capable of computing over 80 statistical network traffic features independently in both directions. A detailed description of each of these 80 features can be found in Table 5 in Appendix 8.1.

5.1.2 Attack Scenarios

The CSE-CIC-IDS2018 dataset encompasses seven distinct attack scenarios, each conducted by an attack infrastructure comprising 50 machines. Figure 18 illustrates the topology of the network under attack, along with the attacking infrastructure. This represents a typical *Local Area Network* topology deployed on the *Amazon* cloud computing platform, *AWS*. It is segmented into five sub-networks, featuring 420 machines and 30 servers, to mirror the machine diversity found in real-world networks.

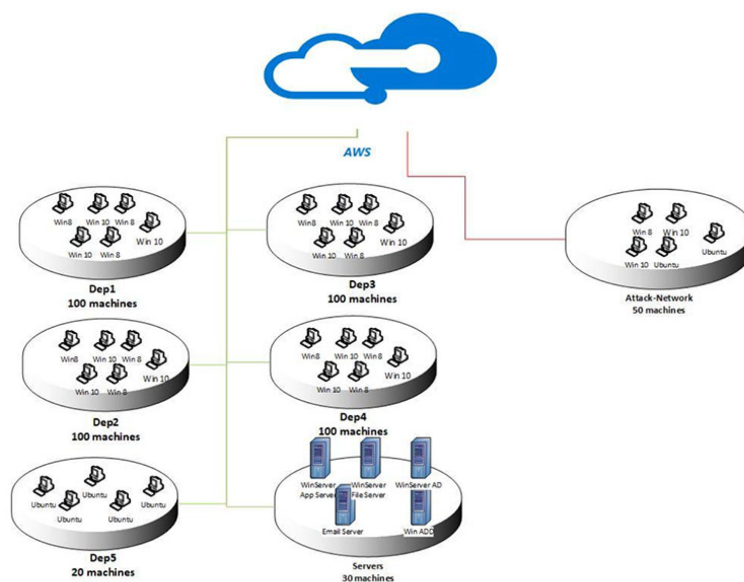

Figure 18: Topology of the attacked network and the attacking infrastructure. Image take from the dataset's presentation page, at this link: `https://www.unb.ca/cic/datasets/ids-2018.html`.

The following provides a brief overview of each attack scenario as detailed on the dataset's presentation page [25].

1. **Brute-force Attack**
   Brute-force attacks are a prevalent method targeting networks, aiming to exploit accounts protected by weak username and password combinations. This specific scenario is designed to simulate the breaching of SSH and FTP accounts. It employs a dictionary-based brute-force attack, utilizing a comprehensive list of 90 million words, against the main server. The attack is executed using *Patator* [26], a versatile, multi-threaded brute-force tool written in Python, known for its effectiveness in such scenarios.

2. **Heartbleed Attack**
   Discovered in 2014, *Heartbleed* is a significant security flaw in the OpenSSL cryptographic library. This vulnerability allowed attackers to read the memory of systems operating with susceptible versions of the library, notably version 1.0.1f. Consequently, it exposed sensitive data, including private keys, user credentials, and confidential information. The dataset's authors utilized *Heartleech* [27] to exploit this vulnerability in their simulation. *Heartleech* is renowned for its ability to identify systems vulnerable to Heartbleed and to extract data from them effectively.

3. **Denial-of-Service Attack**
   A Denial-of-Service (DoS) attack aims to disrupt the normal operations of a targeted server or network by inundating it with an excessive amount of internet traffic. This is typically achieved by overloading the system with an overwhelming number of requests, rendering it unresponsive. In this scenario, the authors employed four distinct tools: *Http Unbearable Load King* (HULK) [28], *GoldenEye* [29], *Slowhttptest* [30], and *Slowloris* [31]. Notably, *Slowloris* has the unique advantage of allowing a single machine to incapacitate a web server of another machine with minimal bandwidth usage and minimal impact on other services and ports. The primary objective of this scenario was to compromise the web server of the targeted network.

4. **Distributed Denial-of-Service Attack**
   Distributed Denial-of-Service (DDoS) attacks are an extension of DoS attacks, launched from multiple sources. In this particular scenario, the authors utilized two different tools: the *High Orbit Ion Cannon* (HOIC) [32] and the *Low Orbit Ion Cannon* (LOIC) [33]. The HOIC is capable of simultaneously targeting 256 URLs. To realistically simulate the distributed aspect of the attack, the authors operated these tools on four separate computers within the attacking network.

5. **Botnet Attack**
   A botnet attack involves a network of compromised computers, referred to as bots, which are controlled by an attacker known as the botmaster. These bots are often utilized to disseminate malware or to execute distributed attacks, such as DDoS attacks. In this scenario, the authors employed two tools: *Zeus* [34], a type of *Trojan horse malware*, and *Ares* [35], an open-source botnet software, to infect computers. Subsequently, the bots were programmed to send a screenshot to the botmaster every 400 seconds.

6. **Web Attacks**
   In this scenario, the authors of the dataset utilized the *Damn Vulnerable Web App* (DVWA) [36], a PHP/MySQL web application intentionally designed with numerous security vulnerabilities. The primary purpose of DVWA is to serve as a tool for security professionals to test their skills and tools in a legal and controlled environment. In this context, it was employed to generate traffic simulating three distinct types of web attacks on the deliberately vulnerable web application: *Cross-Site Scripting* (XSS) [37], *SQL Injection* [38], and unrestricted file upload.

7. **Infiltration of the Network from Inside**
   In this scenario, the authors initiated the attack by sending a malicious file through email to the intended victim. This file was designed to exploit vulnerabilities in specific applications, such as Adobe Acrobat Reader 9. Upon successful exploitation, they utilized the *Metasploit framework* [39] to establish a backdoor in the system.

5.1.3 Labels Distribution

The seven attack scenarios previously described enabled the CSE and CIC teams to generate 14 malicious labels for their dataset. Coupled with the benign label, this results in a total of 15 distinct labels present in the CSE-CIC-IDS2018 dataset. Furthermore, after applying my pre-processing steps, which will be discussed in the upcoming sub-section, I have obtained over 16 million netflows.

Table 1 provides a summary of the distribution of these 15 labels across the 16 million netflows. As mentioned in subsection 4.4.2, there is a significant imbalance in my dataset, which has adversely affected the efficiency of my initial zero-day detection model when combined with the GE2E Embedding Model. This table clearly demonstrates this disparity. Notably, the CSE-CIC-IDS2018 dataset consists of over 82% benign netflows, leaving less than 18% as attack flows. It is this imbalance that necessitated the introduction of balanced accuracy, as detailed in

subsection 2.5.2. Balanced accuracy is essential for evaluating models trained on datasets with such pronounced imbalances.

Label	Number of netflows	Percentage of the dataset
Benign	13,390,249	82.978 %
DDOS attack-HOIC	686,012	4.251 %
DDoS attacks-LOIC-HTTP	576,191	3.571 %
DoS attacks-Hulk	461,912	2.862 %
Bot	286,191	1.773 %
FTP-BruteForce	193,354	1.198 %
SSH-Bruteforce	187,589	1.162 %
Infilteration	160,639	0.995 %
DoS attacks-SlowHTTPTest	139,890	0.867 %
DoS attacks-GoldenEye	41,508	0.257 %
DoS attacks-Slowloris	10,990	0.068 %
DDOS attack-LOIC-UDP	1,730	0.011 %
Brute Force -Web	611	0.004 %
Brute Force -XSS	230	0.001 %
SQL Injection	87	0.001 %
SUM	16,137,183	100.000 %

Table 1: Distribution of the 15 labels across the entire CSE-CIC-IDS2018 dataset, illustrating the proportion of each label within the dataset.

5.1.4 Pre-processing

The pre-processing applied to the CSE-CIC-IDS2018 dataset involved a series of five distinct steps: feature removal, netflow removal, train-validation-test splitting, data normalization, and label encoding. In the following sections, I will provide a detailed explanation of each of these steps.

Removed Feature

As discussed in the previous section, each netflow in the CSE-CIC-IDS2018 dataset comprises 80 features. One of these features is reserved for the label associated with the netflow. Since I handle the label separately from the other features, this effectively leaves us with 79 features per netflow. In Figure 12 and Figure 14, we saw that the models take inputs consisting of 78 features. This discrepancy is due to the removal of one specific feature from the netflows: the "*Timestamp*". The primary reason for excluding the Timestamp feature is its unique discriminating ability; each netflow in the dataset has a distinct value for this feature. Consequently, a

classifier could potentially achieve 100% accuracy on training netflows merely by learning to identify netflows using this feature alone, leading to overfitting.

Removed Netflows

The CSE-CIC-IDS2018 dataset includes some netflows that are considered corrupted due to anomalies in certain features. Specifically, there are netflows with a zero duration, indicated by the feature "*Duration*" having a value of zero. This situation causes the "*Flow Pkts/s*" feature, which measures the rate of packet transfer (number of packets per second), to register an infinite value. Since it's impossible to train a machine learning model with samples containing infinite values for any feature, I chose to remove all netflows that had an infinite value for the "*Flow Pkts/s*" feature. As a result of this decision, 95,760 netflows were removed from the dataset in this pre-processing step.

Train - Validation - Test Splitting

In machine learning, it is customary to partition the entire dataset into three subsets:

1. **Training set:** As the name suggests, this set is used for training the model.
2. **Validation set:** This set is utilized to evaluate the model's performance during training. Specifically, it helps monitor the model's ability to generalize to unseen data. The training process can be halted when there is no further progress or regression in loss on the validation set. This approach is instrumental in preventing overfitting on the training set.
3. **Test set:** Used exclusively for evaluating the model's performance after the training is complete, this set comprises data never exposed to the model during training. It provides an unbiased evaluation of the final model's effectiveness, particularly its ability to generalize to new, unseen data.

For these splits, I first allocated 20% of the netflows from the entire dataset to the test set and the remaining 80% to the training set, randomly. Subsequently, 20% of the netflows from the training set were randomly reassigned to the validation set, and the remaining 80% forming the final training set.

An important consideration in this process was maintaining the same label distribution in each split as in the entire dataset, ensuring that none of the splits are biased towards a specific label. To validate this, Table 2 displays the label distribution within the training, validation, and test sets. We observe that each

split retains the same percentage of each label, although the actual number of netflows per label varies between the splits.

Label	Training set		Validation set		Test set	
Benign	8,569,759	82.978 %	2,142,440	82.978 %	2,678,050	82.978 %
DDOS attack-HOIC	439,047	4.251 %	109,762	4.251 %	137,203	4.251 %
DDoS attacks-LOIC-HTTP	368,762	3.571 %	92,191	3.571 %	115,238	3.571 %
DoS attacks-Hulk	295,624	2.862 %	73,906	2.862 %	92,382	2.862 %
Bot	183,162	1.773 %	45,791	1.774 %	57,238	1.773 %
FTP-BruteForce	123,746	1.198 %	30,937	1.198 %	38,671	1.198 %
SSH-Bruteforce	120,057	1.162 %	30,014	1.162 %	37,518	1.162 %
Infilteration	102,809	0.995 %	25,702	0.995 %	32,128	0.995 %
DoS attacks-SlowHTTPTest	89,530	0.867 %	22,382	0.867 %	27,978	0.867 %
DoS attacks-GoldenEye	26,565	0.257 %	6,641	0.257 %	8,302	0.257 %
DoS attacks-Slowloris	7,034	0.068 %	1,758	0.068 %	2,198	0.068 %
DDOS attack-LOIC-UDP	1,107	0.011 %	277	0.011 %	346	0.011 %
Brute Force -Web	391	0.004 %	98	0.004 %	122	0.004 %
Brute Force -XSS	147	0.001 %	37	0.001 %	46	0.001 %
SQL Injection	56	0.001 %	14	0.001 %	17	0.001 %
SUM	10,327,796	100.000 %	2,581,950	100.000 %	3,227,437	100.000 %

Table 2: Comparative distribution of the 15 labels within the training, validation, and test sets, illustrating the proportion of each label in these subsets of the CSE-CIC-IDS2018 dataset.

Data Normalization

Data normalization is a standard practice in machine learning, serving several crucial functions. I believe the following two reasons are particularly significant:

1. Normalization ensures that each feature contributes equally to the learning process. Without normalization, features with relatively larger scales could dominate and bias the model, overshadowing those with smaller scales.

2. Normalization generally leads to more effective models by enhancing convergence, ensuring numerical stability, and is especially beneficial for algorithms sensitive to input scales, such as deep learning and distance-based models.

In this thesis, I have employed a normalization technique known as *standardization*. This process involves transforming the data so that each feature has a mean of zero and a standard deviation of one. Essentially, it scales the values of each feature to be centered around zero with unit standard deviation. Practically, this is achieved by calculating the mean and standard deviation for each feature, then subtracting the mean from each value and dividing by its standard deviation.
Denoting the total number of netflows in the dataset as N, the value of the fth

feature of the ith netflow as $d_{i,f}$, the standardization technique can be formally defined as follows:

$$\mu_f = \frac{1}{N} \sum_{i=1}^{N} d_{i,f} \tag{5.1}$$

$$\sigma_f = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (d_{i,f} - \mu_f)^2} \tag{5.2}$$

$$d_{i,f} \leftarrow \frac{d_{i,f} - \mu_f}{\sigma_f} \tag{5.3}$$

Label Encoding

The final step in data preparation involves converting the 15 labels in the dataset into numerical values. As indicated in Table 1 and Table 2, the labels in the CSE-CIC-IDS2018 dataset are initially character strings. However, machine learning models require numerical inputs, necessitating a process known as *encoding*, which converts these string labels into integers. The encoding method I employed consists of two steps:

1. First, I established an order for the labels by sorting them lexicographically.
2. Then, each label was mapped to its corresponding position in the sorted order.

To illustrate, consider a small subset of the labels: {"FTP-BruteForce", "SQL Injection", "Benign"}. After sorting, we have: {"Benign", "FTP-BruteForce", "SQL Injection"}. Thus, in this encoding scheme, the label "Benign" is assigned the value 1, "FTP-BruteForce" is assigned 2, and "SQL Injection" is assigned 3.

5.2 Models Training

In chapter 4, I presented the zero-day detection model developed throughout this thesis. Its training pipeline, as illustrated in Figure 16, comprises three trainable models: a clustering model, an embedding creation model, and a distance-based classification model. Additionally, I have discussed a simple deep learning-based netflow classifier (the MLP classifier). This section will detail the training process for all these models.

Before exploring the training procedures in depth, it's important to note a key aspect regarding all the classifiers used in this thesis. They were trained to predict the specific label associated with a netflow or its embedding, rather than merely distinguishing between "Benign" and "Attack". This approach was chosen to evaluate the models' capabilities in detecting each type of label individually. In other words, this allowed me to measure the accuracy for label attack separately. To ultimately achieve a binary NIDS output, which classifies a netflow as either "Benign" or "Attack", the predictions of all classifiers were binarized. This means, denoting the output of a classifier as $\hat{y}_{clf}$ and the NIDS output as $\hat{y}_{nids}$, the following logic applies:

$$\hat{y}_{nids} = \begin{cases} "Benign" & \text{if } \hat{y}_{clf} \text{ corresponds to a benign label,} \\ "Attack" & \text{if } \hat{y}_{clf} \text{ corresponds to an attack label.} \end{cases} \tag{5.4}$$

5.2.1 Deep Learning-based Classifier

Let's now proceed with the training of the simple deep learning netflow classifier, the architecture of which is illustrated in Figure 12. In subsection 2.2.6, I introduced the four key hyper-parameters that govern the training of a deep learning model. To recap, these hyper-parameters are the loss, the optimizer, the threshold, and the patience. Here are the specific values I chose for these hyper-parameters to train the MLP classifier:

1. **Loss:** Categorical cross-entropy loss (cf. subsection 2.2.4).
2. **Optimizer:** The *TensorFlow* [9] implementation of the *Adam* algorithm [16], with all hyper-parameters set to their default values.
3. **Threshold:** 0.001
4. **Patience:** 3 epochs.

The chosen values for the threshold and patience imply that the training will be halted if there is no decrease of at least 0.001 in the best categorical cross-entropy loss value over a span of 3 epochs.

5.2.2 Clustering Model

In subsection 4.4.2, I discussed the significant improvement in my zero-day detection model achieved through clustering benign flows. This section will detail the practical implementation of this clustering. For this purpose, I employed the *Mini-Batch K-Means* (MBK-Means) algorithm, which is described in section 2.3. MBK-Means relies on a single hyper-parameter, K, representing the number of clusters to form.

In the context of this work, K corresponds to the number of sub-benign labels into which the benign label should be divided. I chose to set $K = 40$, and the justification of this decision will be discussed in subsection 5.3.2.

5.2.3 Embedding Creation Models

In this research, two embedding creation models were developed and tested. As outlined in subsection 4.3.1, the first model is a variation of the simple MLP classifier. It involves initially training the MLP classifier (cf. to subsection 5.2.1) and then removing its classification and last dropout layers. Since its core architecture is already trained, it requires no further training. Therefore, this section will focus exclusively on the training of the second embedding creation model.

Both embedding creation models, as mentioned in section 4.3, are based on the same architecture (shown in Figure 14), enabling a fair evaluation of the GE2E loss's impact compared to simpler embedding creation methods. To emphasize this, the training of the second model mirrored the MLP classifier in three of the four hyper-parameters: optimizer, threshold, and patience, as detailed in subsection 5.2.1. For the fourth hyper-parameter, loss, the GE2E loss was used.

As explained in section 2.2.5, the GE2E loss operates on batches containing M samples from N different labels. To accommodate this, both the training and validation sets were converted to meet these requirements through a data generator. The data generator, taking a dataset subset S, is governed by two hyper-parameters: M and N. Its function is to reorganize the netflows in S into several batches, each comprising M netflows from N different labels. This process led to two key design decisions: the selection of N labels for each batch, and the choice of M netflows for each label. The following subsection elaborates on these design choices.

Data Generator: Label Selection

The process to select N labels for forming a batch includes these steps:

1. Create a list L of all labels in S, with l denoting the length of L.

2. Remove from L all labels that do not encompass enough netflows, i.e., a label is removed when the number of available netflows (i.e., the netflows that are not already included in a batch) is fewer than M.

3. If $l < N$, it indicates insufficient labels in L to create more batches. The process stop here.

4. Otherwise, divide L into subsets each containing N labels. If l is not a multiple of N, the last batch is formed from the first k labels, where $k = (l \ // \ N) * N$ and $//$ denotes integer division.

5. Shuffle L to ensure different combinations of N labels in the batches, then return to step 2.

Figure 19 illustrates two iterations of this process using the first seven labels from Table 1 and setting $N = 3$.

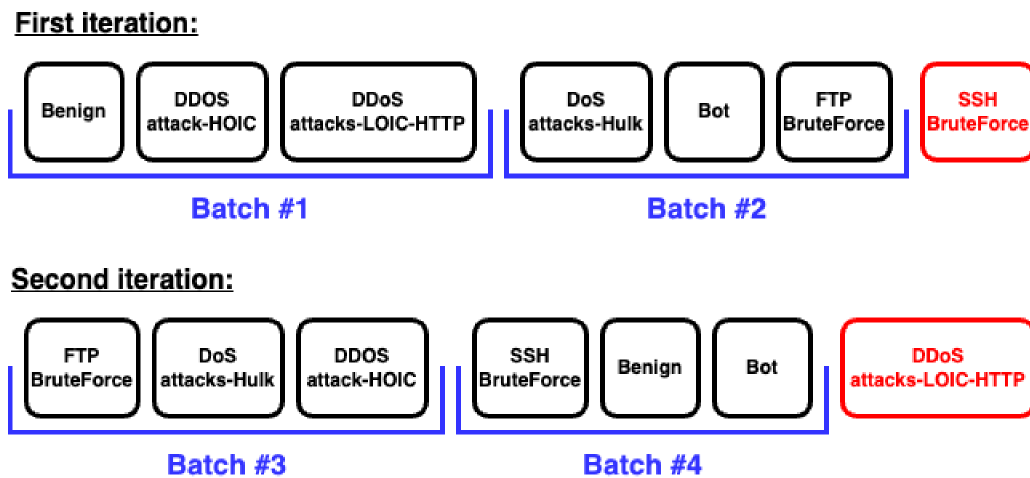

Figure 19: Two iterations of the label selection process. Red boxes indicate labels not considered in each iteration.
This image was created using the online tool *DrawIO*.

Data Generator: Netflows Selection

A key requirement for my data generator was to include the majority of netflows present in S while avoiding any repetition. To achieve this, I grouped the netflows by their labels. Then, for each label, I divided its netflows into several subsets, with each subset containing M netflows. If there were not enough netflows to complete the last subset, I considered only the largest multiple of M netflows. When a label is chosen for a batch, the first available subset of M netflows from that label is included in the batch, and this subset is then removed from the pool of available subsets for that label. This approach ensures that each netflow is selected only once.

Data Generator: Values of N and M

For training a model with the GE2E loss, I chose to set $N = 4$ and $M = 256$. This configuration implies that each batch produced by the data generator will contain four distinct labels, with 256 netflows from each label. Consequently, this results in a total batch size of 1024 netflows.

5.2.4 Distance-based Models

In this thesis, I have utilized two distance-based classifiers: the *K-Nearest Neighbors* (K-NN) and the *Nearest Centroid* (NC). As outlined in chapter 4, I have employed these models in two capacities: firstly, as straightforward netflow classifiers, and secondly, as distance-based classifiers for netflow embeddings. Consequently, I have configured these classifiers to manage both types of input, namely netflows and netflow embeddings. For clarity and generality, I will henceforth use the term "inputs" to refer to either netflows or netflow embeddings throughout this discussion.

As described in section 2.4, the K-NN classifier necessitates two design decisions: the selection of training points and the determination of the value for K. The NC classifier, by design, requires no such choices. Therefore, in the ensuing discussion, I will focus solely on the two design decisions made for the K-NN classifier.

1st Design Choice: Training Points

As indicated in Table 2, the training set contains over 10 million netflows. Handling this volume of training points is beyond the K-NN classifier's capacity. Thus, it was imperative to sample a subset of inputs to form the K-NN classifier's training set. To achieve this, I employed a more strategic sampling technique than mere random sampling. This technique involves a hyper-parameter n, representing the number of inputs per label included in the K-NN's training set, and comprises four steps:

1. Grouping the inputs by their labels.
2. Computing the centroid of the inputs for each label.
3. Calculating the distance of each input to its respective label centroid.
4. Selecting the n closest inputs to the label centroids for inclusion in the K-NN's training set for each label.

The rationale behind this sampling method was to form compact clusters, each representing a distinct label. By "compact cluster", I refer to a cluster whose

samples are closest to the cluster's centroid. Therefore, for each label, I select the n inputs nearest to the label's centroid. I have chosen to set $n = 175$. The reasoning behind this decision is elaborated in subsection 5.3.3.

2nd Design Choice: K

As a reminder, the hyper-parameter K in the K-NN classifier denotes the number of nearest points considered during the majority vote process. I have elected to set $K = 20$. The justification for this choice is provided in subsection 5.3.4.

5.3 Test Scenarios

In this section, I will outline my methodology for simulating zero-day attacks, a critical component in assessing the efficacy of my models for their detection, which is the primary objective of my thesis. Subsequently, I will detail the rationale behind determining the values of three hyper-parameters of my zero-day detection models. These parameters are: firstly, the number of clusters into which the benign label should be divided; secondly, the number of inputs per label selected to construct the training set of the K-NN classifier; and finally, the value of K in the K-NN classifier. All the graphs presented in this section were generated using the plotting tool developed by *Yui-MHCP* [13], which is available in its *GitHub repository*.

5.3.1 Zero-Day Attack Simulation

Simulating zero-day attacks is a crucial element of my research. Given the primary aim of my thesis is to design a model that effectively detects such attacks, it is imperative to meticulously evaluate my models' detection capabilities and the accuracy of the simulation process.

My approach to simulating zero-day attacks is straightforward and involves three steps. To simulate an attack a as a zero-day attack, i.e., an attack that is entirely novel to my models, I execute the following procedures:

1. I combine the training and validation sets into a single dataset, hereafter referred to as the merged set.

2. From this merged set, I transfer all netflows labeled with attack a to the test set.

3. I then re-divide the merged set, allocating randomly 20% of the netflows into the new validation set, and the remaining 80% into the new training set.

This ensures the distribution of the other attacks remains consistent in both the training and validation sets.

I believe this method effectively simulates zero-day attacks. By moving all netflows of a designated zero-day attack into the test set, my models are evaluated on their ability to classify *all* netflows of an attack they have never encountered before.

To assess the models' proficiency in detecting zero-day attacks, I incrementally remove nearly all attacks from the dataset. The order of removal is based on the decreasing number of netflows each attack contributes to the training set. This corresponds to the same order in which the attacks are presented in Table 1 and Table 2.

As discussed in section 2.2.5, to train a model using the GE2E loss, it's necessary to organize netflows into batches containing M netflows from N different labels, with each batch including at least two distinct labels (i.e., $N > 1$). Consequently, attacks with fewer than M netflows are by design treated as zero-day attacks by all models trained with the GE2E loss, as they are not considered by my data generator. Additionally, at least one attack must remain in the training set to maintain the requirement of having two different labels, besides the benign label. In section 5.2.3, I noted setting the hyper-parameter N of my data generator to four. If, after removing certain attacks, fewer than four labels remain, I reduce the value of N to two.

Table 3 illustrates the changing sizes of the training, validation, and test sets upon the sequential removal of each attack. Each row in the table represents the cumulative effect of removing attacks, with, for example, the third row indicating the removal of the first two attacks, "DDOS attack-HOIC" and "DDoS attacks-LOIC-HTTP".

	Removed attack	**Training set size**	**Validation set size**	**Test set size**	**Sum**
0	No attack removed	10,327,796	2,581,950	3,227,437	16,137,183
1	DDOS attack-HOIC	9,888,749	2,472,188	3,776,246	16,137,183
2	DDoS attacks-LOIC-HTTP	9,519,987	2,379,997	4,237,199	16,137,183
3	DoS attacks-Hulk	9,224,363	2,306,091	4,606,729	16,137,183
4	Bot	9,041,200	2,260,301	4,835,682	16,137,183
5	FTP-BruteForce	8,917,454	2,229,364	4,990,365	16,137,183
6	SSH-Bruteforce	8,797,397	2,199,350	5,140,436	16,137,183
7	Infilteration	8,694,588	2,173,648	5,268,947	16,137,183
8	DoS attacks-SlowHTTPTest	8,605,059	2,151,265	5,380,859	16,137,183
9	DoS attacks-GoldenEye	8,578,494	2,144,624	5,414,065	16,137,183
10	DoS attacks-Slowloris	8,571,460	2,142,866	5,422,857	16,137,183
11	DDOS attack-LOIC-UDP	8,570,353	2,142,589	5,424,241	16,137,183

Table 3: Evolution of the number of netflow in the training set, the validation set and the test set, when removing attacks.

5.3.2 Clustering: Number of Benign Clusters

To select the most appropriate value for the K hyper-parameter of the MBK-Means algorithm, that is, the number of clusters into which the benign label must be divided, I decided to conduct an experiment. This involved repeatedly applying the training pipeline of my zero-day detection model (as depicted in Figure 16), each time with a different value for the MBK-Means' K hyper-parameter. I then measured the balanced accuracy of the resulting model in classifying all netflows in the test set. This experiment was executed with two embedding creation models (*MLP Embedding Model* and *GE2E Embedding Model*) and two distance-based classifiers (K-NN and NC). The MBK-Means algorithm is very sensitive to the random initialization of the centroids. Therefore, to account for this randomness, I decided to repeat this experiment four times, each with a different seed for the MBK-Means algorithm, and then take the average of the balanced accuracy from the four iterations.

Based on the results of this experiment shown in Figure 20, I decided to set $K = 40$ for its hyper-parameter when applying the MBK-Means algorithm to cluster the benign netflows. I chose this value because it provided good results for all four models tested.

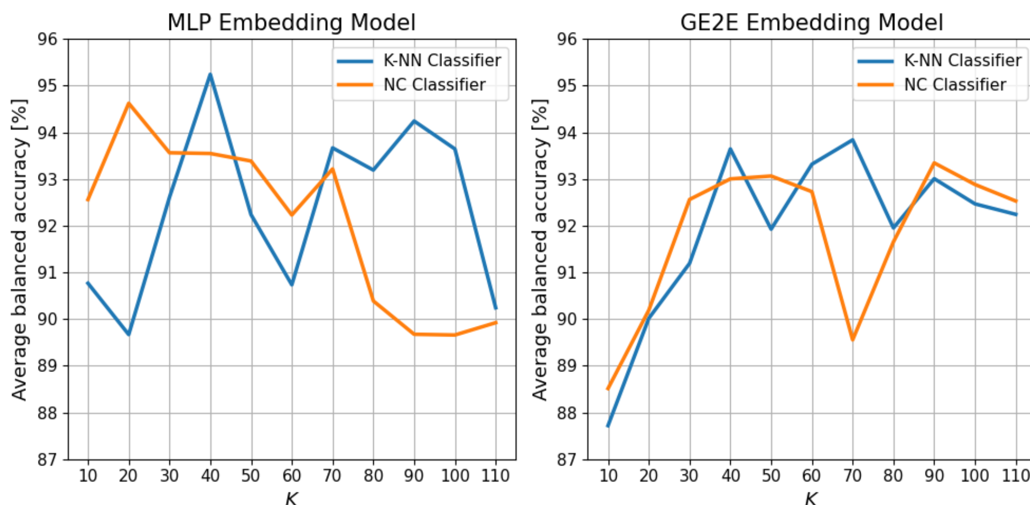

Figure 20: Results of the experiment conducted to determine the optimal value for the K hyper-parameter of the MBK-Means algorithm.

5.3.3 K-NN: Number of Inputs per Label

To select the most appropriate value for the n hyper-parameter of the K-NN classifier, that is, the number of inputs per label to form its training set, I conducted an experiment similar to the previous one. This experiment, also involved the repeated application of the training pipeline for my zero-day detection model. However, unlike the previous experiment where the MBK-Means' K hyper-parameter was varied, this time it was fixed at 40, and the focus was on varying the K-NN's n hyper-parameter while keeping its K hyper-parameter steady at 15. This value of 15 for K was chosen based on earlier manual analyses that indicated it produced good results. This experiment was conducted using the two embedding creation models but was limited to the K-NN classifier, as the objective was to determine the optimal value for one of its hyper-parameters, rather than for the NC classifier. Since the K-NN classifier is deterministic and does not involve randomness, I did not perform multiple rounds with different seeds. Instead, the experiment was repeated four times, each time removing a different proportion of attacks: 0, 1/3, 2/3, and all, which corresponded to removing 0, 3, 7, and 11 attacks, respectively (cf. Table 3).

Based on the results shown in Figure 21, I have decided to set $n = 175$ when training a K-NN classifier for the remainder of this thesis. This value was chosen as it provided good performance for both embedding creation models and across all proportions of attacks removed.

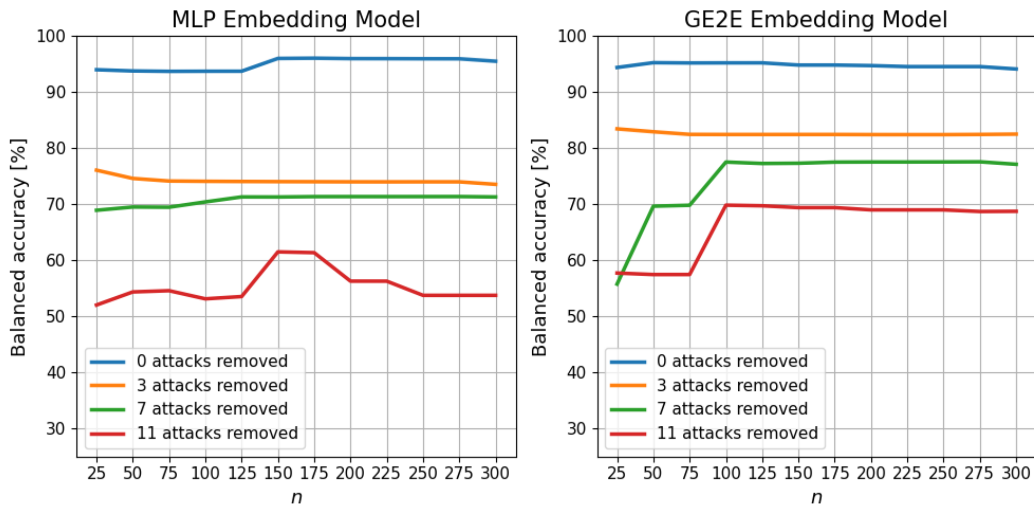

Figure 21: Results of the experiment conducted to determine the optimal value for the n hyper-parameter of the K-NN algorithm.

5.3.4 K-NN: Value of K

To determine the most suitable value for the K hyper-parameter of the K-NN classifier, namely, the number of nearest points considered for the majority vote, I conducted the same experiment as the previous one. However, in this iteration, I tested different values for the K hyper-parameter, while fixing its n hyper-parameter at 175.

Based on the results shown in Figure 22, I have decided to set $K = 20$ for predictions with the K-NN classifier for the remainder of this thesis. This value was chosen as it yielded good performance for both embedding creation models and across all proportions of attacks removed.

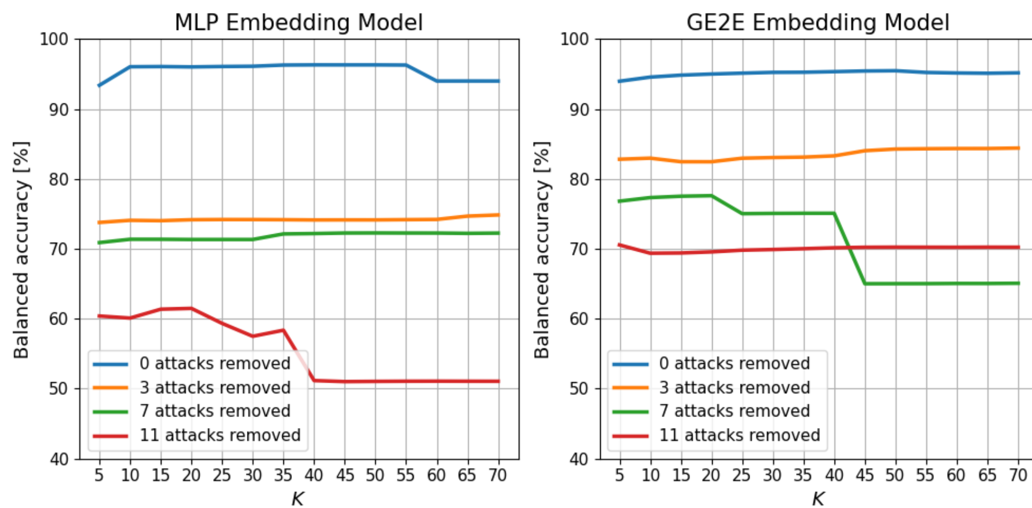

Figure 22: Results of the experiment conducted to determine the optimal value for the K hyper-parameter of the K-NN algorithm.

Chapter 6

Results

# removed attacks:	0	1	2	3	4	5	6	7	8	9	10	11
Baseline Models: simple netflows classifiers												
MLP	96.95	67.22	58.50	66.19	63.13	65.58	63.88	63.17	58.54	50.03	50.02	50.00
K-NN	53.93	54.34	54.37	54.37	54.39	54.42	48.55	61.20	55.92	57.27	57.17	58.99
NC	52.01	52.37	51.73	52.48	54.27	53.31	53.72	52.51	52.77	48.53	45.35	46.45
First type Zero-Day Models: without clustering of benign netflows												
MLP + K-NN	72.66	80.00	68.06	84.13	64.71	85.89	80.54	75.64	64.26	63.34	72.15	66.49
MLP + NC	73.96	83.09	69.34	79.08	65.14	85.60	81.27	72.40	62.59	62.65	56.15	66.96
GE2E + K-NN	66.09	62.59	52.73	59.92	71.24	61.69	61.09	71.53	60.05	65.75	59.30	68.03
GE2E + NC	66.60	63.30	63.39	76.22	60.27	63.83	60.30	78.17	77.55	77.26	58.39	66.03
Second type Zero-Day Models: with clustering of benign netflows												
MLP + K-NN	96.01	73.23	63.78	74.16	72.25	68.02	81.69	71.34	63.63	61.84	62.97	61.49
MLP + NC	93.09	70.29	62.63	69.80	66.77	65.30	77.91	62.82	60.30	49.91	52.30	61.81
GE2E + K-NN	95.00	95.67	83.54	82.45	81.24	83.85	82.74	77.60	75.22	77.47	64.95	69.56
GE2E + NC	93.39	67.09	57.14	54.02	52.31	55.47	54.21	55.53	59.74	49.70	49.57	49.51

Table 4: Balanced accuracy of all models developed throughout this work, observed when progressively removing attacks (cf. Table 3). The color red highlights the least performing model, whereas green denotes the best-performing model. The model marked in green represents the final NIDS proposed in this thesis (cf. section 6.4).

In this chapter, I will analyze the performance of all models developed throughout this work. Table 4 summarizes their balanced accuracies. The first column indicates the evaluated model. **MLP + K-NN** means that the **MLP** Embedding Model is used to create embeddings of netflows, on which the **K-NN** model performs classification. In the following sections, I will delve into the analysis of all models. Starting with simple netflow classifiers, I will then move on to my first type of zero-day detection models, followed by my second type of zero-day detection models. Additionally, I will compare these three different kinds of models with each other. Subsequently, I will present the final NIDS proposed by this thesis and conclude with a final test to evaluate the accuracy per label of the proposed NIDS.
All the graphs presented in this chapter were generated using the plotting tool

developed by *Yui-MHCP* [13], available in its *GitHub repository*. Appendix 8.2 details the environment in which all models presented in this work were trained and evaluated.

6.1 Baseline Models

Let's begin with the baseline models. In this section, I will compare the three simple netflow classifiers presented in section 4.2. The objective is to identify the most effective baseline model, which will then be used exclusively for comparisons with the zero-day detection models throughout the remainder of the chapter. This approach is adopted to ensure more readable graphs when comparing different types of models.

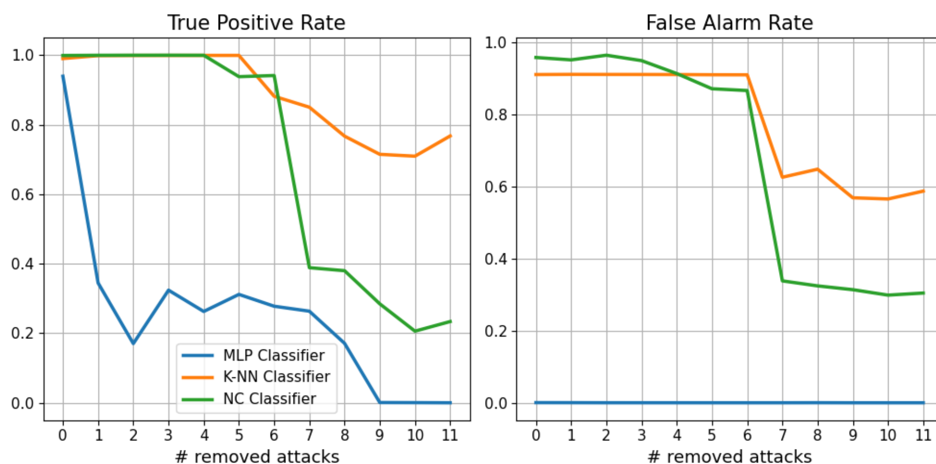

Figure 23: TPR and FAR produced by the baseline models.

From Figure 23, we observe that the MLP classifier has the lowest FAR. This makes sense, as this classifier is significantly biased by the majority label present in its training set, which is the benign label. Consequently, it has learned to represent benign netflows quite effectively, almost never confusing a benign flow with an attack. However, its TPR drops dramatically when the first attack is removed, making it a poor candidate for a zero-day detection model.

The K-NN and NC classifiers demonstrate greater robustness in detecting zero-day attacks, as their TPRs remain nearly constant even after removing the first six attacks. However, at the same time, they generate a significant number of false positives. Upon removing the seventh attack, specifically the "Infiltration" attack as detailed in Table 3, the FAR drops for both classifiers. I believe this is

due to the attack's semantic similarity to benign netflow. As described in subsection 5.1.2, this attack involves infiltrating the network by sending a malicious file to an internal victim. A netflow cannot discern the malicious nature of this attack. This hypothesis is supported by additional data analysis: certain netflows labeled "Infiltration" share identical values with some labeled "Benign" (there is a zero distance between some "Infiltration" netflows and benign ones). Consequently, all benign netflows near the centroid of this label are mistakenly identified as attacks, leading to numerous false positives. When this label is removed, these benign netflows are no longer classified as attacks, which reduces the FAR. However, the model then misclassifies more "Infiltration" netflows in the test set as benign, which also causes a drop in TPR.

The K-NN and NC classifiers are roughly equivalent in terms of efficiency. Specifically, the K-NN demonstrates a higher TPR than the NC, but the NC has a lower FAR compared to the K-NN. As detailed in subsection 2.5.2, the balanced accuracy is the arithmetic mean of the TPR and the complement of the FAR. Table 4 indicates that the K-NN classifier has a marginally better balanced accuracy than the NC. Consequently, I have chosen to use the K-NN classifier as the sole baseline model for comparisons with other models throughout the remainder of this chapter.

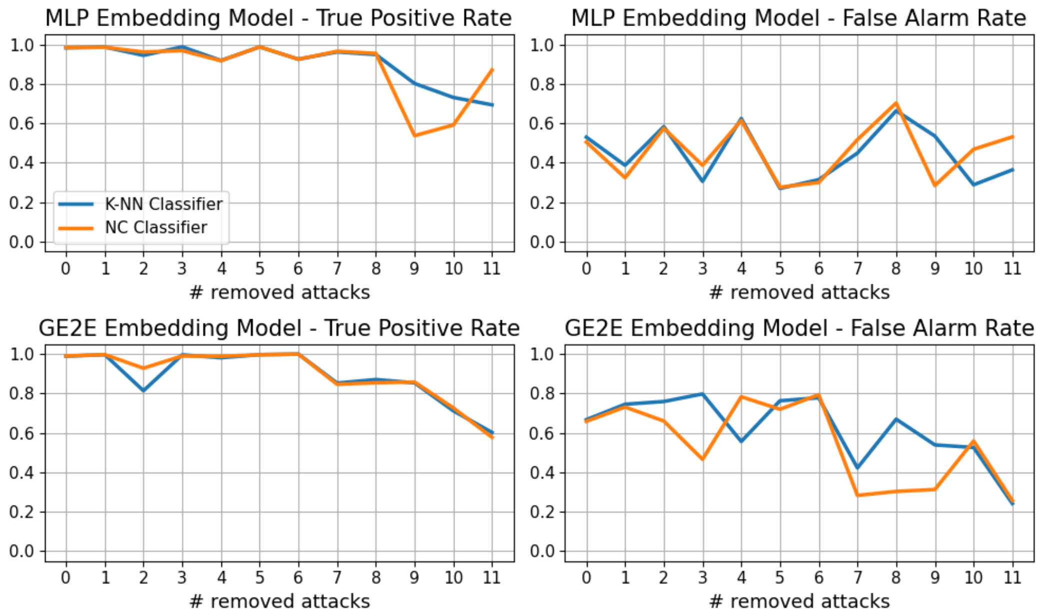

Figure 24: TPR and FAR produced by my first type of zero-day models (without clustering of benign netflows).

6.2 First Type of Zero-Day Detection Models

In this section, I will evaluate the first type of zero-day detection models that I have developed, as presented in subsection 4.4.1. Additionally, I will compare these models with the simple K-NN netflow classifier.

From Figure 24, we observe that both distance-based classifiers, when applied in my first type of zero-day detection models (cf. subsection 4.4.1), exhibit similar trends across all cases. Consequently, for these models, I will focus solely on the K-NN distance-based classifier for the remainder of this chapter. Both embedding creation models also display almost identical trends. Their TPRs remain stable around 1 until the removal of the eighth attack for the MLP Embedding Model and the sixth one for the GE2E Embedding Model, after which their TPRs start to decline, eventually stabilizing around 0.6 when all attacks have been removed. For the MLP Embedding Model, the FAR does not exhibit a specific pattern upon the removal of attacks. In contrast, the FAR for the GE2E Embedding Model generally decreases, albeit with some fluctuations, starting at around 0.7 and ending around 0.25.

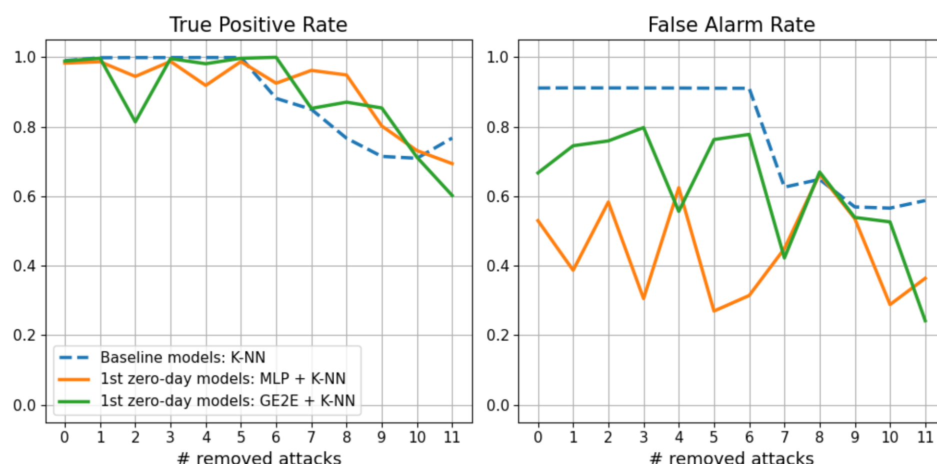

Figure 25: Comparison between my first type of zero-day detection models and the baseline K-NN netflows classifier.

From Figure 25, it can be concluded that my initial attempt at detecting zero-day attacks yields a TPR very close to that of the simple K-NN netflow classifier. However, my models result in a lower FAR compared to the simple classifier. Therefore, even this initial attempt offers benefits in detecting zero-day attacks. This advantage is also reflected in Table 4, where my first zero-day detection models consistently demonstrate higher balanced accuracy than the simple K-NN classifier.

6.3 Second Type of Zero-Day Detection Models

In this section, I will evaluate my second type of zero-day detection models, as presented in subsection 4.4.2. As a reminder, the key difference between my second and first type of zero-day models is that the second type involves clustering the benign netflows into sub-benign labels. Additionally, I will compare these models with those from the first type, as well as with the simple K-NN netflows classifier.

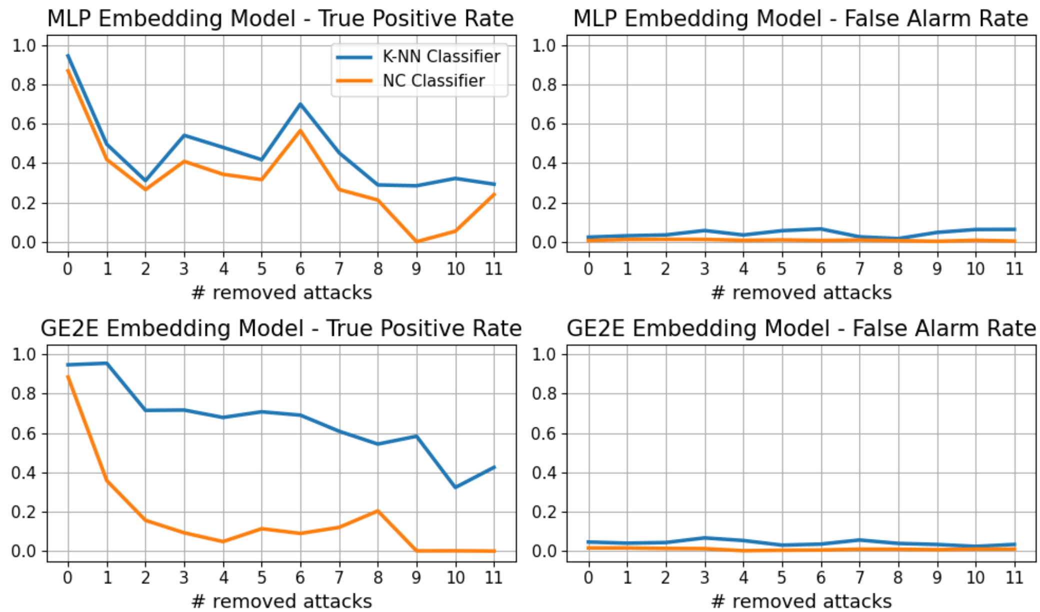

Figure 26: TPR and FAR produced by my second type of zero-day models (with clustering of benign netflows).

From Figure 26, it is observed that the TPR of the NC classifier is consistently lower than that of the K-NN classifier. When paired with the GE2E Embedding Model, after the removal of the first attack, the NC classifier consistently produces a TPR that is at least 40% lower compared to that produced by the K-NN. With the MLP Embedding Model, this difference is smaller. In terms of FAR, both distance-based classifiers, as well as both embedding creation models, are equivalent with a FAR of maximum 10%. Therefore, I will also discontinue considering the NC classifier for my second type of zero-day models for the remainder of this chapter, as it demonstrates lower performance than the K-NN.

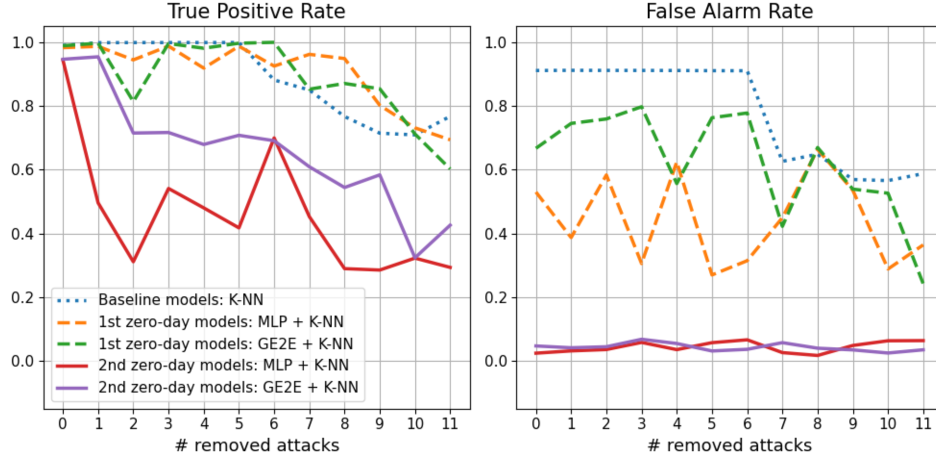

Figure 27: Comparison between my first type of zero-day detection models, my second type of zero-day detection models, and the baseline K-NN netflows classifier.

From Figure 27, it is observed that, for my second type of zero-day detection models, the GE2E Embedding Model almost always yields a higher TPR than the MLP Embedding Model, while maintaining an equivalent FAR. Based on this and previous discussions, we can conclude that the most effective combination for my second type of zero-day detection model is the GE2E Embedding Model paired with the K-NN classifier. This conclusion is also supported by Table 4.

Additionally, this figure reveals that the FAR generated by my second type of zero-day models is consistently lower than that produced by the first type (and, by extension, lower than the baseline models). This suggests that the primary advantage of my second type of zero-day detection models, namely the clustering of benign netflows, lies in its ability to better represent benign netflows, thereby significantly reducing the number of false positives. However, this reduction in FAR comes at the expense of TPR, as the second type of models consistently shows lower TPR than the first type. To evaluate whether the clustering of benign netflows has an overall positive impact on detecting zero-day attacks, I have decided to compare my first and second type of zero-day detection models using the balanced accuracy metric. As explained in subsection 2.5.2, balanced accuracy serves as a sort of arithmetic mean that takes both TPR and FAR into account.

Figure 28: Difference in the balanced accuracy between my second type of zero-day detection models and my first type, for corresponding embedding creation models.

From Figure 28, we can deduce that clustering benign netflows positively impacts the embeddings created by the GE2E loss, showing a 30% gain in balanced accuracy when no attacks are removed. However, it generally has a negative effect on the embeddings produced by the simpler MLP Embedding Model, especially when attacks are removed. This observation is also supported by Table 4. I am not surprised by this outcome. In fact, the clustering of benign netflows was introduced in subsection 4.4.2 primarily to enhance the efficiency of classifications using embeddings generated by the GE2E loss. While this strategy has been effective for the GE2E Embedding Model, it has had an unintended negative impact on the MLP Embedding Model.

6.4 From Models to the Final NIDS

Having compared all eight models (four per type of zero-day detection model) developed in this work, specifically designed to detect zero-day attacks, it's time to integrate everything and select one as the final NIDS.

First, I must decide between my first and second types of zero-day detection models, i.e., choosing to cluster or not cluster the benign netflows into sub-groups. Although my second type of model demonstrates a lower TPR than the first one, it boasts a significantly lower FAR. Additionally, its FAR remains consistently low, even when all attacks are removed. In cybersecurity, a high FAR is particularly problematic because it results in the NIDS frequently generating false alerts. These false alerts necessitate analysis by security analysts, potentially causing them to

waste time or take inappropriate actions against nonexistent attacks. Consequently, I have opted for a model from my second type of zero-day detection models, prioritizing a lower FAR.

Now, as illustrated in Figure 9, I need to choose the embedding creation model and the distance-based classifier for classifying these embeddings. These decisions are straightforward, given my choice of the second type of zero-day models. We observed in the previous section that clustering the benign netflows adversely affected the performance of the MLP Embedding Model and that the NC classifier was consistently outperformed by the K-NN.

The final NIDS proposed in this thesis involves clustering the benign netflows into sub-groups as detailed in subsection 5.2.2, then transforming all netflows into embeddings using the GE2E Embedding Model presented in subsection 4.3.2, and finally classifying these embeddings using the K-NN distance-based classifier presented in section 2.4. Figure 29 and Figure 30 depict the training and prediction pipelines, respectively, of this final NIDS.

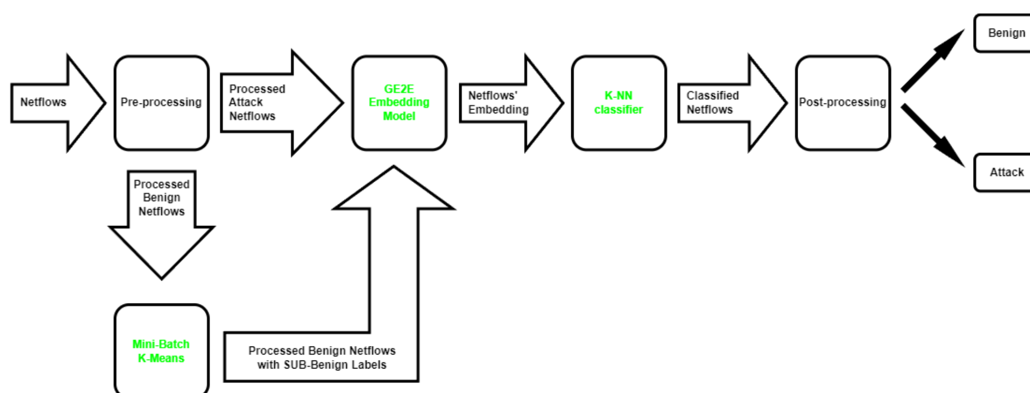

Figure 29: Training pipeline of the final NIDS.
This image was created using the online tool *DrawIO*.

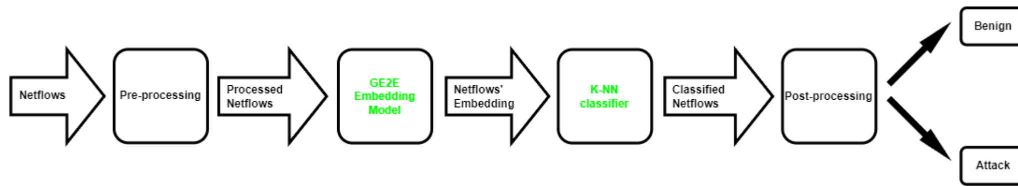

Figure 30: Prediction pipeline of the final NIDS.
This image was created using the online tool *DrawIO*.

6.5 Accuracy per Label

The final test conducted with the NIDS proposed in this thesis involved evaluating the accuracy for each label. Specifically, this entailed determining the percentage of netflows encompassed by each label that were correctly classified by the NIDS. This test was repeated four times, each with a different proportion of removable attacks being excluded: none, 1/3, 2/3, and all. These proportions correspond to the removal of 0, 3, 7, and 11 attacks, respectively (as detailed in Table 3).

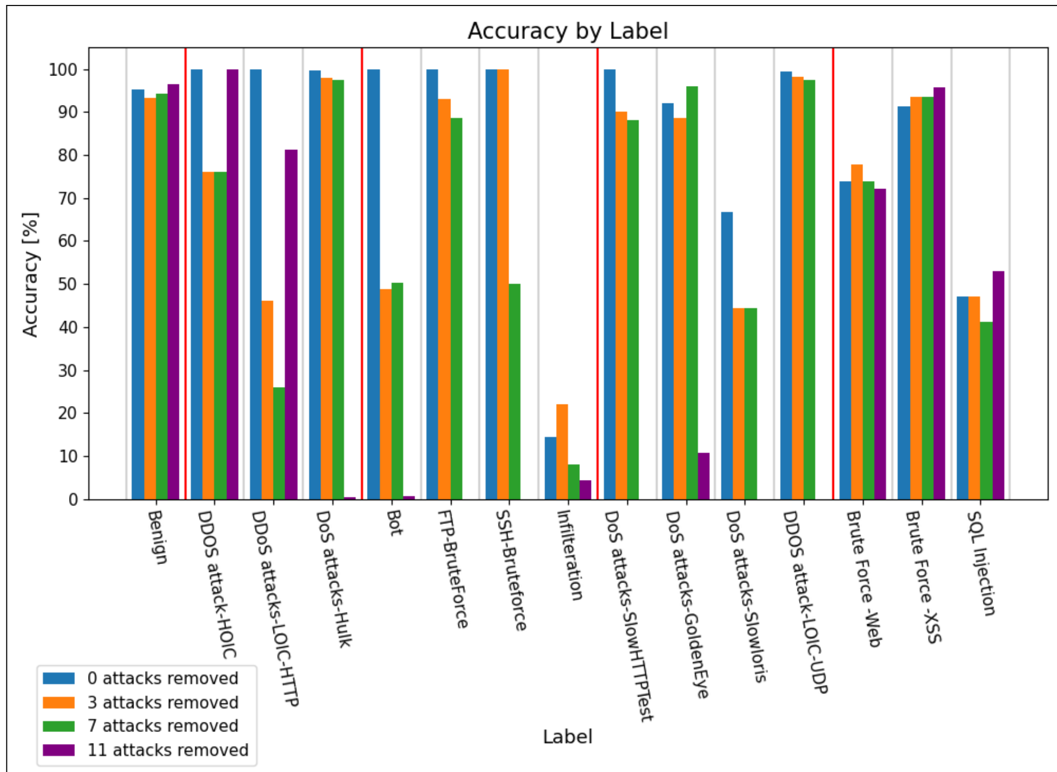

Figure 31: Accuracy for each label produced by the final NIDS when removing 0, 3, 7, and 11 attacks. The attacks, i.e., all labels except 'Benign', are sorted in the order in which they were removed. The red vertical lines indicate the groups of labels removed; for instance, the three labels between the first two red lines are the first three attacks removed, corresponding to the accuracies shown on the orange bars. The three last attacks, located to the right of the final red line, are not removable (cf. subsection 5.3.1).

The first observation from Figure 31 is that the accuracy of benign netflows remains constant and is not impacted by the removal of attacks. This makes sense, as the final NIDS is based on my second type of zero-day detection models. As indicated in Figure 26, this type of model maintains a very low and constant FAR when attacks are removed.

Another insight from Figure 31 is that an attack's accuracy is not directly correlated with whether the attack is considered a zero-day attack or not. For instance, the "DDOS attack-HOIC" initially has an accuracy of about 95%, which decreases to around 75% when the attack is removed but then rebounds to 95% when all removable attacks are excluded. A similar pattern, albeit with different accuracies, is observed with the "DDoS attacks-LOIC-HTTP". Other attacks,

like "DoS attacks-Hulk" and "FTP-BruteForce", are not or only slightly impacted when considered as zero-day attacks. This is a really important result because it demonstrates that the model has learned the semantics of an attack, implying that removing an attack does not directly decrease its ability to correctly classify it. Therefore, this final NIDS is aptly suited for detecting zero-day attacks. Interestingly, the accuracy for some attacks, such as "Bot", decreases even though they have not been removed.

A final noteworthy aspect of Figure 31 concerns the last two attacks, i.e., the two rightmost labels. As detailed in subsection 5.3.1, these attacks are never seen by the GE2E loss due to the design of my data-generator. This explains why their accuracy is unaffected by the removal of other attacks. Notably, the NIDS achieves relatively high accuracy in classifying these attacks, especially "Brute Force -XSS".

Chapter 7

Conclusion

The primary objective of this thesis was to develop a novel network intrusion detection system specifically designed to detect zero-day attacks. To achieve this, I explored a technique involving the creation of embeddings from network flows, followed by the classification of these embeddings. The central goal was to develop a model capable of generating embeddings that not only represent various types of network flows but also effectively capture the essence of flows it has never encountered before, including zero-day attacks. Ideally, this model should produce embeddings that form distinct clusters, with each cluster representing a different type of attack, and a separate cluster for benign flows. With this clustering concept in mind, I chose a distance-based model for the classification of the embeddings.

The final NIDS proposed in this thesis demonstrated strong performance in detecting zero-day attacks, as highlighted by the **GE2E + K-NN** entry in Table 4. Notably, it achieved a maximum FAR of only 10%, significantly lower than that of simpler netflow classifiers, especially the K-NN classifier. Figures Figure 29 and Figure 30 illustrate the training and prediction pipelines of this NIDS, respectively. The NIDS is composed of three distinct models:

1. **Mini-Batch K-Means Clustering Model** (cf. section 2.3), utilized during the training phase to cluster benign network flows into sub-groups.
2. **GE2E Embedding Model** (cf. subsection 4.3.2), employed to convert network flows into embeddings.
3. **K-NN Distance-Based Classifier** (cf. subsection 2.4.1), used for classifying the netflow embeddings.

In total, seven other NIDS were tested throughout this thesis before finalizing the proposed NIDS. This process included testing two distinct embedding creation

models and two different distance-based classifiers. All these models were trained and evaluated using the CSE-CIC-IDS2018 dataset [7].

To conclude this thesis, I wish to propose two directions for future improvements to this work. As depicted in Figure 9, the foundational concept of the proposed NIDS involves an embedding creation model followed by a classifier. Here, the GE2E loss was utilized for creating netflow embeddings. However, other embedding creation techniques could be explored. For instance, one might consider generating netflow embeddings directly from raw traffic packets instead of aggregated netflows. Exploring alternative classification models, aside from distance-based approaches, also presents an intriguing possibility.

The proposed NIDS incorporates clustering of benign netflows during its training process. An improvement to this approach could involve clustering not only benign netflows but also all or selected attack types. This modification could enable the embedding model to learn specific characteristics of each attack more effectively, rather than just an overall pattern. Moreover, experimenting with different clustering algorithms other than mini-batch K-means could be beneficial, especially considering algorithms capable of handling large datasets, such as the CSE-CIC-IDS2018 dataset, which contains more than 16 million netflows.

Chapter 8

Appendix

8.1 Dataset's Features Description

No.	Description of the feature
1	Netflow label
2	Netflow start timestamp
3	Netflow destination port
4	Netflow protocol
5	Netflow duration
6	Total packets sent in the forward direction
7	Total packets sent in the backward direction
8	Sum of the packet sizes in the forward direction
9	Sum of the packet sizes in the backward direction
10	Maximum size of packet in the forward direction
11	Minimum size of packet in the forward direction
12	Average size of packet in the forward direction
13	Standard deviation size of packet in the forward direction
14	Maximum size of packet in the backward direction
15	Minimum size of packet in the backward direction

No.	Description of the feature
16	Average size of packet in the backward direction
17	Standard deviation size of packet in the backward direction
18	Number of bytes transferred per second
19	Number of packets transferred per second
20	Average time between two netflows
21	Standard deviation time between two netflows
22	Maximum time between two netflows
23	Minimum time between two netflows
24	Total time between two packets sent in the forward direction
25	Average time between two packets sent in the forward direction
26	Standard deviation time between two packets sent in the forward direction
27	Maximum time between two packets sent in the forward direction
28	Minimum time between two packets sent in the forward direction
29	Total time between two packets sent in the backward direction
30	Average time between two packets sent in the backward direction
31	Standard deviation time between two packets sent in the backward direction
32	Maximum time between two packets sent in the backward direction
33	Minimum time between two packets sent in the backward direction
34	Only for TCP: Number of times the PSH flag was set in packets sent in the forward direction
35	Only for TCP: Number of times the PSH flag was set in packets sent in the backward direction
36	Only for TCP: Number of times the URG flag was set in packets sent in the forward direction
37	Only for TCP: Number of times the URG flag was set in packets sent in the backward direction
38	Total bytes used for headers in the forward direction

No.	Description of the feature
39	Total bytes used for headers in the backward direction
40	Number of packets sent in the forward direction per second
41	Number of packets sent in the backward direction per second
42	Minimum length of a netflow
43	Maximum length of a netflow
44	Average length of a netflow
45	Standard deviation length of a netflow
46	Minimum inter-arrival time of packet
47	Only for TCP: Number of packets with FIN flag
48	Only for TCP: Number of packets with SYN flag
49	Only for TCP: Number of packets with RST flag
50	Only for TCP: Number of packets with PSH flag
51	Only for TCP: Number of packets with ACK flag
52	Only for TCP: Number of packets with URG flag
53	Only for TCP: Number of packets with CWE flag
54	Only for TCP: Number of packets with ECE flag
55	Download and upload ratio
56	Average size of packet
57	Average segment size observed in the forward direction
58	Average segment size observed in the backward direction
59	Minimum segment size observed in the forward direction
60	Average bulk rate of bytes in the forward direction
61	Average bulk rate of packets in the forward direction
62	Average bulk rate in the forward direction
63	Average bulk rate of bytes in the backward direction
64	Average bulk rate of packets in the backward direction

No.	Description of the feature
65	Average bulk rate in the backward direction
66	The average number of packets in a sub netflow in the forward direction
67	The average number of bytes in a sub netflow in the forward direction
68	The average number of packets in a sub netflow in the backward direction
69	The average number of bytes in a sub netflow in the backward direction
70	Number of bytes sent in initial window in the forward direction
71	Number of bytes sent in initial window in the backward direction
72	Number of packets with at least one byte of TCP data payload in the forward direction
73	Average time a netflow was active before becoming idle
74	Standard deviation time a netflow was active before becoming idle
75	Maximum time a netflow was active before becoming idle
76	Minimum time a netflow was active before becoming idle
77	Average time a netflow was idle before becoming active
78	Standard deviation time a netflow was idle before becoming active
79	Maximum time a netflow was idle before becoming active
80	Minimum time a netflow was idle before becoming active

Table 5: Description of the 80 features in the CSE-CIC-IDS2018 dataset. The feature descriptions are sourced from the dataset's presentation page [25].

8.2 Test Environment

The experiments and evaluations presented in this work were conducted on a dedicated machine, equipped with the following specifications:

CPU	Intel I7 13700K
GPU	NVIDIA GeForce RTX 4090 24 GB
RAM 1-2	2x16GB DDR5 - 6000 MHz
RAM 3-4[1]	2x32GB DDR5 - 5600 MHz
Operating System	Debian 11
Python version	3.9.2
TensorFlow version	2.13.0

8.3 Manuscript Correction Prompt

Here is the prompt that I have used on the *ChatGPT website* to correct my entire manuscript:

```
I’m currently writing my master’s thesis and I will need your help.
I want you todo the following things:
1) Correct the spelling mistakes.
2) Correct grammatical mistakes.
3) Reformulate for better readability while maintaining my writing
style.
4) Do not change any of my mathematical notation, i.e., you cannot
change anything in the \begin{gather}...\end{gather} block.
5) You also cannot change anything between $, i.e., $...$
6) Never remove or add any double backslashes.
Can you do that if I provide you with my LaTeX code?
```

[1] The system’s RAM was increased from 32GB to 96GB to accommodate the entire CSE-CIC-IDS2018 dataset, as the initial capacity was insufficient for handling its size.

Bibliography

[1] A. K. Saxena, S. Sinha, and P. Shukla, "General study of intrusion detection system and survey of agent based intrusion detection system," in *2017 International conference on computing, communication and automation (ICCCA)*. IEEE, 2017, pp. 471–421.

[2] Symantec, "2019 internet security threat report," 2019, accessed on: 27-November-2023. [Online]. Available: https://docs.broadcom.com/doc/istr-24-2019-en

[3] symantec, "2017 internet security threat report," 2017, accessed on: 01-December-2023. [Online]. Available: https://docs.broadcom.com/doc/istr-22-2017-en

[4] Y. Guo, "A review of machine learning-based zero-day attack detection: Challenges and future directions," *Computer Communications*, vol. 198, pp. 175–185, 2023.

[5] IBM, "Cost of a data breach report 2023," 2023, accessed on: 27-November-2023. [Online]. Available: https://www.ibm.com/downloads/cas/E3G5JMBP

[6] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, pp. 1–22, 2019.

[7] C. CSE, "A realistic cyber defense dataset (cse-cic-ids2018)," 2018, accessed on: 27-November-2023. [Online]. Available: https://registry.opendata.aws/cse-cic-ids2018

[8] J. D. Kelleher, *Deep learning.* MIT press, 2019.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens,

B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[10] L. Wan, Q. Wang, A. Papir, and I. L. Moreno, "Generalized end-to-end loss for speaker verification," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 2018, pp. 4879–4883.

[11] J. M. Naik, "Speaker verification: A tutorial," *IEEE communications magazine*, vol. 28, no. 1, pp. 42–48, 1990.

[12] S. Mokhtarani, "Embeddings in machine learning: Everything you need to know," *FeatureForm*, 2021. [Online]. Available: https://www.featureform.co m/blog/embeddings-in-machine-learning

[13] yui, "A deep learning projects centralization," https://github.com/yui-mhcp, 2021.

[14] R. Rojas and R. Rojas, "The backpropagation algorithm," *Neural networks: a systematic introduction*, pp. 149–182, 1996.

[15] C. Lemaréchal, "Cauchy and the gradient method," *Doc Math Extra*, vol. 251, no. 254, p. 10, 2012.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 1177–1178.

[18] E. Fix and J. L. Hodges, "Discriminatory analysis. nonparametric discrimination: Consistency properties," *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.

[19] A. Hussain, F. Aguiló-Gost, E. Simó-Mezquita, E. Marín-Tordera, and X. Massip, "An nids for known and zero-day anomalies," in *2023 19th International Conference on the Design of Reliable Communication Networks (DRCN).* IEEE, 2023, pp. 1–7.

[20] H. Hindy, R. Atkinson, C. Tachtatzis, J.-N. Colin, E. Bayne, and X. Bellekens, "Utilising deep learning techniques for effective zero-day attack detection," *Electronics*, vol. 9, no. 10, p. 1684, 2020.

[21] M. Soltani, B. Ousat, M. J. Siavoshani, and A. H. Jahangir, “An adaptable deep learning-based intrusion detection system to zero-day attacks,” *Journal of Information Security and Applications*, vol. 76, p. 103516, 2023.

[22] G. of Canada, “Communications security establishment,” https://www.cse-cst.gc.ca/en, 2023, accessed on: 16-December-2023.

[23] U. of New Brunswick, “About the canadian institute for cybersecurity,” https://www.unb.ca/cic/about/index.html, 2023, accessed on: 16-December-2023.

[24] CIC, “Applications,” 2016, accessed on: 16-December-2023. [Online]. Available: https://www.unb.ca/cic/research/applications.html#CICFlowMeter

[25] C. CSE, “Cse-cic-ids2018 on aws,” 2018, accessed on: 16-December-2023. [Online]. Available: https://www.unb.ca/cic/datasets/ids-2018.html

[26] Lanjelot, “Patator,” https://github.com/lanjelot/patator, 2023.

[27] R. D. Graham, “Heartleech,” https://github.com/robertdavidgraham/heartleech, 2023.

[28] S. Chatterjee, “Http unbearable load king,” https://github.com/R3DHULK/HULK, 2022.

[29] Jseidl, “Goldeneye,” https://github.com/jseidl/GoldenEye, 2021.

[30] Shekyan, “Slowhttptest,” https://github.com/shekyan/slowhttptest, 2023.

[31] S. Sabri, N. Ismail, and A. Hazzim, “Slowloris dos attack based simulation,” in *IOP Conference series: materials science and engineering*, vol. 1062, no. 1. IOP Publishing, 2021, p. 012029.

[32] Imperva, “What is hoic,” "accessed on: 17-December-2023". [Online]. Available: https://www.imperva.com/learn/ddos/high-orbit-ion-cannon/

[33] P. Farina, E. Cambiaso, G. Papaleo, and M. Aiello, “Understanding ddos attacks from mobile devices,” in *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE, 2015, pp. 614–619.

[34] Radware, “DDoS-Pedia: Zeus,” https://www.radware.com/security/ddos-knowledge-center/ddospedia/zeus/, 2023, accessed on: 31-December-2023.

[35] Ares, “ARES DDOS,” https://ares.tel/, 2023, accessed on: 31-December-2023.

[36] Digininja, “Damn vulnerable web application,” https://github.com/digininja/DVWA, 2023.

[37] S. Gupta and B. B. Gupta, "Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, pp. 512–530, 2017.

[38] A. Sadeghian, M. Zamani, and S. M. Abdullah, "A taxonomy of sql injection attacks," in *2013 International Conference on Informatics and Creative Multimedia.* IEEE, 2013, pp. 269–273.

[39] Rapid7, "Metasploit framework," https://github.com/rapid7/metasploit-framework, 2023.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**