# Detection of Denial of Service Attacks Using Echo State Networks

by

**Kamila Bekshentayeva**

B. Sc., Pennsylvania State University, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

# Declaration of Committee

|  |  |
|---|---|
| **Name:** | **Kamila Bekshentayeva** |
| **Degree:** | **Master of Applied Science** |
| **Title:** | **Detection of Denial of Service Attacks Using Echo State Networks** |
| **Committee:** | **Chair:** Ivan Bajic<br>Professor, Engineering Science |
|  | **Ljiljana Trajković**<br>Supervisor<br>Professor, Engineering Science |
|  | **Mirza Faisal Beg**<br>Committee Member<br>Professor, Engineering Science |
|  | **Uwe Glässer**<br>Examiner<br>Professor, Computing Science |

# Abstract

Denial of Service and Distributed Denial of Service attacks are major threats to communication security. These cyber attacks are evolving and becoming more difficult to identify and, hence, a number of detection approaches have been proposed. Various machine learning techniques have proved useful in detecting network intrusions. We apply echo state networks to detect known DoS and DDoS attacks. Echo state networks are a reservoir computing approach to train recurrent neural networks. The reservoir in the echo state networks serves as a memory and as a nonlinear high dimensional expansion of the input. The performance of echo state network models depends on settings of reservoir hyperparameters: input scaling, spectral radius, leaking rate, size and sparsity of the reservoir, and distribution of nonzero elements. The most important features are selected using an extra-trees classifier. We use network intrusion and Internet routing datasets. We compare echo state network models to bidirectional long short-term memory, one of the widely used recurrent neural networks, and evaluate their performance based on accuracy, F-Score, false alarm rate, and training time.

**Keywords:** Machine learning, supervised learning, classification, recurrent neural networks, reservoir computing, echo state networks, network anomalies, network intrusion detection, (distributed) denial of service attacks

# Dedication

Dedicated to Rafael, Emilia, and Simon.

# Acknowledgements

Working on this Thesis would not have been possible without the support and inspiration I received on the way to completing my Master's degree.

I would like to express my gratitude to my Senior Supervisor Dr. Ljiljana Trajković for her guidance, patience, and time. She has always been inspiring me with her hard work, immense knowledge, and expertise. I am very grateful for her encouragements and hours she invested in my research work.

I would like to thank my committee members Prof. Faisal Beg and Prof. Glässer for the knowledge that I have received in their classes that were influential in selecting the topic for my research. I would also like to thank them for providing valuable and constructive feedback when reviewing my Thesis.

I am extending my gratitude to Prof. Ouldooz Baghban Karimi, late Prof. Steve Whitmore, Michael Hegedus, and Michael Sjoerdsma. I enjoyed being a TA for the classes you were teaching.

I would like to thank the professors of CMPT 726: Ke Li and Greg Mori. I applied machine learning knowledge that I acquired from this course in my Thesis. I am also thanking my team: Matthew Canute, Young-min Kim, Donggu Lee, and Adriena Wong.

Many thanks also go to my colleagues, volunteers, visitors and past members in Communication Networks Laboratory at Simon Fraser University: Zhida Li, Ana Laura Gonzalez Rios, Soroush Haeri, Professor Sorrentino, Maheeppartap Singh, Guangyu Xu, and Alfonso Diaz Alonso for their support during my graduate study.

I would also like to extend my gratitude to friends I met throughout this journey: Tatyana Mozgacheva, Sadika and Veso Jungic, Maheeppartap Singh, Daria Sergienko, Mubina Sarwar, Aigerim Shilibekova, Anna Shatayeva, Neha Sharma, Elif Erdermli, Aisuluu, and Yunduz, who have been emotionally supportive and inspiring in countless ways.

My immense gratitude goes to my husband Rafael and to our parents. I do not believe I would be able to complete my degree without their understanding and constant support in helping me balance the academic and family life. Zhanybek, Irina, and Svetlana Yakovlevna, I will never have enough words to express my gratitude for taking care of Emilia when I was away. Gulzada, Umirbek, and Eric thank you for your unconditional love and support. Emilia and Simon, you are our constant inspiration and all the efforts and accomplishments are dedicated to you.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview of Denial of Service and Distributed Denial of Service Attacks

Denial of Service (DoS) attacks are attempts of an attacker to make services unavailable to legitimate users. Distributed Denial of Service (DDoS) attacks combine the resources of multiple compromised end systems in a coordinated way to exhaust resources of a target system. DoS and DDoS are employed by attackers to overload a network's infrastructure thus causing disruptions and outages to companies and organizations. An attacker may be a cyber criminal, a hacktivist, or a user who pursues financial gain, prestige, or other personal goals from the conducted attacks [1].

DoS and DDoS attacks utilize the best-effort Internet architecture, which was originally designed for functionality without initial security concerns. Even though a victim's end system may be properly secured, it is still exposed to cyber threats because of the interdependent nature of Internet [2].

Malware bots are devices that are affected by malware while their collections are known as botnet. DDoS attacks, illustrated in Fig. 1.1, include a victim (target host), bots (daemon agents), a controller (handler), and a real attacker (the mastermind behind the attack) [3, 4].

The following steps [3] are performed to initiate a DDoS attack:

- Selection of agents: An attacker manually or automatically selects agents and exploits vulnerabilities in their machines.
- Compromise: The attack code is planted and some measures are taken by the attacker to prevent the planted code from being discovered and deactivated. Intermediate layers between bots and victims may be utilized in order to impede the traceback.
- Communication: The attacker communicates with a single or multiple controllers via Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), or User Datagram Protocol (UDP) to control further processes.

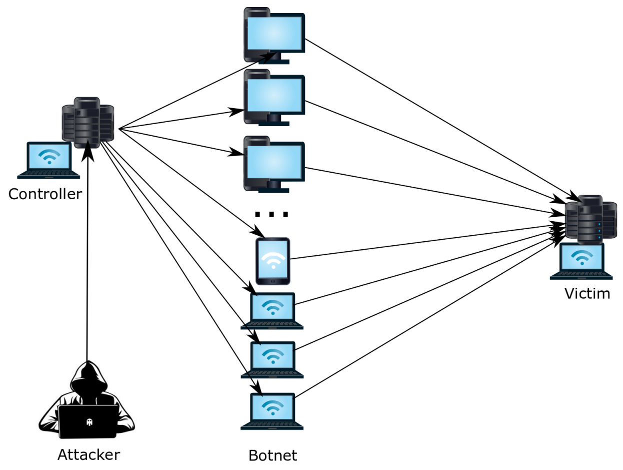


Figure 1.1: DDoS attacks include a victim, bots, a controller, and a real attacker.

- Attack: At this step, the attacker launches the attack and is able to modify features of the attack such as duration, time-to-live, and port numbers.

The first documented DDoS attack was launched in August 1999 using a tool called Trinoo. It targeted a computer in University of Minnesota through at least 227 bots [4]. The first large-scale DDoS attack executed by Mafiaboy from Canada affected Yahoo!, Amazon, Buy.com, CNN, Dell, FIFA, and eBay, leaving them inaccessible, slow, or dysfunctional. It lasted one week and captured world's attention in early 2000. In late 2016, a novel strategy was introduced to amplify the effect of DDoS attacks. It utilized a network of Internet of Things (IoT). (IoT are the Internet-connected devices such as doorbells, light switches, thermometers.) One of the known examples of the DDoS attacks that involved botnets of IoT devices included the Mirai botnet that was responsible for generating one of the largest DDoS attacks and compromised thousands of insecure IoT devices affecting Reddit, Etsy, Spotify, CNN, and the New York Times [6]. Mirai also took down a major DNS provider Dyn via massive DDoS that exceeded 1 Tbps.

A massive (1.35 Tbps) DDoS attacks based on artificial intelligence (AI) self-learning algorithms occurred in the beginning of 2018 and targeted the developer platform Github [5]. Another recent attack in 2019 targeted Amazon Web Services (AWS) causing eight hours of outage and interrupted services and leaving thousands of customers unable to reach cloud services, websites, and applications. The largest (2.3 Tbps) attack of all time - DDoS attack occurred in February 2020, affected Amazon cloud services, and caused three days of elevated threat. These attacks illustrated that DoS and DDoS attacks are evolving and becoming not only more sophisticated and devastating but also more difficult to detect and prevent, especially when using traditional countermeasures.

Every DDoS attack on average costs enterprises over $2M USD [7]. Attackers are enhancing their commercial tactics and advertising their malicious services both on the dark web and on social media channels. Reports [8, 9, 10] suggest that the number of attacks may double by 2023. According to the report [8], 52% of all DoS and DDoS attacks in 2019 lasted less than 15 min (Fig. 1.2) with the longest attack lasting 509 hours. Distribution of popular DDoS attacks by type is shown in Fig. 1.3.

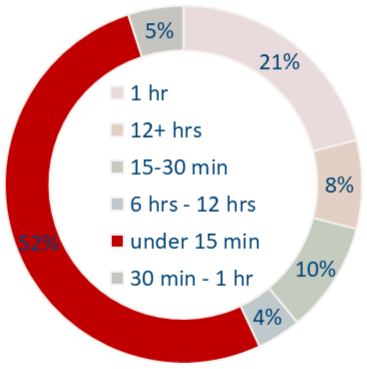


Figure 1.2: DoS and DDoS attacks duration: 52% of all DoS and DDoS attacks in 2019 lasted less than 15 min [8].

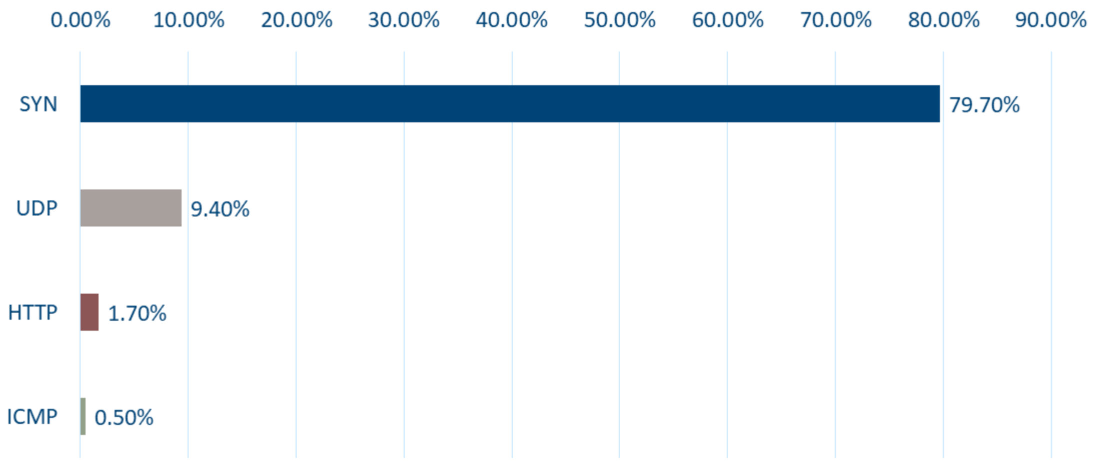


Figure 1.3: Distribution of DDoS attacks by type: SYN attacks constitute approximately 80% of all DDoS attacks [9].

## 1.2 Types of DoS and DDoS Attacks

Generalizing attacks into categories helps design counter strategies for each category. A taxonomy of DDoS attacks is presented in Table 1.1. DDoS attacks may belong to: volumetric application/network level floods, amplification and reflection, network and application protocol layer attacks, and multivector attacks.

Volumetric attacks (floods), measured in gigabits per second, use a voluminous traffic to crash a target such as DNS or web server. A botnet floods a victim with requests that

Table 1.1: A taxonomy of DDoS attacks [3, 11]

| | | |
|---|---|---|
| Degree of automation | Manual | |
| | Semiautomatic | Direct |
| | | Indirect |
| | Automatic | |
| Exploited vulnerability | Flood attack | Application level flood |
| | | Network level flood |
| | Amplification and reflection attack | Smurf attack |
| | | Fraggle attack |
| | Protocol exploit attack | |
| | Malformed packet attack | |
| Attack network used | Attacks based on agent handler network | |
| | IRC botnet based | |
| | Peer to peer network based | |
| Attack rate dynamics | Continuous | |
| | Variable | Fluctuating |
| | | Increasing |
| Victim type | Host | |
| | Resource | |
| | Network | |
| | Application | |
| Impact | Disrupting | |
| | Degrading | |
| Agent set | Constant agent set attacks | |
| | Variable set attacks | |

may not be properly formatted thus consuming victim's bandwidth with ICMP or UDP packets.

UDP is a simple connectionless transport layer protocol. In order to establish faster connections, UDP does not require the source and destination to establish an end-to-end connection for transmission. This connectionless nature of UDP is utilized by attackers for amplification and reflection DDoS attacks, which often occur together. An attacker directs requests to server spoofing the source address (inserting a false source IP address that belongs to a victim). The amplified responses from the server are reflected to the victim. This type of the DDoS attacks include Domain Name System (DNS) and Network Time Protocol (NTP) amplifications.

Network protocol attacks or (TCP) state exhaustion attacks, measured in packets per second, usually target firewalls, load balancers, and servers. In state exhaustion DoS attacks, an attacker exploits weakness within a system's memory allocation mechanism to take up all memory or state resources. An attacker may exploit vulnerabilities of TCP and UDP transport layer protocols. For example, using a TCP's three-way handshake, an attacker sends the initial SYN to establish connection with a receiver that keeps connection open waiting to receive further packets.

The main focus of application layer attacks is to monopolize SMTP, HTTP, or DNS services. An attacker may launch a slow POST operation or perform an HTTP GET flood thus overwhelming the HTTP server until the session times out.

The volume of the involved traffic often defines the difference between various types of attacks. While network and application level flood attacks are easy to identify by the large volume they take to overwhelm the network services, application layer attacks are more challenging to detect because the requests seem to be legitimate and they cause low volume of traffic.

Multivector attacks employ a combination of various types of DoS and DDoS attacks. They may be launched as a flood and evolve into other type of attacks [1].

## 1.3 DoS and DDoS Detection Methods and Anomaly-Based Detection

The continuous growth of vulnerable and connected end systems (computers, smartphones, IoT devices, self-driving cars) increase chances of successful DDoS attacks [1]. Due to the development of new attacks, various of DoS and DDoS detection, mitigation, and prevention techniques have been designed.

The first step in countering an attack is detecting the onset of an attack. Older single source attacks or volumetric attacks are easily detected by the majority of defence systems. As shown in Fig. 1.4, one could simply identify a sudden surge in a number of packets and,

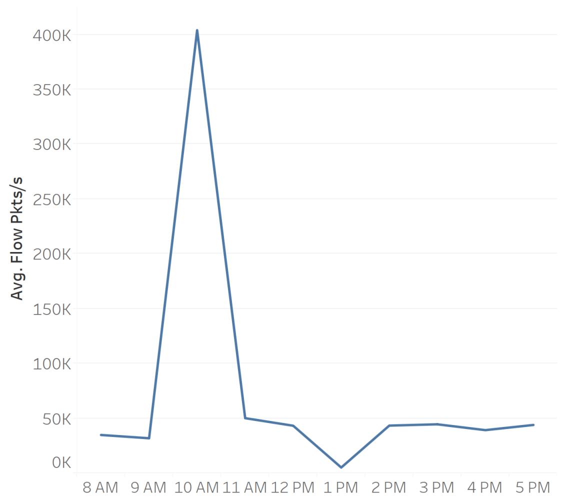


Figure 1.4: A sudden upsurge in a number of packets per second may help detecting whether the network is under attack using only one feature: “average packets/second flow” (CSE-CIC-IDS2018, Friday, February 16, 2018).

thus, successfully detect whether the network is under attack. However, the problem is not only in determining whether the network is under attack but also in accurate classification: whether a request for each source IP and port traffic flow is an anomaly or a regular event because regular traffic should not be blocked in the midst of a DoS attack. The objectives of attack detection are high accuracy, low false alarm rate, and faster detection time. Methods to detect DoS and DDoS attacks include poll-based monitoring and detection, flow-based network parameter detection, network mirrors and deep packet inspection, and anomalies-based detection [1].

Anomaly-based detection has been extensively used in machine learning, statistics, information theory, as well as application domains such as network intrusion detection, fraud detection, fault or damage detection, and intelligent monitoring and forecasting in safety critical systems [12, 13, 14]. Anomaly detection is identifying the patterns in data that do not conform to expected behavior. There are three types of anomalies: point anomalies, contextual anomalies, and collective anomalies [13]. Point anomaly is a data instance that is considered anomalous compared to the remaining observations. Contextual anomaly, also known as conditional anomaly, is an anomalous data instance defined in a specific context. Collective anomaly is a collection of related anomalous data instances. The individual outliers in a collective anomaly may not be considered anomalous. Examples of anomalies are shown in Fig. 1.5 using a simple two-dimensional dataset. It shows anomalies (regions $O_1, O_2, O_3$) with points that lie far away from two regular regions $(N_1, N_2)$. Factors that make anomaly detection a challenging task include difficulty in defining boundaries of normal regions, varying notions of anomaly for various application domains, adapting anomalous observations to appear as normal by adversaries, and lack of labeled data for training.

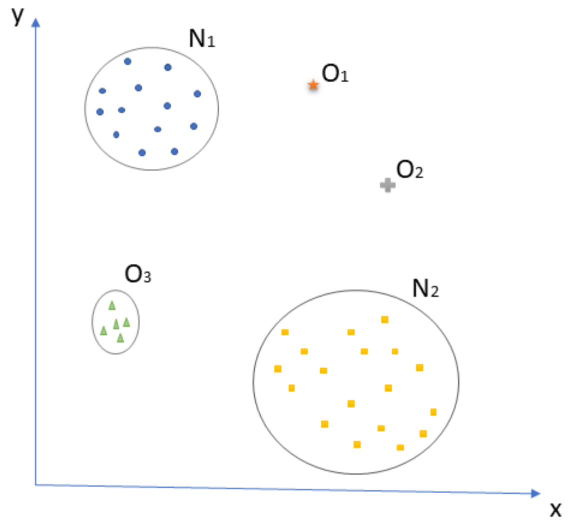


Figure 1.5: An example of anomalies in a two-dimensional dataset. It includes anomalous $(O_1, O_2, O_3)$ and regular $(N_1, N_2)$ regions.

Intrusion detection is one of the applications of anomaly detection. Two types of intrusion detection systems are host-based and network-based [15]. Host-based intrusion detection systems treat intrusions as collective anomalies (subsequences) where the nature of data is considered to be sequential. Network-based intrusion detection deals with the point anomalies in network data [12].

## 1.4 Overview of Machine Learning

Machine learning, defined as "the study of algorithms that improve automatically through experience" [17], involves the design of learning algorithms that optimize their performance as additional data are observed to solve a specific task. The history of machine learning and pattern recognition is long and successful. While the concept of machine learning is introduced in "Computing Machinery and Intelligence" by Alan Turing in 1950s [18], the applications of pattern recognition to the discoveries of regularities in data go back to the $16^{th}$ century [19]. The field of machine learning is one of the fastest developing fields due to growing computing power, availability of distributed and cloud computing, and ability to store and process data on massive scales. Machine learning and pattern recognition have been showing great promise and practical value in solving complex problems and have been adopted in various domains. Applications of machine learning include image recognition to identify specific shapes and colors (faces and fingerprints recognition), security heuristics to distinguish between attack and regular patterns (network intrusion detection systems), generating rules for behavior analytics (marketing and sales), and object recognition and prediction (autonomous driving) [20].

A large set of observations $N = \{x_1, x_2, ..., x_N\}$ called a *training set* is employed for obtaining optimal parameters of an adaptive model. An identity of a corresponding observation point $x_i$ is known as a label or a target $y_i^{target}$ [19]. A machine learning algorithm generates an output as a result of mapping input vector $\mathbf{x}$ to an output vector $\mathbf{y}$ using a function $y(x)$, which is determined during the training (learning) phase. After the model is trained, a *test set* is used to evaluate the ability of the model to correctly categorize new examples different from the observations contained within the training set. This process is known as generalization.

Machine learning includes supervised, unsupervised, and reinforcement learning paradigms. In *supervised learning*, the training data contain the input data points with their corresponding labels. Supervised learning includes classification (logistic regression, classification trees, support vector machines, random forests, artificial neural networks) and regression (linear regression, decision trees, Bayesian networks, fuzzy classification, artificial neural networks) [20]. The goal of *classification* is to predict one of a discrete set of labels. The classifier labels the observations as either an anomaly or a regular instance. The classifier models are usually trained using datasets that contain fewer samples of anomaly class. The performance of the classifier relies on the ability of the model to correctly predict the class. The goal of *regression* is to predict a value within a continuous interval. Examples of linear regression are a linear function of input variables and linear combinations of a fixed set of nonlinear functions of input variables. The latter can be nonlinear with respect to the input while being linear functions of the parameters [19].

In *unsupervised learning*, the training data do not include any labels. The purpose of such machine learning technique is to discover observations or clusters of observations with similar behavior. Unsupervised learning employs clustering (k-means, hierarchical, Gaussian mixture models, genetic algorithms, artificial neural networks) and dimension reduction (principal component analysis, tensor decomposition, multidimensional statistics, random projection artificial neural networks) [20].

In *reinforcement learning* [21], a sequence of actions is performed by an algorithm in a given environment to maximize a cumulative reward. The algorithm is not given examples of suitable outputs and is searching for the optimal ones by trial and error. Basic reinforcement process may be modeled as a Markov Decision Process (MDP) that includes a set of environment and agent states, set of agent actions, a probability of transition from one state to another, and immediate reward after transition.

Most common machine learning approach to classify network intrusions is supervised learning: classification algorithms are utilized to learn classes of regular and anomalous traffic, and detect irregularities by mapping input observations to discrete output values.

## 1.5 Machine Learning Algorithms for Network Anomaly Detection

Various network anomaly detection systems [22, 23, 24, 25, 26] have been proposed to address a dynamically changing landscape of cyber threats. They employed diverse machine learning algorithms [27, 28] such as convolutional neural networks, recurrent neural networks (RNN) [29, 30], deep belief networks, and autoencoders that offered promising performance for anomaly detection [31, 32, 33].

A cascade-structured architecture based on three-step methodology (data augmentation, hyperparameters optimization, and ensemble learning) had been proposed [23]. The approach optimized intrusion detection with neural networks and achieved high classification accuracy. Combination of supervised learning and feature selection algorithms was employed to devise novel intrusion detection solutions that addressed the high false alarm rate by classifying previously unobserved network traffic patterns [24]. Reported results demonstrated that the proposed anomaly-based intrusion detection system (IDS) employing a neural network with a wrapper feature selection outperformed other models. A deep neural network model with four hidden layers yielding high accuracy had been also introduced [34]. RNN-based IDS was proposed for binary and multiclass classification [25].

The performance of the system yielded higher accuracy when compared to traditional approaches that employ J48 algorithm, artificial neural network, random forest, and support vector machine. The proposed systems had been evaluated using KDD Cup 99 [35] and NSL-KDD [36] datasets.

An extensible framework was designed for testing machine learning algorithms in detecting DDoS attacks [26]. The framework utilized fully automated learning approach (without human interaction) that allowed detecting zero-day attacks. Before applying artificial neural network (ANN), the framework performed data collection and traffic filtering. Load and traffic probes were installed on a victim's site that exported usage data and traffic samples to the server. The management interface was employed to retrieve status and make changes to the system. The proposed solution achieved comparable results in detecting attacks not based on the traffic but instead from monitoring the resources usage within the system. Data collection was performed at various levels of the network stack and was fed to the training algorithm.

Support Vector Machine (SVM), a widely used machine learning algorithm, was employed to identify Border Gateway Protocol (BGP) anomalies [37, 38, 39, 40]. Various types of supervised RNN such as long short-term memory (LSTM) [30], gated recurrent unit (GRU) [29, 41], and stack bidirectional LSTM [42] had been applied for data classification and intrusion detection in network traffic. The main disadvantages of conventional machine learning techniques are long training time and computational complexity. In contrast, broad learning system (BLS) employed fewer hidden layers, relied on calculating pseudo-inverse during the training process, and required comparably shorter training time when used for function approximation, time series forecast, and image recognition [43, 44, 45]. BLS, when employed for network anomaly detection, was compared to RNN [46, 47] and SVM [40] showing competitive performance. Various BLS models, such as BLS, RBF-BLS, BLS with cascades of mapped features and cascades of enhancement nodes with and without incremental learning, were evaluated using CIC-IDS2017 and CSE-CIC-IDS2018 datasets [48].

Echo state networks (ESNs), a reservoir computing approach that uses sparse neural networks, was employed in online anomaly detection framework that was implemented on mote-class devices (computers with limited resources in terms of memory size, processing speed, wireless communication throughput, and power budget) [49]. The approach showed comparable accuracy to PC-based intrusion detection implementation. ESN was proved to detect a wider variety of anomalies with lower false alarm rate when compared to rule-based anomaly detection techniques. It was also shown that ESN was a fast and simple approach that was not too resource intensive to be implemented on motes for pattern recognition alongside a fully-functional real-time environmental monitoring. Three neural networks models (feed-forward neural network, LSTM, ESN) were trained and tested using CSE-CIC-IDS2018 network dataset. Additionally, the models were tested using a dataset that was generated by employing a modified HULK attack script. ESN demonstrated a comparable performance in terms of accuracy and F-Score [50].

## 1.6 Research Contributions

Various machine learning approaches were employed for detecting network anomalies [37, 38, 39, 46, 47, 40, 48]. Previous research findings were extended by including ESNs as a feasible reservoir computing approach to identify network intrusions by classifying regular and anomalous instances.

We have selected synthetically generated CIC-IDS datasets: CIC-IDS2017, CSE-CIC-IDS2018, and CIC-DDoS2019 that reflect the characteristics of diverse regular and anomalous current network trends. Important properties and features of DoS and DDoS attacks are visualized using Tableau 2019.4. Border Gateway Protocol (BGP) datasets that were acquired from BGP trace collectors contain worms that may cause DoS and DDoS. We also considered the BGP data collected over periods of large DDoS attacks that targeted Amazon Web Services in 2019 and 2020. In order to improve classification accuracy, we decrease redundancy in data by selecting appropriate features employing extra-trees, a tree-based ensemble method. Furthermore, we compare the performance of classification models using both unbalanced and balanced datasets.

We provide an overview of DoS and DDoS attacks, past network intrusion detection systems and algorithms, description of recurrent neural networks and reservoir computing, characterization of ESNs, the hyperparameters of a reservoir, and review of ESNs related work. By varying the ESN hyperparameters, we generate ESN models to evaluate the influence of the ESN reservoir configuration. The 10-fold cross-validation is employed for model selection. We compare the best ESN model to bidirectional long short-term memory (Bi-LSTM), a widely used recurrent neural network and evaluate both based on accuracy, F-Score, false alarm rate, and training time.

The experiments are performed on Windows 10 64-bit Operating System, and Intel Corei7-8650U CPU at 1.9-2.11 GHz. We use Python 3.8 with the following libraries and packages: numpy, pandas, scipy, and sklearn. We employ PyTorch for Bi-LSTM model.

## 1.7 Organization of the Thesis

The Thesis is organized as follows: We first describe Denial of Service and Distributed Denial of Service attacks, the methods of their detection, and provide an overview of machine learning. Recurrent neural networks, reservoir computing, and ESNs are introduced in Chapter 2. We describe the ESN reservoir hyperparameters and provide details about ESN output training. The description of three CIC-IDS datasets considered in this Thesis and characterization of the collected attacks and their features are given in Chapter 3. Chapter 4 contains the description of the BGP, BGP data collection from RIPE and Route Views, generated BGP datasets, and the anomalous events and their features. In Chapter 5, we describe the employed feature selection algorithm, performance metrics, and experimental procedure. Results, including the comparison with Bi-LSTM model, are given in Chapter 6. We conclude with Chapter 7. The list of references is provided in the Reference Section. The scripts for detecting anomalies are provided in the Appendix.

# Chapter 2

# Echo State Networks

## 2.1 Recurrent Neural Networks

Artificial neural networks are efficient models for problems that are challenging to address using classic machine learning approaches. One of the most commonly applied neural networks is *feed-forward neural network* known as multilayer perceptron. Feed-forward neural networks model [51, 52, 53], described as a series of functional transformations, may be unable to solve cases where a time-series inputs are processed (Fig. 2.1($a$)). *Recurrent Neural Networks* (RNNs) have been introduced specifically for sequential data [54]. RNNs also belong to a class of artificial neural networks and are one of the widely used approaches to detect anomalies in network datasets.

A simple RNN shown in Fig. 2.1($b$) has an input node, a hidden layer (associated with a weight), and an output node. The output may be represented as [55]:

$$\mathbf{y} = \mathbf{W}_L \mathbf{h_L}, \tag{2.1}$$

where $\mathbf{y}$ is an output layer, $\mathbf{W}_L$ are model parameters of the output layer, and $\mathbf{h_L}$ is the last hidden layer. Unlike the case of neural networks with sequential structure shown in Fig. 2.1($a$), the input of RNN is flowing from a previous state of a hidden layer and is fed into the next state along with the new input signal at each step, as seen in the unrolled RNN in Fig. 2.1($c$).

RNNs often employ backpropagation algorithm for training. Backpropagation is an efficient dynamic programming algorithm for computing gradients. It consists of two stages: forward pass and backward pass. In forward pass, the values of all intermediate variables such as pre- and post-activations ($\mathbf{z_1}$, ..., $\mathbf{z_L}$, $\mathbf{h_1}$, ..., $\mathbf{h_L}$) are computed:

$$\mathbf{h_L} = f(\mathbf{z_L}), \tag{2.2}$$

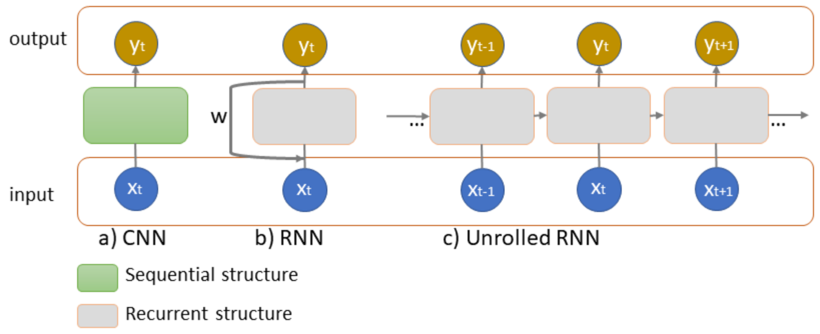


Figure 2.1: Sequential and recurrent structures of neural networks. The input of a neural network with recurrent structure is flowing from a previous state of a hidden layer and is circularly fed into the next state along with the new input signal.

where $\mathbf{h_L}$ is the last hidden layer of total $n$ hidden layers, $f(\cdot)$ is known as an activation function, and $\mathbf{z_L}$ is the last pre-activation layer:

$$\mathbf{z_L} = \mathbf{W_{L-1}}\mathbf{h}_{L-1} \ldots \mathbf{W}_1\mathbf{h_1}, \tag{2.3}$$

where $\mathbf{W}_{L-1}, \ldots, \mathbf{W}_{L-n}$ are model parameters of intermediate hidden layers $\mathbf{h}_{L-1} \ldots \mathbf{h}_{L-n}$:

$$\mathbf{h_{L-n}} = f(\mathbf{W_{L-(n+1)}}\mathbf{h}_{L-(n+1)}), \tag{2.4}$$

with $\mathbf{h_1}$ as a first hidden layer:

$$\mathbf{h_1} = f(\mathbf{z_1}), \tag{2.5}$$

where $\mathbf{z_1}$ is the first pre-activation layer:

$$\mathbf{z_1} = \mathbf{W}_0\mathbf{x}, \tag{2.6}$$

where $\mathbf{W_0}$ are input weights and $\mathbf{x}$ is an input layer.

In backward pass, gradients of the loss function $\mathbf{L}(\mathbf{y})$ are computed first with respect to all later layers, starting from the last layer. The eventual goal of the backward pass is to compute gradient of the loss function with respect to first layer's weights and biases. The partial derivative of the loss with respect to the post-activations of the last hidden layer is:

$$\frac{\partial L}{\partial \mathbf{h_L}} := \frac{\partial \mathbf{y}}{\partial \mathbf{h_L}} \frac{\partial L}{\partial \mathbf{y}} = \mathbf{W}_L^T \frac{\partial L}{\partial \mathbf{y}}. \tag{2.7}$$

The partial derivative of the loss with respect to the pre-activations of the $l^{th}$ hidden layer:

$$\frac{\partial L}{\partial \mathbf{z_l}} := \frac{\partial \mathbf{h_l}}{\partial \mathbf{z_l}} \frac{\partial L}{\partial \mathbf{h_l}} = \begin{pmatrix} f'(z_{l,1}) & 0 & \cdots & 0 \\ 0 & f'(z_{l,2}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(z_{l,n}) \end{pmatrix} \frac{\partial L}{\partial \mathbf{h_l}}. \tag{2.8}$$

The partial derivative of the loss with respect to the post-activations of the $l^{th}$ hidden layer is:

$$\frac{\partial L}{\partial \mathbf{h_l}} := \frac{\partial \mathbf{z}_{l+1}}{\partial \mathbf{h_l}} \frac{\partial L}{\partial \mathbf{z}_{l+1}} = \mathbf{W}_l^T \frac{\partial L}{\partial \mathbf{z}_{l+1}}. \tag{2.9}$$

The partial derivative of the loss with respect to the weights between $l^{th}$ layer and the preceding $l+1$ hidden layer is:

$$\frac{\partial L}{\partial \mathbf{w_{l,j}^T}} := \frac{\partial \mathbf{z_{l+1}}}{\partial \mathbf{w_{l,j}^T}} \frac{\partial L}{\partial \mathbf{z_{l+1}}} = \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{h_l} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \frac{\partial L}{\partial \mathbf{z_{l+1}}}. \tag{2.10}$$

For example, to compute the partial derivative of the loss with respect to the weights of the first hidden layer is:

$$\frac{\partial L}{\partial \mathbf{w_{0,j}^T}} := \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{x} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \prod_{l=1}^{n} \left( \begin{pmatrix} f'(z_{l,1}) & 0 & \cdots & 0 \\ 0 & f'(z_{l,2}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(z_{l,n}) \end{pmatrix} \mathbf{W}_l^T \right) \frac{\partial L}{\partial \mathbf{y}}. \tag{2.11}$$

If the activation function $f(\cdot)$ is a logistic sigmoid or hyperbolic tangent, the derivatives of the large pre-activations are close to zero. The small entries $f'(z_{l,n})$ in (2.11) multiplied with the remaining terms resulting in a small or “vanishing” gradient. If there are many layers, the magnitude of the gradient decays exponentially as the number of such layers grows. Vanishing gradient problem may cause the network to stop updating the weights during training. Selecting a different activation function such as rectified linear unit (ReLU) or softplus may help creating a more stable model.

In the opposite case, when the weights $\mathbf{W}$ are large, the gradient may “explode”. Assuming we use a non-saturating activation functions such as ReLU or softplus, if there are many layers whose weight matrices’ singular values are large, the magnitude of the gradient grows exponentially with the increasing number of such layers.

Vanishing and exploding gradients are encountered in backpropagation process of the simplest RNN models. Solutions that treat the gradient problems by efficiently coping with long-term dependencies include *long short-term memory* (LSTM) [56] and gated recurrent unit (GRU) [29]. Echo state networks (ESNs) [57] do not employ gradient-based iterative optimization algorithms such as backpropagation to compute the optimal weights. Therefore, they do not encounter vanishing or exploding gradient problems.

### 2.1.1 Long Short-Term Memory and Bidirectional Long-Short Term Memory

Vanilla RNN architectures may perform poorly with the data that have long-time gaps [137]. The main advantages of LSTM is that while retaining memory, it allows the model to be trained over long sequences. LSTM employs four gates in order to overcome the gradient problems [138].

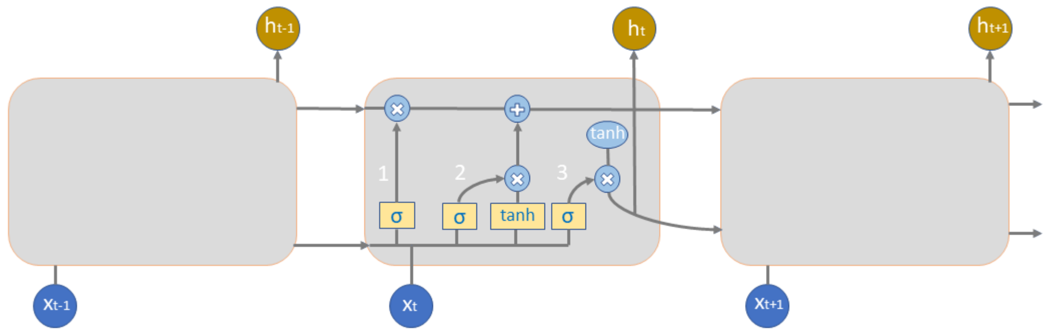


Figure 2.2: A repeating module of LSTM with four interracting neural network layers. The main components: the cell state (indicated by the horizontal line on top of a diagram), three gates (yellow bricks), vectors (grey arrows), and pointwise operators (blue circles).

LSTM is composed of repeating modules chained together. The main components of one repeating module shown in Fig. 2.2 are the cell state and LSTM gates that have the ability to control the information that may be added to or removed from the cell state. The lines in figure indicate vectors while blue circles represent pointwise operations. Sigmoid layers generate values between 0 and 1 that denote the amount of information to be passed to the rest of the network. Steps 1, 2, and 3 [139] are described as:

1. A sigmoid layer called forget gate $f_t$ decides what information may be kept or discarded by accepting its inputs (output of the previous hidden state $h(t-1)$ and a new input $x(t)$) and applying sigmoid activation. The output of the forget gate is:

$$f_t = \sigma(W_f[h_{t-1}, x(t)] + b_f). \tag{2.12}$$

2. A sigmoid layer called input gate $i_t$ determines what new information is added:

$$i_t = \sigma(W_i[h_{t-1}, x(t)] + b_i). \tag{2.13}$$

A vector $\widetilde{C_t}$ of new candidate values to be added to the current state is created after *tanh* is applied:

$$\widetilde{C_t} = tanh(W_c[h_{t-1}, x(t)] + b_c). \tag{2.14}$$

Then, the current cell state $C_t$ is updated as:

$$C_t = f_t * C(t-1) + i_t * \widetilde{C_t}. \tag{2.15}$$

3. The sigmoid output gate layer $o_t$ has a returning value of:

$$o_t = \sigma(W_o[h_{t-1}, x(t)] + b_o) \tag{2.16}$$

that is multiplied by a vector of all possible values between -1 and 1 generated after applying *tanh* activation:

$$h_t = o_t * tanhC(t). \tag{2.17}$$

The parameters $W_*$ and $b_*$ represent the weight and the bias, respectively [139].

Bidirectional LSTM neural network is one of the variants of LSTM that has two hidden layers of opposite directions connected to the same output [140]. Bi-LSTM aims to improve the performance of a model for sequence classification tasks due to its ability to utilize additional information: the output obtains information from past (backward or negative time direction) and future (forward or positive time direction) states at the same time [42]. Therefore, having timesteps of the input sequence, Bi-LSTMs employ two LSTMs instead of one as the input sequence. The general structure of Bi-LSTM is shown in Fig. 2.3.

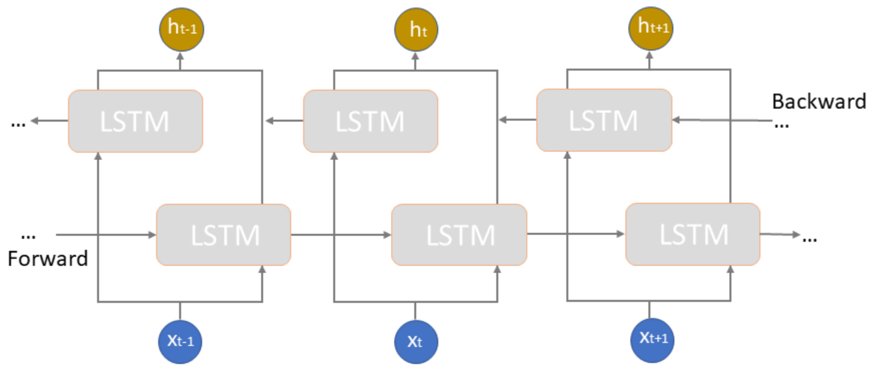


Figure 2.3: Module structure for the Bi-LSTM neural network.

Typically, including future information introduces delays in standard RNN architectures. In Bi-LSTMs, forward and backward cell states are not interacting and, therefore, no delays are associated when using future information. Bi-LSTM may be trained using algorithms similar to RNN because the two directional layers are not connected.

## 2.2 Reservoir Computing as a Paradigm for Training RNNs

Reservoir computing (RC) is an approach to supervised training of RNNs that avoids the gradient related problems. Reservoir is a randomly connected network of nodes (nodes, neurons, and units are used interchangeably) that may be excited by the input $x(n)$ of the network. The reservoir weights are not changed by training. Most common reservoirs are echo state networks (ESNs) in machine learning and liquid state machine (LSM) in computational neuroscience. They both share the idea that it is sufficient to train only the memoryless output weights leaving out the supervised adaptation of input and reservoir weights (reservoir, internal, and recurrent weights are used interchangeably). With this approach, the difficulties of training RNNs, such as discontinuous changes in behavior caused even by small changes due to cyclic dependencies in RNNs, may be prevented while achieving adequate performance in various tasks. Other problems such as non-convergence stemmed from classical gradient descent RNN approach may be avoided when employing ESN and LSM methods thus decreasing computational complexity and costs [58]. Reservoir in ESNs/LSM is a linear combination of the reservoir activations. Because there are no cyclic dependencies between the trained output connections, training an ESN/LSM becomes a simple linear regression task. The difference between gradient-based and RC-based RNN training is shown in Fig. 2.4. Weights in gradient based training are updated iteratively while in ESN the values of the output weights $\mathbf{W}^{out}$ are calculated in a single iteration.

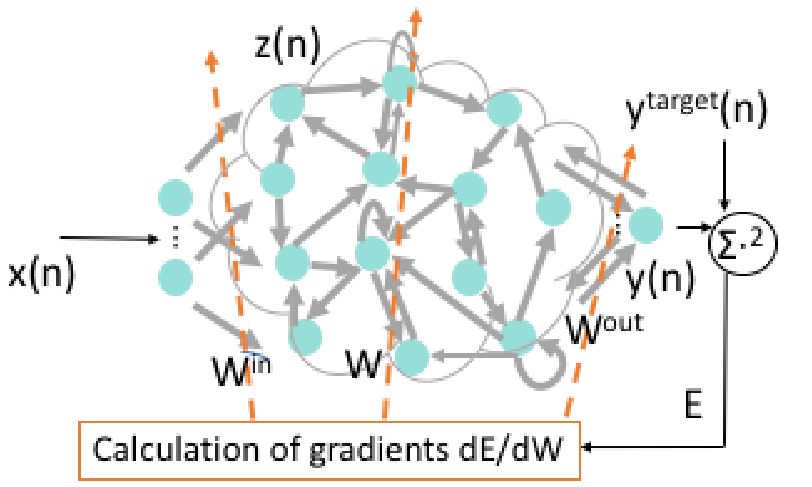
The recurrent networks in RC are usually discrete-time networks of non-linearly activated nodes with the following reservoir state activation equation:

$$\mathbf{z}(n) = f(\mathbf{x}(n)\mathbf{W}^{in} + \mathbf{z}(n-1)\mathbf{W} + \mathbf{y}(n-1)\mathbf{W}^{fb}), \qquad n = 1, ..., N, \tag{2.18}$$

where $\mathbf{z}(n) \in \mathcal{R}^{N_z}$ is a vector of reservoir node activations at a time step $n$, $f(\cdot)$ is the node activation function (usually the $tanh(\cdot)$ applied elementwise), $\mathbf{W}^{in} \in \mathcal{R}^{N_x \times N_z}$ is a randomly generated input weight matrix, $\mathbf{x}(n) \in \mathcal{R}^{N_x}$ is an input, $\mathbf{W} \in \mathcal{R}^{N_z \times N_z}$ is a randomly generated sparse reservoir weight matrix, $\mathbf{W}^{fb} \in \mathcal{R}^{N_z \times N_y}$ is an optional output feedback weight matrix, and $\mathbf{y}(n) \in \mathcal{R}^{N_y}$ is the network output (typically, $N_y = 1$). Discrete time $n = \{1, ..., N\}$ with $N$ being the total number of data points in the training set, $\mathcal{R}$ is a set of real numbers, $N_x$, $N_z$, and $N_y$ are numbers of input, reservoir, and output nodes. [59]. Bias, that may be easily implemented by adding an additional input with a constant value of 1 and a randomly generated column in the input weights matrix, is omitted in (2.18) for simplicity. The output of the network is a combination of the reservoir state and input, defined as:

$$\mathbf{y}(n) = g([\mathbf{z}(n); \mathbf{x}(n)]\mathbf{W}^{out}), \qquad n = 1, ..., N, \tag{2.19}$$

where $\mathbf{W}^{out} \in \mathcal{R}^{(N_x+N_z)\times N_y}$ is the learned output weight matrix, $g(\cdot)$ is the output activation function (commonly $tanh(\cdot)$ or the identity function when using linear regression)



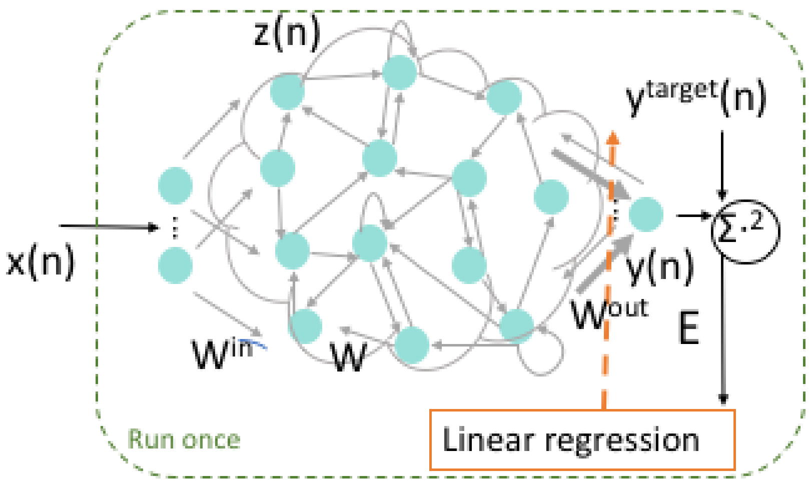


Figure 2.4: Gradient-based (top) and RC-based (bottom) RNN training: $\mathbf{x}(n)$ is input vector, $\mathbf{z}(n)$ is a vector of nodes activations, $dE/dW$ is gradient of loss function $E$ with respect to weights. The operator $\sum(\cdot)^2$ indicates the sum of squared error between the output and target vectors. The optimal weights ($W^{in}$, $W$, $W^{out}$) that are being computed are in bold.

applied componentwise, and ";" stands for a vertical concatenation of vectors (or matrices). The standard supervised training in case of ESN starts with the training input sequence $\mathbf{x}(n)$ that is fed to the reservoir with reservoir's initial state equal to zero. The internal states are collected over the entire training period. Output weights $\mathbf{W}^{out}$ are then derived as the linear regression weights of the label $\mathbf{y}^{target}(n)$.

A new perspective on RC [58] states that the reservoir (recurrent part) is *trained differently* than the output, moving away from the traditional view that only the output is trained while the reservoir remains unchanged. The partition of reservoir and output training creates ground for two independent research directions: reservoir training and output training. The best results may be combined to test different types of RNN models and supervised, unsupervised, reinforcement, and biologically inspired adaptation techniques. The original

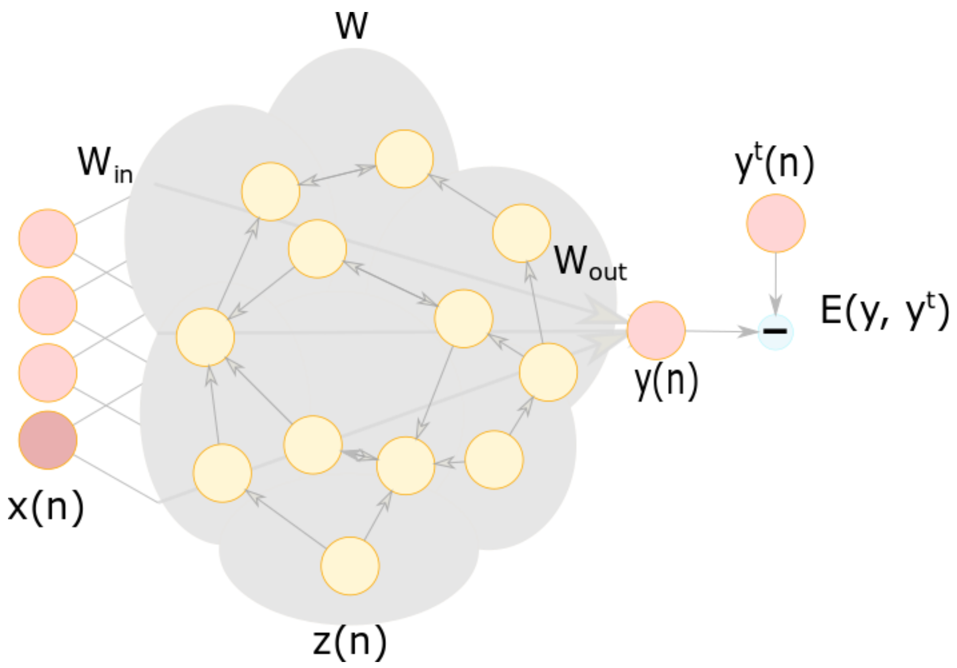


Figure 2.5: A graphical representation of training the echo state network: input $\mathbf{x}(n) \in \mathcal{R}^{N_x}$, input weights matrix $\mathbf{W}^{in} \in \mathcal{R}^{N_x \times N_z}$, reservoir activation $\mathbf{z}(n) \in \mathcal{R}^{N_z}$, reservoir weights matrix $\mathbf{W} \in \mathcal{R}^{N_z \times N_z}$, output $\mathbf{y}(n) \in \mathcal{R}^{N_y}$, output weights matrix $\mathbf{W}^{out} \in \mathcal{R}^{(N_x+N_z)\times N_y}$, labels $\mathbf{y^t}(n) \in \mathcal{R}^{N_y}$, and root mean-square error $E(\mathbf{y}, \mathbf{y^{target}})$.

approach is also used for its simplicity and effectiveness. Original and modern paradigms of RC were employed for temporal pattern recognition and classification, time series prediction, and controlling of nonlinear systems [58]. Reservoirs resemble the spatial encoding of the temporal information in the brain that was used for sensory-motor sequence and natural language learning models [60, 61]. Other RC applications include speech and handwriting recognition [62, 63, 64], robot motor control [65], financial forecasting [68, 66, 67, 69], and medical applications [71, 70].

## 2.3 Description of Echo State Networks

ESNs are a practical, conceptually simple, and computationally non-expensive to implement reservoir computing approach [57]. The ESN approach where the reservoir is generated randomly and only the output is trained without the need to fully adapt all weights provided important insights for training RNNs. A graphical illustration of ESN and the idea of training ESN is shown in Fig. 2.5.

ESN is employed as a supervised machine learning approach for time series data for the tasks where for the given input $\mathbf{x}(n) \in \mathcal{R}^{N_x}$, the output of the model $\mathbf{y}(n) \in \mathcal{R}^{N_y}$ matches as closely as possible the provided target $\mathbf{y^{target}}(n) \in \mathcal{R}^{N_y}$. According to the original reservoir computing approach [57], the four steps associated with applying ESNs are:

1. Generate random reservoir with parameters: $\mathbf{W}^{in} \in \mathcal{R}^{N_x \times N_z}$ (input weight matrix), $\mathbf{W} \in \mathcal{R}^{N_z \times N_z}$ (reservoir or recurrent weight matrix), and leaking rate $\alpha \in (0, 1]$ (used in updating the reservoir state);

2. Calculate reservoir activation states $\widetilde{\mathbf{z}}(n)$ from the training input $\mathbf{x}(n)$. Let $\mathbf{s}(n) = [s_1(n), s_2(n), ..., s_{N_x}(n)]^T \in \mathcal{R}^{N_x \times 1}$, where $n = 1, 2, ..., N$, be the collected time series, $\mathbf{x}(n) = \mathbf{s}(n)$ is the input, and $\mathbf{y}(n) = \mathbf{s}(n + h)$ is the output at time step $n$ with the prediction horizon $h$. The echo states $\widetilde{\mathbf{z}}(n) \in \mathcal{R}^{N_z \times 1}$ are generated from the input $\mathbf{x}(n)$ and the echo states $\mathbf{z}(n - 1)$. The initial state of the reservoir is a zero vector. The common ESN reservoir state update equations are:

$$\widetilde{\mathbf{z}}(n) = tanh([\mathbf{x}(n)]\mathbf{W}^{in} + \mathbf{z}(n-1)\mathbf{W}) \tag{2.20}$$

$$\mathbf{z}(n) = (1-\alpha)\mathbf{z}(n-1) + \alpha\widetilde{\mathbf{z}}(n), \tag{2.21}$$

   where $\mathbf{z}(n) \in \mathcal{R}^{N_z}$ is the reservoir state update, and $\widetilde{\mathbf{z}}(n) \in \mathcal{R}^{N_z}$ is its activation at time step $n$, with $N_z$: number of reservoir nodes. The most commonly used sigmoid wrapper $tanh(\cdot)$ is applied element-wise. Input weight matrix is $\mathbf{W}^{in} \in \mathcal{R}^{N_x \times N_z}$, the reservoir or recurrent weight matrix is $\mathbf{W} \in \mathcal{R}^{N_z \times N_z}$, and $\alpha \in (0, 1]$ is the leaking rate. In cases where the model is used without the leaky integration, $\alpha = 1$ and $\mathbf{z}(n) \equiv \widetilde{\mathbf{z}}(n)$. The state of the ESN contains information about the input history where recent input vanishes gradually as time elapses.

   Matrix $\mathbf{Z} \in \mathcal{R}^{N \times (N_z + N_x)}$ is generated by concatenating the column vectors $[\mathbf{z}(n); \mathbf{x}(n)]$ horizontally over the training data points $n$. Labels $\mathbf{y}^{\mathbf{target}}(n) \in \mathcal{R}^1$ are collected into a matrix $\mathbf{Y} \in \mathcal{R}^{N \times 1}$. $\mathbf{Z}$ and $\mathbf{Y}$ have a row for each training time step $n$.

3. Use linear regression to obtain the output weights $\mathbf{W}^{out}$ from the reservoir by minimizing the mean square error of the network output with respect to the labels $\mathbf{Y}$:

$$\mathbf{W}^{out} = \mathbf{Z}^{\dagger}\mathbf{Y}. \tag{2.22}$$

   The matrix $(\mathbf{Z}^T\mathbf{Z})^{-1}\mathbf{Z}^T$ is known as the pseudoinverse (or Moore-Penrose inverse) of $\mathbf{Z}$, and is denoted as $\mathbf{Z}^{\dagger}$. In order to match the output of the model with the labels, the loss function - root mean-squared error (RMSE) $E(\mathbf{y}, \mathbf{y}^{\mathbf{target}})$ (2.23) should be minimized [72], where RMSE is averaged over the output dimensions $N_y$:

$$E(\mathbf{y}, \mathbf{y}^{\mathbf{target}}) = \frac{1}{N_y} \sum_{n=1}^{N_y} \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i(n) - y_i^{target}(n))^2}. \tag{2.23}$$

4. Evaluate the network by applying collected output weights $\mathbf{W}^{out}$ with the new input $\mathbf{x}(n)$ to compute $\mathbf{y}(n)$. The output layer is a linear combination of the input and the reservoir state:
$$\mathbf{y}(n) = [\mathbf{z}(n); \mathbf{x}(n)]\mathbf{W}^{out}, \tag{2.24}$$
where $\mathbf{y}(n)$ is the output of the network and $\mathbf{W}^{out} \in \mathcal{R}^{(N_z+N_x)\times 1}$ is the learned output weight matrix. Vertical vector (or matrix) concatenation is represented by $[\cdot;\cdot]$.

ESN is said to have the echo state property if it is able to “wash out” the initial state of the reservoir at a rate independent of the input sequence. ESN with the echo state property is designed so that the effect of the past input on the reservoir gradually fades away [57]. Echo state property ensures that, provided an input sequence, future trajectories of initial states become indistinguishable [73].

## 2.4 Echo State Networks: Related Work

During the past two decades, ESNs have been employed in a variety of domains and tasks, including time series forecasting [66, 67, 68, 69], wireless communnication networks [74], speech and handwriting recognition [63, 62], music imitation [75], fetal ECG monitoring [76], and robot control [65].

Application of neural networks in predicting financial time series has been of a great interest. A variety of artificial neural networks have been considered, including RNNs. However, predicting nonlinear and volatile stock data using vanilla RNN methods have encountered challenges due to its slow convergence and high computational cost. ESNs proved powerful in approximating dynamical systems and have been applied in prediction of future stock prices outperforming classic neural networks [66, 67]. Most of the key reservoir hyperparameters of standard ESNs are configured by trial and error. A deterministically constructed ESN and its forecasting performance have been investigated exhibiting higher accuracy with minimum complexity when compared to a standard ESN model [68]. The problem of overfitting is solved with the proposed multi-objective diversified ESN for predicting trends in stock prices [69].

ESNs have been considered for optimization of uplink-downlink decoupling for small cell networks where users may be associated with different base stations and may access both licensed and unlicensed long-term evolution (LTE) bands. ESN acts as a framework of an allocating algorithm that allows base stations to select optimal resource allocation strategies based on the limited information about the states of network and users. The developed solution shows high performance gains based on rate and load balancing when compared to conventional approaches such as Q-learning. Furthermore, the ESN resource allocation framework significantly decreases information exchange for the cellular networks [74].

The models that may support the long-term dependencies are proved effective in imitating music [75]. Currently, most methods in reproducing and creating music modeling are based on RNNs such as LSTM [77, 78, 79]. The ESN model has been applied to generate musical compositions as complex systems of probabilistic relationship [75]. Setting of ESN reservoir hyperparameters in combination with other variables are explored in order to provide a comparable result. ESN is shown as a feasible, light-weight, and least computationally "costly" solution for Mozart's music imitation when selecting the best values for leaking rate, input scaling, and spectral radius [75].

ESNs have been applied in detecting fetal QRS (a combination of three graphical deflections seen on a typical electrocardiogram: Q wave, R wave, and S wave) complexes by monitoring the fetal electrocardiogram (fECG) and parameters that are useful in portraying fetal heart health information. The long-standing challenge in noninvasive fECG is low signal-to-noise ratio due to the noise caused by strong signal interferences of maternal ECG (mECG), fetal brain activity, and myographic signals. ESN is trained to recognize fetal QRS (fQRS) along with the options of dynamic programming employed to combine data coming from sensors. The proposed method has high score with fast processing time. The results may be improved by using larger ESN reservoir. However, that might increase the processing time [76].

ESN variations have been proposed in order to improve ESNs' performance in pattern recognition, prediction, and classification tasks. In order to address dynamic time warping (measuring similarity between two temporal sequences) in pattern recognition, a modified version of ESNs named time warping invariant ESN (TWIESNs) has been designed. This modification employs leaky integrator node reservoirs and has been applied for handwriting recognition tasks [80]. A leaky integrator node is a biologically inspired model of a neuron, which accumulates its input signals, but also exponentially leaks the accumulated excitation over time (2.25–2.26):

$$\widetilde{\mathbf{z}}(n) = f([\mathbf{x}(n)]\mathbf{W}^{in} + \mathbf{z}(n)\mathbf{W}) \tag{2.25}$$

$$\mathbf{z}(n) = c^{-1}(-\alpha)\mathbf{z}(n) + \widetilde{\mathbf{z}}(n)), \tag{2.26}$$

where $f(\cdot)$ is the activation function and $c$ is the positive time constant. To adapt the network to the temporal characteristics during training with slow dynamical system for classifying slow noisy time series data using Japanese vowel dataset, ESN nodes are considered as leaky integrated nodes. In this case, a stochastic gradient descent approach has been introduced to optimize a leaking rate [62].

Multivariate time series employ more than one time-dependent variable. Collinearity problem arises when one variable is correlated with another, making it hard to determine their separate influences. To address a collinearity problem that is observed in high-dimensional reservoirs to predict multivariate time-series, the weights are derived using a new model-adaptive elastic ESN is designed that utilizes adaptive elastic net algorithm. The proposed model proves effective when evaluated on two benchmark multivariate chaotic datasets and two real-world applications [81].

The advantages of ESNs allow computation on non-conventional hardware platforms. ESNs are used in opto-electronic, optical systems, randomly crystallized nonlinear electronic circuits [82, 83, 84, 85].

## 2.5 Purpose and Hyperparameters of the Reservoir in ESNs

The reservoir in ESNs, being an input driven dynamical system, allows obtaining the desired $y_i^{target}$. It serves as a a memory as well as a nonlinear high dimensional expansion $\mathbf{z}(n)$ of the input $\mathbf{x}(n) \in \mathcal{R}^{N_x}$, where the input becomes linearly separable in the expanded space $R^{N_z}$. The reservoir is characterized by the tuple ($\mathbf{W}^{in}$, $\mathbf{W}$, $\alpha$), where the generation of both input weight $\mathbf{W}^{in}$ and the recurrent connection weight $\mathbf{W}$ matrices is random. The global ESN parameters are: size of the reservoir $N_z$, reservoir sparsity, spectral radius $\rho$ of $\mathbf{W}$, $\mathbf{W}^{in}$ scaling, and leaking rate $\alpha$.

- **Reservoir size** or number of nodes in the reservoir $N_z$ determines the ESN's memory capacity. The larger the number of reservoir nodes the better the performance because it becomes easier to obtain linear combination of the inputs to approximate the target. However, large reservoirs are computationally expensive. The number of reservoir nodes $N_z$ is selected to be at least equal to the approximate number of values the reservoir keeps from the input to solve the task. The maximum number of stored values, called memory capacity, may not exceed $N_z$. When performing regression on a reservoir that is larger than the training output, there are fewer equations than unknowns, making the system underdetermined. This problem can be mitigated when using regularization by ridge regression or by including scaled noise to the input. When comparing various approaches, the reservoir size may be limited for convenience. Several models reported in the literature employ relatively small reservoir sizes (  40 nodes) for optimal performance [86].

- **Reservoir sparsity** is the ratio of zero elements. A reservoir weight matrix with a sparsity of 0.5, implies that half of its elements are zero. Sparsity is one of the hyperparameters that is often tuned last since its impact may be minimal. In the original ESN models, the reservoir connections are sparse, which enables fast reservoir updates. This hyperparameter is referred to as connectivity: the ratio of nonzero rather than zero elements. Connectivity determines the ratio of the connections between nodes in the reservoir out of all possible connections.

- **Spectral radius** $\rho(\mathbf{W})$ is the highest absolute value of the eigenvalues of a reservoir weight matrix. Spectral radius sets the reservoir weight matrix $\mathbf{W}$ or, alternatively, sets the width of the distribution of its nonzero elements. A reservoir with large $\rho(\mathbf{W})$ has higher memory capacity. It is usually recommended to select a $\rho(\mathbf{W})$ slightly below 1 to allow large memory while still ensuring the echo state property.

- **Input weights scaling** is used to scale large input weights and enable the reservoir to be driven more by the reservoir dynamics rather than the input. The input scaling controls the amount of the nonlinearity that may also grow with the increasing spectral radius of the reservoir. As the spectral radius controls the impact of the internal dynamics of the reservoir, both spectral radius and input weight scaling have to be adjusted so that input and reservoir have the appropriate impact on the reservoir dynamics.

- **Leaking rate** $\alpha$ is related to the reservoir's update dynamics and may be tuned by trial and error. By default, the ESN does not use a leaking rate ($\alpha = 1$). In this case, the value of a reservoir state activation is equal to reservoir's update at time step $n$ and the reservoir has no memory of its previous value. When using a leaking rate, the reservoir state keeps portion of its original value $(1 - \alpha)\mathbf{z}(n - 1)$. A new value $\alpha\widetilde{\mathbf{z}}(n)$ based on new input and the reservoir state controlled by the amount of $\alpha$ is assigned to the reservoir state update. The reservoir values change more gradually for lower $\alpha$, which induces slow dynamics of $\mathbf{z}(n)$ thus increasing the duration of the short-term network memory.

The proper selection of the hyperparameters needed to produce the reservoir are summarized in Table 2.1 [72]. The most important hyperparameters to optimize are input scaling, spectral radius ($\rho$), and leaking rate ($\alpha$). It is recommended to change hyperparameters' values one at a time [72].

## 2.6 Finding Optimal Parameters in ESNs

Finding optimal parameters in ESNs is comparably a fast process, and determining the training and validation errors is a straightforward way to evaluate the reservoir. Cross-validation is common for static non-temporal machine learning tasks. However, it is not widely used in temporal modeling due to the challenges in splitting time series for training and validation. Temporal dependencies may leak across the cuts and have to be disentangled. K-fold cross-validation might be an excess, unless the data is scarce in quantity, and obtaining training error has a benefit of using fewer data without a need to rerun the entire network. Smaller reservoir sizes and/or shorter datasets speed up the training [72].

The output (2.24) [72] may be represented in a matrix form:

Table 2.1: Key points and recommendations when selecting hyperparameters for ESN reservoir

| **Hyperparameters** | **Key points** | **Recommendations** |
|---|---|---|
| Size of the reservoir ($N_z$) | The larger number of nodes $N_z$ in reservoir, the better the performance (if proper regularization against overfitting is applied) | Select $N_z$ to be at least equal to the number of values the reservoir has to memorize from the input |
| Sparsity of the reservoir | The sparser the connections (when most elements in $\mathbf{W}^{in}$ are 0), the better the performance and faster reservoir updates | Connect each node in the reservoir to a small fixed number of other nodes, irrespective of $N_z$ |
| Distribution of nonzero elements | Nonzero element of $\mathbf{W}$ (typically sparse matrix) and $\mathbf{W}^{in}$ (typically dense matrix) have either symmetrical uniform, discrete bi-valued, or normal distribution centered around 0 | |
| Spectral radius $\rho(\mathbf{W})$ (maximal eigenvalue of $\mathbf{W}$) | Spectral radius $\rho(\mathbf{W})$ defines how fast the influence of input dies out in reservoir with time (e.g., the larger the radius, the longer the memory of the input) | Set the $\rho(\mathbf{W}) < 1$ to ensure the echo state property |
| Input scaling | Input scaling determines the amount of nonlinearity of $z(n)$ and the influence of the input on $z(n)$ as opposed to the history of the input | Normalize the data in order to keep the inputs bounded and avoid outliers (e.g., apply $tanh(\cdot)$) |
| Leaking rate ($\alpha$) | Usually small dynamics of the reservoir extends the duration of the memory in ESN | Select the leaking rate to match the speed of dynamics of the input and/or the target (usually tuned by trial and error) |

$$\mathbf{Y} = \mathbf{Z}\mathbf{W}^{out} \tag{2.27}$$

$$\mathbf{Y}^{target} = \mathbf{Z}\mathbf{W}^{out}, \tag{2.28}$$

where $\mathbf{Y}$ and $\mathbf{Y}^{target} \in \mathcal{R}^{N \times N_y}$ and $\mathbf{Z} \in \mathcal{R}^{N \times (N_z + N_x)}$ are the representation of vectors $\mathbf{y}(n)$ and $[\mathbf{z}(n); \mathbf{x}(n)]$, respectively. The matrix $\mathbf{Z}$ is used instead of $[\mathbf{X}; \mathbf{Z}]$ for simplicity. The optimal output weights $\mathbf{W}^{out}$ are obtained by minimizing the squared error between $\mathbf{Y}$ (2.27) and $\mathbf{Y}^{target}$ (2.28) by solving the overdetermined ($N >> N_z + N_x$) system of equations (2.27, 2.28). Ridge regression (2.29) is the most preferred option [72] to solve the system and learn output weights:

$$\mathbf{W}^{out} = (\mathbf{Z}^{\mathbf{T}}\mathbf{Z} + \beta\mathbf{I})^{-1}\mathbf{Z}^{T}Y^{target} \tag{2.29}$$

$$\mathbf{W}^{out} = \underset{\mathbf{W}^{out}}{\operatorname{argmin}} \frac{1}{N_y} \sum_{i=1}^{N_y} (\sum_{n=1}^{N} (y_i(n) - y_i^{target}(n))^2 + \beta \left\| w_i^{out} \right\|^2), \tag{2.30}$$

where $\mathbf{I}$ is the identity matrix and $\beta$ is regularization coefficient used to reduce the effect of overfitting, which is usually indicated by large values of the output weights. The term $\beta \left\| w_i^{out} \right\|^2$ is a regularization term that introduces weight decay and penalizes large values of $\mathbf{W}^{out}$, with $|| \cdot ||$ standing for Eucleadian norm. Including scaled white noise to the input serves a similar purpose as regularization.

The dimensions of the matrices $(\mathbf{Z}^{T}\mathbf{Y}^{target}) \in \mathcal{R}^{N_z \times N_y}$ and $\mathbf{Z}^{\mathbf{T}}\mathbf{Z} \in \mathcal{R}^{N_z \times N_z}$ do not depend on the size of the training set $N$, which allows training with nearly unlimited data size. The size of data $N$ also does not influence the training time and memory. The matrices may be easily updated by adding the results with the new data.

If $\mathbf{Z}^{\mathbf{T}}\mathbf{Z}$ is invertible, (2.30) becomes:

$$\mathbf{W}^{out} = \mathbf{Z}^{\dagger}\mathbf{Y}^{target}. \tag{2.31}$$

It is also a solution to (2.21), where $\mathbf{Z}^{\dagger}$ is Moore-Penrose pseudoinverse. Even though this solution shows high stability and requires little or no regularization, its disadvantage is that it requires sizeable memory for large design matrices $(\mathbf{Z}^{T}\mathbf{Z})^{-1}\mathbf{Z}^{T}$ thus restraining the size of the reservoir $N_z$ and/or number of training samples. The pseudoinverse solution is recommended, time and memory permitting.

For classification tasks, a model is trained to decide a class for an input sequence given a $y^{target}(n)$ that is equal to 1 for the class of interest and 0 otherwise. The class is decided by:

$$\text{class } \mathbf{x}(n) = \operatorname*{argmax}_k \left( \frac{1}{|\tau|} \sum_{n \in \tau} y_k(n) \right) = \operatorname*{argmax}_k \left( \left( \sum \mathbf{y} \right)_k \right), \tag{2.32}$$

where $y_k(n)$ is the element of the $k^{th}$ column of $\mathbf{y}(n)$ and $\tau$ is an integration interval. The $\sum \mathbf{y}$ is equal to $\mathbf{y}(n)$ time-averaged over $\tau$:

$$\begin{aligned} \sum \mathbf{y} = \frac{1}{|\tau|} \sum_{n \in \tau} \mathbf{y}(n) = \frac{1}{|\tau|} \sum_{n \in \tau} [\mathbf{x}(n); \mathbf{z}(n)] \mathbf{W}^{out} = \\ \mathbf{W}^{out} \frac{1}{|\tau|} \sum_{n \in \tau} [\mathbf{x}(n); \mathbf{z}(n)] = \mathbf{W}^{out} \sum \mathbf{z}. \end{aligned} \tag{2.33}$$

Equation (2.33) is an effective way to find $\sum \mathbf{y}$ because it involves only one multiplication with $\mathbf{W}^{out}$. For a given input sequence, it is more convenient to find one $\mathbf{W}^{out}$ that minimizes the error between $E(\mathbf{y}^{target}, \sum \mathbf{y})$ instead of calculating $\mathbf{W}^{out}$ for every $n \in \tau$ that minimizes $E(\mathbf{y}^{target}(n), \mathbf{y}(n))$.

## 2.7 Output Feedbacks in ESNs

The dynamics of training process is varied as the trained output is fed back to the reservoir creating recurrence between the output and reservoir [72]. The update equation with the looping output $y(n-1)$ for the next update step is:

$$\widetilde{\mathbf{z}}(n) = tanh([\mathbf{x}(n)]\mathbf{W}^{in} + \mathbf{z}(n-1)\mathbf{W} + \mathbf{y}(n-1)\mathbf{W}^{fb}). \tag{2.34}$$

Input $x(n)$ and input weights $\mathbf{W}^{in}$ are virtually equivalent to $y(n-1)$ and $\mathbf{W}^{fb}$. However, because of stability issues, feedback should be utilized when learning is otherwise not possible. When training output with feedbacks, two methods may be followed:

- Target forcing or breaking the feedback loop during the learning process (feeding $y^{target}(n)$ instead of $y(n)$);

- Adapting $\mathbf{W}^{out}$ online in the presence of a feedback.

Chapter 3

# CIC-IDS Datasets: CIC-IDS2017, CSE-CIC-IDS2018, and CIC-DDoS2019

The performance of network intrusion detection models relies on datasets that are public, labeled, and contain diverse traffic. With changing network behavior and intrusion patterns, it is challenging to identify datasets that reflect the current network trends.

The Canadian Institute for Cybersecurity (CIC) has devised testbed framework [87, 88, 89] to generate CIC-IDS2017 [90], CSE-CIC-IDS2018 [91] (in collaboration with the Communications Security Establishment (CSE)), and CIC-DDoS2019 [92] network traffic datasets. Two profile classes are used: B-Profile and M-Profile. The profiles are employed with various protocols and network topologies. Features such as certain patterns in payload and its size, number of packets per flow, and request time distribution are extracted from client traffic into individual profiles. They are clustered into B-Profiles. These B-profiles are groups of users with similar behavior representing regular (benign) background traffic. For these datasets, B-Profiles incorporate the abstract behavior of 25 users based on the most frequently used protocols such as HTTP, HTTPS, FTP, SSH, SMTP, POP3, and IMAP. M-Profiles generate attacks scenarios: infiltration of the network, denial of service, brute force, and a variety of web application attacks.

Extraction of more than 80 features including duration, size of packets, and number of packets was performed using CICFlowMeter, an application written in Java for generating and analyzing network traffic flows [93]. The application generates bi-directional flows where the first packet defines the forward or backward directions. Hence, the features are calculated for both directions. CICFlowMeter analyzes network traffic and labels the flows based on time stamp, source and destination IP addresses and ports, protocols, and type of attacks.

## 3.1 CIC-IDS2017 Dataset

The CIC-IDS2017 testbed includes an attacker and victim networks. The attacker network consists of one router, one switch, and four terminals with Kali Linux and Windows 8.1 operating systems. The victim network consists of three servers, one firewall, two switches, and ten terminals interconnected by a security authentication server. One switch in the victim-network serves as a mirror port and captures the incoming and outgoing traffic.

The CIC-IDS2017 dataset includes intrusions that rely on various network vulnerabilities [87] and are executed using attack tools: Patator, Slowloris, Heartleech, Damn Vulnerable Web App, Metasploit, Ares, and Low Orbit Ion Cannon.

The data capture began with regular traffic at 9:00 on Monday July 3, 2017. Intrusion attacks were initiated at 9:20 on Tuesday, July 4, 2017 and ended at 17:00 on Friday, July 7, 2017 [90, 93]. The 2,830,743 data points collected during five days include regular traffic and attacks.

We use DoS data collected on Wednesday, July 5, 2017 with the average packet size shown in Fig. 3.1. Malicious data points are labeled GoldenEye, Hulk, SlowHTTPTest, and

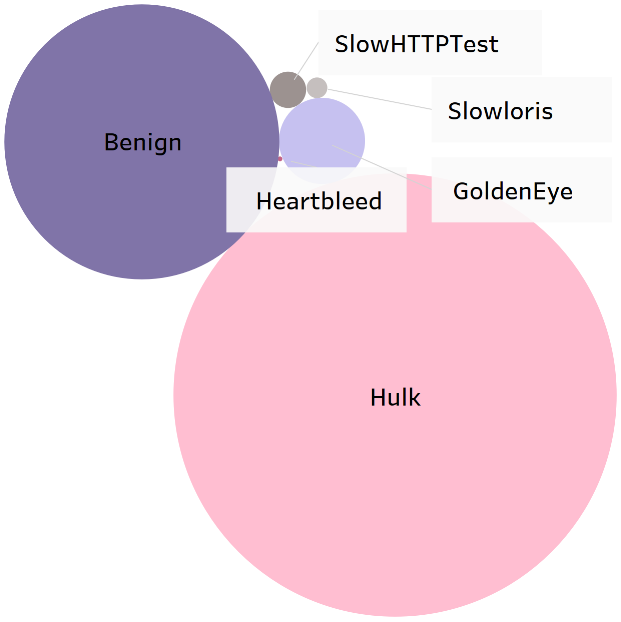


Figure 3.1: CIC-IDS2017: Average packet size. Color indicates flow labels. Size shows average packet size. Hulk attack has the largest average packet size.

Slowloris having 10,293, 230,124, 5,499, and 5,796 intrusions, respectively. The number of regular (benign) and malicious packets is shown in Fig. 3.2.

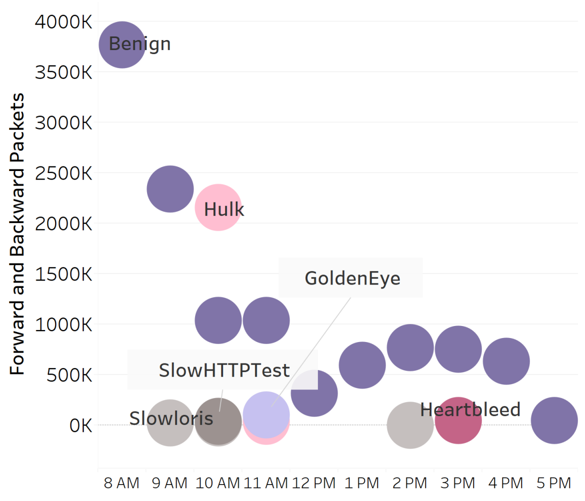


Figure 3.2: CIC-IDS2017: Number of forward and backward packets.

## 3.2 CSE-CIC-IDS2018 Dataset

The CSE-CIC-IDS2018 testbed consists of victim (420 terminals and 30 servers split into 5 subsets) and attacker networks (50 terminals) implemented using Amazon Web Services. Ubuntu, Windows 8.1, and Windows 10 operating systems are installed on host machines while servers use Windows 2012 and Windows 2016.

The CSE-CIC-IDS2018 dataset captures ten days between Wednesday, February 14, 2018 and Friday, March 2, 2018 [91]. There are 6.1 million rows, each consisting of a bidirectional flow (biflow) representing the source-to-destination packets for nine attack types. Malicious (attack) data contain seven most common attack scenarios: botnet, brute-force, DoS, DDoS, heartbleed, network infiltration, and web attacks.

In this Thesis, we considered the GoldenEye, Hulk, SlowHTTPTest, and Slowloris DoS attacks collected on Thursday, February 15, 2018 and Friday, February 16, 2018. The content of the dataset collected during these days is shown in Fig. 3.3. Average size of packets is shown in Fig. 3.4.

## 3.3 CIC-DDoS2019 Dataset

The CIC-DDoS2019 testbed contains victim and attack networks. The victim network includes one Ubuntu 16 webserver, four personal computers with Windows 7, Windows Vista, Windows 8.1, and Windows 10 operating systems, one Fortinet firewall, and two switches. The attack network employs a third party infrastructure and utilizes tools and packages necessary to initiate 12 types of DDoS attacks.

The set, collected on January 12, 2019 from 10:30 to 17:15, includes 12 attacks: NTP, DNS, LDAP, MSSQL, NetBIOS, SNMP, SSDP, UDP, UDP-Lag, WebDDoS, SYN, and TFTP. The set, collected on March 11, 2019 from 9:40 to 17:35, includes 7 attacks: PortScan,

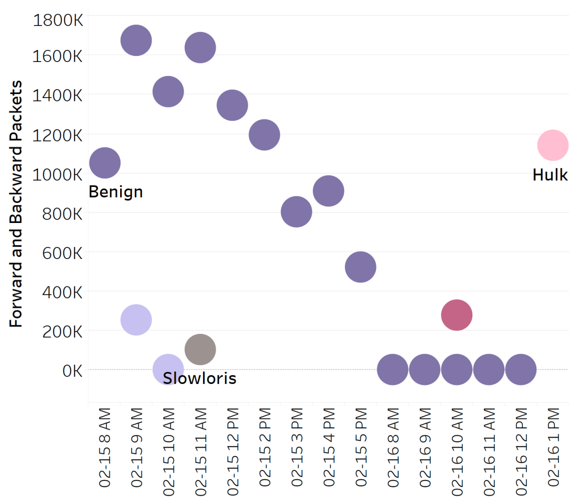


Figure 3.3: CSE-CIC-IDS2018: Number of forward and backward packets.

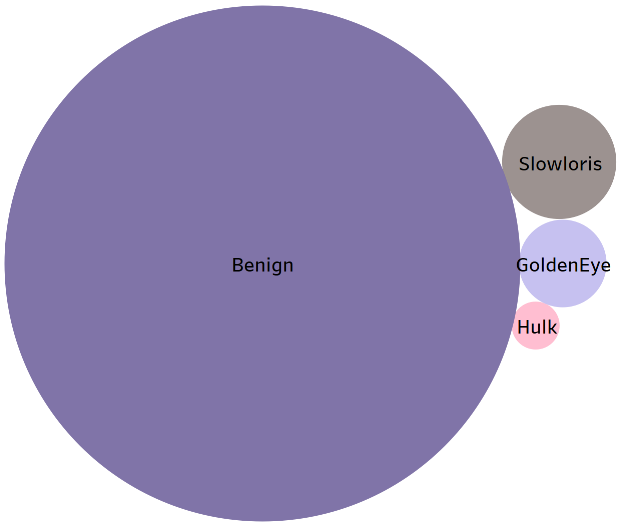


Figure 3.4: CSE-CIC-IDS2018: Average packet sizes. Color indicates flow labels. Size shows average packet sizes. Benign packets have the largest average packet size.

NetBIOS, LDAP, MSSQL, UDP, UDPLag, and SYN. In this Thesis, we use LDAP, NTP, SYN, UDP-lag, and WebDDoS attacks collected on January 12, 2019, 9:00–13:00.

The content and the collection time of the selected set of regular and malicious traffic are shown in Fig. 3.5 with the average packets size shown in Fig. 3.6.

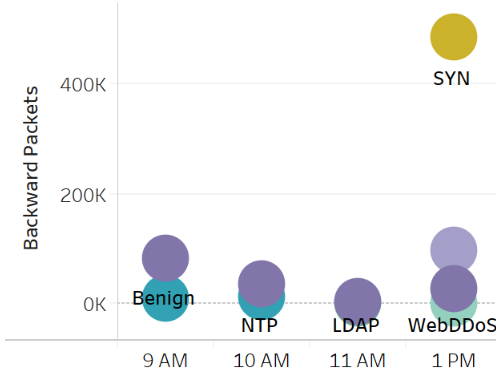


Figure 3.5: CIC-DDoS2019: Number of forward and backward packets.

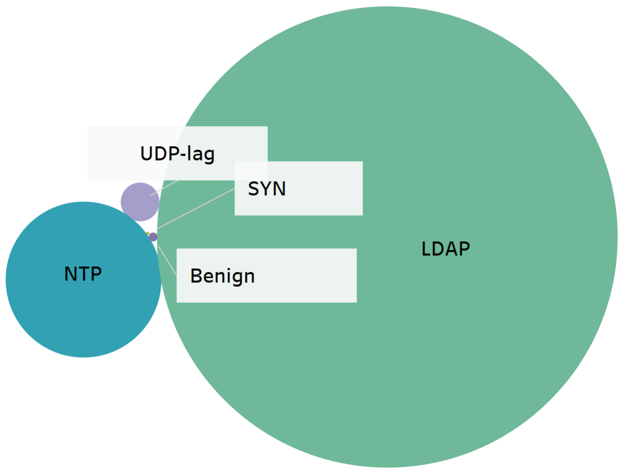


Figure 3.6: CIC-DDoS2019: Average packet size. Colors indicate flow labels. Size of the circles shows the average size of packets. LDAP has the largest packet size.

## 3.4 Attacks and Features

The attacks considered in this Thesis as well as the features related to these attacks are described in this Subsection. We considered CIC-IDS2017 and CSE-CIC-IDS2018 that include: GoldenEye, HULK, SlowHTTPTest, and Slowloris attacks. LDAP, NTP, SYN, UDP-lag, and WebDDoS are the attacks encountered in the CIC-DDoS2019 dataset.

*GoldenEye* is a Python-based application for security testing that may be also used for malicious activities. GoldenEye creates application-layer attacks. Their main goal is to occupy all available HTTP/HTTPS sockets by sending *keep alive* requests. The time that it takes to successfully start a GoldenEye DDoS attack is 6.2 ms compared to Slowloris that may be launched in 100 ms [117].

*HULK (HTTP Unbearable Load King)* creates high volumes of obscure traffic that brings down web servers with voluminous requests from a number of random user agents.

*SlowHTTPTest* is a tool for implementing most common application-layer DoS attacks such as Slow HTTP Post, Slowloris, and Slow Read. HTTP is designed to wait for complete requests before they are processed. Exploiting this feature, application-layer DoS attacks drain the concurrent connections pool by sending incomplete requests.

*Slowloris* is a slow rate and low volume traffic generation from a single source that makes it difficult to detect using traditional Network Intrusion Detection Systems' (NIDS) approaches. This attack occurs at the application-layer and is characterized by sending fractional HTTP GET requests without termination code in order to keep the connection open. Hence, the server is forced to wait indefinitely for the remaining data. By sending HTTP headers one by one, attacker keeps the session alive overflowing buffer with live sessions and resulting in busy resources to establish new legitimate connection requests from other users.

*Lightweight Directory Access Protocol (LDAP)* is a widely used, open, vendor-neutral, industry standard application protocol for accessing and supporting distributed directory information services over an IP network [118]. LDAP allows an access to organized set of records such as email directory, information about users, systems, and services [119]. LDAP Amplification attacks utilize this protocol by sending requests to a publicly available vulnerable LDAP server with open TCP port 389. This triggers amplified replies approximately 50 times larger than the initial small queries. The amplified replies are reflected to a target server.

*Network Time Protocol (NTP)* is one of the oldest currently used Internet protocols that provides clock synchronization between computer systems over packet-switched, variable-latency data networks [120]. NTP amplification attack allows an attacker to use spoofed IP address of the victim's NTP infrastructure and send small NTP queries to the Internet servers that, in turn, generate and reflect amplified NTP responses.

*SYN* flood, often created via distributed botnet, overwhelms available resources of the target systems. It may affect firewalls or other defense components of a target. SYN packets are sent to victim at a very high rate using up the TCP connection pool that causes dropping of legitimate packets. Approximately 80% of all DDoS attacks in 2018 were SYN flood attacks [9].

*UDP-lag* attack aims to disrupt the connection between the client and the server. An example is online gaming where the players wish to slow down or interrupt their opponents.

The attack may be initiated either via a hardware switch known as a lag switch, or by a software program that allows monopolization of bandwidth [92].

*Web DoS* attacks may be divided into three categories. In type-I attacks, the same page is repeatedly being requested. This may be detected using thresholds that prevent sources from requesting the same page more than N times per time period. In type-II attacks, random web pages are being requested. In type-III attacks, an attacker scans websites and creates navigation patterns that resemble genuine human web surfers [92]. Both, type-II and type-III web DoS attacks are more difficult to detect and block compared to type-I attacks.

Packet length features such as the sizes of header, trailer, and payload are valuable in detecting volumetric nature of amplification DDoS attacks (floods) or monopolizing nature of application layer attacks. Regular packets are often under 1,000 bytes in length as shown in Figs. 3.7, A.1, and A.2. Note that Heartbleed attack packets reach on average approximately 15,000 bytes.

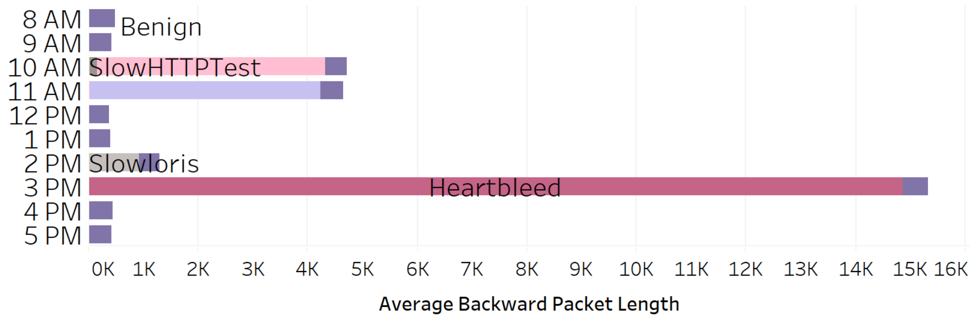


Figure 3.7: CIC-IDS2017: Average backward packet length. Heartbleed, GoldenEye, and Hulk have larger packet length compared to benign traffic.

Standard deviation of packet length helps differentiate regular from anomalous traffic because regular traffic exhibits high variation in packet length, as shown in Fig. 3.8. For example, regular packets in CIC-IDS2017 dataset have high variation of packet length, as shown in Fig. 3.8. The length of malicious packets in the TCP state exhaustion attacks such as SYN and ICMP packets is often small. Moreover, the attackers often generate fixed-sized packets. Therefore, the minimum average segment size in a malicious flow may be smaller than in regular flows. In CSE-CIC-IDS2018 datasets, standard deviation of packet length in case of benign packets is high most of the time, as shown in Fig. A.3. This is easier observed if 9 AM and 11 AM benign traffic is excluded from the graph. Slowloris, however, shows comparably high standard deviation. Malicious packets may have high length variation, as in the case of NTP attack shown in Fig. A.4. In such cases, machine learning models may not be able to learn from this feature in order to distinguish benign and malicious traffic.

Features related to flow interarrival time (IAT) such as minimum/maxium/mean interarrival time and flow duration, help detect the DDoS attacks such as DDoS UDP-lag and

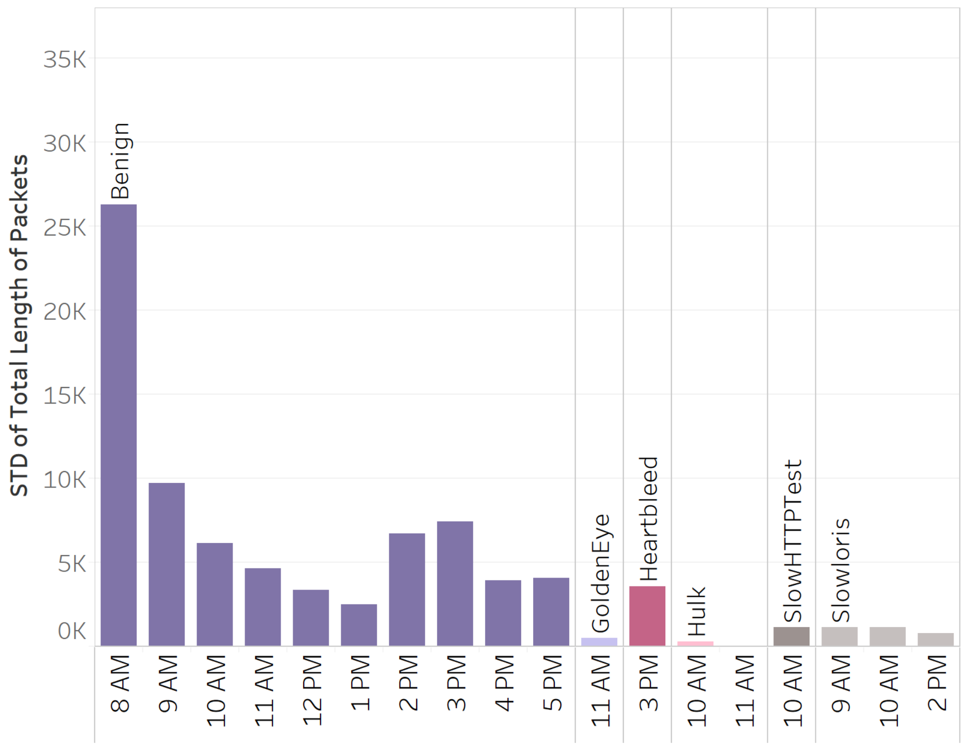


Figure 3.8: CIC-IDS2017: Standard deviation of packet length. Benign packets usually have high variation in length.

DDoS-NTP that cause bursty behavior. Average flow IAT is the highest for GoldenEye, Hulk, SlowHTTPTest, and Slowloris attacks in CIC-IDS2017, as shown in Fig. 3.9. While GoldenEye and Slowloris keep comparable IAT as in 2017, Hulk and SlowHTTPTest have lower flow IATs in CSE-CIC-IDS2018 as shown in Fig. A.5. Application layer DoS attacks exhibit longer flow duration as shown in Fig. 3.10.

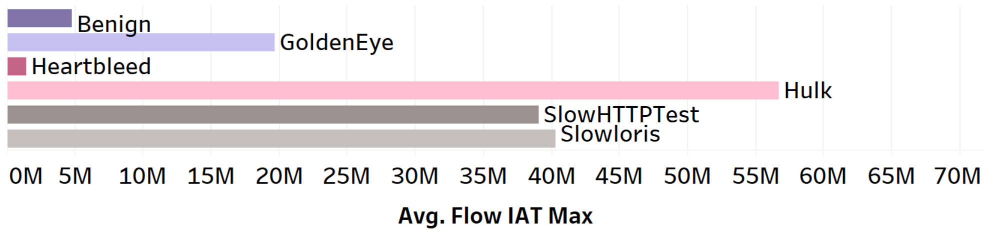


Figure 3.9: CIC-IDS2017: Average flow interarrival time (IAT) values are the highest for GoldenEye, Hulk, SlowHTTPTest, and Slowloris.

In TCP, flags (control bits) indicate a particular state of connection and provide useful information. Most commonly used flags are ACK, SYN, URG, RST, PSH, and FIN where each flag can be set to 1 (on) or 0 (off). They are often used by attackers to disrupt the target's normal operation. Therefore, "ACK Flag Count", "SYN Flag Count", and "URG Flag Count" shown in Fig. 3.11, may be important features that help detect malicious

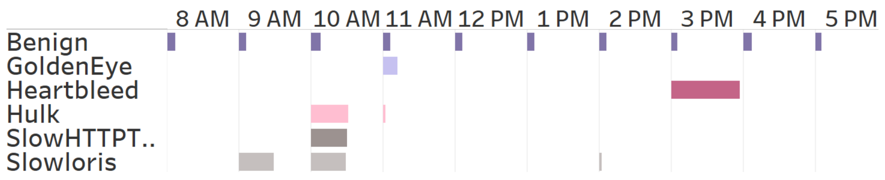


Figure 3.10: CIC-IDS2017: Average duration of collected flows.

traffic. An attacker may use the control bits by overwhelming the server with TCP ACK

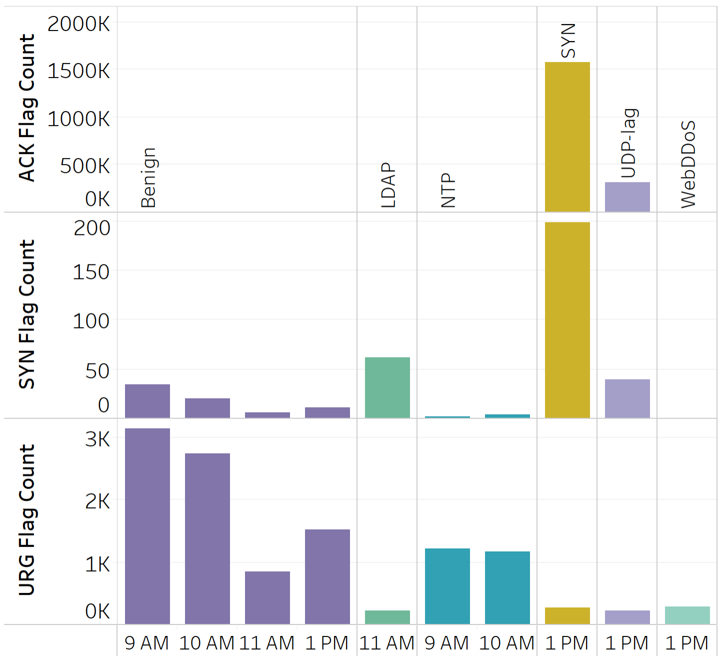


Figure 3.11: CIC-DDoS2019: ACK, SYN, and URG flag counts. SYN attacker may bring down a network connection by requesting seemingly legitimate connections through a series of TCP requests with TCP SYN and ACK flags set to 1.

packets as shown in Fig. A.6, where Hulk attack employs a large number of packets with ACK set to 1.

SYN attacker may bring down a network connection by requesting seemingly legitimate connections through a series of TCP requests with TCP flags SYN and SYN ACK set to 1, as shown in Fig. 3.11. By using up available connections, the server will not be able to respond to legitimate connection requests.

# Chapter 4

# Border Gateway Protocol Datasets

## 4.1 Border Gateway Protocol

Border Gateway Protocol (BGP) is a routing protocol that plays an essential role in forwarding Internet Protocol traffic between the source and destination Autonomous Systems (ASes) [94]. Autonomous System (AS) is a group consisting of one or more independent networks of BGP peers (neighbors) with uniform routing policies enforced by a single network administrative domain [95]. BGP enables ASes to exchange reachability information with neighboring ASes and propagate the information about the availability of routes within an AS thus allowing all sub-networks to be interconnected and known to the Internet. BGP routers exchange four types of messages: open, update, keep-alive, and notification. BGP is an incremental protocol that sends updates only if there are reachability or topology changes within the network. Therefore, only updates regarding new prefixes or withdrawals of the existing prefixes are exchanged [94].

Transmission of the BGP routing information is susceptible to various cyber threats including worms, malicious attacks, power outages, blackouts, and misconfigurations of BGP routers. These threats affect the Internet servers and hosts and are manifested by anomalous traffic behavior. These events may spread false routing information throughout the Internet by either dropping packets or directing traffic through unauthorized ASes [96].

## 4.2 Border Gateway Protocol Data Collections

BGP routing information is shared by Internet service providers (ISPs) located worldwide [97]. BGP trace collectors obtain and store the routing tables into publicly available archives. Routing tables contain entries from each peering AS that indicate the preferred paths to destination prefixes at a given time. BGP routing update messages are available from global BGP monitoring systems such as RIPE [98] and Route Views [99]. The Internet routing data used in this Thesis contain BGP anomalous events: Slammer [100], Nimda [101], and Code Red I [102] were acquired from the Routing Information Service

(RIS) project. The Internet routing data that contain BGP anomalous events: AWS (Amazon Web Services) DDoS attacks, used in this Thesis, were acquired from the Routing Information Service (RIS) and Route Views projects.

Routing Information Service (RIS) is the project originated in 2001 by the Réseaux IP Européens (RIPE) Network Coordination Centre (NCC) [98] with the main goal to collect and store chronological routing data offering a unique view of the Internet topology. The RIPE BGP update messages are publicly available to the research community. Data were exported every fifteen minutes until July 2003. The interval between consecutive exports was later increased to five minutes.

Route Views [99] is the University of Oregon project that aims to store real-time BGP routing data from backbone routers located in various geographical locations since 1997. The publicly available data have been used for routing analysis, AS path visualization, analysis of IPv4 address space utilization, topological studies, and generation of geographic host locations. Backbone routers (Cisco, Juniper), configured as IPv4 or IPv6 Route-Views-like route servers, connect as peers via multi-hop BGP sessions. Route Views project employs three types of collectors: FRR, Quagga, and Cisco.

BGP update messages are collected and stored in the multi-threaded routing toolkit (MRT) binary format [103]. The BGP update messages are converted from MRT into American Standard Code for Information Interchange (ASCII) format by using the *zebra-dump-parser* [104]. A C# tool [105] is used to generate datasets by extracting 37 numerical features from BGP update messages.

## 4.3 Anomalies and Features

Considered BGP datasets contain worms that may cause DoS and DDoS. We consider Slammer [100], Nimda [101], and Code Red I [102] worms obtained from collector rrc04 located at at CIXP, Geneva that stores route updates since 2001. These anomalies caused increases in the number of announcement and withdrawal messages exchanged by the Border Gateway Protocol (BGP) routers. The examples of the collected traces of Slammer, Nimda, and Code Red I worms are shown in Figs. 4.1–A.10. Details including the period and duration of these anomalous events are given in Table 4.1.

Table 4.1: Examples of BGP Internet worms

| **Dataset** | **Class** | **Beginning of the event** | **Duration (min)** |
|---|---|---|---|
| Slammer | Anomaly | 25.01.2003 at 5:31 GMT | 869 |
| Nimda | Anomaly | 18.09.2001 at 13:19 GMT | 1,301 |
| Code Red I | Anomaly | 19.07.2001 at 13:20 GMT | 600 |

*Slammer*: The Structured Query Language (SQL) Slammer worm attacked Microsoft SQL servers on January 25, 2003 [100]. Microsoft SQL servers were infected through a small piece of code that generated IP addresses at random. Furthermore, code replicated itself by infecting new machines through randomly generated targets. If the destination IP address was a Microsoft SQL server or a user's PC with the Microsoft SQL Server Data Engine (MSDE) installed, the server became infected and began infecting other servers. The number of infected machines doubled approximately every nine seconds. As a result, the update messages consumed most of the routers' bandwidth, which in turn slowed down the routers and, in some cases, caused the routers to crash.

*Nimda*: The Nimda worm exploited vulnerabilities in the Microsoft Internet Information Services (IIS) web servers for the Internet Explorer 5 on September 18, 2001 [101]. It propagated fast through email messages, web browsers, and file systems. The worm propagated by sending an infected attachment that was automatically downloaded after viewing the email messages. A user could also download the worm from the website or access an infected file through the network. The worm modified the content of the web document file in the infected hosts and copied itself in local host directories.

*Code Red I*: Although the Code Red I worm attacked Microsoft IIS web servers earlier, the peak of infected computers was observed on July 19, 2001. The worm affected approximately half a million IP addresses a day [102]. It took advantage of vulnerability in the Internet Information Services (IIS) indexing software. The worm replicated itself by exploiting weakness of the IIS servers and, unlike the Slammer worm, Code Red I searched for vulnerable servers to infect. It triggered a buffer overflow in the infected hosts by writing to the buffers without checking their limits. Rate of infection was doubling every 37 minutes.

We also consider the BGP data collected from RIPE rrc14 collector over periods of largest DDoS attacks that affected Amazon Web Services (AWS) in 2019 (DDoS2019-v1 and DDoS2019-v2) and in 2020 (DDoS2020). BGP data for AWS 2019 and 2020 attacks were also collected from Route Views route-views4 collector (Quagga type). The RIPE collector rrc14 is located in Palo Alto, USA. The Route Views collector route-views4 is located in Eugene, Oregon, USA. Details including the period and duration of the events are shown in Table 4.2.

Table 4.2: Examples of major DDoS attacks

| **Dataset** | **Class** | **Beginning of the event** | **Duration of the event** |
|---|---|---|---|
| DDoS2019-v1 | Anomaly | 22.10.2019 | 8 hours |
| DDoS2019-v2 | Anomaly | 22.10.2019 | 61.5 hours |
| DDoS2020 | Anomaly | 17.02.2020 | 3 days |

*October 2019 DDoS Attack on AWS (DDoS2019):* AWS experienced a DDoS attack on October 22, 2019 that caused an eight hour outage and interrupted services [107]. The attack affected the Amazon route 53 DNS webservice leaving thousands of customers unable to access cloud services, websites, and applications [108]. After the malicious DNS query hit the AWS network, approximately 50% of packets were dropped. The attack lasted from 10:30 AM to 6:30 PM PDT and was persistent in San Francisco and intermittent in Boston,

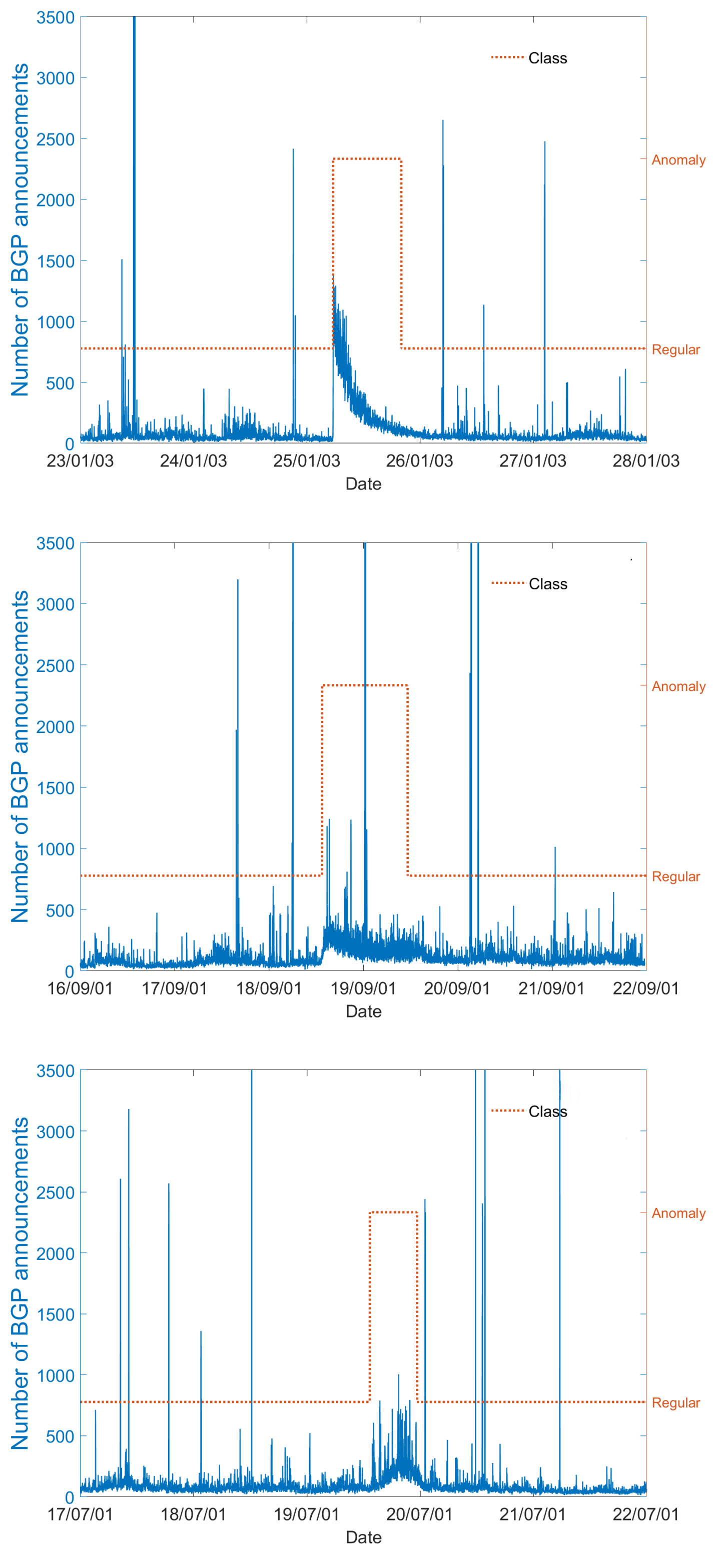


Figure 4.1: Slammer (top), Nimda (middle), and Code Red I (bottom): Number of BGP announcements. The red dotted line indicates two classes: regular and anomaly.

Chicago, and Dallas. AWS Shield [109] attempted to mitigate the malicious traffic affecting legitimate traffic, which caused further DNS failures and congestion due to the increased traffic. Meanwhile, on October 23, 2019, a wave of ransom driven DDoS attacks hit the banking industry in South Africa that left Johannesburg's emergency call centers and e-services, including online banking and billing system inaccessible to customers. The attack was launched with a ransom note (data that are encrypted and locked down by malware) that was delivered to staff email addresses. The attackers were demanding four Bitcoins (equivalent to $37,000 USD) threatening to upload the hacked data online. However, no private data breach occurred and DDoS attack only caused increased network traffic and service disruptions [110]. This attack may have impacted the BGP update messages collected for AWS DDoS attack that occurred on October 22, 2019 (DDoS2019-v1), as shown in Fig. A.11. We used BGP data collected from RIPE (Fig. 4.2) and from Route Views (Fig. 4.3) for AWS DDoS attack on October 22, 2019 (DDoS2019-v2) and DDoS attack on October 23, 2019 (DDoS2019-v2).

*February 2020 DDoS Attack on AWS:* The largest ever DDoS attack of 2.3 Tbps occurred on February 17, 2020 and caused three days of elevated threat [111]. It was a Connectionless Lightweight Directory Access Protocol (CLDAP) reflection attack that targeted Amazon cloud web services. Both CLDAP and its older alternative LDAP are widely used protocols for authenticating username and password information. According to AWS Shield report, this attack was 44% larger than any network volumetric event previously detected on AWS [111]. In such attacks, an attacker sends a CLDAP request to a LDAP server with a spoofed sender IP address that is the target's IP address. The server prepares a bulked-up response to the target's IP address starting the reflection attack. The goal is to flood the target with a massive amount of data to disrupt normal traffic, leaving unresponsive the web services hosted on the server. The server, unaware of the attack, receives multiple seemingly legitimate requests to establish communication and replies with a SYN-ACK causing the server's connection tables to fill and, thus, deny access to legitimate users [112]. The numbers of BGP announcements and announced Network Layer Reachability Information (NLRI) prefixes of a dataset collected from RIPE are shown in Fig. 4.4. Even though the attack lasted until February 20, 2020, we are able to see high occurrences of the BGP updates starting February 21, 2020, which may influence the training of a machine learning model. The BGP data for the same anomaly collected from Route Views are shown in Fig. 4.5.

GMT (UTC) time is used for all update messages in order to synchronize RIPE and Route Views collection times. While the available datasets contain data over much longer periods of time, we have selected for our analysis five-, six-, and seven-day periods to minimize storage and computational requirements. Furthermore, selecting longer periods of regular data would make datasets even more unbalanced. The AS-path is a BGP update message attribute that enables the protocol to select the best path for routing packets. It indicates a path that a packet may traverse to reach its destination. If a feature is

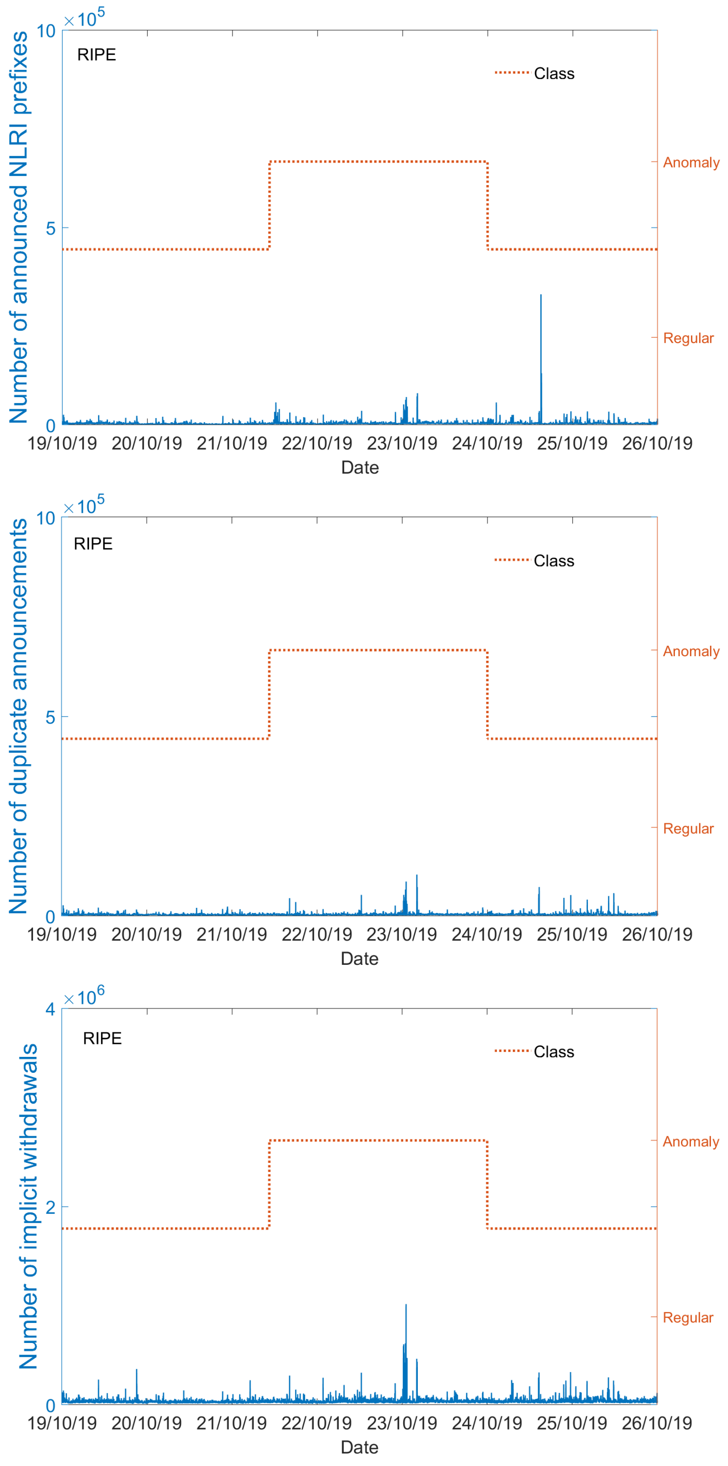


Figure 4.2: DDoS2019-v2 collected from RIPE: Number of announced NLRI prefixes (top), number of duplicate announcements (middle), and number of implicit withdrawals (bottom). The red dotted line indicates two classes: regular and anomaly.

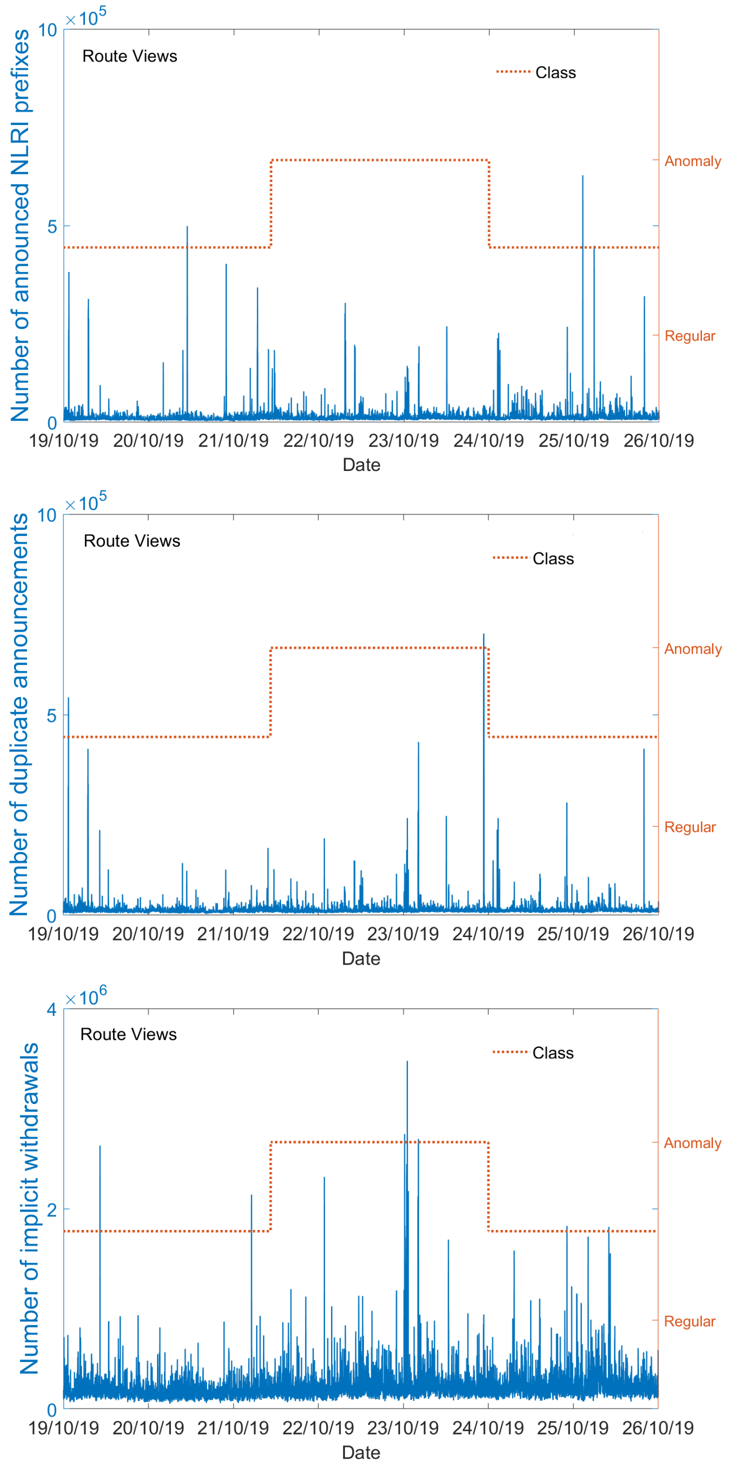
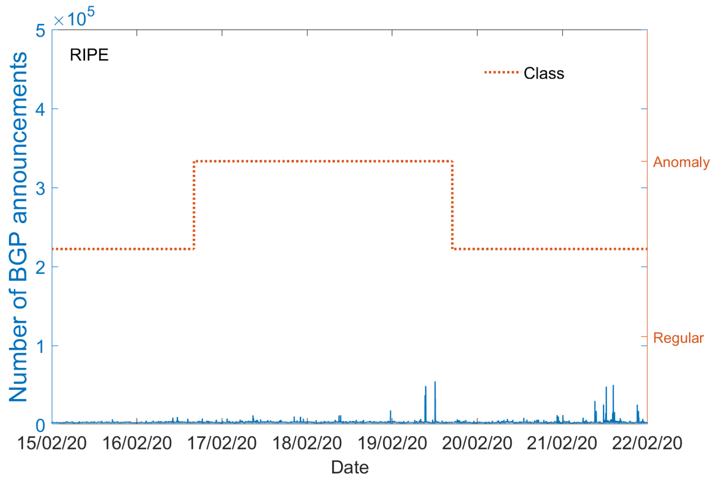


Figure 4.3: DDoS2019-v2 collected from Route Views: Number of announced NLRI prefixes (top), number of duplicate announcements (middle), and number of implicit withdrawals (bottom). The red dotted line indicates two classes: regular and anomaly.



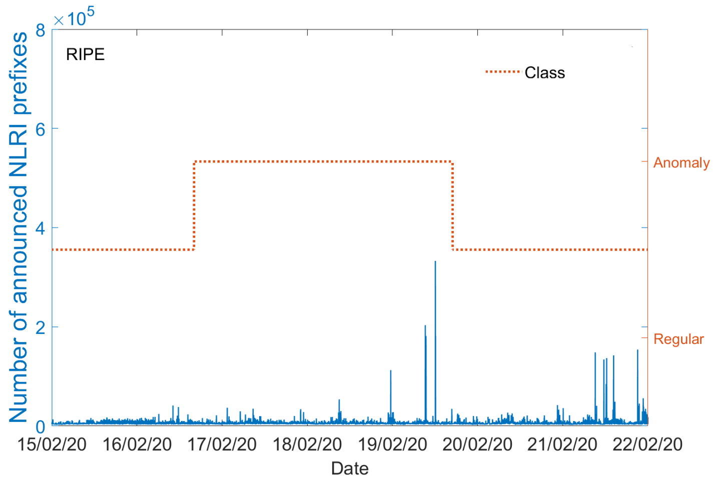


Figure 4.4: DDoS2020 collected from RIPE: Number of BGP announcements (top) and number of announced NLRI prefixes (bottom). The red dotted line indicates two classes: regular and anomaly.

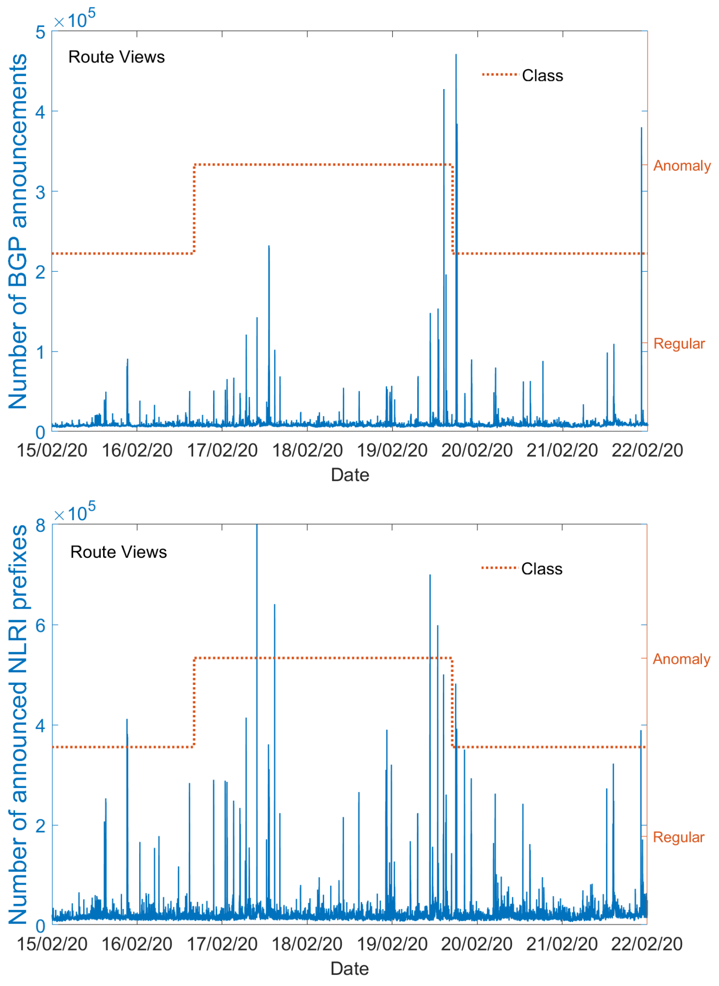


Figure 4.5: DDoS2020 collected from Route Views: Number of BGP announcements (top) and number of announced NLRI prefixes (bottom). The red dotted line indicates two classes: regular and anomaly.

derived from the AS-path attribute, it is categorized as an AS-path feature. Otherwise, it is categorized as a volume feature. There are three types of features: continuous, categorical, and binary. Extracted AS-path and volume features are shown in Table 4.3. Definitions of the extracted features are listed in Table 4.4.

Table 4.3: List of features extracted from BGP update messages

| **Feature** | **Name** | **Category** |
|---|---|---|
| 1 | Number of announcements | *volume* |
| 2 | Number of withdrawals | *volume* |
| 3 | Number of announced NLRI prefixes | *volume* |
| 4 | Number of withdrawn NLRI prefixes | *volume* |
| 5 | Average *AS-path* length | *AS-path* |
| 6 | Maximum *AS-path* length | *AS-path* |
| 7 | Average unique *AS-path* length | *AS-path* |
| 8 | Number of duplicate announcements | *volume* |
| 9 | Number of implicit withdrawals | *volume* |
| 10 | Number of duplicate withdrawals | *volume* |
| 11 | Maximum edit distance | *AS-path* |
| 12 | Arrival rate | *AS-path* |
| 13 | Average edit distance | *volume* |
| 14 – 23 | Maximum *AS-path* length, where $n = (11, ..., 20)$ | *AS-path* |
| 24 – 33 | Maximum edit distance = $n$, where $n = (7, ..., 16)$ | *AS-path* |
| 34 | Number of Interior Gateway Protocol (IGP) packets | *volume* |
| 35 | Number of Exterior Gateway Protocol (EGP) packets | *volume* |
| 36 | Number of incomplete packets | *volume* |
| 37 | Packet size (B) | *volume* |

Table 4.4: Definition of *volume* and *AS-path* features extracted from BGP update messages

| **Feature** | **Name** | **Definition** |
|---|---|---|
| 1 | Number of announcements | Routes available for delivery of data |
| 2 | Number of withdrawals | Routes no longer reachable |
| 3/4 | Number of announced/withdrawn NLRI prefixes | BGP update messages that have type field set to announcement/withdrawal |
| 5/6/7 | Average/maximum/average unique *AS-path* length | Features related to *AS-path* |
| 8/10 | Number of duplicate announcements/withdrawals | Duplicate BGP update messages with type field set to announcement/withdrawal |
| 9 | Number of implicit withdrawals | BGP update messages with type field set to announcement and different AS-path attribute for already announced NLRI prefixes |
| 11/13 | Average/maximum edit distance | Average/maximum of edit distances of messages |
| 34/35/36 | Number of IGP, EGP, or incomplete packets | BGP update messages generated by IGP, EGP, or unknown sources |

BGP update messages are either announcement or withdrawal messages for the NLRI prefixes. The NLRI prefixes that have identical BGP attributes are encapsulated and sent in a single BGP packet [113]. Hence, a BGP packet may contain more than one announced or withdrawn NLRI prefix. The average and the maximum number of AS peers are used for calculating AS-path lengths. Large length of the AS-path BGP attribute implies that the packet is routed to its destination via a longer path, which causes large routing delays during BGP anomalies. Duplicate announcements are the BGP update packets that have identical NLRI prefixes and the AS-path attributes. Conversely, a duplicate announcement is a redundant prefix advertisement with the same attributes as the most recent update for this prefix in the same session and not interleaved with a withdrawal or a session reset. Implicit withdrawals are prefixes that have been implicitly withdrawn by sending the same prefix with new attributes [114]. The edit distance is a metric to quantify the similarity of strings. A router uses edit distance to measure the difference between two AS paths. The edit distance between two AS-path attributes is the minimum number of deletions, insertions, or substitutions that need to be executed to match the two attributes [115]. The more frequent changes in an AS path, the larger is the edit distance, which makes the routing update less trustworthy [116]. During the Slammer, Nimda, Code Red I events, the paths change frequently, as shown in Fig. A.9. The maximum AS-path length and the maximum edit distance are used to count Features 14 to 33. We also consider Features 34, 35, and 36 based on distinct values of the origin attribute that specifies the origin of a BGP update packet and may assume three values: IGP, EGP, and incomplete. Even though the EGP protocol is the predecessor of BGP, EGP packets still appear in traffic traces containing BGP updates messages. Under a worm attack, BGP traces contained large number of EGP packets. Furthermore, incomplete update messages imply that the announced NLRI prefixes are generated from unknown sources. They usually originate from BGP redistribution configurations [113]. Examples of the impact of BGP anomalies on features, such as number of BGP announcements, number of announced NLRI prefixes, edit distances, duplicate announcements, implicit withdrawals, number of EGP packets are shown in Figs. 4.1–A.10.

# Chapter 5

# Feature Selection, Performance Metrics, and Experimental Procedure

## 5.1 Feature Selection

Feature selection is usually the first step in the classification process. Selecting relevant features may decrease the redundancy while enhancing the classification performance and reducing training time in deep neural networks [121]. Purpose of feature selection is to identify a set of useful features with the preservation of important discriminatory information [19]. The selected features are used as an input to classification algorithms. Simulations with ESNs employing two feature selection algorithms have been performed to illustrate that it is important to disregard irrelevant features from directly contributing to the output in order to decrease the generalization error (error when trained ESNs are evaluated on previously unseen data) [122].

Decision tree is an algorithm that is employed for classification or regression tasks by posing conditions on a given point. A simple condition may be of the form: "Is feature $i$ smaller than the value $v$?" The conditions are represented as non-leaf nodes and predicted values are represented as leaves of a decision tree. Decision trees are trained downward from the root in a greedy recursive way. When a node with an associated split is created, the children of the node are constructed by the same tree-growing procedure. The data used to train left subtree are the points satisfying $x_i < v$. Similarly, the right subtree is trained with the data points satisfying $x_i \geq v$. Trees are grown until no further splits are made and a prediction (rather than a split) is associated with a node. Building the trees is associated with understanding of how to select the split-feature, split-value pairs and when to stop growing the trees. Decision trees select a split by considering all possible splits and choosing the split that has the smallest uncertainty. Ways to quantify the uncertainty are *entropy* and *Gini impurity*. The heuristics that we consider when deciding when to stop splitting are:

- Limited depth: Do not split if the node is beyond some fixed depth in the tree.
- Node purity: Do not split if the proportion of training points in a class is sufficiently high.
- Information gain criteria: Do not split if the gained information/purity is sufficiently close to zero.

In order to select relevant features, we have employed tree-based ensemble learning method called extremely randomized trees, or extra-trees [123]. Ensemble learning is a technique to overcome the overfitting by combining the predictions of many varied models into a single prediction. The varied models are decision trees trained in a randomized way to reduce correlation among them. The extra-trees algorithm generates a large number of decision trees from the training set. The majority voting from decision trees is employed for classification tasks. The main differences between extra-trees and other related ensemble decision trees algorithms such as random forest and bagging are that the extra-trees algorithm randomly chooses cut-points to split nodes (instead of using greedy algorithm) and it employs the entire training set to grow trees. The main hyperparameters to adjust are the number of attributes (features) $K$ used at each node and a minimum sample size *nmin* in a node to create a new splitting point. $K$ defines the strength of the feature selection process while *nmin* determines the strength of averaging output noise. The number of decision trees in the ensemble determines the strength of the variance reduction of the ensemble model aggregation. The common value for the number of decision trees is 100, which is large enough to ensure convergence. Typically selected value for *nmin* is 2 for classification and 5 for regression [123].

The scikit-learn Python machine learning library contains an implementation of extra-trees (ExtraTreesRegressor and ExtraTressClassifier classes). We rank the features of the employed datasets using *sklearn.ensemble.ExtraTreesClassifier()* [124] with parameters *n_estimators* = 100, *min_samples_split* = 2, and default values for the remaining settings. Gini impurity is the function used to measure the quality of a split. It measures how often a randomly chosen element from the set would be incorrectly labeled. It is defined as:

$$G(Y) = \sum_k P(Y=k) \sum_{i \neq k} P(Y=i), \tag{5.1}$$

$$G(Y) = \sum_k P(Y=k)(1 - P(Y=k)), \tag{5.2}$$

$$G(Y) = 1 - \sum_k P(Y=k)^2, \tag{5.3}$$

where $Y$ are the labels. Gini impurity is slightly faster to compute than entropy because there is no need to take logs. Feature importance is calculated as the decrease in node impurity weighted by the probability of reaching the leaf node. The node probability may be calculated by the number of samples that reach the node, divided by the total number of samples. The higher the value, the more important is the feature. Most relevant features in the CIC-IDS datasets and their importance are shown in Table 5.1. Most relevant features in Slammer, Nimda, Code Red I, DDoS2019, and DDoS2020 BGP datasets and their importance are shown in Tables 5.2–5.3.

## 5.2 Performance Metrics

We evaluate the ESN performance by calculating the confusion matrix shown in Table 5.4, where:

- True Negative (TN): correctly classified regular data points as regular
- False Negative (FN): incorrectly classified anomalous data points as regular (type II error)
- False Positive (FP): incorrectly classified regular data points as anomaly (type I error)
- True Positive (TP): correctly classified anomalous data points as anomaly

Various measures calculated to evaluate classification algorithms are: accuracy, F-Score, precision, recall (sensitivity), and specificity. Accuracy reflects the proportion of the accurately predicted results:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}. \tag{5.4}$$

Even though it is common to use accuracy as a metric for a classification model performance, it may be a misleading measure for imbalanced datasets because it accepts equal cost for misclassifications despite the distribution of classes in a dataset. For example, given a sample with 95 regular instances and 5 anomalous, classifying all instances as regular will yield 95% accuracy, which seems high. However, no anomalous points were classified correctly. In this case of uneven class distribution, it would be more effective to use F-Score (5.5). F-Score considers the false predictions and is described as a harmonic mean of the precision and recall (sensitivity):

$$\text{F-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5.5}$$

It measures the discriminating ability of the classifier to identify classified and misclassified anomalies. In case of misclassified anomalous points in the given example, where the accuracy score is 95%, the F-Score is zero.

The precision identifies true anomalies among all data points that are classified as anomalies and shows how many positively identified instances were relevant:

$$\text{Precision} = \frac{TP}{TP + FP}. \tag{5.6}$$

Table 5.1: Most relevant features and their importance for CIC-IDS datasets. Feature importance score in parentheses: decreased impurity weighted by the probability of reaching the node

| **CIC-IDS2017 Wednesday, July 5** | |
|---|---|
| 1. feature 84: BiFlowsCount (0.189) | 11. feature 58: Average Packet Size (0.026) |
| 2. feature 5: Protocol (0.049) | 12. feature 46: Packet Length Mean (0.025) |
| 3. feature 53: ACK Flag Count (0.044) | 13. feature 29: Fwd IAT Max (0.025) |
| 4. feature 80: Idle Mean (0.039) | 14. feature 2: Source Port (0.024) |
| 5. feature 18: Bwd Packet Length Mean (0.038) | 15. feature 75: min_seg_size_forward (0.023) |
| 6. feature 24: Flow IAT Max (0.036) | 16. feature 19: Bwd Packet Length Std (0.021) |
| 7. feature 4: Destination Port (0.034) | 17. feature 44: Min Packet Length (0.020) |
| 8. feature 60: Avg Bwd Segment Size (0.033) | 18.feature 54: URG Flag Count (0.017) |
| 9. feature 47: Packet Length Std (0.029) | 19. feature 16: Bwd Packet Length Max (0.017) |
| 10. feature 82: Idle Max (0.027) | 20. feature 28: Fwd IAT Std (0.017) |
| **CSE-CIC-IDS2018 Thursday, February 15** | |
| 1. feature 70: Fwd Seg Size Min (0.218) | 11. feature 68: Init Bwd Win Byts (0.020) |
| 2. feature 67: Init Fwd Win Byts (0.086) | 12. feature 18: Flow IAT Mean (0.020) |
| 3. feature 79: BiFlowsCount (0.069) | 13. feature 22: Fwd IAT Tot (0.020) |
| 4. feature 1: Protocol (0.043) | 14. feature 26: Fwd IAT Min (0.019) |
| 5. feature 0: Dst Port (0.041) | 15. feature 25: Fwd IAT Max (0.018) |
| 6. feature 48: PSH Flag Cnt (0.034) | 16. feature 23: Fwd IAT Mean (0.016) |
| 7. feature 12: Bwd Pkt Len Max (0.030) | 17. feature 75: Idle Mean (0.015) |
| 8. feature 15: Bwd Pkt Len Std (0.029) | 18. feature 3: Flow Duration (0.015) |
| 9. feature 49: ACK Flag Cnt (0.026) | 19. feature 20: Flow IAT Max (0.015) |
| 10. feature 21: Flow IAT Min (0.021) | 20. feature 41: Pkt Len Max (0.015) |
| **CSE-CIC-IDS2018 Friday, February 16** | |
| 1. feature 0: Dst Port (0.246) | 11. feature 44: Pkt Len Var (0.024) |
| 2. feature 8: Fwd Pkt Len Max (0.128) | 12. feature 37: Bwd Header Len (0.023) |
| 3. feature 11: Fwd Pkt Len Std (0.095) | 13. feature 19: Flow IAT Std (0.018) |
| 4. feature 41: Pkt Len Max (0.056) | 14. feature 54: Pkt Size Avg (0.018) |
| 5. feature 55: Fwd Seg Size Avg (0.047) | 15.feature 25: Fwd IAT Max (0.017) |
| 6. feature 10: Fwd Pkt Len Mean (0.046) | 16. feature 14: Bwd Pkt Len Mean (0.016) |
| 7. feature 43: Pkt Len Std (0.044) | 17.feature 68: Init Bwd Win Byts (0.015) |
| 8. feature 15: Bwd Pkt Len Std (0.033) | 18. feature 48: PSH Flag Cnt (0.016) |
| 9. feature 12: Bwd Pkt Len Max (0.032) | 19.feature 67: Init Fwd Win Byts (0.012) |
| 10.feature 23: Fwd IAT Mean (0.026) | 20. feature 6: TotLen Fwd Pkts (0.011) |
| **CIC-DDoS2019 Saturday, January 12** | |
| 1. feature 85: BiFlowsCount (0.206) | 11. feature 21: Flow Packets/s (0.029) |
| 2. feature 2: Source Port (0.086) | 12. feature 46: Packet Length Mean (0.021) |
| 3. feature 4: Destination Port (0.081) | 13. feature 55: CWE Flag Count (0.020) |
| 4. feature 54: URG Flag Count (0.076) | 14. feature 5: Protocol (0.020) |
| 5. feature 53: ACK Flag Count (0.056) | 15. feature 58: Average Packet Size (0.020) |
| 6. feature 13: Fwd Packet Length Min (0.051) | 16. feature 57: Down/Up Ratio (0.019) |
| 7. feature 59: Avg Fwd Segment Size (0.042) | 17. feature 20: Flow Bytes/s (0.018) |
| 8. feature 42: Fwd Packets/s (0.038) | 18. feature 72: Init_Win_bytes_forward (0.014) |
| 9. feature 44: Min Packet Length (0.038) | 19. feature 51: RST Flag Count (0.011) |
| 10. feature 14: Fwd Packet Length Mean (0.030) | 20. feature 35: Bwd IAT Min (0.009) |

Table 5.2: Most relevant features and their importance: BGP datasets: Slammer, Nimda, and Code Red I. Feature importance score in parentheses: decreased impurity weighted by the probability of reaching the node

| | |
|---|---|
| **Slammer** | |
| 1. feature 34: IGP packets (0.116) | 11. feature 37: Packet size (B) (0.029) |
| 2. feature 1: Number of announcements (0.112) | 12. feature 6: Maximum *AS-path* length (0.024) |
| 3. feature 36: Number of incomplete packets (0.102) | 13. feature 13: Interarrival time (0.023) |
| 4. feature 3: Number of announced NLRI prefixes (0.094) | 14. feature 7: Average unique *AS-path* length (0.020) |
| 5. feature 9: Number of duplicate withdrawals (0.084) | 15. feature 5: Average *AS-path* length (0.019) |
| 6. feature 8: Number of duplicate announcements (0.073) | 16. feature 11: Average edit distance (0.018) |
| 7. feature 10: Number of implicit withdrawals (0.072) | 17. feature 20: Maximum edit distance $n = 13$ (0.016) |
| 8. feature 4: Number of withdrawn NLRI prefixes (0.071) | 18. feature 35: Number of EGP packets (0.009) |
| 9. feature 2: Number of withdrawals (0.043) | 19. feature 28: Maximum *AS-path* length $n = 10$ (0.004) |
| 10. feature 12: Maximum edit distance (0.031) | 20. feature 26: Maximum *AS-path* length $n = 8$ (0.004) |
| **Nimda** | |
| 1. feature 34: Number of IGP packets (0.136) | 11. feature 8: Number of duplicate announcements (0.047) |
| 2. feature 1: Number of announcements (0.129) | 12. feature 13: Interarrival time (0.023) |
| 3. feature 3: Number of announced NLRI prefixes (0.100) | 13. feature 7: Average unique *AS-path* length (0.020) (0.019) |
| 4. feature 4: Number of withdrawn NLRI prefixes (0.079) | 14. feature 5: Average *AS-path* length (0.019) |
| 5. feature 9: Number of duplicate withdrawals (0.075) | 15. feature 35: Number of EGP packets (0.013) |
| 6. feature 12: Maximum edit distance (0.067) | 16. feature 6: Maximum *AS-path* length (0.011) |
| 7. feature 37: Packet size (B) (0.059) | 17. feature 11: Average edit distance (0.010) |
| 8. feature 2: Number of withdrawals (0.055) | 18. feature 16: Maximum edit distance $n = 9$ (0.004) |
| 9. feature 36: Number of incomplete packets (0.054) | 19. feature 14: Maximum edit distance $n = 7$ (0.004) |
| 10. feature 10: Number of implicit withdrawals (0.049) | 20. feature 32: Maximum *AS-path* length $n = 14$ (0.004) |
| **Code Red I** | |
| 1. feature 34: Number of IGP packets (0.137) | 11. feature 8: Number of duplicate announcements (0.049) |
| 2. feature 1: Number of announcements (0.137) | 12. feature 13: Interarrival time (0.025) |
| 3. feature 3: Number of announced NLRI prefixes (0.096) | 13. feature 7: Average unique *AS-path* length (0.021) (0.019) |
| 4. feature 4: Number of withdrawn NLRI prefixes (0.079) | 14. feature 5: Average *AS-path* length (0.020) |
| 5. feature 9 : Number of duplicate withdrawals (0.070) | 15. feature 35: Number of EGP packets (0.012) |
| 6. feature 12: Maximum edit distance (0.065) | 16. feature 11: Average edit distance (0.009) |
| 7. feature 36: Number of incomplete packets (0.058) | 17. feature 6: Maximum *AS-path* length (0.009) |
| 8. feature 37: Packet size (B) (0.057) | 18. feature 32: Maximum *AS-path* length $n = 14$ (0.004) |
| 9. feature 2: Number of withdrawals (0.055) | 19. feature 18: Maximum edit distance $n = 11$ (0.004) |
| 10. feature 10: Number of implicit withdrawals (0.049) | 20. feature 29: Maximum *AS-path* length $n = 11$ (0.003) |

Table 5.3: Most relevant features and their importance: BGP datasets: DDoS2019 and DDoS2020. Feature importance score in parentheses: decreased impurity weighted by the probability of reaching the node

| **DDoS2019** | |
|---|---|
| 1. feature 35: EGP packets (0.116) | 11. feature 12: Maximum edit distance (0.029) |
| 2. feature 1: Number of announcements (0.112) | 12. feature 4: Number of withdrawn NLRI prefixes (0.024) |
| 3. feature 10: Number of implicit withdrawals (0.102) | 13. feature 7: Average unique *AS-path* length (0.023) |
| 4. feature 3: Number of announced NLRI prefixes (0.094) | 14. feature 7: Average unique *AS-path* length (0.020) |
| 5. feature 2: Number of withdrawals (0.084) | 15. feature 13: Interarrival time (0.019) |
| 6. feature 34: Number of IGP packets (0.073) | 16. feature 11: Average edit distance (0.018) |
| 7. feature 37: Packet size (B)(0.072) | 17. feature 21: Maximum edit distance $n = 14$ (0.016) |
| 8. feature 3: Number of announced NLRI prefixes (0.071) | 18. feature 5: Average *AS-path* length (0.009) |
| 9. feature 8: Number of duplicate announcements (0.043) | 19. feature 20: Maximum edit distance $n = 13$ (0.004) |
| 10. feature 9: Number of duplicate withdrawals (0.031) | 20. feature 22: feature 22: Maximum edit distance $n = 15$ (0.004) |
| **DDoS2020** | |
| 1. feature 8: Number of duplicate announcements (0.136) | 11. feature 12: Maximum edit distance (0.047) |
| 2. feature 2: Number of withdrawals(0.129) | 12. feature 35: Number of EGP packets (0.023) |
| 3. feature 9: Number of duplicate withdrawals (0.100) | 13. feature 13: Interarrival time (0.019) |
| 4. feature 36: Number of incomplete packets (0.009) | 14. feature 11: Average edit distance (0.019) |
| 5. feature 3: Number of announced NLRI prefixes (0.075) | 15. feature 6: Maximum *AS-path* length (0.013) |
| 6. feature 34: Number of IGP packets (0.013) (0.004) | 16. feature 5: Average *AS-path* length (0.011) |
| 7. feature 1: Number of announcements (0.059) | 17. feature 7: Average unique *AS-path* length (0.010) |
| 8. feature 37: Packet size (B) (0.055) | 18. feature 20: Maximum edit distance $n = 13$ (0.004) |
| 9. feature 4: Number of withdrawn NLRI prefixes (0.054) | 19. feature 23: Maximum edit distance $n = 16$ (0.004) |
| 10. feature 10: Number of implicit withdrawals (0.049) | 20. feature 22: Maximum edit distance $n = 15$ (0.004) |

Table 5.4: Confusion matrix

| | **Predicted class** | |
|---|---|---|
| **Actual class** | Negative (regular) | Positive (anomaly) |
| Negative (regular) | TN | FP |
| Positive (anomaly) | FN | TP |

Recall measures the ability of the model to identify correctly predicted anomalies and shows how many relevant instances are selected:

$$\text{Recall} = \frac{TP}{TP + FN}. \tag{5.7}$$

Specificity or true negative rate, measures the proportion of actual negatives that are correctly identified:

$$\text{Specificity} = \frac{TN}{TN + FP}. \tag{5.8}$$

False alarm rate (FAR) is a common measure used for evaluating intrusion detection models:

$$\text{FAR} = 1 - \text{Specificity} = \frac{FP}{TN + FP}. \tag{5.9}$$

## 5.3 Experimental Procedure

The experiments are performed on Windows 10 64-bit Operating System and Intel Core i7-8650U CPU at 1.9-2.11 GHz. We use Python 3.8 and import the following libraries and packages:

- **numpy**: Library that supports large and multidimensional arrays and matrices, as well as high-level mathematical functions to manipulate these arrays and matrices [125].
- **pandas**: Library for data manipulation and analysis [126].
- **scipy**: Package that provides a wide range of functions to work around with different format of files; we imported: "sparse" for sparse matrices manipulation; **scipy.io**: Module that allows to read data from and write data to a variety of file formats; **scipy.stats** a module that includes a large number of probability distributions, along with a growing library of statistical functions.
- **scikit-learn/sklearn**: Library that contains various classification, regression, and clustering algorithms (such as random forests). It is designed to interoperate with the Python numerical (numpy) and scientific libraries (scipy); employed were: "preprocessing", **sklearn.decomposition** "PCA" for linear dimensionality reduction, **sklearn.linear_model** "Ridge" for linear least squares with L2 regularization, **sklearn.metrics** "accuracy_score", "f1_score", **sklearn.model_selection** "train_test_split" [127].
- **math**: Provides an access to mathematical functions defined by the C standard [129].

### 5.3.1 Data Preprocessing: CIC-IDS Datasets

The numbers of features in CIC-IDS2017, CSE-CIC-IDS2018, and CIC-DDoS2019 datasets are 84, 79, and 85, respectively. As discussed in Section 5.1, we extract 20 most important features from each dataset to train the model. Categorical features are converted to numeric (by applying pandas.to_numeric), with setting the invalid parsing to Not a Number (NaN). The values of raw data varied broadly. Therefore, in order to normalize the range of features, we apply min-max scaling, so that each feature contributes proportionately. Normalization keeps the input data bounded avoiding outliers. Min-max scaling translates each feature leaving it in the given range of the training set (between zero and one). When using min-max scaling, NaN are treated as missing values and kept in normalized data. The scaling is given by:

$$x' = \frac{x - min(x)}{max(x) - min(x)}. \tag{5.10}$$

After normalization NaN are converted to 0. A target data frame is created while converting all the 'BENIGN' labels to 0 and the remaining to 1.

Bias in the training dataset may affect the performance of machine learning models. Oversampling and undersampling are the resampling techniques applied to imbalanced datasets with a skewed class distribution [130]. Oversampling is duplicating samples from the minority class (the class with the smallest number of data points). Undersampling is removing samples from the majority class. These resampling techniques are illustrated in Fig. 5.1. The total, regular, and anomalous number of points before applying resampling with the CIC-IDS datasets are shown in Table 5.5.

Table 5.5: Number of data points before resampling

| **Dataset** | **Class** | **Number of data points** |
|---|---|---|
| **CIC-IDS2017, Wednesday, July 5, 2017** | **Total** | **692,703** |
| | Regular | 440,031 |
| | Anomaly | 252,672 |
| **CSE-CIC-IDS2018, Thursday, February 15, 2018** | **Total** | **1,048,575** |
| | Regular | 996,077 |
| | Anomaly | 52,498 |
| **CSE-CIC-IDS2018, Friday, February 16, 2018** | **Total** | **1,048,575** |
| | Regular | 446,772 |
| | Anomaly | 601,802 |
| **CIC-DDoS2019, Saturday, January 12, 2019** | **Total** | **1,000,000** |
| | Regular | 3,654 |
| | Anomaly | 996,346 |

We apply both random oversampling and random undersampling. Random oversampling involves randomly selecting examples from the minority class, and including them with replacement to the dataset achieving the desired split across the classes. The examples from the minority are selected from the original dataset and inserted to the new "more balanced" dataset multiple times. These samples are returned or "replaced" in the original dataset, allowing to be selected again. Random undersampling involves randomly selecting samples from the majority class to be removed from a dataset. Number of points in the majority class is decreased until the desired class distribution is achieved. A drawback of undersampling is that valuable information may be removed from the majority class. Due to removal of random samples, it may be infeasible to keep "more important" information in the majority class. Random oversampling and random undersampling are known as "naive resampling" because no assumptions about the data is made and no heuristics are used. Random resampling is simple to implement and fast to execute even with large datasets. It may be employed for binary and multi-class classification tasks [130].

CIC-IDS2017 dataset initially contains 692,703 data points. The dataset is unbalanced: 440,031 regular instances and 252,672 anomalies. When creating balanced datasets, we apply random resampling: We experiment with **oversampling** (oversampling anomalies), followed by **undersampling** (undersampling regular points).

The number of data points in CSE-CIC-IDS2018, on both days (Thursday and Friday) is 1,048,575. CSE-CIC-IDS2018 dataset collected on Thursday, February 15, 2018 includes

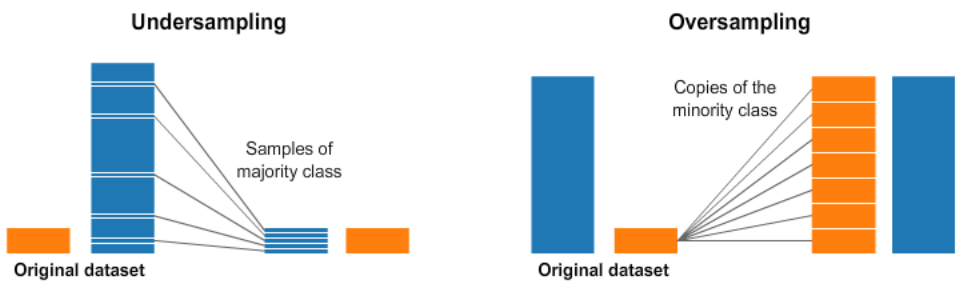


Figure 5.1: Resampling: a widely adopted technique for dealing with unbalanced datasets. It consists of removing samples from the majority class (undersampling) or adding more samples from the minority class (oversampling) [131].

996,077 regular points and 52,498 anomalies. For the experiments with balanced datasets, we **oversample** underrepresented attack class (anomalies). The number of data points after applying **undersampling** of regular class is 104,996. There is no need to apply resampling to CSE-CIC-IDS2018 dataset collected on Friday, February 16, 2018 since the dataset contains 57% anomaly and 43% regular data points.

CIC-DDoS2019 dataset consists of multiple datasets. We merge DrDoS_NTP.csv, DrDoS_LDAP.csv, Syn.csv, and UDPLag.csv files. Since the dataset, with the total of 1,000,000 points, is highly unbalanced, we only conduct experiments with balanced (resampled) dataset. We apply minority (regular class) **oversampling**. The final dataset has 1,000,000 data points containing 50% regular data points.

The number of total, regular, and anomalous data points after applying oversampling and undersampling is shown in Table 5.6. Because the size of training data impacts the

Table 5.6: Number of data points after oversampling and undersampling

| **Dataset** | **Class** | **After oversampling** | **After undersampling** |
|---|---|---|---|
| **CIC-IDS2017, Wednesday, July 5** | **Total** | **800,000** | **505,344** |
| | Regular | 399,919 | 252,672 |
| | Anomaly | 400,081 | 252,672 |
| **CSE-CIC-IDS2018, Thursday, February 15** | **Total** | **1,000,000** | **104,996** |
| | Regular | 500,120 | 52,498 |
| | Anomaly | 499,880 | 52,498 |
| **CIC-DDoS2019, Saturday, January 12** | **Total** | **1,000,000** | |
| | Regular | 500,153 | |
| | Anomaly | 499,847 | |

performance of machine learning models, the datasets are of comparable sizes (exception: undersampled datasets). In order to speed up the computation, a fraction of each dataset may be used: we use half of each dataset.

### 5.3.2 Data Preprocessing: BGP Datasets

We use BGP datasets that contain update messages collected during the time periods when the Internet experienced major anomalies. The amount of total, regular, and anomalous points in Slammer, Nimda, and Code Red I datasets are provided in Table 5.7. The amount of total, regular, and anomalous points in DDoS2019 and DDoS2020 datasets are provided in Table 5.8.

Table 5.7: Number of total, regular, and anomalous points in Slammer, Nimda, and Code Red I datasets

| Dataset | Class | Number of points |
|---|---|---|
| **Slammer** | **Total** | **7,200** |
| | Regular | 6,331 |
| | Anomaly | 869 |
| **Nimda** | **Total** | **8,609** |
| | Regular | 7,308 |
| | Anomaly | 1,301 |
| **Code Red I** | **Total** | **7,200** |
| | Regular | 6,600 |
| | Anomaly | 600 |

Table 5.8: Number of total, regular, and anomalous points in AWS 2019 and AWS 2020 datasets

| Dataset | Class | Number of points |
|---|---|---|
| **DDoS2019-v1** | **Total** | **7,200** |
| | Regular | 6,719 |
| | Anomaly | 481 |
| **DDoS2019-v2** | **Total** | **10,080** |
| | Regular | 6,390 |
| | Anomaly | 3,690 |
| **DDoS2020** | **Total** | **10,080** |
| | Regular | 5,709 |
| | Anomaly | 4,371 |

For Slammer and Code Red I anomalies, we consider a five-day period: one day of the attack (anomalous data points) and two days prior and two days after the attack (regular data points). Slammer and Code Red I attacks lasted 869 and 600 minutes, respectively. The duration of regular periods within two days before and after the Slammer and Code Red I are 6,331 and 6,600 minutes, respectively. The Nimda attack lasted 1,301 minutes. We use two and a half days prior to the Nimda event and two days after the attack as regular data points. Oversampling the anomalous points in BGP datasets in order to have balanced datasets, causes significant improvement (around 100%) in the performance of machine learning models implying that the models learn well when smaller datasets, such as BGP datasets, get resampled. Therefore, for the experiments in this Thesis, we are not balancing the BGP datasets via resampling techniques. The data represented by matrices: 7,200 by 37, 8,609 by 37, and 7,200 by 37, where 37 corresponds to 37 features.

For DDoS2019-v1 dataset, we consider a five-day period with 8 hours of anomaly. For DDoS2019-v2, we select approximately 2.5 days of anomaly that includes the ransom-driven DDoS attack on October 23, 2019, and two days prior and two days after the attacks for regular data points. DDoS2020 dataset contains 3 days of anomalies, and four days of regular points.

The 20 most important features from each dataset are used to train the model, as discussed in Section 5.1. Similarly to CIC-IDS datasets, categorical features found in features columns are converted to a numeric type by applying pandas.to_numeric. To normalize the range of features, we apply min-max scaling so that each feature contributes proportionately.

Training set is a set of observations used for training (fitting) a model. Test set is a set of new observations. Number of data points and properties of a model often determine how to split a dataset. Most commonly, the training and test sets are split in proportions 7:3 or 8:2. If a model requires a large number of data points to train, the test set may be reduced to improve the performance [19]. We have divided the data into 80% or 60% for training and 20% or 40% for test, as shown in Tables 6.2, 6.4, 6.5, 6.7, and 6.9.

## 5.4 Cross-Validation in ESN

Choosing the best hyperparameters is known as *model selection.* It is not recommended to select models using a test set. Instead, a training set is split into smaller subsets where validation subsets are used to evaluate the model. *K-fold cross-validation*, the most popular cross-validation technique, allows $(K-1)/K$ of the data to be used for training and the rest to assess performance. When employing a k-fold cross-validation, the entire training set is used for both training and validating where each subset is used for validation only once. The *10-fold cross-validation* is illustrated in Fig. 5.2. In the 10-fold cross-validation, the original training dataset is randomly partitioned into 10 equal-sized subsets. Nine subsets are used for training and one is left as a validation set to evaluate the model. This process continues until all subsets have been used for both training and validation and have returned 10 estimation results. The average of the 10 results is the final estimation of the model performance useful for model selection. When the model achieves low training loss and low test loss, the model is said to *generalize.*

The k-fold cross-validation may be effective in preventing models' underfitting and overfitting. Underfitting occurs when a model is not fitted well enough: when training and test losses are high. A different selection of hyperparameters may solve this issue. In practice, underfitting is easy to detect while overfitting is rather challenging. When model overfits, the model is fitted too well to the data and its test loss is much higher than training loss.

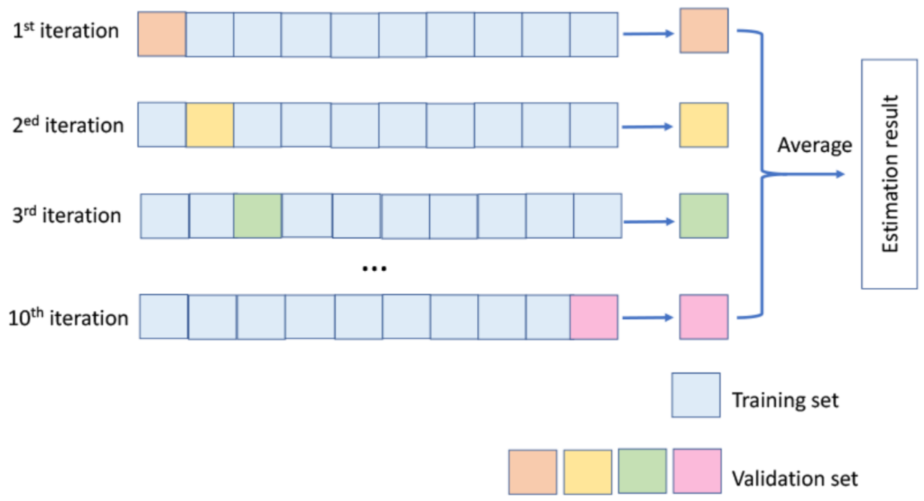


Figure 5.2: Illustration of the 10-fold cross validation. The original training dataset is partitioned into 10 folds. Each fold is used once as a validation set during the training process. The final estimation is the average number of the 10 validation results.

In general, training a model on larger datasets makes it perform better on test data. Therefore, when the training set is smaller, the validation loss is a less accurate gauge of true performance on the testing set. This is one of the drawbacks of cross-validation. When training set is smaller, machine learning models may underfit. Another drawback of k-fold cross-validation is computational cost: number of training runs is increased by k. This may be problematic for models where the training is computationally expensive [19]. For example, this may occur with ESN when increasing the number of reservoir nodes. Typically, ESN and other time series algorithms are validated on a single data subset. Cross-validation may optimize the basic ESN performance [136]. We use a variation of 10-fold cross-validation called time series cross-validation in order to test if having several splits and averaging the results would produce a better performance.

## Chapter 6

# Performance Results

We generate the reservoir by creating a class (ESN_Reservoir) with hyperparameters:

- *proc_nodes*: $N^z$ determines the size of the reservoir, number of reservoir/internal/processing nodes (units/neurons);
- *s_radius*: Largest eigenvalue of reservoir connection weights matrix $\mathbf{W}$.
- *leak*: Leaking rate of the reservoir.
- *connectivity*: Ratio of nonzero elements in the reservoir, related to the sparsity of the reservoir. This hyperparameter is not used in a deterministic topology.
- *inp_scale*: Scaling of input weights matrix $W^{in}$.
- *noise*: Deviation of the Gaussian noise added to the state update.
- *deterministic_res*: Deterministic (also known as recursive or circular) reservoir with each weight having the same value.

Reservoirs that vary in size, number of nodes, or other hyperparameters may behave differently in various learning tasks [62]. However, contribution of hyperparameters to the performance of ESN models may be ambiguous. Therefore, thorough hyperparameters initialization may improve ESN's predictive or classification capabilities.

The input and reservoir weights are generated randomly. To generate the input weights, we use a binomial distribution with n = 1, p = 0.5, and the size = $N^x \times N^z$. The reservoir weights $\mathbf{W}$ are usually sparse (not fully connected) because sparse connections yield better performance and speed up the updates [72]. It is recommended to connect each internal node to a small number of other internal nodes (10 nodes). The connectivity in this Thesis is set to 25% (alternatively, sparsity is set to 75%). Generated reservoir weights are set to be uniformly distributed between $[-0.5, 0.5]$ centered around 0.0.

A simple deterministically constructed circular reservoir was sufficient for a variety of tasks to obtain performances comparable to the ESN with randomly generated reservoirs [134]. In order to evaluate the reservoir with deterministic topology, that is designed with minimal complexity we may select a deterministic reservoir. The reservoir weights $\mathbf{W} \in R^{N^z \times N^z}$ are then generated in a deterministic manner where each connection is having the same weight.

A larger value of spectral radius is employed for the tasks where a longer history of the input is required. If the output $\mathbf{y}(n)$ relies on a more recent history of the input $\mathbf{x}(n)$, a smaller value of the radius is recommended. To assure an echo state property, the ability of the reservoir to gradually wash out the effect of past input on the present state, the spectral radius value is kept below 1. [72]. We divide $\mathbf{W}$ by the spectral radius to obtain a matrix with a unit spectral radius, which is later scaled with the given value of spectral radius. Spectral radius scales the matrix $\mathbf{W}$ (scales the width of the distribution of nonzero elements):

$$\mathbf{W} = \frac{\mathbf{W} * \rho}{max|\lambda|}. \tag{6.1}$$

Usually, the level of input weights scaling is determined by trial and error. The input weights are scaled by 0.3. Scaling of both $\mathbf{W^{in}}$ and $\mathbf{W}$ indicates how much the current state $z(n)$ relies on the input. Furthermore, principal component analysis (PCA) characterized by dimensionality reduction with data preservation may be included before feeding the input to the reservoir.

The training/test phase begins upon calculating the initial reservoir state matrix (when loading *x_train*/*x_test*). Then, we set the sequence of reservoir states at each instance - the state matrix $\mathbf{Z}$ (matrix $\mathbf{Z} \in \mathcal{R}^{N \times (N_z + N_x)}$ is generated by concatenating the column vectors $[\mathbf{z}(n); \mathbf{x}(n)]$ horizontally over the training data points $n$) using (6.2) and (6.3), where for every time step $n$, the current state of a reservoir is:

$$\widetilde{\mathbf{z}}(n) = tanh(\mathbf{x}(n)\mathbf{W}^{in} + \mathbf{z}(n-1)\mathbf{W}), \tag{6.2}$$

and the update is:

$$\mathbf{z}(n) = (1-\alpha)\mathbf{z}(n-1) + \alpha\widetilde{\mathbf{z}}(n), \tag{6.3}$$

where $\mathbf{z}(n-1) \in \mathcal{R}^{N_z}$ is the previous state. Furthermore, we include noise for regularization. The addition of noise during the training of a machine learning model has a regularization effect and improves the robustness of the model [132]. It shows a similar impact on the loss function (2.23) as the addition of a penalty term, as in the case of weight regularization methods. A schematic of ESN is shown in Fig. 6.1. After training the model, we evaluate its performance using the metrics described in Section 5.2.

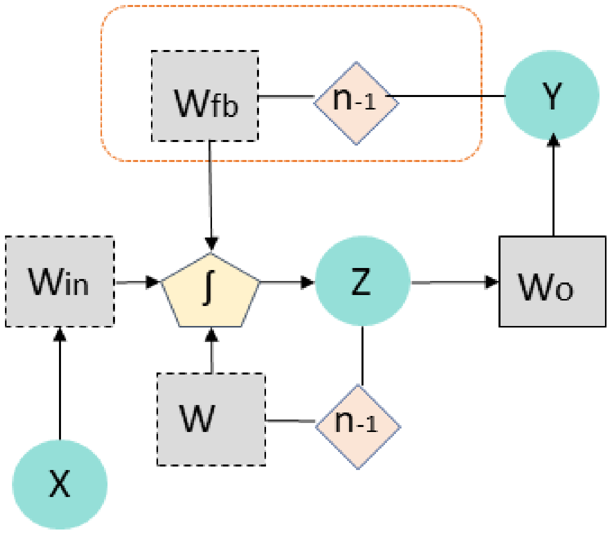


Figure 6.1: ESN structure. Shown are: input X , reservoir state Z, and output Y; trainable matrix $W_o$; $W_{in}$ and $W_{fb}$ are randomly initialized matrices; $\int$ represent non-linear transformation; "$n_{-1}$" indicates the unit delay. $W_{fb}$ and adjacent "$n_{-1}$" unit delay are optional feedback.

## 6.1 Performance of ESN Models with Balanced and Unbalanced Datasets

We follow the guides for applying the ESN for time-series classification [72, 128] and conduct grid-search by varying reservoir hyperparameters' settings. The results are generated based on the scripts provided in the Appendix: Section 7. In order to evaluate the influence of the hyperparameters on the ESN performance, we select the following values, as shown in Table 6.1: Deterministic Reservoir Weights $\mathbf{W}$ are set to either *False* or *True*; leaking rate ("$\alpha$"): *None* and 0.2; spectral radius ("$\rho(\mathbf{W})$"): 0.1 and 0.9; number of reservoir nodes ("$N_z$"): 10 and 30 (higher values for reservoir nodes produced memory errors and thus are not considered); connectivity: 0.25; input scaling: 0.3; and noise level: 0.01. We develop the ESN models when varying one hyperparameter at a time and fixing values for the rest, as shown in Table 6.1.

Table 6.1: ESN models and hyperparameters: deterministic reservoir weights $\mathbf{W}$, leaking rate ($\alpha$), spectral radius ($\rho(\mathbf{W})$), and number of reservoir nodes ($N_z$)

| | Deterministic $\mathbf{W}$ | $\rho(\mathbf{W})$ | $\alpha$ | $N_z$ |
|---|---|---|---|---|
| **ESN1** | False | 0.9 | 0.2 | 10 |
| **ESN2** | **True** | 0.9 | 0.2 | 10 |
| **ESN3** | False | **0.1** | 0.2 | 10 |
| **ESN4** | False | 0.9 | **None** | 10 |
| **ESN5** | False | 0.9 | 0.2 | **30** |

Number of data points in training and test sets for the CIC-IDS datasets is provided in Table 6.2 while the validation and test performance results are given in Table 6.3. We

Table 6.2: Number of data points in training and test sets

| Dataset | Class | Number of data points | Training set | Test set |
|---|---|---|---|---|
| **CIC-IDS2017,** | **Total** | **346,352** | **277,081** | **69,271** |
| **Wednesday, July 5** | Regular | 219,984 | 175,855 | 44,129 |
| | Anomaly | 126,368 | 101,226 | 25,142 |
| **CSE-CIC-IDS2018,** | **Total** | **525,288** | **419,430** | **104,858** |
| **Thursday, February 15** | Regular | 497,973 | 398,349 | 99,624 |
| | Anomaly | 26,315 | 21,081 | 5,234 |
| **CSE-CIC-IDS2018,** | **Total** | **525,288** | **419,430** | **104,858** |
| **Friday, February 16** | Regular | 223,208 | 178,483 | 44,725 |
| | Anomaly | 301,080 | 240,947 | 60,133 |
| **CIC-DDoS2019,** | **Total** | **500,000** | **400,000** | **100,000** |
| **Saturday, January 12** | Regular | 249,977 | 200,016 | 49,961 |
| | Anomaly | 250,023 | 199,984 | 50,039 |

perform a model selection based on 10-fold cross-validation results that indicate that the ESN5 is the best model among evaluated ESNs in terms of accuracy, F-Score, and false alarm rate. Increasing the number of reservoir nodes $N_z$ enhances the performance of ESN models: ESN5 model with 30 reservoir nodes shows better performance than ESN1 model with 10 reservoir nodes. Reducing the radius of the reservoir degrades the performance: the ESN3 model with a low spectral radius performs worse than the ESN1 model with a larger spectral radius. Additional benefits of using cross-validation are improved stability: training on multiple folds could be a form of regularization. Moreover, if data have occasional imperfections, averaging validation over many folds reduces their effects [136].

In order to balance CIC-IDS2017, Wednesday, July 5, 2017, CSE-CIC-IDS2018, Thursday, February 15, 2018, and CIC-DDoS2019, Saturday, January 12, 2019, we use the resampling techniques: random oversampling and undersampling, as described in Subsection 5.3.1. As shown in Table 6.6, there is a difference compared to the cases with unbalanced datasets. The results vary depending on the used resampling technique. We observe that resampling the CIC-IDS datasets neither significantly improves nor reduces the performance of ESN models. Training a model on larger data makes it perform better on test data [133]. Imbalanced and balanced CIC-IDS datasets are large enough for the models to learn diverse examples and show comparable classification performance with no overfitting. For balanced (resampled) CIC-IDS datasets, ESN5 is the best model in terms of accuracy, F-Score, and false alarm rate.

Number of data points in training and test sets for BGP datasets: Slammer, Nimda, and Code Red I are given in Table 6.7 while the performance results are given in Table 6.8. ESN3 is the best model to detect anomalies in Slammer dataset while ESN2 is the best model to detect anomalies in Nimda and Code Red I datasets. ESN models are able to detect only half of the anomalies in Nimda and Code Red I datasets yielding low F-Score.

Table 6.3: Performance of ESN models when evaluated using CIC-IDS2017 Wednesday, July 5, 2017 (unbalanced), CIC-CSE-IDS2018 Thursday, February 15, 2018 (unbalanced), CIC-CSE-IDS2018 Friday, February 16, 2018 (balanced), and CIC-DDoS2019 Saturday, January 12, 2019 (balanced)

**CIC-IDS2017, Wednesday, July 5 2017**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.929 | 0.928 | 0.129 | 0.927 | 0.907 | 0.106 |
| **ESN2** | 0.927 | 0.925 | 0.136 | 0.958 | 0.945 | 0.058 |
| **ESN3** | 0.895 | 0.894 | 0.176 | 0.915 | 0.893 | 0.120 |
| **ESN4** | 0.900 | 0.899 | 0.189 | 0.919 | 0.899 | 0.120 |
| **ESN5** | **0.967** | **0.950** | **0.057** | **0.973** | **0.965** | **0.038** |

**CIC-CSE-IDS2018, Thursday, February 15 2018**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.938 | 0.937 | 0.109 | 0.983 | 0.854 | 0.017 |
| **ESN2** | 0.927 | 0.925 | 0.113 | 0.980 | 0.828 | 0.020 |
| **ESN3** | 0.798 | 0.787 | 0.401 | 0.961 | 0.679 | 0.032 |
| **ESN4** | 0.855 | 0.851 | 0.259 | 0.979 | 0.824 | 0.021 |
| **ESN5** | **0.945** | **0.944** | **0.011** | **0.997** | **0.973** | **0.003** |

**CIC-CSE-IDS2018, Friday, February 16 2018**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.988 | 0.990 | 0.115 | 0.997 | 0.998 | 0.006 |
| **ESN2** | 0.934 | 0.937 | 0.123 | 0.980 | 0.828 | 0.020 |
| **ESN3** | 0.985 | 0.988 | 0.117 | 0.996 | 0.996 | 0.010 |
| **ESN4** | 0.991 | 0.993 | 0.032 | 0.999 | 0.999 | 0.003 |
| **ESN5** | **0.995** | **0.996** | **0.009** | **0.999** | **0.999** | **0.000** |

**CIC-DDoS2019, Saturday, January 12 2019**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.989 | 0.989 | 0.017 | 0.994 | 0.994 | 0.012 |
| **ESN2** | 0.992 | 0.991 | 0.013 | 0.999 | 0.997 | 0.000 |
| **ESN3** | 0.922 | 0.921 | 0.154 | 0.927 | 0.932 | 0.146 |
| **ESN4** | 0.957 | 0.957 | 0.077 | 0.981 | 0.999 | 0.000 |
| **ESN5** | **0.998** | **0.998** | **0.002** | **0.999** | **0.999** | **0.001** |

Table 6.4: Number of data points after oversampling

| Dataset | Class | After oversampling | Training set | Test set |
|---|---|---|---|---|
| **CIC-IDS2017, Wednesday, July 5** | **Total** | **400,000** | **320,000** | **80,000** |
| | Regular | 200,376 | 160,415 | 40,039 |
| | Anomaly | 199,624 | 159,585 | 39,961 |
| **CSE-CIC-IDS2018, Thursday, February 15** | **Total** | **500,000** | **400,000** | **100,000** |
| | Regular | 250,060 | 200,403 | 50,343 |
| | Anomaly | 249,940 | 199,597 | 49,657 |
| **CIC-DDoS2019, Saturday, January 12** | **Total** | **500,000** | **400,000** | **100,000** |
| | Regular | 249,977 | 200,016 | 50,039 |
| | Anomaly | 250,023 | 199,984 | 49,961 |

Table 6.5: Number of data points after undersampling

| Dataset | Class | After undersampling | Training set | Test set |
|---|---|---|---|---|
| **CIC-IDS2017, Wednesday, July 5** | **Total** | **252,672** | **202,137** | **50,535** |
| | Regular | 126,388 | 100,918 | 25,470 |
| | Anomaly | 126,284 | 101,219 | 25,065 |
| **CSE-CIC-IDS2018, Thursday, February 15** | **Total** | **52,498** | **41,998** | **10,500** |
| | Regular | 26,280 | 21,012 | 5,294 |
| | Anomaly | 26,218 | 20,986 | 5,206 |

Numbers of data points in training and test sets for BGP datasets that contain DDoS attacks of 2019 and 2020 are given in Table 6.9 while the performance results are shown in Table 6.10. The models when trained using the DDoS2019 dataset that contains only one attack (DDoS2019-v1) that occurred on October 22, 2019 categorize almost all the points as regular producing close to 0 recall and thus low F-Score. The classifier is not able to adequately predict anomalies because the dataset does not clearly reflect the difference between regular and anomalous traces shown in Fig. A.11; therefore, the results of ESN models trained with this dataset are not included. When labeling both October 22 and 23, 2019 as days of anomalies, shown in Fig. 4.2), performance of the ESN models improves. However, the models still underfit and are unable to capture underlying trend of the data, due to the observed anomalous behavior outside of the "anomalous" days as well as some regular behavior during the "anomalous" days. Similar behavior is observed in case of DDoS2020: large spikes in BGP announcements, announced NLRI prefixes, and other features are seen on a "regular" day on February, 22, 2020 (Fig. 4.4). However, no known anomaly was reported during that period. There is a difference when training the ESN models using datasets collected either from RIPE or Route Views. ESN5 is the best model to detect anomalies in these datasets.

The employed datasets influence the performance of ESN models. The ESN models evaluated using CIC-IDS datasets perform better than using BGP datasets. Note that CIC-IDS datasets are synthetically generated and contain records of various application layer

Table 6.6: Performance of ESN models when evaluated using resampled CIC-IDS2017 Wednesday, July 5, 2017 and CIC-CSE-IDS2018 Thursday, February 15, 2018 datasets

**CIC-IDS2017, Wednesday, July 5 2017**

| | Oversampling | | | Undersampling | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.926 | 0.930 | 0.127 | 0.925 | 0.929 | 0.135 |
| **ESN2** | 0.946 | 0.948 | 0.099 | 0.920 | 0.924 | 0.140 |
| **ESN3** | 0.911 | 0.917 | 0.159 | 0.818 | 0.840 | 0.321 |
| **ESN4** | 0.896 | 0.906 | 0.202 | 0.924 | 0.928 | 0.133 |
| **ESN5** | **0.971** | **0.972** | **0.052** | **0.960** | **0.960** | **0.074** |

**CIC-CSE-IDS2018, Thursday, February 15 2018**

| | Oversampling | | | Undersampling | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.981 | 0.982 | 0.035 | 0.970 | 0.971 | 0.059 |
| **ESN2** | 0.976 | 0.976 | 0.046 | 0.981 | 0.981 | 0.038 |
| **ESN3** | 0.891 | 0.902 | 0.215 | 0.837 | 0.860 | 0.322 |
| **ESN4** | 0.982 | 0.982 | 0.036 | 0.823 | 0.850 | 0.355 |
| **ESN5** | **0.990** | **0.991** | **0.018** | **0.988** | **0.989** | **0.022** |

Table 6.7: Number of total, regular, and anomalous points in Slammer, Nimda, and Code Red I datasets

| **Dataset** | **Class** | **Entire dataset** | **Training set** | **Test set** |
|---|---|---|---|---|
| **Slammer** | **Total** | **7,200** | **5,760** | **1,440** |
| | Regular | 6,331 | 5,058 | 1,273 |
| | Anomaly | 869 | 702 | 167 |
| **Nimda** | **Total** | **8,609** | **6,887** | **1,722** |
| | Regular | 7,308 | 5,841 | 1,467 |
| | Anomaly | 1,301 | 1,046 | 255 |
| **Code Red I** | **Total** | **7,200** | **5,760** | **1,440** |
| | Regular | 6,600 | 5,272 | 1,328 |
| | Anomaly | 600 | 488 | 112 |

Table 6.8: Performance of ESN models when evaluated using BGP datasets: Slammer, Nimda, and Code Red I

**Slammer**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.587 | 0.549 | 0.446 | 0.907 | 0.699 | 0.080 |
| **ESN2** | 0.625 | 0.654 | 0.366 | 0.908 | 0.710 | 0.083 |
| **ESN3** | 0.536 | 0.563 | 0.453 | **0.930** | **0.726** | **0.036** |
| **ESN4** | 0.505 | 0.524 | 0.471 | 0.927 | 0.712 | 0.036 |
| **ESN5** | **0.636** | **0.669** | **0.341** | 0.900 | 0.699 | 0.095 |

**Nimda**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.463 | 0.465 | 0.512 | 0.805 | 0.502 | 0.166 |
| **ESN2** | **0.507** | **0.529** | **0.473** | **0.821** | **0.470** | **0.130** |
| **ESN3** | 0.446 | 0.439 | 0.507 | 0.843 | 0.167 | 0.024 |
| **ESN4** | 0.436 | 0.433 | 0.514 | 0.841 | 0.122 | 0.021 |
| **ESN5** | 0.492 | 0.497 | 0.513 | 0.818 | 0.516 | 0.150 |

**Code Red I**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.619 | 0.612 | 0.331 | 0.910 | 0.432 | 0.040 |
| **ESN2** | 0.636 | 0.671 | 0.358 | **0.919** | **0.424** | **0.027** |
| **ESN3** | 0.678 | 0.700 | 0.270 | 0.913 | 0.046 | 0.002 |
| **ESN4** | **0.907** | **0.876** | **0.001** | 0.901 | 0.536 | 0.075 |
| **ESN5** | 0.598 | 0.605 | 0.401 | 0.910 | 0.547 | 0.062 |

Table 6.9: Number of total, regular, and anomalous points in DDoS2019 and DDoS2020 datasets

| Dataset | Class | Entire dataset | Training set | Test set |
|---|---|---|---|---|
| **DDoS2019** | **Total** | **10,080** | **6,048** | **4,032** |
| | Regular | 6,390 | 3,823 | 2,567 |
| | Anomaly | 3,690 | 2,225 | 1,465 |
| **DDoS2020** | **Total** | **10,080** | **8,064** | **2,016** |
| | Regular | 5,709 | 4,572 | 1,136 |
| | Anomaly | 4,371 | 3,492 | 880 |

protocols: HTTPS, HTTP, SMTP, POP3, IMAP, SSH, and FTP. Therefore, features such as distributions of packet sizes, number of packets per flow, certain patterns in the payload, and size of payload may provide more information about regular and anomalous network behavior for machine learning models. While Slammer, Nimda, and Code Red I as well as DDoS2019 and DDoS2020 datasets contain records of BGP protocol only, RIPE and Route Views BGP trace collection projects provide only estimates of AS-level Internet topologies. They collect routing updates from a smaller number of AS peers. Their view of Internet connectivity may be limited and including additional data from route servers and looking glasses may help capture more complete AS-level topology [97]. The labeling of the regular and anomaly data was based on the known time of the anomalies. However, indicated periods of anomaly may also contain the regular BGP traces that were categorized as anomalous. The CIC-IDS and BGP datasets that we consider consist of approximately $10^5 - 10^6$ and $10^3 - 10^4$ data points, respectively. Hence, training ESN models on larger datasets may improve their performance.

## 6.2 Comparing Performance of ESN and Bi-LSTM in Detecting the Denial of Service Attacks

We use PyTorch [141], an open-source Python-based scientific computing package, to create LSTM model. PyTorch, developed by Facebook's AI Research lab (FAIR), is employed for various applications including computer vision and natural language processing. Tensors are used in PyTorch to encode the model's parameters, inputs, and outputs. Tensors are a specialized data structure similar to arrays and matrices. PyTorch Tensors, unlike NumPy ndarrays, can run on GPUs. We use torch.nn that provides building blocks for any neural network that consists of various modules (layers). The nested structure allows for building neural network architectures of any complexity.

Our RNN contains: one bidirectional LSTM layer with input nodes = number of features and 16 output nodes, dropout rate 0.5, and batch size 10. The LSTM layer employs the ReLU activation function followed by fully-connected layer with 32 input and 2 output nodes. The last layer returns logits - raw values which are passed to the F.softmax module. The logits

Table 6.10: Performance of ESN models: BGP DDoS2019 and DDoS2020 datasets collected from RIPE and Route Views

**DDoS2019, RIPE**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | **0.622** | **0.621** | **0.351** | 0.571 | 0.502 | 0.465 |
| **ESN2** | 0.618 | 0.623 | 0.389 | 0.579 | 0.558 | 0.527 |
| **ESN3** | 0.549 | 0.546 | 0.398 | 0.481 | 0.522 | 0.702 |
| **ESN4** | 0.564 | 0.552 | 0.399 | 0.525 | 0.505 | 1.000 |
| **ESN5** | 0.602 | 0.611 | 0.361 | **0.677** | **0.617** | **0.371** |

**DDoS2019, Route Views**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.560 | 0.528 | 0.378 | 0.613 | 0.433 | 0.259 |
| **ESN2** | 0.555 | 0.587 | 0.374 | 0.611 | 0.551 | 0.406 |
| **ESN3** | 0.551 | 0.552 | 0.394 | 0.615 | 0.261 | 0.130 |
| **ESN4** | 0.526 | 0.528 | 0.399 | 0.624 | 0.193 | 0.084 |
| **ESN5** | **0.590** | **0.659** | **0.350** | **0.618** | **0.540** | **0.373** |

**DDoS2020, RIPE**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.520 | 0.506 | 0.452 | 0.439 | 0.610 | 0.988 |
| **ESN2** | 0.529 | 0.491 | 0.400 | 0.437 | 0.606 | 0.994 |
| **ESN3** | 0.529 | 0.491 | 0.390 | 0.437 | 0.607 | 0.998 |
| **ESN4** | 0.513 | 0.512 | 0.453 | 0.436 | 0.607 | 1.000 |
| **ESN5** | **0.539** | **0.536** | **0.444** | **0.453** | **0.610** | **0.955** |

**DDoS2020, Route Views**

| | Validation | | | Test | | |
|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Acc. | F-Score | FAR |
| **ESN1** | 0.513 | 0.491 | 0.400 | 0.477 | 0.609 | 0.877 |
| **ESN2** | 0.516 | 0.496 | 0.399 | 0.577 | 0.610 | 0.565 |
| **ESN3** | 0.508 | 0.483 | 0.410 | 0.437 | 0.603 | 0.982 |
| **ESN4** | 0.503 | 0.473 | 0.408 | 0.441 | 0.604 | 0.971 |
| **ESN5** | **0.553** | **0.554** | **0.413** | **0.595** | **0.621** | **0.536** |

Table 6.11: Performance of Bi-LSTM and ESN5 model based on accuracy, F-Score, false alarm rate (FAR), and training time

| | Bi-LSTM | | | | ESN5 | | | |
|---|---|---|---|---|---|---|---|---|
| | Acc. | F-Score | FAR | Time (s) | Acc. | F-Score | FAR | Time (s) |
| **CIC-IDS Datasets:** | | | | | | | | |
| CIC-IDS2017 | 0.995 | 0.994 | 0.002 | 2,200 | 0.973 | 0.965 | 0.038 | 988 |
| CSE-CIC-IDS2018, Thursday | 0.996 | 0.962 | 0.004 | 3,417 | 0.997 | 0.973 | 0.003 | 2,335 |
| CSE-CIC-IDS2018, Friday | 0.976 | 0.979 | 0.000 | 3,149 | 0.999 | 0.999 | 0.000 | 2,369 |
| CIC-DDoS2019 | 1.000 | 1.000 | 0.000 | 2,619 | 0.999 | 0.999 | 0.001 | 1,690 |
| **BGP Worm Datasets:** | | | | | | | | |
| Slammer | 0.958 | 0.827 | 0.024 | 34 | 0.900 | 0.699 | 0.095 | 8 |
| Nimda | 0.863 | 0.375 | 0.029 | 41 | 0.818 | 0.516 | 0.150 | 7 |
| Code Red I | 0.929 | 0.491 | 0.021 | 37 | 0.910 | 0.547 | 0.062 | 6 |
| **BGP DDoS Datasets:** | | | | | | | | |
| DDoS2019, RIPE | 0.388 | 0.478 | 0.837 | 111 | 0.677 | 0.617 | 0.371 | 12 |
| DDoS2019, Route Views | 0.654 | 0.791 | 1.000 | 99 | 0.618 | 0.540 | 0.373 | 6 |
| DDoS2020, RIPE | 0.346 | 0.514 | 1.000 | 107 | 0.453 | 0.610 | 0.955 | 9 |
| DDoS2020, Route Views | 0.760 | 0.864 | 1.000 | 101 | 0.595 | 0.621 | 0.536 | 11 |

are scaled to values [0, 1] representing the model's predicted probabilities for each class. A module called torch.optim allows implementation of various optimization algorithms used for building neural networks. Most of the commonly employed optimization methods are supported. We use torch.optim.Adam() to calculate the gradients and update the weights when training the model. The learning rate is a hyperparameter that controls how much to change the model in response to the evaluated loss each time the model weights are updated. Selecting a smaller learning rate would result in a slower training causing slow convergence. With a slow learning rate gradient descent may stuck in a local minimum. Large learning rate may result in learning a sub-optimal set of weights or an unstable training process (by diverging from the global minimum). When selecting the learning rate, we are trying various values such as $0.1, 0.01, 0.001$ uniformly sampled on a logarithmic scale and observing when the objective value (nn.CrossEntropyLoss()) would stop oscillating. With the selection of learning rate 0.001 the loss goes down considerably. After training the network and evaluating the loss, we include additional metrics such as accuracy, F-Score, false alarm rate, and training time.

The Bi-LSTM and ESN5 results are shown in Table 6.11. When evaluated using CIC-IDS datasets and BGP Nimda and Code Red I datasets, ESN and Bi-LSTM show comparable performance. When evaluated using BGP Slammer dataset and BGP DDoS2019 and DDoS2020 Route Views datasets, Bi-LSTM outperforms ESN. When evaluated using BGP DDoS2019 and DDoS2020 RIPE datasets, ESN slightly outperforms LSTM. The ESN training time is faster because ESN is not employing backpropagation. Low F-Score is produced when missing anomalies and classifying them as regular.

# Chapter 7

# Conclusions and Future Work

Over the past decades, the Internet has been subjected to various types of malicious intrusions such as DoS and DDoS. DoS and DDoS detection is becoming a challenging task due to changing network behavior and attacks patterns, especially when using classic intrusion detection methods. This Thesis applies machine learning techniques to detect DoS and DDoS attacks and to show that echo state networks (ESNs) is a feasible method. ESNs, a reservoir computing approach, help train RNNs.

We have selected synthetically generated datasets: CIC-IDS2017, CSE-CIC-IDS2018, and CIC-DDoS2019, that reflect the characteristics of diverse regular and current malicious network behavior. Synthetically generated datasets contain regular data samples and randomly added artificial anomalies. Datasets are often unbalanced and contain only a small portion of anomalous data points, which may affect the classification results. One approach to create a balanced dataset is to use oversampling in order to adjust the class distributions. We compare the performance of classification models using both unbalanced and balanced datasets. We also used data from deployed networks collected from public repositories Réseaux IP Européens (RIPE) and Route Views. We observed how recent large DDoS attacks are reflected in BGP traffic records. The results show that ESN models vary their performance depending on the employed datasets. Possible reasons why ESN models' performance is better with CIC-IDS datasets than with BGP datasets are that synthetically generated CIC-IDS datasets contain records of various application layer protocols: HTTPS, HTTP, SMTP, POP3, IMAP, SSH, and FTP. Additionally, the variety of features may provide more information of regular and anomalous behavior for machine learning models. BGP trace collector projects, RIPE and Route Views, do not provide entire AS-level Internet topologies data, and including additional data from route servers and looking glasses may help capture more complete AS-level topology. The classifier may also have been influenced by the labeling of regular and anomaly data as indicated periods of anomaly might contain the regular BGP traces that are categorized as anomalous. Finally, training machine learning models on datasets with larger size significantly improves their performance.

CIC-IDS datasets used in the Thesis are approximately in the order of $10^5 - 10^6$, compared to BGP datasets available in the order of $10^3$.

Even though k-fold cross-validation may be found problematic in terms of the computational cost (number of training runs is increased by k) and is not common for models that rely on time-series data, it may help generate a machine learning model by selecting optimal hyperparameters. The cross-validation process in ESN may still be computationally less expensive when compared to other models (those that employ backpropagation) because there is no need to rerun the entire network when training the model.

Performance of ESN depends on feature selection because selecting appropriate features decreases the redundancy and enhances the classification performance. The valuable features that are selected include standard deviation and mean of packet length, flow inter-arrival time related features, and control flags (ACK, SYN, URG, RST, PSH, FIN). They may be effective for machine learning classifiers because they reveal variations in packets' lengths, bursty behavior, and misused control bits that help detect anomalous traffic. We compare the performance of ESN to Bi-LSTM, an RNN approach, in order to examine the effectiveness of each approach in identifying network anomalies. Bi-LSTMs treat vanishing gradient and exploding gradient problems by efficiently coping with long-term dependencies and gaps in data. ESN and Bi-LSTM models evaluated in this Thesis demonstrate comparable accuracy, F-Score, and false alarm rates. The ESN training time is shorter because ESNs do not employ backpropagation.

Varying the reservoir configurations, we select echo state network models and observe the effect of hyperparameters. Increasing the number of processing nodes enhances the performance of echo state networks. Decreasing the radius of the reservoir slightly degrades the performance. Echo state networks involve a degree of uncertainty in tuning some of the hyperparameters by trial and error. Therefore, they are less commonly used compared to LSTM or CNN. They require further research to examine their advantages and constraints for identifying anomalies. The efficiency may be improved by further tuning the hyperparameters of the reservoir and experimenting with output training and feedbacks. Bi-directional (two-way) technique may be employed to capture dependencies in the data forward and backward in time by using a large untrained reservoir.

# Bibliography

[1] E. Chou and R. Groves, *Distributed Denial of Service (DDoS): Practical Detection and Defense.* 1st Ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[2] T. Wong, K. Law, J. Lui, and H. Wong, "An efficient distributed algorithm to identify and traceback DDoS traffic," *Comput. J.*, vol. 49, no. 6, pp. 418–442, Nov. 2006.

[3] M. Bhuyan, H. Kashyap, D. Bhattacharyya, and J. Kalita, "Detecting distributed denial of service attacks: methods, tools and future directions," *Comput. J.*, vol. 57, no. 4, pp. 537–556, Apr. 2014.

[4] F. Lau, S. H. Rubin, M. H. Smith, and Lj. Trajkovic, "Distributed denial of service attacks," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Nashville, TN, Oct. 2000, pp. 2275–2280.

[5] What was the largest DDoS attack of all time? [Online]. Available: https://www.cloudflare.com/learning/ddos/famous-ddos-attacks. Accessed: Apr. 07, 2021.

[6] Mapping Mirai: A botnet case study. [Online]. Available: https://www.malwaretech.com/2016/10/mapping-mirai-a-botnet-case-study.html. Accessed: Apr. 07, 2021.

[7] DDoS breach costs rise to over $2M for enterprises finds Kaspersky lab report. [Online]. Available: https://usa.kaspersky.com/about/press-releases/2018_ddos-breach-costs-rise-to-over-2m-for-enterprises-finds-kaspersky-lab-report. Accessed: Apr. 07, 2021.

[8] Cisco Annual Internet Report (2018–2023) White Paper. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed: Apr. 07, 2021.

[9] Kaspersky report finds over half of Q3 DDoS attacks occurred in September. [Online]. Available: https://usa.kaspersky.com/about/press-releases/2019_kaspersky-report-finds// over-half-of-q3-ddos-attacks-occurred-in-september. Accessed: Apr. 07, 2021.

[10] Corero Security: 2019 Half-Year DDoS Trends Report. [Online]. Available: https://www.corero.com/blog/infographic-2019-mid-year-ddos-trends-report/. Accessed: Apr. 07, 2021.

[11] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *ACM SIGCOMM Comput. Commun. Rev.*, 34, Apr. 2004, pp. 39–53.

[12] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: a survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, July 2009.

[13] Z. Zohrevand, U. Glasser, H. Y. Shahir, M. A. Tayebi, and R. Costanzo, "Hidden Markov based anomaly detection for water supply systems," in *Proc. 2016 IEEE Int. Conf. Big Data*, Washington, DC, Dec. 2016, pp. 1551–1560.

[14] Z. Zohrevand, U. Glässer, M. A. Tayebi, H. Yaghoubi Shahir, M. Shirmaleki, and A. Yaghoubi Shahir, "Deep learning based forecasting of critical infrastructure data," in *Proc. 2017 ACM Conf. Inf. Knowl. Manage. (CIKM '17)*, New York, NY, USA, Nov. 2017, pp. 1129–1138.

[15] D. E. Denning, "An Intrusion-Detection Model," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 2, pp. 222–232, Feb. 1987.

[16] P. Gogoi, D. K. Bhattacharyya, B. Borah, and J. K. Kalita, "A survey of outlier detection methods in network anomaly identification," *Comput. J.*, vol. 54, no. 4, pp. 570–588, Sept. 2011.

[17] T. M. Mitchell, *Machine Learning.* New York: McGraw-Hill, 1997.

[18] A. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, Oct. 1950.

[19] C. M. Bishop, *Pattern Recognition and Machine Learning.* Secaucus, NJ, USA: Springer-Verlag, 2006.

[20] P. Louridas and C. Ebert, "Machine learning," *IEEE Softw.*, vol. 33, no. 5, pp. 110–115, Sept./Oct. 2016.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 2018.

[22] W. Alhakami, A. Alharbi, S. Bourouis, R. Alroobaea, and N. Bouguila, "Network anomaly intrusion detection using a nonparametric Bayesian approach and feature selection," *IEEE Access*, vol. 7, pp. 52181–52190, Apr. 2019.

[23] M. Labonne, A. Olivereau, B. Polvé, and D. Zeghlache, "A cascade-structured meta-specialists approach for neural network-based intrusion detection," in *Proc. 16th IEEE Annu. Consumer Commun. Netw. Conf.*, Las Vegas, NV, USA, Jan. 2019, pp. 1–6.

[24] K. A. Taher, B. M. Y. Jisan, and M. M. Rahman, "Network intrusion detection using supervised machine learning technique with feature selection," in *Proc. Int. Conf. Robotics, Elect. Signal Process. Techn.*, Dhaka, Bangladesh, Jan. 2019, pp. 643–646.

[25] C. Yin, Y. Zhu, J. Fei, and X. He, "A deep learning approach for intrusion detection using recurrent neural networks," *IEEE Access*, vol. 5, pp. 21954–21961, 2017.

[26] S. Seufert and D. O'Brien, "Machine learning for automatic defence against distributed denial of service attacks,"in *Proc. IEEE Int. Conf. Commun.,* Glasgow, 2007, pp. 1217–1222.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* Cambridge, MA, USA: The MIT Press, 2016.

[28] K. P. Murphy, *Machine Learning: A Probabilistic Perspective.* Cambridge, MA, USA: The MIT Press, 2012.

[29] K. Cho, B. van Merriënboer, C. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translations," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Doha, Qatar, Oct. 2014, pp. 1724–1734.

[30] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: a search space odyssey," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017.

[31] G. Karatas, O. Demir, and O. K. Sahingoz, "Deep learning in intrusion detection systems," in *Proc. Int. Congr. Big Data, Deep Learn. Fighting Cyber Terrorism*, Ankara, Turkey, Dec. 2018, pp. 113–116.

[32] T. Kim, S. C. Suh, H. Kim, J. Kim, and J. Kim, "An encoding technique for CNN-based network anomaly detection," in *Proc. IEEE Int. Conf. Big Data*, Seattle, WA, USA, Dec. 2018, pp. 2960–2965.

[33] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, "Enhanced network anomaly detection based on deep neural networks," *IEEE Access*, vol. 6, pp. 48231–48246, Aug. 2018.

[34] Y. Jia, M. Wang, and Y. Wang, "Network intrusion detection algorithm based on deep neural network," *IET Inform. Secur.*, vol. 13, no. 1, pp. 48–53, Jan. 2019.

[35] KDD Cup 1999 Data [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. Accessed: Apr. 07, 2021.

[36] NSL-KDD Dataset [Online]. Available: https://www.unb.ca/cic/datasets/nsl.html. Accessed: Apr. 07, 2021.

[37] Q. Ding, Z. Li, S. Haeri, and Lj. Trajković, "Application of machine learning techniques to detecting anomalies in communication networks: datasets and feature selection algorithms," in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds., Berlin: Springer, 2018, pp. 47–70.

[38] Z. Li, Q. Ding, S. Haeri, and Lj. Trajković, "Application of machine learning techniques to detecting anomalies in communication networks: classification algorithms," in *Cyber Threat Intelligence*, M. Conti, A. Dehghantanha, and T. Dargahi, Eds., Berlin: Springer, 2018, pp. 71–92.

[39] P. Batta, Z. Li, and Lj. Trajković, "Evaluation of support vector machine kernels for detecting network anomalies," in *Proc. IEEE Int. Symp. Circuits Syst.*, Florence, Italy, May 2018, pp. 1–4.

[40] A. L. Gonzalez Rios, Z. Li, G. Xu, A. Diaz Alonso, and Lj. Trajković, "Detecting network anomalies and intrusions in communication networks," in *Proc. 23rd IEEE Int. Conf. Intell. Eng. Syst.*, Gödöllö, Hungary, Apr. 2019, pp. 29–34.

[41] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, “Deep recurrent neural network for intrusion detection in SDN-based networks,” in *Proc. 4th IEEE Conf. Netw. Softwarization Workshops (NetSoft),* Montreal, QC, Canada, 2018, pp. 202–206.

[42] Z. Ran, D. Zheng, Y. Lai, and L. Tian, “Applying stack bidirectional LSTM model to intrusion detection,” *Comput., Mater. & Continua*, vol. 65, no. 1, pp. 309–320, May 2020.

[43] C. L. P. Chen and Z. Liu, “Broad learning system: an effective and efficient incremental learning system without the need for deep architecture,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 1, pp. 10–24, Jan. 2018.

[44] C. L. P. Chen, Z. Liu, and S. Feng, “Universal approximation capability of broad learning system and its structural variations,” *IEEE Trans. Neural Netw. Learn. Syst.*, pp. 1–14, Sept. 2018.

[45] Z. Liu and C. L. P. Chen, “Broad learning system: structural extensions on single-layer and multi-layer neural networks,” in *Proc. Int. Conf. Secur., Pattern Anal., Cybern.*, Shenzhen, China, Dec. 2017, pp. 136–141.

[46] Z. Li, P. Batta, and Lj. Trajković, “Comparison of machine learning algorithms for detection of network intrusions,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Miyazaki, Japan, Oct. 2018, pp. 4248–4253.

[47] Z. Li, A. L. Gonzalez Rios, G. Xu, and Lj. Trajković, “Machine learning techniques for classifying network anomalies and intrusions,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Sapporo, Japan, May 2019.

[48] A. L. Gonzalez Rios, Z. Li, K. Bekshentayeva, and Lj. Trajkovic, “Detection of denial of service attacks in communication networks,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Seville, Spain, Oct. 2020.

[49] M. Chang, A. Terzis, and P. Bonnet, “Mote-based online anomaly detection using echo state networks,” *Distrib. Comput. in Sensor Syst.,* Berlin, Germany: Springer Berlin Heidelberg, 2009, vol. 5516, pp. 72–86.

[50] K. Bekshentayeva, M. Canute, Y. Kim, D. Lee, and A. Wong, “Network intrusion detection using various deep learning approaches,” Simon Fraser Univ., Burnaby, BC, Canada, CMPT 419_726 Project Rep., Dec. 2019.

[51] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, Dec. 1943.

[52] B. Widrow and M. E. Hoff, “Adaptive switching circuits,” in *Proc. IRE WESCON Conv. Rec., Part 4*, New York, 1960, pp. 96–104.

[53] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms,* Washington DC, USA: Spartan Books, 1962.

[54] J. Schmidhuber and D. Prelinger, “Discovering predictable classifications,” *Neural Comput.*, vol. 5, no. 4, pp. 625–635, July 1993.

[55] K. Li, Class lecture, Lecture 19 “Neural Networks: Backpropagation in MLPs” CMPT 726_419 Machine Learning, SFU School of Computing Science, Burnaby BC, Canada, March 24, 2021.

[56] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Apr. 1997.

[57] H. Jaeger, “The ‘echo state’ approach to analysing and training recurrent neural networks-with an erratum note,” German Nat. Res. Center for Inf. Technol. GMD, Bonn, Germany, Tech. Rep. 148, 2001.

[58] M. Lukoševičius, H. Jaeger, and B. Schrauwen, “Reservoir computing trends”, *KI. Künstliche Intelligenz*, vol. 26, no. 4, pp. 365–371, Nov. 2012.

[59] M. Lukoševičius, “Echo State Networks with trained feedbacks,” Jacobs Univ. Bremen, Jacobs University Tech. Rep. 4, 2007.

[60] P. Dominey, “Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning,” *Biol. Cybern.*, vol. 73, no. 6, pp. 265–274, Nov. 1995.

[61] P. Dominey and F. Ramus, “Neural network processing of natural language. Sensitivity to serial,temporal and abstract structure of language in the infant,” *Lang. Cogn. Processes*, vol. 15, no. 1, pp. 87–127, June 2000.

[62] H. Jaeger, M. Lukosevicius, D. Popovici, and U. Siewert, “Optimization and applications of echo state networks with leaky-integrator neurons,” *Neural Netw.*, vol. 20, no. 3, pp. 335–352, Dec. 2007.

[63] M. D. Skowronski and J. G. Harris, “Noise-robust automatic speech recognition using a discriminative echo state network,” in *Proc. IEEE Int. Symp. Circuits Syst.*, New Orleans, LA, USA, 2007, pp. 1771–1774.

[64] F. Triefenbach, A. Jalalvand, B. Schrauwen, and J. P. Martens, “Phoneme recognition with large hierarchical reservoirs,” in *Proc. Advances Neural Inf. Process. Syst.*, vol. 23, MIT Press, Cambridge, Dec. 2011, pp. 2307–2315.

[65] M. Salmen and P. Plöger, “Echo state networks used for motor control,” in *Proc. IEEE Int. Conf. Robotics Automation*, Barcelona, Spain, Apr. 2005, pp. 1953–1958.

[66] X. Lin, Z. Yang, and Y. Song, “Short-termstock price prediction based on echo state networks,” *Expert Syst. Appl.*, vol. 36, no. 3, pp. 7313–7317, Dec. 2009.

[67] X. Lin, Z. Yang, and Y. Song, “Intelligent stock trading system based on improved technical analysis and Echo State Network,” *Expert Syst. Appl.*, vol. 38, no. 9, pp. 11347–11354, Apr. 2011.

[68] J. Dan, W. Guo, W. Shi, B. Fang, and T. Zhang, “Deterministic Echo State Networks based stock price forecasting,” *Abstr. Appl. Anal.*, vol. 2014, pp. 1–6.

[69] Z. Liu, Z. Liu, Y. Song, Z. Gong, and H. Chen, “Predicting stock trend using multi-objective diversified Echo State Network,” in *Proc. 7th Int. Conf. Inf. Sci., Technol.*, Da Nang, 2017, pp. 181–186.

[70] P. Buteneers, D. Verstraeten, P. van Mierlo, T. Wyckhuys, D. Stroobandt, R. Raedt, H. Hallez, and B. Schrauwen, "Automatic detection of epileptic seizures on the intracranial electroencephalogram of rats using reservoir computing," *Artif. Intell. Med.*, vol. 53. no. 3, pp. 215–223, Sept. 2011.

[71] P. -J. Kindermans, P. Buteneers, D. Verstraeten, and B. Schrauwen, "An uncued brain-computer interface using reservoir computing," in *Proc. Workshop Mach. Learn. Assistive Technol.*, British Columbia, Canada, Dec. 2010.

[72] M. Lukosevicius, "A practical guide to applying Echo State Networks," in *Neural Networks: Tricks of the Trade (2nd ed.)*, G. Montavon, G. B.'Orr, and K. -R. Müller, Eds., Berlin, Heidelberg, Springer, 2012, vol. 7700, pp. 659–686.

[73] I. B. Yildiz, H. Jaeger, and S. J. Kiebel, "Re-visiting the echo state property," *Neural Netw.*, vol. 35, pp. 1–9, Nov. 2012.

[74] M. Chen, W. Saad, and C. Yin, "Echo State Networks for self-organizing resource allocation in LTE-U with uplink–downlink decoupling," *IEEE Trans. Wireless Commun.*, vol. 16, no. 1, pp. 3–16, Jan. 2017.

[75] A. Krušna and M. Lukosevicius, "Predicting Mozart's next note via Echo State Networks," in *Proc. Symp. Young Scientists Technol., Eng. Math.*, Gliwice, Poland, May 2018, pp. 84–91.

[76] M. Lukoševičius and V. Marozas, "Noninvasive fetal QRS detection using Echo State Network," in *Proc. Noninvasive Fetal ECG - PhysioNet Comp. in Cardiology Challenge*, Zaragoza, Spain, 2013, pp. 205–208.

[77] G. M. Rader, "A method for composing simple traditional music by computer," *Commun. ACM*, vol. 17, no. 11, pp. 631–638, 1974.

[78] H. Chu, R. Urtasun, and S. Fidler, "Song from PI: A musically plausible network for pop music generation." [Online]. Available: https://arxiv.org/abs/1611.03477H. Accessed: Apr. 07, 2021.

[79] A. Huang and R. Wu, "Deep learning for music," [Online] Available: https://arxiv.org/abs/1606.04930. Accessed: Apr. 07, 2021.

[80] M. Lukosevicius, D. Popovici, H. Jaeger, and U. Siewert, "Time Warping Invariant Echo State Networks," IRC-Library, Inf. Resour. Center der Jacobs Univ. Bremen, Jacobs University Tech. Rep. 2, 2006.

[81] M. Xu and M. Han, "Adaptive elastic Echo State Network for multivariate time series prediction," *IEEE Trans. Cybern.*, vol. 46, no. 10, pp. 2173–2183, Oct. 2016.

[82] B. Schrauwen B, M. D'Haene, D. Verstraeten, and D. Stroobandt, "Compact hardware liquid state machines on FPGA for real-time speech recognition," *Neural Netw.*, vol. 21, no. 2–3, pp. 511–523, Nov. 2008.

[83] K. Vandoorne, J. Dambre, D. Verstraeten, B. Schrauwen, and P. Bienstman, "Parallel reservoir computing using optical amplifiers," *IEEE Trans. Neural Netw.*, vol. 22, no. 9, pp. 1469–1481, Sept. 2011.

[84] A. Z. Stieg, A. V. Avizienis, H. O. Sillin, C. Martin-Olmos, M. Aono, and J. K. Gimzewski, “Emergent criticality in complex Turing B-type atomic switch networks,” *Adv. Mater*, vol. 24, no. 2, pp. 286–293, Nov. 2012.

[85] L. Larger, M. C. Soriano, D. Brunner, L. Appeltant, J. M. Gutierrez, L. Pesquera, C. R. Mirasso, and I. Fischer, “Photonic information processing beyond Turing: an optoelectronic implementation of reservoir computing,” *Opt. Express*, vol. 20, pp. 3241–3249, 2012.

[86] L. Maciel, F. Gomide, D. Santos, and R. Ballini, “Exchange rate forecasting using echo state networks for trading strategies,” in *Proc. Comput. Intell. Fin. Eng. Econ. Conf.*, London, UK, Mar. 2014, pp. 40–47.

[87] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, “Towards a reliable intrusion detection benchmark dataset,” *Softw. Netw.*, vol. 2017, no. 1, pp. 177–200, July 2017.

[88] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, “Toward generating a new intrusion detection dataset and intrusion traffic characterization,” in *Proc. 4th Int. Conf. Inform. Syst. Secur. Privacy*, Funchal, Portugal, Jan. 2018, pp. 108–116.

[89] I. Sharafaldin, A. H. Lashkari, S. Hakak and A. A. Ghorbani, ”Developing realistic distributed denial of service (DDoS) attack dataset and taxonomy,“ in *Proc. Int. Carnahan Conf. Secur. Technol. (ICCST),* CHENNAI, India, 2019, pp. 1–8.

[90] Intrusion Detection Evaluation dataset (CIC-IDS2017). [Online]. Available: https://www.unb.ca/cic/datasets/ids-2017.html. Accessed: Apr. 07, 2021.

[91] A Realistic Cyber Defense dataset (CSE-CIC-IDS2018). [Online]. Available: https://registry.opendata.aws/cse-cic-ids2018/. Accessed: Apr. 07, 2021.

[92] DDoS Evaluation Dataset (CICDDoS2019). [Online]. Available: https://www.unb.ca/cic/datasets/ddos-2019.html. Accessed: Apr. 07, 2021.

[93] CICFlowMeter. [Online]. Available: http://netflowmeter.ca/ netflowmeter.html. Accessed: Apr. 07, 2021.

[94] Y. Rekhter and T. Li, “A Border Gateway Protocol 4 (BGP-4),” RFC 1771, IETF, Mar. 1995. [Online]. Available: http://tools.ietf.org/rfc/rfc1771.txt [Jan. 2021].

[95] D. P. Watson and D. H. Scheidt, “Autonomous systems,” Johns Hopkins APL Technical Digest, vol. 26, no. 4, pp. 368–376, Oct.–Dec. 2005.

[96] D. Dolev, S. Jamin, O. Mokryn and Y. Shavitt, “Internet resiliency to attacks and failures under BGP policy routing,” *Comp. Netw.*, vol. 50, no. 16, pp. 3183–3196, Jan. 2006.

[97] B. Zhang, R. Liu, D. Massey, and L. Zhang, “Collecting the Internet ASlevel topology,” *ACM Computer Communication Review*, vol. 35, no. 1, pp. 53–62, Jan. 2005.

[98] RIPE NCC: RIPE Network Coordination Center. [Online]. Available: http://www.ripe.net/data-tools/stats/ris/ris-raw-data. Accessed: Apr. 07, 2021.

[99] University of Oregon Route Views project. [Online]. Available: http://www.routeviews.org. Accessed: Apr. 07, 2021.

[100] Center for Applied Internet Data Analysis. The Spread of the Sapphire/Slammer Worm [Online]. Available: http://www.caida.org/publications/papers/2003/sapphire/. Accessed: Apr. 07, 2021.

[101] Sans Institute. Nimda worm — why is it different? [Online]. Available: http://www.sans.org/reading-room/whitepapers/malicious/nimda-worm-different-98. Accessed: Apr. 07, 2021.

[102] Sans Institute. The mechanisms and effects of the Code Red worm. [Online]. Available: https://www.sans.org/reading-room/whitepapers/dlp/mechanisms-effects-code-red-worm-87. Accessed: Apr. 07, 2021.

[103] MRT routing information export format. [Online]. Available: http://tools.ietf.org/html/draft-ietf-grow-mrt-13. Accessed: Apr. 07, 2021.

[104] Zebra-dump-parser. [Online]. Available: https://github.com/ rfc1036/zebra-dump-parser. Accessed: Apr. 07, 2021.

[105] BGP C-Sharp Tool [Online]. Available: https://github.com/communication-networks-laboratory/BGP_c_sharp_tool. Accessed: Apr. 07, 2021.

[106] Institute for Critical Infrastructure Technology: Rise of the machines. [Online]. Available: https://icitech.org/wp-content/uploads/2016/12/ICIT-Brief-Rise-of-the-Machines.pdf. Accessed: Apr. 07, 2021.

[107] DDoS attack: How AWS going down affected business. [Online]. Available: https://www.panopta.com/resources/ddos-attack-on-amazon/. Accessed: Apr. 07, 2021.

[108] Learning from the Amazon Web Services (AWS) DDoS attack. [Online]. Available: https://www.corero.com/blog/learning-from-the-amazon-web-services-aws-ddos-attack/. Accessed: Apr. 07, 2021.

[109] AWS Shield. Managed DDoS protection. [Online]. Available: https://aws.amazon.com/shield/. Accessed: Apr. 07, 2021.

[110] South African banks resilient in the face of latest DDoS attacks. [Online]. Available: https://www.sabric.co.za/media-and-news/press-releases/south-african-banks-resilient-in-the-face-of-latest-ddos-attacks/. Accessed: Apr. 07, 2021.

[111] AWS shield threat landscape report - Q1 2020. [Online]. Available: https://aws-shield-tlr.s3.amazonaws.com/2020-Q1_AWS_Shield_TLR.pdf. Accessed: Apr. 07, 2021.

[112] DDoS attack on AWS sets a new size record. [Online]. Available: https://www.corero.com/blog/ddos-attack-on-aws-sets-a-new-size-record/. Accessed: Apr. 07, 2021.

[113] D. Meyer, "BGP communities for data collection," RFC 4384, IETF, Feb. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4384.txt. Accessed: Apr. 07, 2021.

[114] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. F. Wu, and L. Zhang, “Observation and analysis of BGP behavior under stress,” in *Proc. 2nd Workshop on Internet Meas.*, New York, NY, USA, 2002, pp. 183–195.

[115] S. Deshpande, M. Thottan, T. K. Ho, and B. Sikdar, “An online mechanism for BGP instability detection and analysis,” *IEEE Trans. Comput.*, vol. 58, no. 11, pp. 1470–1484, Nov. 2009.

[116] D. Blazakis and J. S. Baras, “Analyzing BGP ASPATH behavior in the Internet,” in *Proc., 9th IEEE Glob. Int. Symp.,* Barcelona, Spain, Apr. 2006.

[117] T. Shorey, D. Subbaiah, A. Goyal, A. Sakxena, and A. K. Mishra, “Performance comparison and analysis of Slowloris, GoldenEye and Xerxes DDoSattack tools,” in *Proc. Int. Conf. Adv. Comput. Commun. Inform.*, Bangalore, India, Sept. 2018, pp. 318–322.

[118] J. Sermersheim, “Lightweight Directory Access Protocol (LDAP): The Protocol,” IETF, RFC 4511, June 2006. Available: https://www.ietf.org/rfc/rfc4511.txt. Accessed: Apr. 07, 2021.

[119] Directory Services LDAP [Online]. Available:
https://docs.oracle.com/cd/A87860_01/doc/ois.817/a83729/adois09.html.
Accessed: Apr. 07, 2021.

[120] D. L. Mills, “Network Time Protocol (NTP),” IETF, RFC 958, Sept. 1985. Available: https://tools.ietf.org/html/rfc958. Accessed: Apr. 07, 2021.

[121] J. Woo, J. Song, and Y. Choi, “Performance enhancement of deep neural network using feature selection and preprocessing for intrusion detection,” in *Proc. Int. Conf. Artif. Intell. Inform. Commun.*, Okinawa, Japan, Feb. 2019, pp. 415–417.

[122] H. U. Kobialka and U. Kayani, 2010) “Echo State Networks with Sparse Output Connections,” in *Artif. Neural Netw. – 2010. Lecture Notes in Computer Science* K. Diamantaras, W. Duch, and L. S. Iliadis, Eds., ICANN 2010, Berlin, Heidelberg, Springer, 2010, vol. 6352.

[123] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006.

[124] Sklearn.ensemble.ExtraTreesClassifier. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTrees
Classifier.html. Accessed: Apr. 07, 2021.

[125] C. R. Harris, et al., “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sept. 2020.

[126] W. McKinney, “Data structures for statistical computing in Python,” in *Proc. 9th Python in Sci. Conf.*, Austin, TX, USA, 2010, pp. 51–56.

[127] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al., “Scikit-learn: machine learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825-2830, Jan. 2011.

[128] F. M. Bianchi, S. Scardapane, S. Løkse, and R. Jenssen, "Reservoir computing approaches for representation and classification of multivariate time series," *IEEE Trans. Neural Netw. Learn. Syst.* pp. 1–11, June 2020.

[129] G. Van Rossum, *The Python Library Reference, release 3.8.2.* Python Softw. Found., 2020.

[130] H. He and Y. Ma, *Imbalanced Learning: Foundations, Algorithms, and Applications.* Hoboken, New Jersey, USA: Wiley-IEEE Press, 2013.

[131] Resampling strategies for imbalanced datasets. [Online]. Available: https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets#t1. Accessed: Apr. 07, 2021.

[132] C. M. Bishop, "Training with noise is equivalent to Tikhonov regularization," *Neural Comput.*, vol. 7, no. 1, pp. 108–116, Jan. 1995.

[133] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-hevel performance in face verification," in *Proc. IEEE Conf. on Comp. Vision and Pattern Recogn.*, Columbus, OH, USA, June 2014, pp. 1701–1708.

[134] A. Rodan and P. Tino, "Minimum complexity echo state network," *IEEE Trans. on Neural Netw.*, vol. 22, no. 1, pp. 131–144, Jan. 2011.

[135] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," [Online]. Available: https://arxiv.org/pdf/1207.0580.pdf. Accessed: Apr. 07, 2021.

[136] M. Lukoševičius and A. Uselis, "Efficient implementations of echo state network cross-validation," [Online]. Available: https://arxiv.org/abs/2006.11282. Accessed: Apr. 07, 2021.

[137] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[138] M. Lamons, R. Kumar, and A. Nagaraja, *Python Deep Learning Projects*, Packt Publishing, 2018. [E-book] Available: O'Reilly Online Learning (formerly Safari Books Online).

[139] Understanding LSTM Networks. [Online].
Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.
Accessed: Apr. 07, 2021.

[140] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[141] PyTorch. [Online]. Available: https://pytorch.org/docs/stable/nn.html. Accessed: May 31, 2021.

# Appendix A

# Figures: CIC-IDS and BGP Datasets

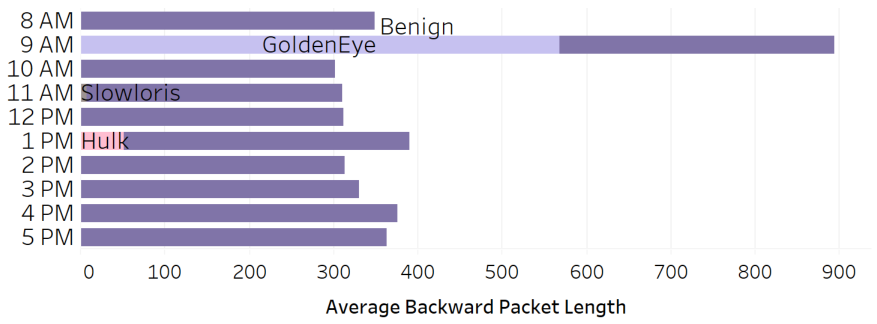


Figure A.1: CSE-CIC-IDS2018: Average backward packet length. The GoldenEye packet length is comparable to benign traffic.

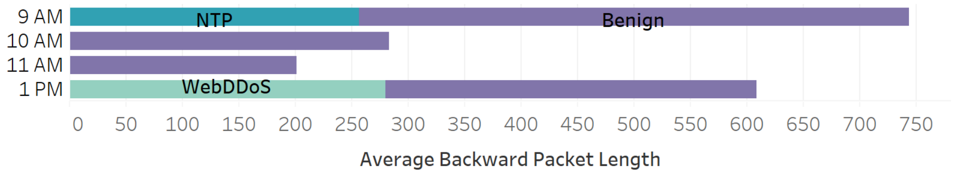


Figure A.2: CIC-DDoS2019: Average backward packet length. The length of NTP and Web-DDoS packets is smaller or comparable to benign traffic. The length of other attacks packets is negligibly small.

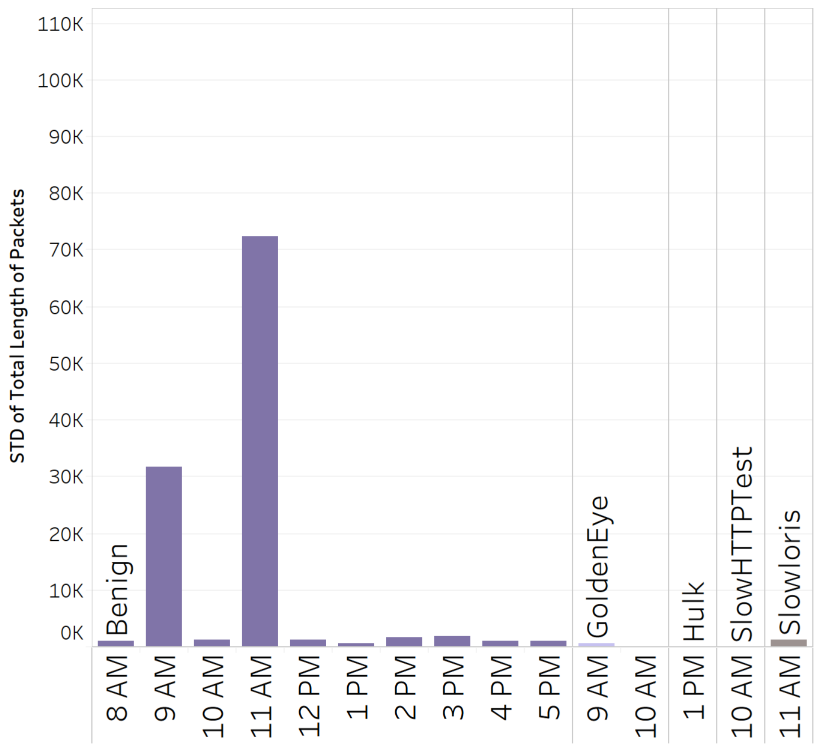


Figure A.3: CSE-CIC-IDS2018 dataset: Standard deviation of total length of packets. While benign packets usually have high variation in length, Slowloris attack also exhibits high standard deviation.

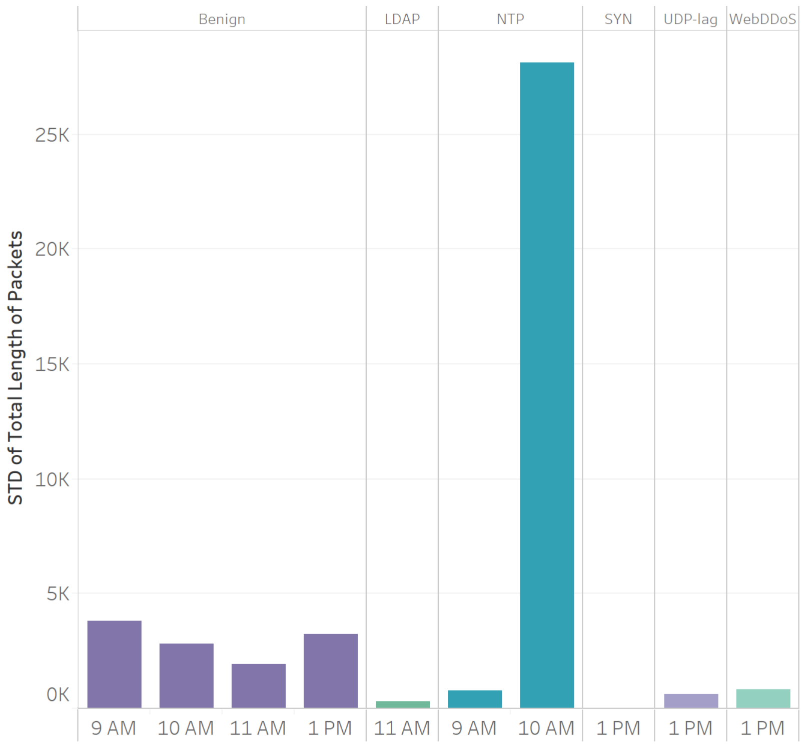


Figure A.4: CIC-DDoS2019: Standard deviation of packet length. NTP attack packet length has higher standard deviation.

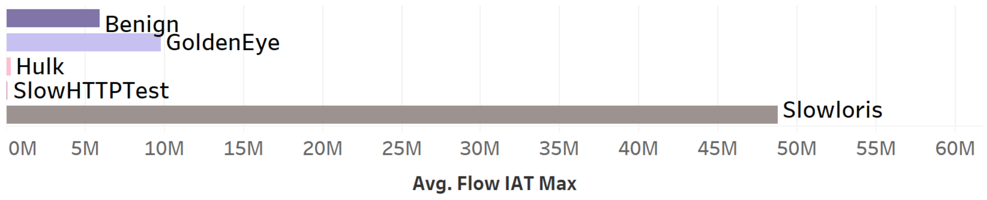


Figure A.5: CSE-CIC-IDS2018: While GoldenEye and Slowloris keep comparable IAT to 2017, Hulk and SlowHTTPTest have lower flow IATs in case of the CSE-CIC-IDS2018 dataset.

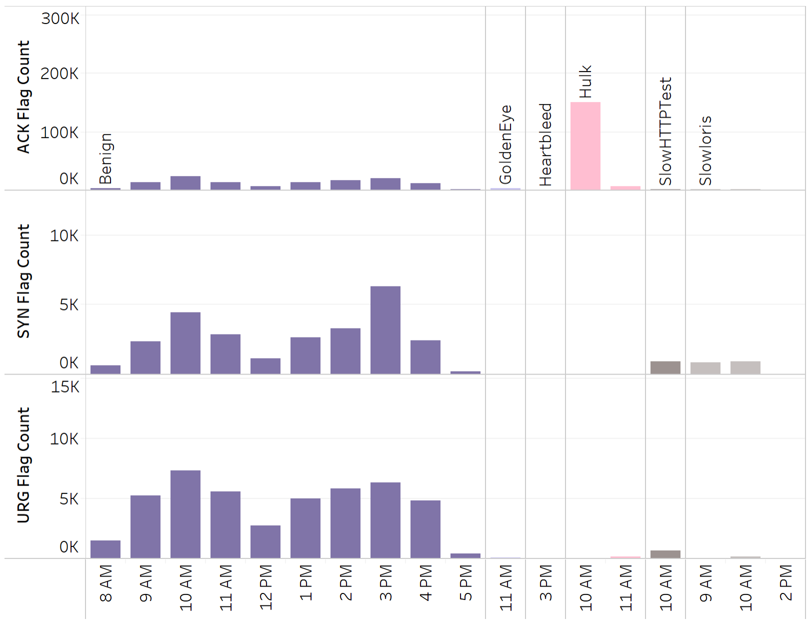


Figure A.6: CIC-IDS2017: ACK, SYN, and URG flag counts. Hulk attack overwhelms the server with ACK packets. Other attacks are not employing TCP control bits for malicious purposes.

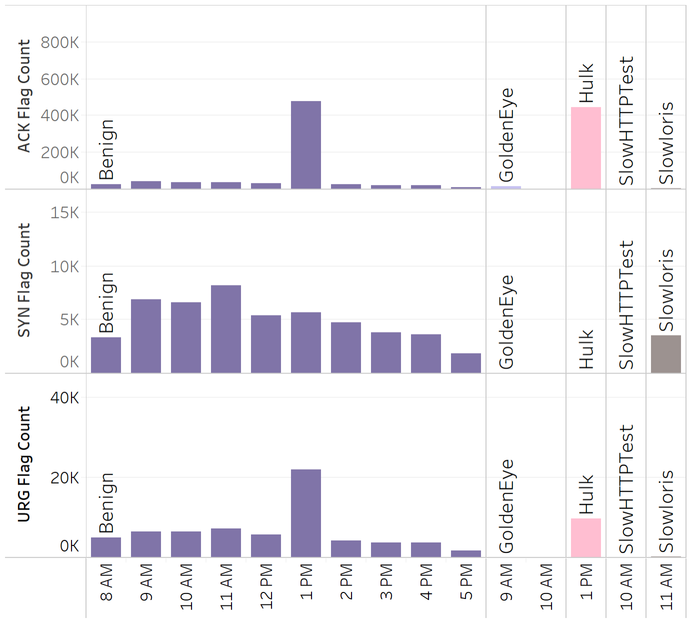
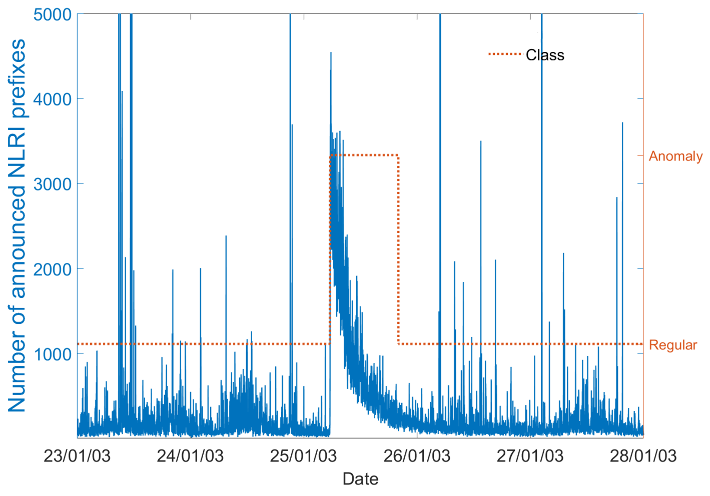
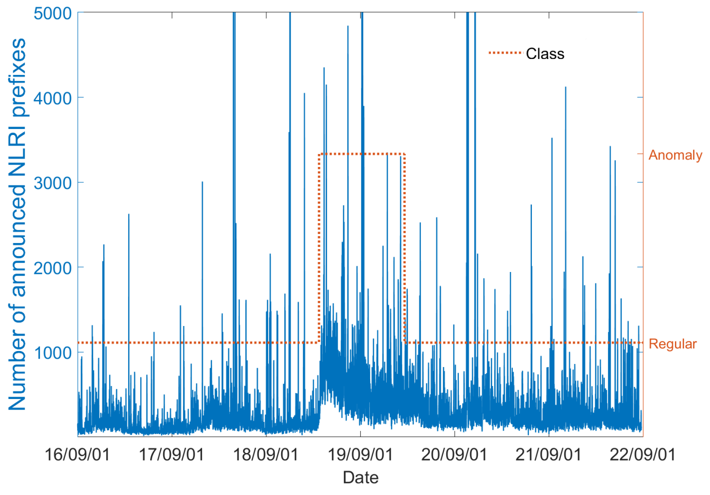


Figure A.7: CSE-CIC-IDS2018: ACK, SYN, and URG flag counts. A large amount of ACK and URG packets is observed at 1 PM.





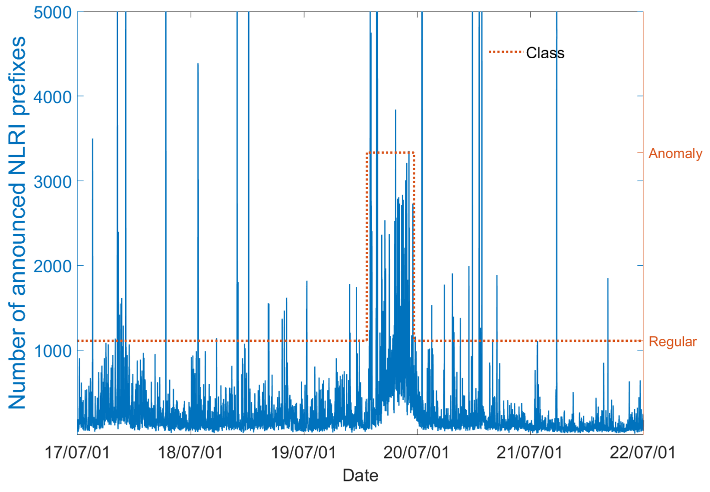
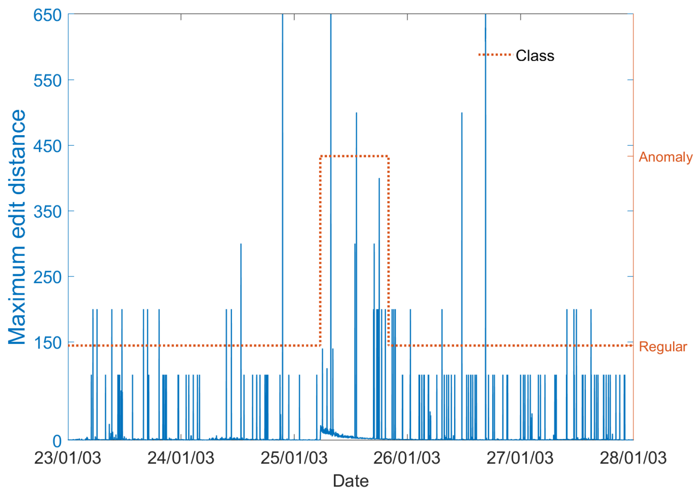
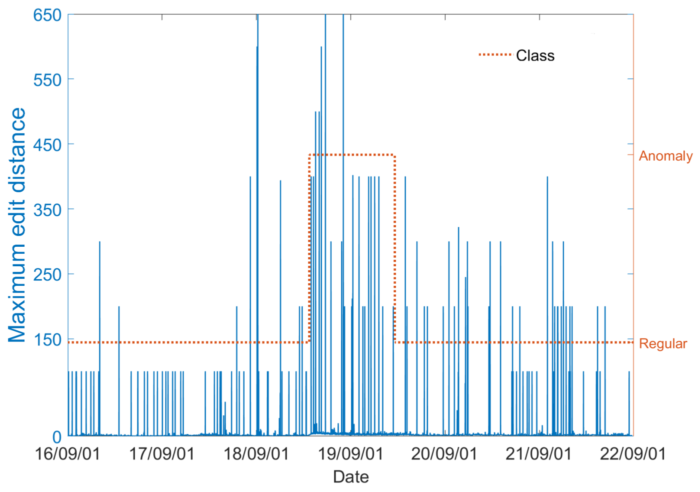


Figure A.8: Slammer (top), Nimda (middle), and Code Red I (bottom): Number of announced NLRI (Network Layer Reachability Information) prefixes. The red dotted line indicates two classes: regular and anomaly.





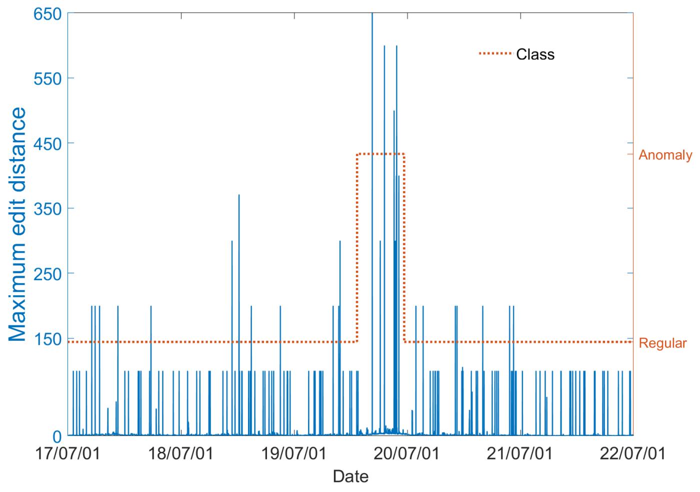
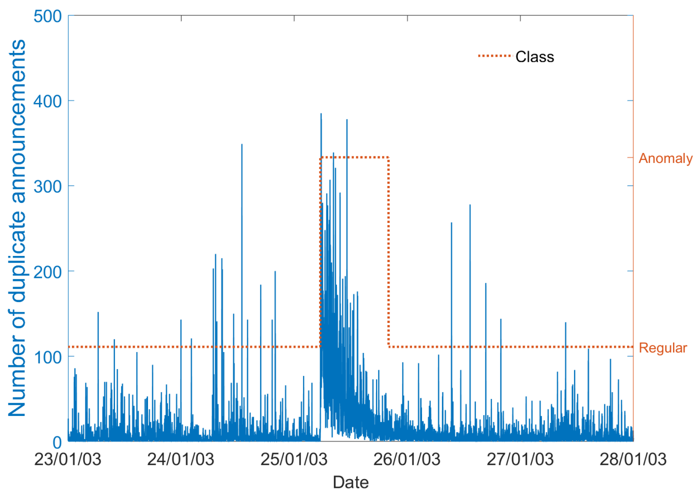
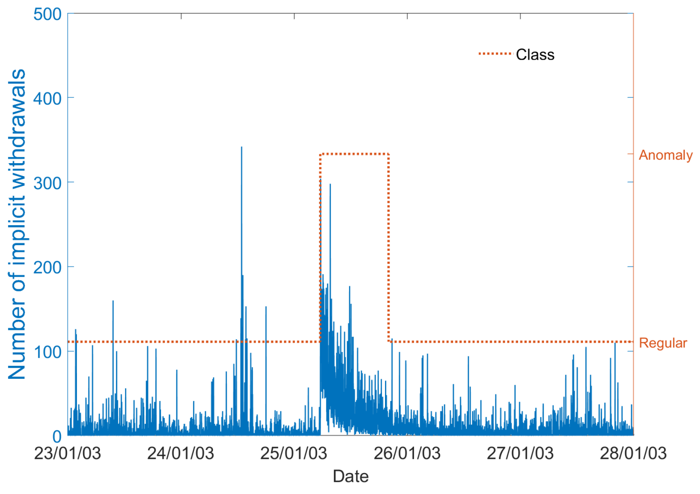


Figure A.9: Slammer (top), Nimda (middle), and Code Red I (bottom): Maximum edit distance. The red dotted line indicates two classes: regular and anomaly.





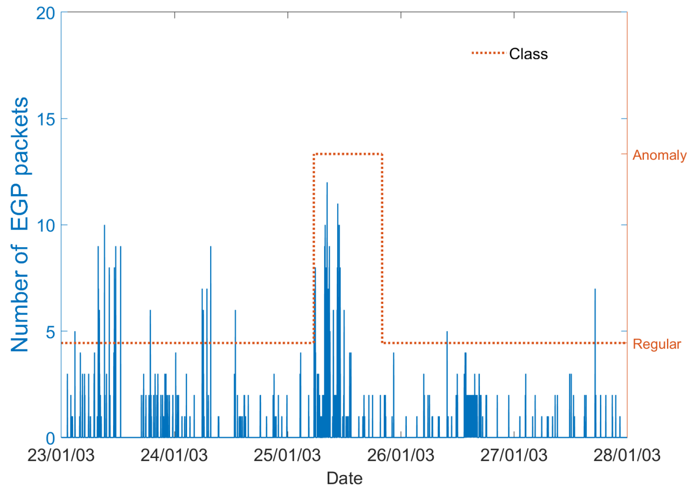
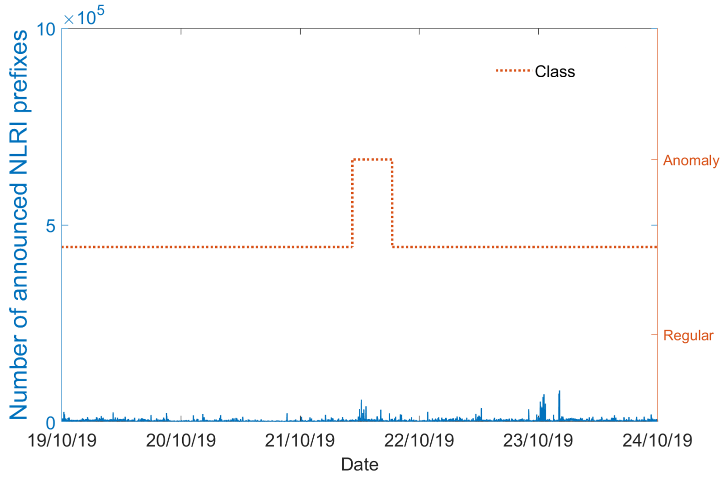
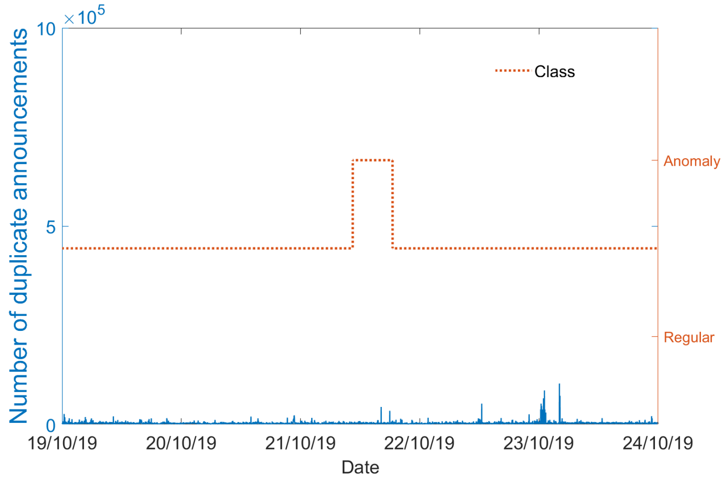


Figure A.10: Slammer: Number of duplicate announcements (top), number of implicit withdrawals (middle), and number of EGP packets (bottom). The red dotted line indicates two classes: regular and anomaly.





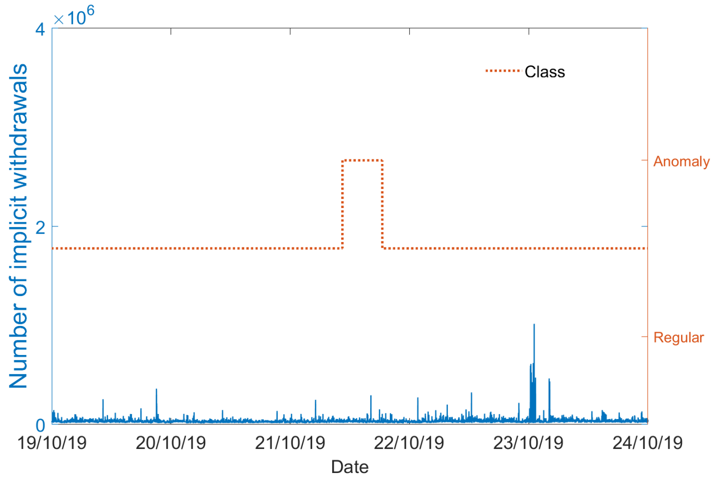


Figure A.11: DDoS2019-v1: Number of announced NLRI prefixes (top), number of duplicate announcements (middle), and number of implicit withdrawals (bottom). The red dotted line indicates two classes: regular and anomaly.

# Appendix B

# Scripts for Detecting Anomalies Using CIC-IDS2017, CSE-CIC-IDS2018, CIC-DDoS2019, and BGP Datasets

**Script 1. Read a .csv file, count instances in each class, oversample to create a balanced dataset (skip oversampling if no need to create balanced datasets), save a new .csv**

```
# Import libraries:
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('Wednesday2017raw.csv') # open '... .csv'
df.head() # output header, 5 first rows, and all columns

df[" Label"].value_counts() # count instances in each class

# Create a target vector:
def targetify(s):
    if s == 'BENIGN':
        return 0
    else:
        return 1

df['Target']=df[' Label'].apply(targetify)

# Output amount of instances in each class, create a bar plot:
target_count = df.target.value_counts()
print('Class 0:', target_count[0])
print('Class 1:', target_count[1])
print('Proportion:', round(target_count[0] / target_count[1], 2), ': 1')
target_count.plot(kind='bar', title='Count (target)');

# Divide by class:
df_class_0 = df[df['target'] == 0]
df_class_1 = df[df['target'] == 1]

# Oversample the minority class:
df_class_1_over = df_class_1.sample(count_class_1, replace=True)
df_test_over = pd.concat([df_class_0, df_class_1_over], axis=0)

# Output the results of oversampling with a bar plot:
print('Oversampling:')
print(df_test_over.target.value_counts())
df_test_over.target.value_counts()
.plot(kind='bar', title='Count (target)');
df_test_over[" Label"].value_counts()

# Shuffle, trim, and save a new .csv file:
df = df_test_over.sample(frac=1)
df.head(800000).to_csv('oversampled_....csv') # save 800,000 points
dfs = pd.read_csv('oversampled_CIC-IDS2017_Wed.csv')
dfs.head()
dfs[" Label"].value_counts()
```

**Script 2. Select 20 most important features**

```
# Import libraries:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
from sklearn import preprocessing
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split

df1 = pd.read_csv('oversampled_CIC-IDS2017_Wed.csv')
# Create a new feature "BiFlowsCount" and sort by "Timestamp":
df2 = df1.groupby([' Timestamp'])[' Flow Duration'].count()
df2 = pd.DataFrame(df2).reset_index()
df2.columns=[' Timestamp','BiFlowsCount']
df = df1.merge(df2, left_on=' Timestamp', right_on=' Timestamp')
df = df.sort_values(' Timestamp')
df.columns # output column names (features)

# List all column names except target:
features = ['Flow ID', ' Source IP',...]
```

```
# Define X and convert all categorical values to numeric:
X = df[features]
X[features] = X[features].apply(pd.to_numeric, errors='coerce', axis=1)

y = df['Target'] # define y

X = X.fillna(0) # fill "not a number" with 0 in X

# Split data into train and test sets and print shape of each set:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print (X_train.shape, y_train.shape)
print( X_test.shape, y_test.shape)

# Features ranking:
forest = ExtraTreesClassifier(n_estimators=100, random_state=2)
forest.fit(X_train, y_train)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in
forest.estimators_],
axis=0)
indices = np.argsort(importances)[::-1]

print("Feature ranking:")
for f in range(X_train.shape[1]):
    print("%d. feature %d: %s (%f)" %
    (f + 1, indices[f], X_train.columns[indices[f]],
    importances[indices[f]]))

# Plot the top x feature importance of the forest:
top_x = 20
plt.figure()
plt.bar(range(X_train.shape[1]), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(top_x), indices)
plt.xlim([-1, top_x])
plt.show()

plt = pd.Series(forest.feature_importances_,
index=X_train.columns).nlargest(top_x).plot(kind='barh')
fig = plt.get_figure()
fig.savefig('figure.jpg', bbox_inches = "tight")
```

**Script 3. Apply echo state networks [72, 128]**

```
# Import libraries:
import numpy as np
import pandas as pd
import scipy.io
from scipy import sparse
from sklearn.decomposition import PCA
from sklearn.linear_model import Ridge
from sklearn.metrics import accuracy_score, f1_score
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import math

# Create a class ESN_Reservoir with the hyperparameters:
    proc_nodes, s_radius, leak, connectivity, inp_scale, noise,
    deterministic_res:
class ESN_Reservoir(object):
def __init__(self, proc_nodes=30, s_radius=0.9, leak=None,
             connectivity=0.25, inp_scale=0.3, noise=0.01,
             deterministic_res=False):

# Initialization:
self._proc_nodes = proc_nodes
self._inp_scale = inp_scale
self._noise = noise
self._leak = leak

# Input weights rely on the size of input data
    (they are configured when data is given):
self._input_W = None

# Generate reservoir weights:
if deterministic_res:
    self._reservoir_W =
    self._initialize_reservoir_W_Deterministic(proc_nodes, s_radius)
else:
    self._reservoir_W =
    self._initialize_reservoir_W(proc_nodes, connectivity, s_radius)

def _initialize_reservoir_W_Deterministic(self, proc_nodes, s_radius):
    reservoir_W = np.zeros((proc_nodes, proc_nodes))
    reservoir_W[0,-1] = s_radius
    for i in range(proc_nodes-1):
        reservoir_W[i+1,i] = s_radius
return reservoir_W

def _initialize_reservoir_W(self, proc_nodes, connectivity, s_radius):
# Generate sparse, uniformly distributed reservoir weights:
    np.random.seed(args.seed)
    reservoir_W = sparse.rand(proc_nodes, proc_nodes,
    density=connectivity).todense()
```

```
# Distribute the nonzero values between [−0.5, 0.5]:
    reservoir_W[np.where(reservoir_W > 0)] −= 0.5
# Calculate the radius of the reservoir:
    E, _ = np.linalg.eig(reservoir_W)
    e_max = np.max(np.abs(E))
    reservoir_W /= np.abs(e_max)/s_radius
return reservoir_W

# (4) Compute past state:
def _design_matrix(self, X, n_drop=0):
    N, T, _ = X.shape
    past_state = np.zeros((N, self._proc_nodes), dtype=float)

# (5) Generate state matrix with the size [N, T−n_drop, _proc_nodes]:
design_matrix =
np.empty((N, T − n_drop, self._proc_nodes), dtype=float)
    for t in range(T):
        current_input = X[:, t, :]
        # State equation:
        state =
        self._reservoir_W.dot(past_state.T)
        + self._input_W.dot(current_input.T)
        # Apply noise:
        np.random.seed(args.seed)
        state += np.random.rand(self._proc_nodes, N)*self._noise
        # Nonlinearity and leakage may be included:
            if self._leak is None:
                past_state = np.tanh(state).T
            else:
                past_state =
                (1.0 − self._leak)*past_state + np.tanh(state).T
        # Store everything after the dropout period
            if (t > n_drop − 1):
                design_matrix[:, t − n_drop, :] = past_state
    return design_matrix

# (2) Compute the input weights:
def reservoir_states(self, X, n_drop=0):
    N, T, V = X.shape
    np.random.seed(args.seed)
    if self._input_W is None:
        self._input_W =
        (2.0*np.random.binomial(1, 0.5 ,
        [self._proc_nodes, V]) − 1.0)*self._inp_scale

# (3) Compute the forward state matrix:
states = self._design_matrix(X, n_drop)
```

```
return states

def Embed_Reservoir(self, X, pca, ridge_embedding,  n_drop=0,
        test = False):

# (1) Obtain the initial reservoir states:
res_states = self.reservoir_states(X, n_drop=0)

# (6) Reshape the initial state matrix res_states from
    [N, T-n_drop, _proc_nodes] to res_states[N*(T-n_drop),
    _proc_nodes]:
N_samples = res_states.shape[0]
res_states = res_states.reshape(-1, res_states.shape[2])

# (7) PCA:
if test:
    red_states = pca.transform(res_states)
else:
    red_states = pca.fit_transform(res_states)
    red_states = red_states.reshape(N_samples,-1,red_states.shape[1])
    print("red_states:" + str(red_states.shape))

# Ridge embedding:
coeff_tr = []
biases_tr = []
for i in range(X.shape[0]):
    ridge_embedding.fit(red_states[i, 0:-1, :],
    red_states[i, 1:, :])
    coeff_tr.append(ridge_embedding.coef_.ravel())
    biases_tr.append(ridge_embedding.intercept_.ravel())
    #print(np.array(coeff_tr).shape,np.array(biases_tr).shape)
    input_repr = np.concatenate((np.vstack(coeff_tr),
    np.vstack(biases_tr)), axis=1)
return input_repr

# Dataset and features:
df1 = pd.read_csv('oversampled_CIC-IDS2017_Wed.csv') # Open a .csv file
df2 = df1.groupby([' Timestamp'])[' Flow Duration'].count()
df2 = pd.DataFrame(df2).reset_index()
df2.columns=[' Timestamp','BiFlowsCount']
df = df1.merge(df2, left_on=' Timestamp', right_on=' Timestamp')
df = df.sort_values(' Timestamp')
num_features = 20 # Change the number of features here
# Input ranked by importance features:
features_wed = ['...']
features = features_wed[0:num_features]
# Select a fraction of a dataset:
```

```
fraction = 0.5
print(str(num_features) + " features")
print("fraction:" + str(fraction))
data = df.sample(frac=fraction, replace=True, random_state=1)

# Get X and y. Normalize X and 3D-reshape for reservoir:
num_col = data.shape[1]
num_row = data.shape[0]
X_data = data[features]
X_data[features] = X_data[features].apply(pd.to_numeric,
     errors='coerce', axis=1)
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(X_data.values)
X = np.nan_to_num(x_scaled)
if len(X.shape) < 3:
    X = np.atleast_3d(X)

def targetify(s):
    if s == 'BENIGN':
        return 0
    else:
        return 1
y = data[' Label'].apply(targetify)
print("Finished loading X and y......")

# Split into training (80%) and test data (20%):
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.2, random_state=100)
print("X_train shape:" + str(X_train.shape),
    "y_train shape:" + str(y_train.shape))
print("X_test shape:" + str(X_test.shape),
    "y_test shape:" + str(y_test.shape))

pca = PCA() # linear dimensionality reduction
ridge_embedding = Ridge(alpha=10, fit_intercept=True)
readout = Ridge(alpha=5)

# Change the number of processing (internal) nodes:
n = 30
print(str(n) + " processing nodes")

# Run through reservoir:
res = ESN_Reservoir(proc_nodes=n, s_radius=0.9, leak=0.2,
     connectivity=0.25, inp_scale=0.3, noise=0.01, deterministic_res=False)
input_repr = res.Embed_Reservoir(np.array(X_train), pca, ridge_embedding,
n_drop=0, test = False)
print("Finished loading training reservoir embedding......")
```

```
input_repr_te = res.Embed_Reservoir(np.array(X_test), pca, ridge_embedding,
n_drop=0, test = True)
print("Finished loading testing reservoir embedding......")

# Fit output:
readout.fit(input_repr, y_train)
pred_class = readout.predict(input_repr_te)
true_class = list(y_test)

# Evaluate:
compdf = pd.DataFrame({'pred_class':pred_class, 'true_class':true_class})
compdf = compdf.sort_values('pred_class', ascending=False)
print(str(compdf.head(10)))

def myRound(x, r):
    if x>r/float(1000):
        return 1
    else:
        return 0

def eqArray(a,b):
    return np.where(a == b, 1, 0)
predictions = list(compdf['pred_class'].apply(myRound, r=225))
true_class = list(compdf['true_class'])
accuracy =
    np.sum(list(map(eqArray, predictions, true_class))) / len(true_class)
accuracy

from sklearn.metrics import confusion_matrix
confm = confusion_matrix(true_class, predictions)
confm
```