BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

MITIGATION OF DOS ATTACKS USING MACHINE LEARNING

POTLAČENÍ DOS ÚTOKŮ S VYUŽITÍM STROJOVÉHO UČENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. PATRIK GOLDSCHMIDT**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN KUČERA**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2020/2021

Master's Thesis Specification

Student: **Goldschmidt Patrik, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Cybersecurity
Title: **Mitigation of DoS Attacks Using Machine Learning**
Category: Networking
Assignment:

1. Get acquainted with Denial of Service (DoS) attacks and DDoS Protector, a high-speed device for network traffic cleaning.
2. Study the theory of Recurrent Neural Networks (RNN), Machine Learning (ML) in general, and their application in mitigation of DoS Attacks.
3. According to available literature, choose or design a suitable method for mitigation of DoS attacks using ML.
4. Implement the method using a suitable toolkit.
5. Perform experiments using an available data set and evaluate achieved results.
6. Discuss achieved results and the possibilities of further improvements.

Recommended literature:

- According to the instructions.

Requirements for the semestral defence:

- Points 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Kučera Jan, Ing.**
Consultant: Žádník Martin, Ing., Ph.D., UPSY FIT VUT
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2020
Submission deadline: July 30, 2021
Approval date: February 9, 2021

Abstract

Distributed Denial of Service (DDoS) attacks are an ever-increasing type of security incident on modern computer networks. This thesis aims to detect these attacks and provide relevant information in order to mitigate them in real-time. This functionality is achieved by data stream mining and machine learning techniques. The output of the work is a series of tools executing the process of the whole machine learning pipeline – from custom feature extraction through data preprocessing to exporting a trained model ready for deployment. The experimental results evaluated on various real and synthetic datasets indicate an accuracy of over 99% with an ability to reliably detect an ongoing attack within the first 4 seconds of its start.

Abstrakt

Útoky typu odoprenia služby (DDoS) sú v dnešných počítačových sieťach stále frekventovanejším bezpečnostným incidentom. Táto práca sa zameriava na detekciu týchto útokov a poskytnutie relevantných informácii za účelom ich mitigácie v reálnom čase. Spomínaná funkcionalita je dosiahnutá s využitím techník prúdového dolovania z dát a strojového učenia. Výsledkom práce je sada nástrojov zastrešujúca celý proces strojového učenia – od vlastnej extrakcie príznakov cez predspracovanie dát až po export natrénovaného modelu pripraveného na nasadenie v produkcii. Experimentálne výsledky vyhodnotené na viacerých reálnych a syntetických dátových sadách poukazujú na presnosť systému väčšiu ako 99% s možnosťou spoľahlivej detekcie prebiehajúceho útoku do 4 sekúnd od jeho začiatku.

Keywords

DoS attack, DDoS attack, DDoS detection, DDoS mitigation, Machine learning, data stream mining

Kľúčové slová

DoS útok, DDoS útok, DDoS detekcia, DDoS mitigácia, strojové učenie, prúdové dolovanie z dát

Reference

GOLDSCHMIDT, Patrik. *Mitigation of DoS Attacks Using Machine Learning.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Kučera

Rozšírený abstrakt

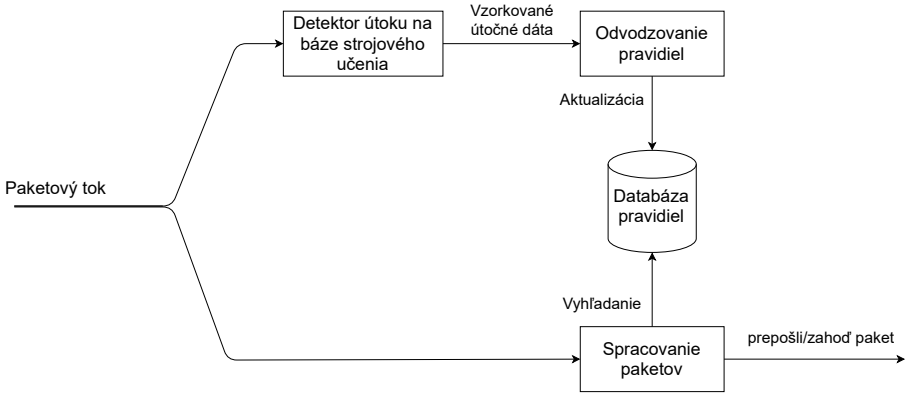
Problematika bezpečnosti je jeden zo základných faktorov pri návrhu a prevádzke informačných systémov. V bezpečnostnej terminológii sa typicky snažíme o dosiahnutie troch základných princípov – dôvernosti, integrity a dostupnosti. Odoprenie služby (DoS) a distribuované odoprenie služby (DDoS) sú v súčasnosti jedny z najčastejších kybernetických útokov cieliace na narušenie dostupnosti. Ich typické prevedenie je vo forme zasielania veľkého množstva paketov so zámerom vyčerpať výpočtové alebo sieťové zdroje ich cieľa a tým narušiť jeho bežnú prevádzku.

Detekcia a mitigácia týchto útokov historicky prebiehala na základe monitorovania sieťovej prevádzky a hľadania rôznych anomálii na báze signatúr alebo vzorov správania. Tieto údaje však museli byť manuálne definované na základe expertných znalostí a skúseností. S pribúdajúcim počtom a rôznorodosťou útokov sa však tento trend časom stal dlhodobo neudržateľný. Jedno z prípadných riešení tohto problému ponúkajú technológie umelej inteligencie a strojového učenia. Ich využitie umožňuje odvodenie podobných rozhodovacích pravidiel, a teda automatizované získavanie signatúr s oveľa väčšou rýchlosťou, škálovateľnosťou a typicky aj presnosťou.

Táto práca sa zaoberá návrhom a implementáciou systému pre detekciu a mitigáciu týchto typov útokov pomocou strojového učenia. Cieľom práce bolo vyvinúť systém detegujúci DDoS incident v reálnom čase. Projekt bol vyvíjaný s podporou bezpečnostného výskumu spoločnosti CESNET a ich projektu *VI20192022137 Adaptivní ochrana před DDoS útoky* s podporou Ministerstva vnitra České republiky.

Súčasná orientácia výskumu na detekciu DDoS útokov smeruje k využívaniu sieťových tokov za účelom klasifikácie. Tento prístup so sebou však prináša viaceré úskalia, ako napríklad stratu kontextu medzi paketmi s rozdielnym zdrojovým portom. Z tohto dôvodu bol navrhnutý detekčný mechanizmus vybudovaný na báze IP adries. Klasifikácia podľa IP adries prináša vyšší level abstrakcie, čím umožňuje vidieť kontext medzi viacerými spojeniami z rovnakej IP adresy. Tento fakt následne umožňuje počítanie rôznych štatistických údajov využívaných pre účely klasifikácie útočníkov.

Pretože od systému vyžadujeme beh v reálnom čase, klasifikácia každého prichádzajúceho paketu nie je realizovateľná. Z tohto dôvodu je systém navrhnutý pre zbieranie štatistických údajov a ich následnú klasifikáciu na pozadí. Výsledok tohto procesu potom môže poslúžiť na odvodenie rýchlych pravidiel, ktoré môžu byť využité pre okamžité rozhodnutie či paket zahodiť alebo preposlať v ráde nanosekúnd (Obrázok 1).



Obrázok 1: Návrh systému na mitigáciu DDoS.

Keďže blok pre spracovanie paketov a databázu pravidiel môžeme uvažovať ako aktuálny CESNET DDoS Protector (systém, do ktorého je metóda vyvíjaná) a niekoľko algoritmov na odvodzovanie pravidiel je už implementovaných, táto práca sa zameriava výhradne na blok detektoru útoku na báze strojového učenia.

Za účelom výpočtu štatistík je z každého paketu extrahovaných 8 hodnôt: časová značka príchodu paketu, zdrojová IP, cieľová IP, L4 protokol, zdrojový port, cieľový port, dĺžka hlavičiek paketu a dĺžka obsahu paketu. Tieto hodnoty sú následne spracovávané pomocou technológie dolovania prúdových dát – princípu časových okien. Extrahované hodnoty z paketov sú zoskupované na základe svojej zdrojovej IP adresy v konkrétnom časovom okne daného príchodom paketu. Po uplynutí určitej doby sa aktuálne okno ukončí a je nahradené novým prázdnym oknom. Tento princíp umožňuje spracovávať teoreticky nekonečný tok dát v reálnom čase a počítať nad ním štatistické ukazovatele. Za účelom šetrenia pamäti boli využité ďalšie algoritmy a dátové štruktúry špecifické pre prúdové spracovanie dát, ako napr. prúdový rozptyl alebo HyperLogLog pre výpočet kardinality.

Po získaní určitého počtu okien pre konkrétnu IP adresu prebehne ich sumarizácia, čím sa vypočítajú ďalšie medzi-oknové štatistiky užitočné pre klasifikáciu. Takýmto spôsobom je vypočítaný vektor o veľkosti 32 prvkov obsahujúci údaje ako napr. entropia zdrojových portov, odchýlka príchodu paketov, alebo priemerný počet zaslaných paketov behom jedného spojenia. Štatistiky v takejto forme sú následne pripravené na klasifikáciu.

Systém bol implementovaný v jazyku Python ako séria niekoľkých skriptov. Každý skript vykonáva určitú časť procesu strojového učenia (napr. skript pre extrakciu dát a tvorbu dátovej sady) a poskytuje rôznu funkcionalitu na základe dodaného konfiguračného súboru a parametrov príkazového riadku. Takýmto spôsobom môže byť systém jednoducho ovládaný na základe aktuálnych potrieb bez nutnosti zásahov do programového kódu.

Experimentálne vyhodnotenie systému bolo prevedené na siedmich zmiešaných dátových sadách. Tri z týchto sád pochádzajú z reálneho sieťového záchytu, zvyšné sady boli umelo generované v laboratórnom prostredí. Na základe dosiahnutých výsledkov (Tabuľka 1) vyvodzujeme, že systém je schopný detegovať prebiehajúci útok s vysokou úspešnosťou (nad 99%) v priebehu 4 sekúnd od začiatku útoku. Dodatočné experimenty taktiež preukázali schopnosť klasifikácie pomalých DoS útokov.

Model	**fit_time**	**s_time**	**accuracy**	**acc_std**	**f-score**	**prec**	**recall**
Adaboost	3.2352	0.0820	0.9946	0.0004	0.9946	0.9960	0.9932
Naive Bayes	0.0526	0.0205	0.7231	0.0025	0.6279	0.9573	0.4672
Extra Trees	1.9194	0.1248	0.9979	0.0009	0.9979	0.9989	0.9970
Gradient Boosting	14.1202	0.0258	0.9974	0.0007	0.9974	0.9981	0.9968
Logistic Regression	0.4203	0.0179	0.9366	0.0045	0.9369	0.9316	0.9423
Linear Discriminant An.	0.2157	0.0170	0.9263	0.0051	0.9267	0.9219	0.9315
Multilayer Perceptron	28.3703	0.0290	0.9952	0.0018	0.9952	0.9947	0.9957
Nearest centroid	0.0342	0.0140	0.7939	0.0074	0.8068	0.7592	0.8609
Support Vector Machines	6.8902	2.6329	0.9704	0.0023	0.9705	0.9677	0.9734
Decision Trees	0.6140	0.0147	0.9946	0.0014	0.9946	0.9944	0.9949
Random Forest	6.1454	0.1095	0.9979	0.0009	0.9979	0.9989	0.9970
XGBoost	10.3377	0.0284	0.9985	0.0007	0.9985	0.9985	0.9984

Tabuľka 1: Porovnanie výkonnosti modelov pre klasifikáciu 4-sekundových blokov pomocou technológie krížovej validácie (*s_time* = *score_time*).

Mitigation of DoS Attacks Using Machine Learning

Declaration

I hereby declare that I have authored this Master's thesis independently, under the supervision of Ing. Jan Kučera. I have not used any other than the declared sources and publications, and that I have explicitly marked all material that has been quoted either literally or by content from the used sources. According to my knowledge, the thesis or its parts have not been presented to any examination authority, nor have they been published. I am aware that the respective work might be considered plagiarism, and legal actions may be taken if the above statements were not true.

.....
Patrik Goldschmidt
July 30, 2021

Acknowledgements

I would like to express my gratitude towards my supervisor *Ing. Jan Kučera*, who was enormously helpful the whole time we were co-working on the DDoS Protector project. Many thanks also go to my consultant, *Ing. Martin Žádník, Ph.D.*, who had introduced me to CESNET's security research projects back in 2017.

Contents

Chapter 1

Introduction

The matter of security is an essential factor to consider when designing and maintaining a computer system. Many types of cyber-attacks can be performed to compromise a system or disrupt its regular operation. Denial of Service (DoS) and Distributed Denial of Service (DDoS) attacks are among the oldest types of these threats. They aim to disrupt the system's availability so that regular users are not able to access its resources. Despite their maturity, they are still becoming increasingly popular as functional computer networks and services became an essential part of any organization in today's interconnected world.

DDoS attacks have been historically mitigated by monitoring network traffic and performing decisions based on their statistics (such as anomaly detection) or per-packet rules. These methods are still relatively successful in mitigating attacks with known or predictable behavioral patterns. However, all of the decision rules and thresholds have to be specified manually based on expert knowledge or patterns of previously discovered attacks. In recent years, this approach has become rather problematic to develop and maintain as attackers perform less predictable types of attacks able to bypass these rules.

As a possible countermeasure, modern mitigation approaches often experiment with various machine learning (ML) techniques, theoretically able to infer decision rules themselves and thus not requiring any manual intervention. Most of the research nowadays focuses on DoS/DDoS detection based on network flows. Such systems rely on a supposition that attackers use a small number of source IP addresses and ports, and so flows can provide relevant information for machine learning classifiers. This assumption is generally too restricting, and so flow-based mitigation may fall short when attackers utilize tools generating malicious traffic with randomized IP addresses and ports.

This work aims to tackle this issue by designing and creating an ML-based mechanism not reliant on network flows but rather classifying according to per-IP data. Therefore, a significantly greater generalization of the problem is created, making the method more usable in practical scenarios as fewer assumptions about the attack have to be made prior. The work has been supported by CESNET's security research *VI20192022137 Adaptive Protection Against DDoS Attacks*, co-funded by the Ministry of Interior, Czech Republic.

The following document firstly discusses DoS and DDoS attacks principles and presents concepts of their mitigation with both traditional and machine learning-based approaches (Chapter 2). Chapter 3 examines various ML principles usable for our purposes and elaborates on their usage in the current research. Chapter 4 proposes our system and a machine learning pipeline. Chapter 5 takes a brief look at how it is implemented and explains the usage of created programs. Chapter 6 then presents achieved results, whereas Chapter 7 summarizes the thesis and suggests possible improvements and future work.

Chapter 2

Denial of Service Attacks

Denial of Service (DoS) and its distributed variant DDoS are cyber-attacks that aim to interrupt the regular operation of the network resource or service in order to make it unavailable for other users. The goal of these attacks is to exhaust the network, memory, or computing resources of the target so it cannot process more messages from clients. This state typically causes any incoming requests to be dropped, hence creating a denial of service situation for regular clients attempting to access a particular resource.

While DoS attacks can be effective in specific scenarios, such as Wi-Fi deauthentication DoS[1], they are usually restricted and not widely applicable. For this reason, distributed versions of DoS are more frequently employed when resources on the Internet are targeted. These can generally cause significantly greater damage and may target almost any publicly available network resource such as web servers, email servers or online gaming platforms.

DDoS is a structured network attack, typically coming from various sources that are merged to form a large packet stream able to disrupt the target's operation or its underlying network infrastructure. The attack is commonly performed using a hierarchical structure. The attacker typically gives a signal to numerous other computers (handlers), which in turn command and control a vast quantity of agents to perform an actual attack (Figure 2.1). Agent computers (called bots or zombies) mostly consist of compromised hosts scattered across different geographical locations, which are in full control of an attacker. Handler computers directly issue commands for agents, e.g., to establish a large number of sessions or generate a certain type of traffic at the same time. If the target network is not explicitly protected, the tremendous amount of generated traffic is generally enough to make the target irresponsive.

According to Cisco, the total number of DDoS attacks will double from 7.9 million in 2018 to 15.4 million by 2023 [28]. The study [62] also states that more than 323 thousand dollars in excess power and added bandwidth consumption was spent by a four-day DDoS attack executed through a network of hacked Internet of Things (IoT) in 2016. This included devices such as Internet routers, security cameras, and digital video recorders. The technical report from Radware [86] declares that 50% of organizations currently consider DDoS attacks as the largest threat to their business model. According to these findings, we may conclude that research in the DDoS attacks detection and mitigation field is a crucial part of the overall system's cybersecurity and will become increasingly important in emerging IoT network architectures and the beginnings of Industry 4.0.

[1] An attack during which the attacker floods a Wi-Fi network with deauthentication frames, causing clients to disconnect from the network.

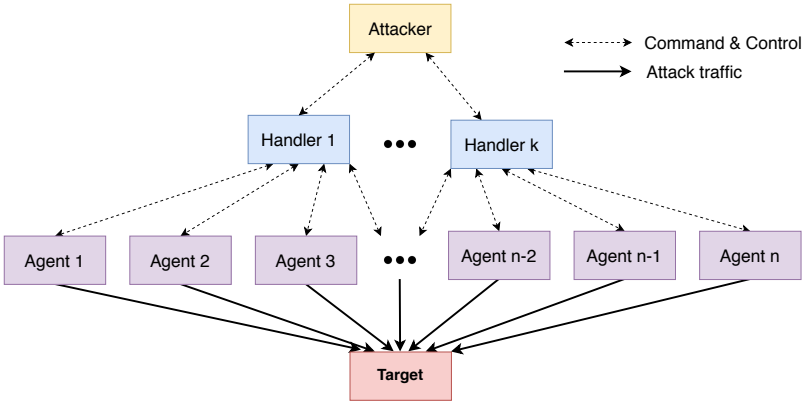


Figure 2.1: DDoS attack architecture.

2.1 Types and Techniques

In general, DDoS attacks can be classified according to the resource they target. By this definition, three main types of DDoS can be recognized as volumetric, protocol, and application attacks [44]. Volumetric attacks attempt to use massive amounts of traffic to exhaust the underlying architecture of the target. The goal of these pure brute-force attacks is to saturate the network bandwidth, intermediary network devices' or the target's processing capabilities, so clients' packets either do not arrive at their destination at all or cannot be processed by the target due to the lack of resources [74]. The most popular techniques used to perform these attacks are ICMP flood, IP/ICMP fragmentation, UDP flood, IPSec flood, or reflection amplification attacks.

Protocol attacks do not aim to exhaust network bandwidth but rather try to utilize characteristics of network or transport layer protocols to perform DoS in a more efficient way. The most popular technique for this attack type is the TCP SYN Flood attack, which takes advantage of a limited number of TCP connections the machine can query. The attack still requires large amounts of segments, but its required amount for a successful denial of service situation is significantly lower than in volumetric attacks due to the TCP properties. In some literature such as [116], protocol and volumetric attacks are not explicitly separated and are often considered as one type. Alternatively, techniques such as ICMP flood and fragmentation attacks are sometimes considered to belong to both categories.

In contrast to previous two attack types, application-layer attacks focus on exploiting weaknesses of the particular applications. These types of attacks are the most sophisticated and require knowledge about the application's version architecture and behavior in order to be executed successfully. Detection and mitigation of these attacks is generally harder, as they are often executed in a low-volume manner and traffic they produce tends to be indistinguishable from that generated by legitimate users [115]. Attack techniques under this category include Slowloris, HTTP(s) flooding, Large payload POST and others.

The most prevalent DDoS attack type in the last years is TCP SYN Flood. According to Kaspersky [63], 78.28% of all attacks were performed as SYN Flood in Q4 2020. Other attacks in a given quarter were UDP (15.17%), Other TCP attacks (5.47%), GRE flood (0.69%), and HTTP flood (0.39%).

2.2 Mitigation Strategies

DDoS attacks have been traditionally detected using statistical mechanisms and mitigated either according to their statistical properties (anomaly-based mitigation) or by the usage of specific techniques against particular attack types (signature-based mitigation). For instance, an ongoing attack may be detected using metrics like correlation, entropy, covariance, packet rate, average packet size, and others [75]. Packets that trigger a certain predefined condition are considered to deviate from regular legitimate traffic, and thus the system marks them as a potential attack and drops them if required. Over the past two decades, these principles proved to be rather popular, according to numerous research articles published in the field. Nevertheless, both statistical triggers and anomaly patterns have to be specified manually by a field expert and fine-tuned for the protected network's specific properties. These methods are also highly unscalable, badly generalize to various attacks, and typically cannot react to zero-day threats not specified in the detection databases [79].

On the other hand, newer methods using artificial intelligence (AI) and machine learning (ML) generally provide a more flexible way for attack detection and mitigation at the cost of slightly higher utilization of computer resources. Their employment is typically faster than traditional methods because attack threshold values and modifiers do not need to be tweaked out after the method is installed. Supposing a balanced dataset representing both regular and attack traffic, methods based on AI and ML can provide a scalable, robust way to detect and mitigate DDoS attacks, usually outperforming systems statically programmed by humans [79].

2.3 Traditional and Machine Learning Mitigation

As briefly outlined in the previous section, traditional DDoS mitigation strategies are often unscalable and typically cannot generalize well to different attack vectors. This section will elaborate a little more on these issues by presenting concrete examples of traditional and ML methods. Both are designed to mitigate the most popular DDoS attack – TCP SYN Flood. This attack is based on opening a large number of TCP connections with the server and not responding to its responses, so the connections stay in a half-open state. The server's memory eventually gets consumed, and legitimate clients are not able to access its resources. Mitigation approaches to combat this threat will be described, compared, and conclusions will be drawn.

2.3.1 Downfalls of Traditional Approaches

A traditional approach to mitigate SYN Floods includes implementing mechanisms based on the SYN Cookies algorithm. These are placed on the server in order to validate the authenticity of the incoming SYN segments originated from clients requesting to establish a connection. This solution is still one of the best ways to prevent the attack, but employing a separate algorithm on an application server is not always desired. For these reasons, several network-based mitigation methods to protect against the attack, such as SYN Cookies variants, RST Cookies, or simply policing the maximum number of allowed SYNs per IP address [40], have also been developed.

Several attempts of network-based mitigation based on packet contents and various statistical data have also been conducted. A typical example of this approach is [76], which uses a statistical variation of the SYN traffic arrival. Packet content analysis detection such

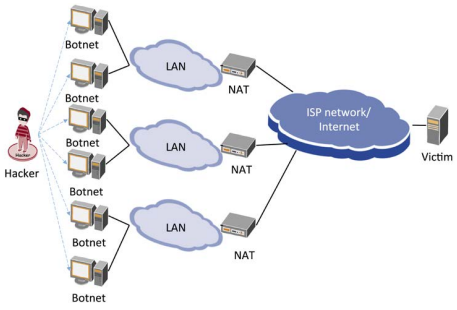


Figure 2.2: DDoS attack NAT illustration. Source: [106].

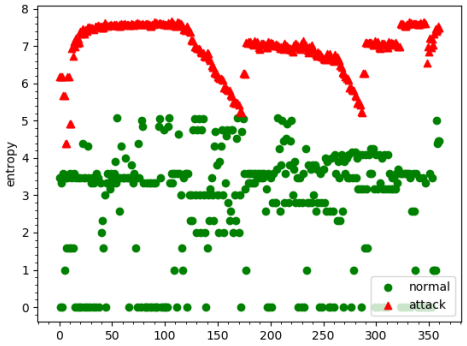


Figure 2.3: Attack and regular traffic port entropy comparison. Source: [106].

as [45] detects a potential SYN Flood by searching for anomalies based on the payload of IP and TCP headers such as ToS, IHL, TTL, and TCP flags.

Statistical and header contents analysis may be an efficient way for attack detection but requires a field expert to specify the correct statistics, modifiers and thresholds so that the mechanism functions properly. This issue was experienced from the author's own perspective as he designed a system for automatic SYN flood mitigation method switching [42]. In this case, the number of parameters that had to be manually set up was so overwhelming that the mechanism either worked "rather fine" or did not work at all. This behavior can be generalized to most of the heuristic approaches, plus the parameters may vary slightly from environment to environment, making the methods relatively inflexible and hard to enhance and maintain. For this reason, manual observation and parameter tweaking may be replaced by artificial intelligence techniques, which should be able to deduce the parameters by themselves and even find the patterns in data that a human would not have noticed.

2.3.2 Machine Learning Approach

A recent paper from 2019 [106] shows that the SYN Flood attack may be mitigated using K-nearest neighbors algorithm based on a simple principle of source port entropy. While analyzing the CAIDA 2007 dataset of DDoS attack traffic from August 2007, the researchers found that each attack source IP opened circa 270 source ports while performing the attack. An explanation for this phenomenon is that the attacker typically uses a botnet with several infected computers in each network. When the attack is launched, all the computers start to produce large quantities of traffic destined for the same target. Since all of the nodes have to pass through the Network Address Translation (NAT) gateway, the private IPs of zombie computers are translated into few (or one) public IPs with different ports (Figure 2.2).

After calculating the port entropy between regular and attack traffic (Figure 2.3), the results clearly show the difference between the two – entropy for regular connection is mostly low, whereas the entropy for attacking hosts is predominantly high. According to these findings, the authors constructed a K-nearest neighbor classifier with an accuracy of 98.2% on the given dataset. Implemented in Software Defined Networking (SDN) environment, the authors were able to classify a single packet in 0.4ms, and the legitimate packet delay averaged 109ms. In contrast, when no mitigation was active, regular traffic often did not reach the destination or experienced significant latency from 7.5s to 9.5s [106].

A vast number of similar machine learning solutions utilizing various statistical and packet features have also been proposed. These will be discussed in the following chapter.

2.3.3 Discussion

As mentioned in Subsection 2.3.2, the K-nearest Neighbors ML algorithm successfully classified most of the traffic only according to one feature. This approach could further be combined with other statistical properties to create an even more robust solution. The average packet delay of 109ms is not perfect but definitely serves the purpose of mitigating the attack while providing stable access to resources for regular clients.

In contrast, traditional network-based mitigation methods built upon SYN-authentication such as TCP Reset Cookies would obtain 100% accuracy due to the nature of spoofed IP addresses in the dataset. This is because the method requires clients to authenticate before forwarding their SYN messages, thus effectively denying all SYN traffic from spoofed IPs [41]. Nevertheless, the experiment would not be replicable because the method requires to interact with the clients. Therefore, replaying a dataset to test the accuracy would not be possible. According to our previous measurements, the delay added by this method is less than 1ms after the client successfully authenticates [41]. This time is significantly lower than in the ML method's case, and the traditional approach also provides better protection in this particular case. Why would we hence bother with ML-based methods?

As outlined earlier in this chapter, traditional methods do not scale and generalize well. TCP RST Cookies and other SYN-authentication mitigation methods may work fine for most typical SYN Flood attacks but are significantly less effective against attacks with non-spoofed IP addresses. In addition, they cannot be used against other TCP DDoS threats, which are also relatively popular. These attacks do not typically have their associated bulletproof deflection technique, and thus methods based on statistical properties need to be used. In their case, defining and fine-tuning thresholds for various statistics is also not a trivial task – it is often imprecise, time-consuming, and requires adjusting for every network.

On the other hand, mitigation based on machine learning can cover a much wider range of attacks, not requiring a specific method for each attack technique, and so might be developed and employed much faster. For instance, K-nearest Neighbors with entropy may be generalized to provide several other mitigation capabilities, like protection against TCP, UDP, and HTTP Floods, which would behave very similarly to SYN Flood from the perspective of port entropy. Due to high generalization capabilities, ML approaches may hence detect even a new type of DDoS attack that has never been seen before. This would be possible only according to its behavioral patterns (such as packet arrival variance, average packet length, or mentioned port entropy), similar to other attacks that the method already encountered during the training phase.

From this perspective, we can clearly see the robustness of the machine learning solutions and why it is beneficial to employ them as detection and mitigation mechanisms. Nevertheless, this does not disqualify traditional approaches from being used. These solutions often offer excellent mitigation capabilities, and their performance is typically much better, allowing them to process more packets with lesser latency. Therefore, it is the most beneficial to use traditional approaches if possible, but their combination with ML methods can significantly improve the detection and mitigation capabilities of other types of attacks, for which traditional approaches are either unavailable or unreliable.

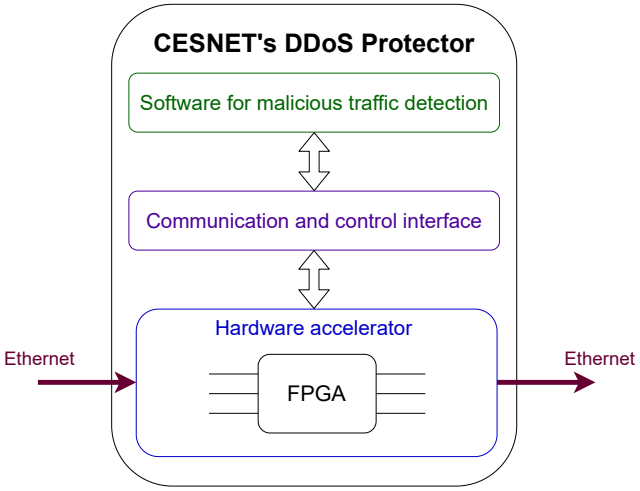


Figure 2.4: CESNET's DDoS Protector architecture.

2.4 CESNET's DDoS Protector

Several existing DDoS protection solutions such as CloudFlare [29], Akamai [3], Imperva [54], and many others are available. CESNET a.l.e., as a Czech operator and developer of national e-infrastructure for science, research, and education, is also developing its own solution to tackle this problem. The project utilizes a hardware-accelerated traffic filtering using FPGA technology[2], own firmware, and a software-based malicious traffic detection core (Figure 2.4).

The product was created to mitigate volumetric DDoS attacks such as DNS amplification DDoS, which produce tremendous amounts of traffic and which needs to be filtered on the network level before reaching end devices. At the time of writing this thesis, the usage of FPGA technology allowed traffic processing of up to 400 Gbps [59]. Most of the traffic is simply forwarded to their destination, but interesting packets like TCP SYNs can be analyzed to determine whether to forward or drop them. The analysis can be done directly in the hardware, but more advanced mitigation mechanisms (like RST Cookies) require the packet to be software processed.

In order to software-process a packet, it needs to be passed from the network interface card to the application running in the operating system (OS). This process is done without the intervention of the OS kernel to maximize the processing performance. For this reason, the CESNET's DDoS Protector is currently being rebuilt upon DPDK architecture[3]. Packets are then polled from input buffers and passed straight to software modules, such as data loggers or mitigation algorithms.

Currently implemented software mitigation methods focus mostly on TCP SYN Flood mitigation. These include mechanisms like RST Cookies, TCP Handshaker, and SYN Drop. The purpose of this thesis is to design and implement a mitigation method based on machine learning that could detect and mitigate a wider range of attacks and be integrable to the DDoS Protector in the future.

[2]Field-Programable Gate Array (FPGA) is a technology of integrated circuits that can be configured by a customer after manufacturing.

[3]Data Plane Development Kit (DPDK) is a set of libraries and network interface controller drivers for offloading packet processing from the operating system kernel to processes running in user space.

Chapter 3

Machine Learning in DDoS Mitigation

Machine learning techniques for DDoS detection and mitigation are almost always based on supervised learning algorithms. These can be defined as functions mapping an input to an output based on example input-output pairs [94]. Example pairs are called training data, which the machine learning method processes during the training phase. This procedure is supposed to "teach" the method to correlate different input features with desired outputs, so it will be able to determine the output correctly even for data that were not seen during the training. More formally, the training process is used to infer the function $f(\overline{x})$, where $\overline{x}$ is an input feature vector, and the result of $f(\overline{x})$ is a class to which the data belong to. In our case, the feature vector will consist of packet contents or statistical information about the traffic, and the result will define if the packet belongs to the legitimate traffic or an attack.

Machine learning mechanisms used for this purpose can be classified into three categories according to the data they use:

- Packet analyzers
- Statistical data analyzers
- Combined

Methods based on packet analysis classify packets only according to their contents. These values typically include IP addresses, port numbers, TCP flags, TCP window sizes, IP flags, and various data from application protocols. This approach is a typically used in non-AI-based systems such as firewalls. Nevertheless, classification of network traffic based on this principle can be done by ML principles as well. However, a smart attacker may masquerade an attack as legitimate traffic by setting packet fields in the same way as the legitimate traffic. Methods based only upon packet analysis may thus fail to detect the security incident.

Other solutions for DDoS detection do not rely on packet content analysis, but rather on the statistical behavior of the traffic as a whole. Attack traffic typically shares various behavioral patterns that may be used for successful detection. As mentioned in Section 2.3.1, traditional systems utilizing this approach have to contain various thresholds and modifiers, which have to be set up manually by a field expert. Machine learning methods are able to infer these values on their own, thus providing a robust and scalable way of mitigation

against various attack types. As discussed later in this chapter, some solutions may utilize both types of information to perform network traffic classification.

Sections in this chapter will summarize research in the machine learning DDoS detection and mitigation field. Various ML models commonly used to achieve this goal will be discussed. For each model, we firstly present its general concepts, mathematical background, and finally, its usage against DDoS attack with concrete examples based on existing research. Machine learning methods examined in this chapter include:

- Naïve Bayes
- K-means
- K-nearest Neighbors
- Support Vector Machines
- Random Forest
- Artificial Neural Networks

3.1 Naïve Bayes

Naïve Bayes (Multinomial naïve Bayes in our context) method is one of the simplest algorithms used in the field of machine learning and classification. As the name suggests, its probabilistic model is built upon the Bayes theorem. Suppose we have an instance to be classified represented by vector $\overline{x} = (x_1, x_2, \ldots, x_n)$ and a probability that the given instance belongs to the class C_k as $p(C_k|x_1, x_2, \ldots, x_n)$. Therefore, we are calculating a probability that the class C_k contains $\overline{x}$, when we know parameters of the instance. When the Bayes theorem is applied, the conditional probability is decomposed as Eq 3.1:

$$p(C_k|\overline{x}) = \frac{p(C_k)p(\overline{x}|C_k)}{p(\overline{x})} \tag{3.1}$$

where

- $p(C_k|\overline{x})$ is the postterior probability (the probability that the class C_k contains $\overline{x}$)
- $p(C_k)$ is the prior probability (the probability of class C_k occurrence)
- $p(\overline{x}|C_k)$ is the likelihood (the probability that $\overline{x}$ belongs to C_k)
- $p(\overline{x})$ is the evidence (the probability of $\overline{x}$ occurrence)

A classifier that assigns a class label $\hat{y} = C_k$ for some k can then be constructed according to Eq. 3.2:

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k) \tag{3.2}$$

The Bayes hypothesis can be easily mapped into a DDoS traffic classification problem, where $\overline{x}$ is the received packet, and C_k represents either attack or legitimate traffic. The method can be used for off-line packet capture classification and forensics, as suggested

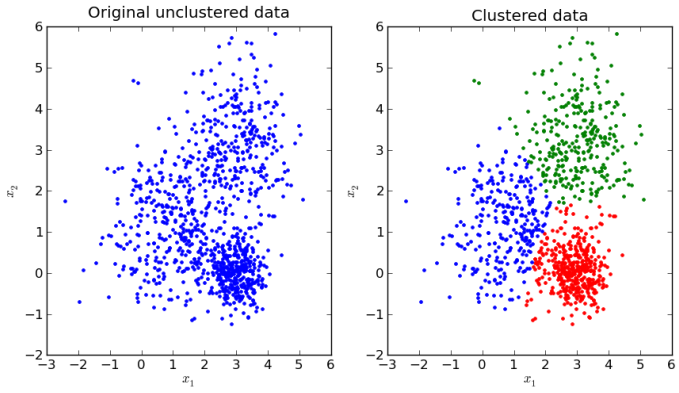


Figure 3.1: K-means algorithm result example. Retrieved from [111].

by [120] and [34]. In this case, the whole dataset can be analyzed, and statistical features can be computed for each flow during the preprocessing phase. When these statistics are obtained, the Naïve Bayes model should be able to classify particular flows correctly. Nevertheless, we are typically interested in classifying and filtering real-time traffic. In this case, statistical features for flows cannot be computed exactly, but the method has to work with statistics per time window, such as via data mining techniques in streams. When dealing with streams, the model may struggle to provide reliable classifications only according to the Bayes theorem and a training dataset due to its lesser robustness. Therefore, we conclude that the Naïve Bayes may not be the best choice for real-time DDoS protection and hence will not be examined any further.

3.2 K-means

K-means is an analysis technique used for determining K clusters in the bulk of data. It is categorized as an unsupervised machine learning algorithm, though K-means for DDoS detection is typically used in a semi-supervised way. In this case, the context about the attack or non-attack is provided at least for some data during the training. Therefore, created clusters may be categorized according to the ratio of labeled samples they contain. K-means algorithm works by using a centroid[1] as a prototype for a cluster. Initially, all centroids are selected randomly, and their position is iteratively updated according to the minimum sum of squares of the points within the cluster.

Mathematically, suppose a set of observations $x_1, x_2, \ldots, x_n$, whereas each observation x_i is represented by a d-dimensional vector. K-means aims to partition the n observations into k sets $S = \{S_1, S_2, \ldots, S_k\}$ so that the within-cluster sum of squares (i.e. variance) is minimal. Formally, the algorithm is defined by Eq. 3.3, where $\boldsymbol{\mu}_i$ is the mean of points in S_i. An example of the K-means clustering algorithm is shown in Figure 3.1.

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} \ S \tag{3.3}$$

She, Wen, Zheng, and Lyu used K-means to detect Application-Layer DDoS attacks [100]. Authors state that user webpage browsing behavioral patterns are rather deterministic.

[1] Mean position of all the points in all of the coordinate directions [5].

Statistical features of sessions like the total number of HTTP requests, the total size of all requests, request rate, and an average access frequency of the request can distinguish legitimate traffic from application-level flood attacks. The mechanism firstly creates clusters from legitimate browsing sessions in the training dataset. Since the created clusters are in the form of spheres, a single instance can be validated whether it belongs to a particular sphere or not. If the analyzed instance is not matched with an existing sphere, it is considered an anomaly and probably represents an attack. The best result was achieved for $K = 9$, which provided a 97.56% detection rate and a 2.67% rate of false positives.

Similar results with a 98% detection rate were achieved by [84] with modified K-means on DARPA 98 dataset. As in the previously mentioned article, a windowing principle was used to compute statistics for the stream of data. In this case, the author used landmark windowing for which 9 features were collected.

As shown in the previous paragraphs, the K-means method can be used to classify traffic relatively reliably. When a principle of time windows is implemented, statistics can be collected on-the-go, and therefore providing real-time DDoS protection. However, the collected statistics still need to be specified manually. Inappropriate features may create too many overlapping clusters and degrade detection capabilities.

3.3 K-nearest Neighbors

The classification with K-nearest neighbors is based on finding K nearest instances to the analyzed instance I. This is most commonly done by calculating Euclidean metric d between instances x and x' (Eq. 3.4). Other types of metrics like Manhattan, Chebyshev, and Hamming can be considered according to the type of solved problem. Nevertheless, the majority of solutions for DDoS detection use the Euclidean metric.

$$d(x, x') = \sqrt{(x_1, x'_1)^2 + \cdots + (x_n, x'_n)^2} \tag{3.4}$$

More formally, suppose a positive integer K, an unseen observation x, and a similarity metric d. The KNN classifier firstly runs through the whole dataset computing d with x for each training observation. Let $\mathbb{A}$ be a set consisting of K nearest instances to observation x. Conditional probability for each class is then estimated as the fraction of points in $\mathbb{A}$ with that given class label. Let $I(x)$ be the indicator function that evaluates to 1 when the argument x is true and 0 otherwise. KNN classifier can then be expressed as Eq. 3.5 [121].

$$P(y = j \mid X = x) = \frac{1}{K} \sum_{i \in \mathbb{A}} I(y^{(i)} = j) \tag{3.5}$$

For the classifier to function properly, the value K must also be appropriately set. Small K restrains the region of a given prediction, thus forces the classifier to ignore the context of other nearby instances. This provides the most flexible fit with low bias but high variance. Higher K averages more neighboring instances, making it more resilient to outliers. Larger values of K will have smoother decision boundaries, which mean lower variance but increased bias (Figure 3.2) [121]. Typically, K is set between 7 and 15, but again, this is highly dependent on a given task.

Apart from [106] discussed in Section 2.3.2, many other studies utilized K-nearest neighbors to classify network traffic and detect anomalies. [73] used KNN to detect DDoS proactively in the early stages of the attack. For this purpose, the entropy of source and destination IPs/ports with packet type entropy, number of packets, and other statistics

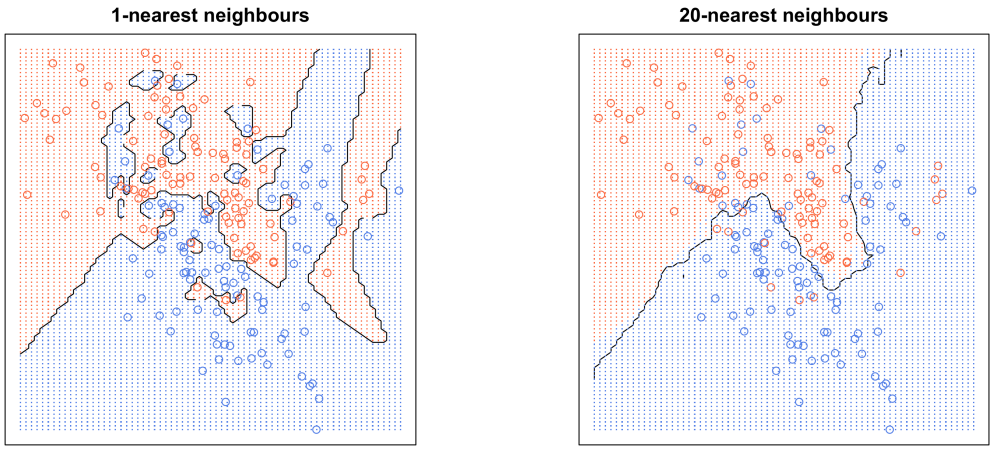


Figure 3.2: K-nearest neighbors K value comparison. Retrieved from [121].

were used. All of these can be collected and computed in real-time, thus allowing an algorithm to be integrated into IDS/IPS systems. Achieved detection accuracy was 92%. Similarly, other research projects like [32] and [78] use KNN with entropy and statistical data to detect an attack with promising results as well.

As outlined in this subsection, K-nearest neighbors classification allows robust detection and mitigation of DDoS attacks. Due to low computational demands, the method can be employed in real-world scenarios as a part of IDS/IPS systems or implemented on SDN-based networks. The detection mechanism for these methods is based on statistical traffic properties. These have to be observed and specified by a field expert, but many publications in the field have mostly done the job already.

3.4 Support Vector Machines

Support Vector Machines (SVMs) are a sophisticated ML technique used for both classification and regression tasks. The main idea of the SVM classification is to construct a linear decision boundary, so the gap (margin) between the classified classes is as large as possible. The decision boundary is generally an N-dimensional hyperplane[2] (Figure 3.3), theoretically reaching up to infinite dimensions. Since it is not always possible to classify data using a hyperplane, the elements are internally classified in higher-dimensional space as they are originally observed. Using this method, the data that would normally be linearly inseparable can be classified using a linear classifier. The relationship between the elements in higher-dimensional space is computed by kernel functions (such as radial or polynomial), which allow efficient computation without performing the actual transformation. If the classified data partially overlap, a soft margin variant, which does not try to separate two groups strictly but allows a small number of misclassifications, could be used.

If we consider perfectly separable data, the SVM searches for a hyperplane that has the maximum margin from the closest points of different classes (hard-margin). Defining a hyperplane as $H : w^T\phi(x) + b = 0$, the problem of finding the best decision boundary can be represented according to Eq. 3.6.

[2]Subspace whose dimension is one less than that of its ambient space (space surrounding an object.)

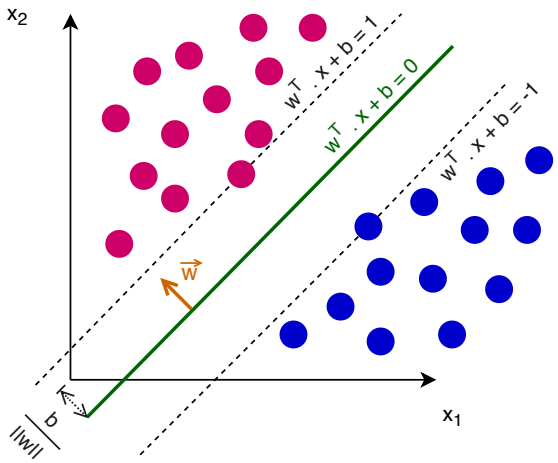


Figure 3.3: SVM maximum-margin hyperplane.

$$w^* = \arg\max_{\mathbf{w}} \frac{1}{||w||^2} \left[\min_{\mathbf{n}} y_n[w^T \phi(x_n) + b]\right] \tag{3.6}$$

where:

- w is a normal vector to the hyperplane (hyperplane parameter)
- Expression $\min_{\mathbf{n}} y_n[w^T \phi(x_n) + b]$ represents a distance of the closest point to H.
- y_n defines the class (1 or -1) to which the point x_n belongs.
- Function $\phi(x)$ is the transformation of the point x. $\phi(x) : \mathbb{R}^m \to \mathbb{R}^n$, $x \in \mathbb{R}^m$.

The above equation can be solved by the steps of normalization, transformation, and Lagrange multipliers[3] application. Two different views on the optimization problem – primal or dual form, may be taken. There, dual form is preferred because it provides a lower bound to the solution [10]. By applying a kernel trick[4] upon it, we receive an optimization problem fully independent of $\phi(x)$ terms, which can be efficiently calculated to make a classification prediction. In practice, primal and dual forms are not computed directly, but many modern approaches for finding SVM classifiers utilize techniques like sub-gradient descent [98] or coordinate descent methods [52].

In the context of DDoS detection, many research articles employed SVMs in Software-defined networks (SDNs) utilizing SDN switches to collect data and perform SVM computations on the controller. For example, [117] uses SVM data classification in the SDN environment based on the statistical properties of the flows. Statistics like the number of unique IP addresses, number of unique ports, standard deviation of packets, standard deviation of bytes, and others, are computed in time windows of length T. Several of these windows are then used to classify the flow as a regular or potential attack. The authors achieved around 95% accuracy for a custom dataset. Similar results were also obtained

[3]Strategy for finding the local maxima and minima of a function subject to equality constraints (such as the condition that one or more equations have to be satisfied by the chosen variable values) [50].

[4]Computation that allows operating in higher dimensional feature space without the need for transformation from the original feature space.

by [61]. This work also employed the SVM method in the SDN environment and measured about 95% accuracy in two combined public datasets in an off-line mode.

Non-SDN applications of SVMs comprise several research papers such as [89], which utilizes Enhanced Support Vector Machine (ESVM) and string kernels to detect ongoing DDoS in real-time. The traffic is classified into one of the 7 classes (normal or a particular DDoS type, e.g., ICMP Flood) according to statistical data such as the number of packets, session rate, and a protocol type. Classification accuracy of 99% was achieved on a live generated attack using common DDoS tools. An intriguingly similar article [103] also uses ESVM to classify the traffic into 10 classes represented by different attack types. The authors use 14 statistical features collected per flow for the classification. The model was trained on KDD99[5] as well as a custom dataset, achieving over 90% accuracy in both cases. Another interesting approach by [112] uses SVM in conjunction with Random Forest (RDF) (discussed in Subsection 3.5). RDF algorithm is used to identify the most significant features out of 42 provided in the KDD99 dataset and train the SVM model with them. This way, the presented solution has achieved high classification precision and F-score[6].

As outlined in this subsection, Support Vector Machines are a robust ML model usable mostly for classification tasks. Its principles allow us to process multi-dimensional data in a relatively fast way while achieving rather fast convergence. Several conducted research projects mentioned in previous paragraphs successfully used SVMs in DDoS detection performed offline and online with fair results. Therefore, SVMs are definitely a mechanism that may be considered for our needs as well.

3.5 Random Forest

Random Forest is an ML technique especially popular in the computer networking field since it provides a way to visualize decisions made by the model. Therefore, the model does not act like a black-box that receives an input and produces an output, but the end-user is able to follow the model's decision flow. This is indeed useful when dealing with packets because features (packet fields) and their thresholds contributing to the classification process may be analyzed. This fact allows to tweak or modify them if the model's behavior is undesired or additional fine-tuning is needed. When discussing the Random Forest technique, a method called Decision trees has to be examined first. Therefore, this section will firstly examine Decision trees and their flaws, which will eventually lead to the Random Forest algorithm and its usage in modern DDoS mitigation systems.

3.5.1 Decision Trees

Classification with a decision tree is done by constructing an n-ary tree based on a training dataset and then performing the tree traversal for each analyzed data sample. Initially, only the prior probabilities of the particular classes are known. The main idea of decision trees is to increase the probability of a successful classification with each new level of the tree. Therefore, we need to define a sequence of features and their associated thresholds to perform these splits on. After the split on some feature is defined, the dataset is divided into several subsets according to the number of thresholds of that feature. This process is then repeated for every subset, each time with a different splitting feature. The mechanism thus constructs a tree, in which each node consists of a dataset subset and defines probabilities

[5] kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.
[6] Statistical measure of the test's accuracy.

Protocol	Port	Packet size	SEQ Number	Attack
TCP	80	1500	9999	True
TCP	443	719	55634117	False
TCP	13401	1301	143577913	False
TCP	21	512	9999	True
UDP	53	240	-	True
UDP	53	240	-	True
UDP	53	314	-	False
UDP	10005	1500	-	False

Table 3.1: Packet classification dataset example.

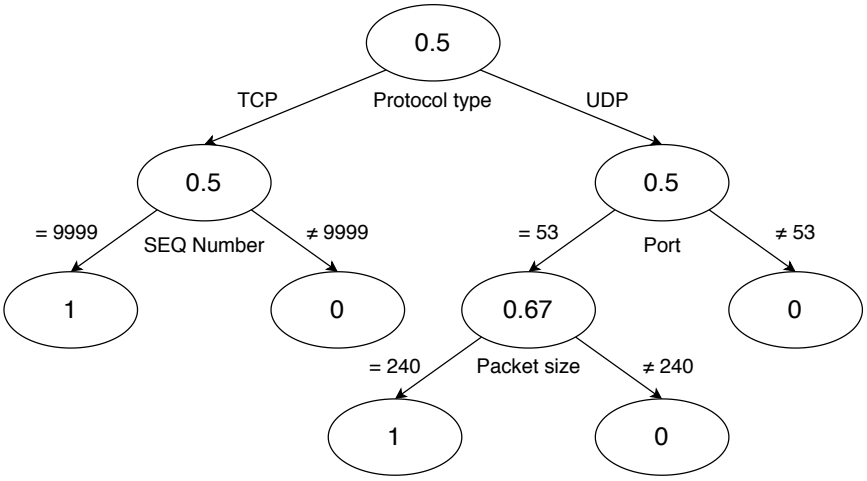


Figure 3.4: Decision tree based on Table 3.1 example.

of the observed classes based on its subset contents. The resulting probability of the data being classified is given by a leaf node or a node at a certain predefined level.

For example, consider a simple packet classification problem consisting of the dataset with 4 features and 8 samples given by Table 3.1. Our task is to construct a decision tree that classifies a new unseen packet as either an attack or regular traffic. An example decision tree for the given dataset is shown in Figure 3.4. The values of the tree nodes represent the probability that a classified packet belongs to an attack. As it may be seen, the first split on the protocol feature does not improve the accuracy but allows precise classification in the following tree levels. Some features were not used in certain tree branches at all because the mechanism does not consider them essential for classification purposes.

The learning phase of the Decision Tree model comprises the estimation of features on the particular tree levels and the definition of thresholds used to branch the tree. For this purpose, either *information gain* based on entropy or *Gini impurity index* is typically used. Entropy, originally defined by Shanon, is a measure of information, choice, and uncertainty [99]. Supposing a random discrete variable $\mathbf{X}$ with possible outcomes $x_1, x_2, \ldots, x_n$ and their probabilities $P(x_1), P(x_2), \ldots, P(x_n)$, the entropy $H(\mathbf{X})$ is computed according to Eq. 3.7. Based on logarithm properties, the entropy is also often calculated as Eq. 3.8.

In the original Shanon's proposal and for the purposes of classification trees, a logarithm of base 2 is used. The computed value is hence in bits.

$$H(\mathbf{X}) = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{3.7}$$

$$H(\mathbf{X}) = \sum_{i=1}^{n} P(x_i) \log \frac{1}{P(x_i)} \tag{3.8}$$

Entropy can be used to describe (im)purity of the analyzed subset of the dataset. Low entropy signifies that the dataset contains mostly data classified into the same class. On the other hand, data from many different classes would produce high entropy. Therefore, given entropy as a measure of impurity in a collection of training examples, we can measure the effectiveness of a feature (attribute) in classifying the training data using *information gain*. Information gain is simply the expected reduction in entropy caused by partitioning the examples according to this attribute [70]. Formally, information gain $IG(S, A)$ of an attribute A relative to the collection of examples S is computed according to Eq. 3.9.

$$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v) \tag{3.9}$$

where:

- $Values(A)$ is the set of all possible values for attribute A
- S_v is the subset of S for which attribute A has value v
- $H(S)$ is the entropy for S

With the concept of information gain, we are able to rate the splits upon different features and choose the one with the biggest value. If we applied this technique for the training dataset in Table 3.1, a tree different from Figure 3.4 would be generated because the first split does not provide any information gain. This is because the probabilities of correct classification are the same in the root node and the first level of the tree (greedy-splitting). However, non-greedy heuristic techniques considering more splits are once may be used, which may generate the same tree as in the example.

Another popular way to determine tree splits is by using the *Gini impurity index*. Instead of measuring entropy, Gini impurity works with variance in the class allocation. In other words, it is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset [114]. Therefore, Gini impurity for category K would be calculated according to Eq. 3.10.

$$Gini(K) = \sum_{i \in N} P_{i,K}(1 - P_{i,K}) = 1 - \sum_{i \in N} P_{i,K}^2 \tag{3.10}$$

where:

- N is the list of classes
- $P_{i,K}$ is the probability that category K has class i

Features for tree splitting are then determined by computing weighted sums of Gini impurities for all categories across all features (Gini impurity indexes). Considering $P_{k,a}$ as a fraction of category k in feature a and M as a list of all feature a categories, the Gini impurity index is computed as Eq. 3.11. The index with the lowest value signifies the best purity of the data (most samples are matched to the correct class), and thus its corresponding feature should be used for splitting on the current tree level.

$$I_{Gini}(a) = \sum_{k \in M} P_{k,a} \cdot Gini(k) \tag{3.11}$$

In addition to the tree split determination, entropy and Gini index can also be used for feature selection. Note that both methods compute how pure a subset of a dataset is in slightly different ways. The purity of the dataset describes a portion of records that are classified correctly. Therefore, these metrics can be used to select features that contribute to the successful classification the most. When we select only top N features with the highest informative value, the classified problem's dimensionality is effectively reduced. This strategy can be used in the preprocessing phase for various classification tasks, such as when a method that can not work with a large number of dimensions is used.

As discussed in many articles, the chosen metric for splitting is not significant because both the Gini impurity index and information gain produce somewhat similar results. As observed by [87], they differ only in approximately 2% of cases. However, Gini impurity does not require calculating the logarithm, thus being a little faster and a slightly more popular variant used in implementations such as Classification and Regression Trees (CART).

Decision Trees Drawbacks and Solutions

The most common problem with decision trees is overfitting. The tree may be theoretically split up to the point that only 1 record in their associated dataset subset remains. This behavior is undesirable, and therefore a cut-off condition denying further tree splitting is typically defined. The condition is commonly defined by minimum records in the subset of the dataset, and of course, by subset purity. If all subset data are classified into the same class, there is no point in splitting the tree anymore. Other techniques, such as pruning, can also be applied to further reduce the number of splits by eliminating unnecessary branches.

Another severe problem of decision trees is their variance. Models with high variance tend to react very sensitively against small changes in the training dataset. According to the calculation of tree split metrics, it may be seen that adding/removing a single record from the dataset would produce slightly different values, which may, in turn, construct a completely different tree as before. This behavior is also highly undesirable because the model may have problems generalizing and may provide poor or inefficient classification results.

For these reasons, multiple decision trees are commonly grouped together as a single ensemble learning technique. Ensemble techniques combine several weak learners[7] (such as shallow trees) with the goal of achieving similar or better results as more complex strong learners[8], while considerably reducing learning variance and bias [11]. Common ensemble learning algorithms include Random Forest or Extra Trees based on the bagging technique (explained below), and AdaBoost and Gradient Boosting algorithms based on technique of boosting.

[7] Classifier only slightly correlated with the true classification

[8] Classifier well-correlated with the true classification.

Random Forest is an ensemble learning method assembled of multiple decision trees that process the classified element independently. The classification result is then determined by combining results from all trees (such as majority vote or mean). This approach is based on the wisdom of the crowd principle[9]. According to this concept, we can say that a large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models [118]. Therefore, the fundamental idea is to generate many random uncorrelated trees that will participate in the classification process. Generation of these trees is done with *boostrap aggregating* and *random feature selection* techniques.

Bootstrap aggregating (bagging) is a mechanism used to generate new datasets by randomly resampling the original one. Each resampled dataset is then used to train a decision tree that will be a part of the random forest. This technique takes advantage of the decision trees' high variance and supposes that each dataset will generate a unique tree with minimal correlation to the others. Another way to minimize the correlation is to randomly reduce the number of features in each of these datasets (random feature selection). This way, each tree will classify only according to the random features subset while being trained on the random dataset subset. Combining these two techniques provide the generation of relatively random trees with low correlation. After constructing a number of these trees (forest), each tree performs classification on its own, and the final result is determined by combining their result, such as with the arithmetic mean formula in Eq. 3.12.

$$p(c|\mathbf{v}) = \frac{1}{T}\sum_{t=1}^{T} p_t(c|\mathbf{v}) \tag{3.12}$$

where:

- $p(c|\mathbf{v})$ is the probability of class c for vector feature $\mathbf{v}$
- T is the number of trees in the forest
- $p_t(c|\mathbf{v})$ is the probability of class c for feature vector $\mathbf{v}$ computed in tree t

Other ways of determining the final result can also be employed. Alongside the presented arithmetic mean, geometric or harmonic means can be used. Another popular technique of result determination is the majority vote – the class on which the most trees agree on wins. One way or the other, the random forest principle enhances classification accuracy, reduces the variance, and helps to avoid overfitting when compared to regular decision tree classification. Random Forest is a popular method in bioinformatics, data mining, finance, and many more for both classification and regression tasks.

3.5.2 Random Forest in DDoS Mitigation

The Random forest model is often a favored way for DDoS detection and network data processing in general. Its overall popularity is mostly based on the model's robustness and properties, such as the ease of implementation and visualization. For example, [35] presented an IDS system based on Random Forest, trained and evaluated on the NSL-KDD dataset[10] with 42 features. The authors performed a feature selection using symmetrical uncertainty (based on information gain). The model was able to detect several types of

[9] A collective opinion of a group of individuals rather than that of a single expert.
[10] https://www.unb.ca/cic/datasets/nsl.html

attacks, including DoS, with a 99.6% accuracy. Similar research was also conducted by [39], which also used the NSL-KDD dataset, Random forest, and information gain metric, achieving over 99% accuracy. Both of these papers described an off-line detection; however, the problem can be transformed into an on-line attack detection task using data mining techniques in data streams, similar to the research presented in previous subsections.

Random forest was also used by [23] to protect DNS servers against DNS DDoS Water Torture Attack with 99.2% accuracy. Jia et al. [57] used Random Forest as a part of the heterogeneous ensemble model with 41 features. Although the ensemble learning achieved over 99% accuracy on the KDD99 dataset, it was outperformed by a single Random forest model. As outlined, the Random forest algorithm can be used for feature selection as well. This functionality was utilized by [112] in conjunction with an SVM model, as discussed in Section 3.4.

3.5.3 Summary

As discussed in this subsection, Random forest consisting of multiple decision trees is a powerful model for not only classification but regression tasks as well. It is especially popular for network data processing since it allows simple visualization and fine-tuning. The collected features have to be specified manually, but the mechanism can select the most important features on its own, allowing to be deployed either standalone or as a part of a more robust ML solution. Features selected by the model can be viewed by the user, providing further intelligence about the solved problem.

3.6 Artificial Neural Networks

Artificial neural networks (ANNs) are computational systems which attempt to simulate the decision process in networks of nerve cell of the biological central nervous system. Simulating a biological system, ANNs are designed to perform elementary computational operations to solve complex, nonlinear, stochastic, or mathematically ill-defined problems in a highly parallelized manner [43]. According to these properties, they are often able to carry out tasks such as detection and recognition, in which humans and animals excel, but conventional systems perform poorly.

This subsection will introduce the concept of a biological and artificial neuron, present various types of artificial neural networks and discuss their usability for real-time DDoS detection and mitigation.

3.6.1 Biological Neuron

The biological networks of humans are composed of approximately 100 billion nerve cells (neurons) that are densely interconnected with thousands of connections per cell. A neuron (Figure 3.5) is a simple processing unit that combines signals from other neurons through input paths called dendrites. The signals from all dendrites are combined in the cell core (nucleus). If this combined signal is strong enough, the neuron “fires”, producing an output signal along a path called the axon. The axon splits up and connects to thousands of dendrites of other neurons through synapses. Synapses, located in dendrites, are junctions controlling the flow of electrical signals. Each synaptic junction has a specific conductance strength that defines the magnitude of the signal. This strength is modified as the brain learns new information. Therefore, synapses act as the brain’s basic “memory units” [109].

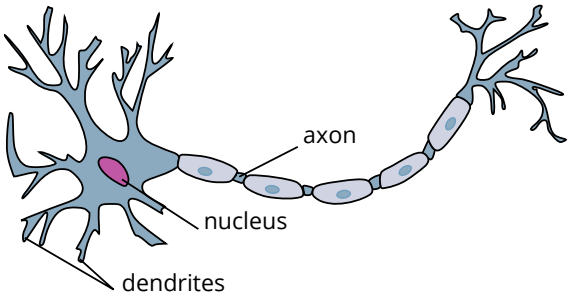


Figure 3.5: Simplified structure of the biological neuron. Retrieved from [1] (modified).

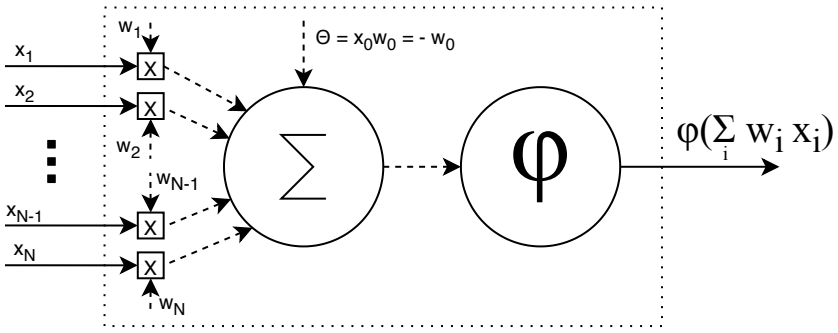


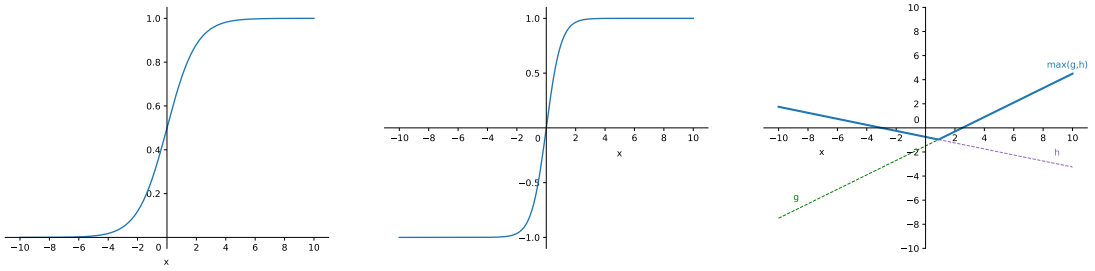
Figure 3.6: Artificial neuron scheme.

3.6.2 Artificial Neuron

Artificial neurons are essential building blocks of artificial neural networks discussed in this section. Their general model (Figure 3.6) is built upon biological neuron concepts presented in the previous subsection. The artificial neuron can hence be viewed as a mathematical model with N inputs x_i representing dendrites. Each input has an associated weight parameter w_i modifying the input value. The "firing" process is represented by summing all inputs multiplied with their corresponding weights and running the result through the activation function φ. Therefore, the output y of the artificial neuron can be defined according to Eq. 3.13. The result of summation before activation function application is called the neuron's internal state.

$$y = \varphi\Big(\sum_{i=0}^{N} x_i w_i\Big) \tag{3.13}$$

Activation function φ is a non-linear function defining the output of the neuron. Its essential idea is to produce a reaction of the neuron according its internal state. If the input value exceeds the threshold defined by the activation function (typically 0), the neuron output is a positive value. Otherwise, the output is 0 or a negative value, as defined by an employed function. This process is an analogy to operations in the biological neuron's nucleus and impulse transfer through the axon. Historically, the most commonly used activation function was sigmoid ($\sigma(x) = \frac{1}{1+e^{-x}}$) (Figure 3.7a) due to a friendly interpretation of neuron firing rate: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). However, other activation functions such as hyperbolic tangent ($tanh$) (Figure 3.7b), *Maxout* ($f(x) = max(w_1^T x + b_1, w_2^T x + b_2)$) (Figure 3.7c), Rectified Linear



(a) Sigmoid. (b) Hyperbolic tangent. (c) Maxout ($n = 2$).

Figure 3.7: Examples of various activation functions.

Unit ($ReLU(x) = max(0, x)$), and others are used nowadays due to sigmoid's limitations in modern ANN learning algorithms [2].

As illustrated in Figure 3.6, the neuron contains a hidden input x_0 associated with a trainable weight w_0. The input is commonly set to -1 (as in the figure) or 1. Together, they form a bias Θ. Therefore, an alternative way to define neuron output instead of Eq. 3.13 is $y = \varphi\big(\sum_{i=1}^{N} x_i w_i + \Theta\big)$. Whereas input weights influence the steepness of the activation function, the bias is used to shift it on the horizontal axis. Both the steepness and shift extents are learned by adjusting corresponding weights during training [67].

The Perceptron[11] (Rosenblatt 1958 [92]) is one of the first artificial neuron models that was proposed. The model computes its output as given by Eq. 3.13, while being activated by the Heaviside step function (Eq. 3.14), thus producing a binary output of either 0 or 1. Training is performed by updating its weights based on the class classification error. Today's neuron models also compute their output according to Eq. 3.13 but use continuous activation functions, such as in Figure 3.7. The output of these functions enhances the training process by allowing the usage of more sophisticated algorithms. Nevertheless, a single neuron of any type can only produce a hyperplane decision boundary. Therefore, it can only be used for binary classification problems that are linearly separable. More advanced (linearly inseparable) problems have to be solved using neural networks.

$$H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{3.14}$$

3.6.3 Neural Networks

As outlined at the end of previous subsection, ANNs were created to tackle the issue of linearly inseparable tasks, which cannot be solved by a single neuron. However, when we put several neurons together to form a layer and interconnect at least two of these layers together, a ML model able to learn complex non-linear patterns is created. This property of multilayer neural networks is known as Universal Aproximation Theorem [30] [51].

[11] Note that the term Perceptron often refers a general artificial neuron model in some literature. Multilayer Perceptron networks may thus refer to general feedforward ANNs. In this thesis, the perceptron represents only a neuron model proposed by Rosenblatt in 1958.

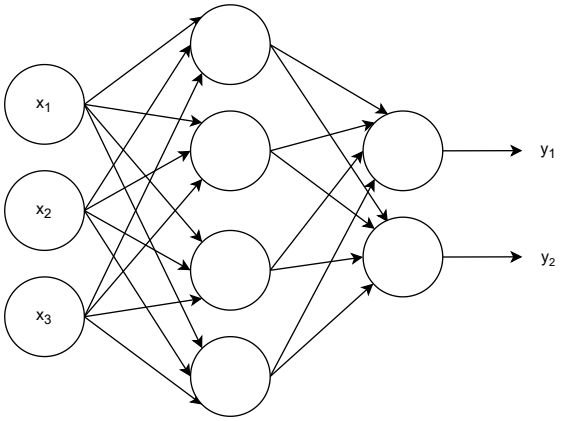


Figure 3.8: A simple 3-layer feed-forward artificial neural network.

The simplest neural network architecture is a network consisting of two neural network layers. According to conventions, the first layer always represents the input to the network. This way, the simplest NN is commonly represented as a 3-layer acyclic graph (Figure 3.8). In our case, the network receives three inputs (x_1, x_2, x_3) and produces two outputs (y_1, y_2). The network thus solves a binary classification problem for three input features. The numbers of input and output neurons are given by the solved problem, whereas the number of hidden neurons is chosen with the network design. Network layers that are not input nor output are called *hidden* layers. When training a network, each neuron in each layer updates its weights so that the output of a layer as a whole is some representation of the original vector the network received as its input. Neural network layer is thus a function $\mathbb{R}^{n_{m-1}} \to \mathbb{R}^{n_m}$, where n_m represents the number of neurons in the m^{th} layer.

In classification problems, we typically want neurons in the last (output) layer to represent the probabilities of the classes the input vector may belong to. For this reason, the activation function Softmax (Eq. 3.15) is commonly used. Softmax turns the internal states of neurons into probabilities that sum to one. The function thus outputs a vector representing probability distributions of a list of potential outcomes. Top K classes (typically one) with the biggest value are then chosen and presented as the final classification result.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \quad \text{and} \quad \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K \tag{3.15}$$

The simplest neural network architecture presented in this section belongs to the category of feed-forward networks. These types of networks do not contain cycles, so each layer works only with the output of the previous layer. Nevertheless, networks with different architectures that allow loops – Recurrent Neural Networks, also exist. These two main architectures can be modified by certain functions and neuron types, creating even more combinations of various NNs. The following subsections will discuss a few of these main network types and elaborate on their usability for DDoS detection.

3.6.4 Feedforward Architectures

As outlined at the end of Subsection 3.6.3, the example network in Figure 3.8 belongs to the feedforward neural networks (FNN) category. The essential building blocks of these

networks are fully-connected layers, which connect output of each neuron in the previous layer to every neuron in that layer. Connections between neurons in the same layer do not exist. Information from the input is thus "fed forward" from one layer to the next.

Neural networks with one hidden layer are considered to be *shallow* machine learning models. According to Universal Aproximation Theorem [30], they should be able to approximate any function. Nevertheless, it was found out that shallow models typically do not achieve desired accuracy for complex tasks, such as image recognition. For this reason, many ML models nowadays utilize principles of *deep* learning, which perform multiple levels of non-linear operations before producing the output [8]. In the context of neural networks, the principles of deep learning are represented by deep neural network (DNN) architectures, which have more than one hidden layer. Such networks are typically able to learn more complex patterns in the data and thus achieve better accuracy. Nevertheless, these types of networks are generally harder to train, requiring more training samples and more computing time. DNNs are also generally prone to overfitting due to additional layers of abstractions that allow the model to pick rare dependencies in the training data [8].

Both shallow and deep feedforward neural networks are mostly used for supervised learning tasks such as classification. Therefore, they may be suitable for DDoS detection. Nevertheless, only a detection based on statistical data without other context is possible because these networks do not have any "memory". For this reason, sequential or time-dependent data (such as network packets as they come) can not be processed by FNNs. To tackle this issue, recurrent neural networks, presented in Subsection 3.6.5, are used.

3.6.5 Recurrent Architectures

In contrast to feedforward networks, recurrent neural network (RNN) architectures do not form an acyclic oriented graph but rather allow cycles composed of neuron connections. With this design, the network gains the ability to model time or sequential dependencies of the input data. Sequential data are fed into the network by steps, one element (vector) of the sequence at the time. Computations in RNNs are performed in a cyclic manner, where the same operation is applied to every element of the processed sequence. The fundamental idea of these networks is to propagate the result computed at time t into the next computation at $t + 1$. The propagation is achieved by mentioned cycles, either self-loops for delay by one timestep or larger cycles across multiple layers.

A simple three-layer RNN is represented in Figure 3.9. The input layer acts the same as in the feedforward architecture – just passes its values to the next layer. RNN neurons in the hidden layers contain self-loops to pass its previous result to the current computation. The output layer is composed of regular linear neurons, typically activated with Softmax for classification tasks. Note that neurons with a self-loop need an extra trainable weight matrix to represent the importance of the past information for the computation process.

RNN networks are often unfolded in time/steps for better demonstration purposes. If we unfold the network from Figure 3.9, Figure 3.10 is obtained. From this representation, equations 3.16 and 3.17 to calculate outputs of hidden RNN and output neurons become more apparent.

$$h^{<t>} = \varphi_r(Vh^{<t-1>} + W_r x^{<t>} + b_r) \tag{3.16}$$

$$y^{<t>} = \varphi_o(W_o h^{<t>} + b_o) \tag{3.17}$$

where the upper index $< t >$ represents a computation made in the timestep t. Therefore:

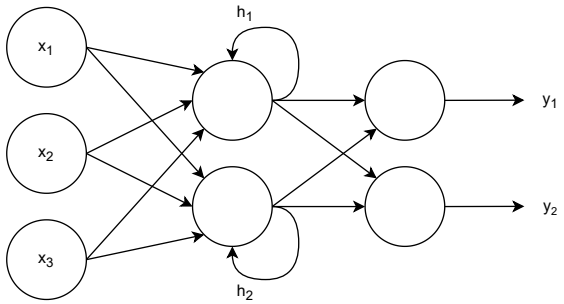


Figure 3.9: Simple 3-layer recurrent neural network.

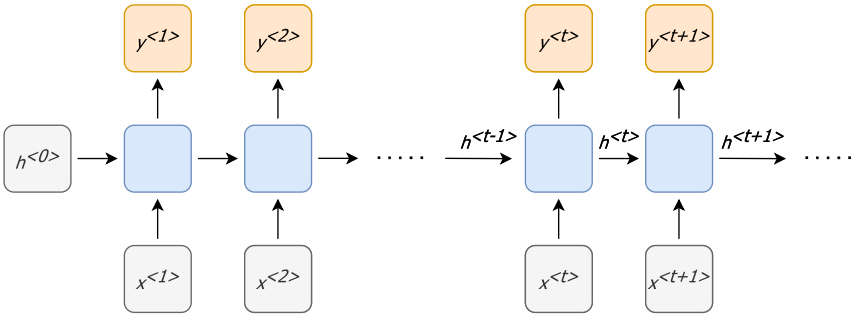


Figure 3.10: Unfolded RNN from Figure 3.9.

- $x^{<t>}$, $y^{<t>}$ are network input and output vectors in t
- $h^{<t>}, h^{<t-1>}$ are the recurrent layer outputs in t and $t-1$
- φ_r, φ_o are activation functions for the recurrent and the output layer
- W_r, W_o are weight vectors for the recurrent and the output layer
- V is the weight vector for self-loops in the recurrent layer
- b_r, b_o are biases for neurons in recurrent and output layers

In addition to the simple RNN variant presented in previous paragraphs (also known as Elman Recurrent Neural Network), numerous other architectures also exist. For instance, this network can be expanded into a deep recurrent neural network by adding one or more hidden layers to the existing model. Such networks have been confirmed to outperform the conventional, shallow RNNs [81]. RNNs can also act as memory cells. For example, the Hopfield network can be used as robust content-addressable memory, resistant to connection alteration. Similarly, Bidirectional Associative Memory (BAM) network architecture provides associative memory functionality. In the context of DDoS detection, classic architectures based on Elman Recurrent Networks and their variants are typically used.

RNN architectures can be generally classified into four categories according to the length of their input (T_x) and output (T_y) sequences. Traditional networks can be considered one-to-one model, in which $T_x = T_y = 1$. In this case, the network receives one input vector and an initial state, for which it produces one output vector. Other models like one-to-many ($T_x = 1$, $T_y > 1$), many-to-one ($T_x > 1$, $T_y = 1$), and many-to-many ($T_x > 1$, $T_y > 1$) in variants $T_x = T_y$ and $T_x \neq T_y$ also exists. The architecture type mainly depends on the task

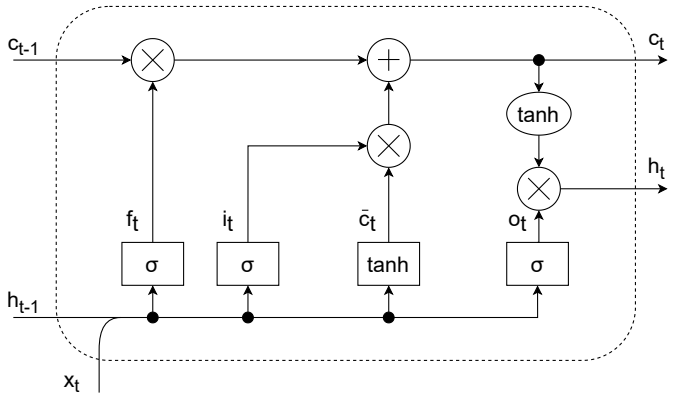


Figure 3.11: Long-Short Term Memory cell structure.

for which the network is built for. In the context of DDoS detection and mitigation, many-to-many models are typically used due to processing many packets (flows) and labeling each of them as either attack or non-attack.

Shallow RNN network architectures were a great success initially, but it was shortly discovered that they could memorize only the last few inputs and thus struggled heavily when longer dependencies had to be processed. This was primarily caused by a simple structure of the original RNN cells, which contain only one hyperbolic tangent block. These standard types of cells cause problems for learning long data sequences by forgetting information about the network prediction error, formally defined as *vanishing gradient* problem [48]. Similar situation – *gradient exploding* happens when information about network prediction errors exponentially increases and thus loses its informative value. This issue has been tackled by new RNN cell designs like LSTM and GRU, discussed in the following paragraphs.

Vanishing and Exploding Gradient Solutions – LSTM & GRU

Vanishing and exploding gradient problems can be solved by numerous means, such as modifying the network training algorithm, gradient clipping, or architecture modification [47]. The most popular solutions in practice are Long-Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) architecture types. These replace the regular recurrent cell design with a custom structure, which prevents discussed gradient problems. For this reason, networks employing these architectures can work with significantly longer sequences (dozens to hundreds) with relative ease by design [49].

A single LSTM cell is composed of an input gate, output gate, and forget gate (Figure 3.11). When the input in the current timestep (x_t) enters the cell, it is concatenated with the previous cell state h_{t-1} and processed in this form by these gates. The forget gate f_t squashes its input into the number between 0 and 1, representing how much information from the previous cell state should be forgotten (Eq. 3.18). Similarly, the input gate i_t defines values that should be updated. When multiplied with the candidate vector $\tilde{c}_t$, the new information that should be added to the cell is obtained. The new cell state is determined by multiplying the old cell state with the forget gate's output to forget some information and adding it to $i_t \cdot \tilde{c}_t$ in order to obtain new information (Eq. 3.19). The cell output h_t is computed by combining the new cell state with the output gate o_t (Eq. 3.20).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3.18}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \tag{3.19}$$

$$h_t = o_t \cdot \tanh(c_t) \tag{3.20}$$

Gated Recurrent Unit (GRU) cells are relatively similar to the LSTM architecture. They are composed of an update gate and reset gate. As in LSTM, the update gate controls what part of the current input should be remembered for the next cell state. Updating is done by an additive operation, enabling to keep specific features from the input and allowing an effective error propagation without gradient vanishing. Similarly, the reset gate is used to forget the past cell state information via multiplication with the sigmoid operation result. In contrast to LSTM, the GRU cell does not contain an output gate. For this reason, the GRU's memory content is fully exposed without any control, the cell state in timestep t thus being the same as its output in t. The computation of candidate value in LSTM and GRU is also slightly different. As evident from Figure 3.11, candidate $\tilde{c}_t$ is computed independently of the previous cell state c_{t-1}. On the other hand, GRU computes its candidate value by considering both the input and the previous state.

In theory, LSTMs should be able to remember longer sequences and train slightly slower due to their more complex structure. Nevertheless, empirical evaluations such as [26] have shown that both LSTM and GRU perform similarly, while significantly outperforming regular RNNs. The choice of the cell type should therefore depend on a particular task. In practice, both architectures are typically tested, and the one performing better is employed.

3.6.6 Convolutional Architectures

Convolutional Neural Networks (ConvNets, CNNs) are a subclass of deep feedforward neural networks. CNNs consist of multiple fully-connected layers as regular feedforward networks but also add several CNN-specific layers. The most important, convolutional layer, then gives a name for this network architecture type. These networks are mostly used to process large multidimensional data, such as images, which cannot be efficiently processed by regular FNNs due to low parameter scalability with such inputs.

Convolutional networks implicitly expect high-dimensional data at its input. For this reason, neurons in layers are also arranged in several dimensions, such as 3D for image processing. An example of a handwritten digits CNN classifier is shown in Figure 3.12. As it may be seen, the network is composed of convolutional, pooling, ReLU, and fully-connected layers, which are the core of every CNN.

Convolutional layers are composed of a set of learnable filters (kernels). Performing the convolution involves sliding these filters across the width and height of the input volume and compute dot products between filters' values and the given position of the input. Sliding produces 2-dimensional activation maps that give the responses of the particular filter at every spatial position [66]. This way, the network is able to learn filters that cause the activations on certain positions of the input, such as vertical lines (edges) and others. Used kernels can be of different sizes (such as 3x3x1), with additional hyperparameters such as stride, zero-padding, and output depth.

Pooling layers are designed to reduce the spatial size of their inputs. They operate independently on every depth dimension with the purpose of resizing it spatially. Pooling is typically done with maximum or average functions, which process several elements given by the filter's size and output their maximum or average. Similar to convolution, the filter is moved across the whole input, resulting in a reduced-size output.

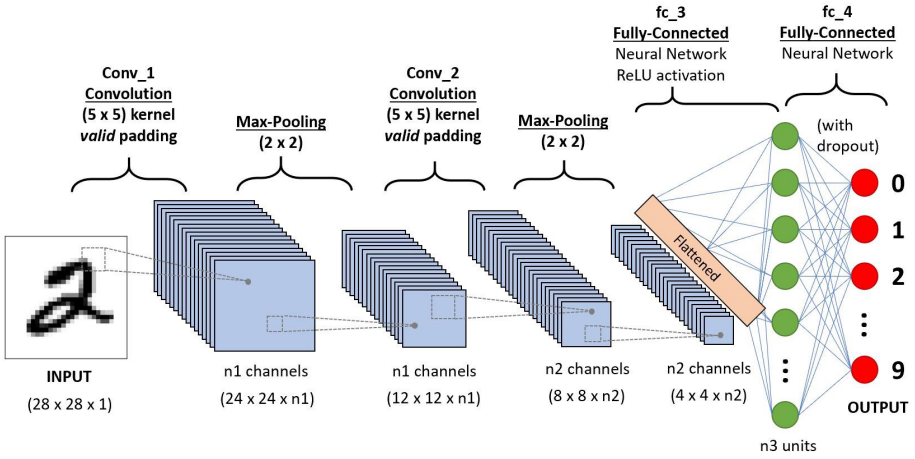


Figure 3.12: Example CNN for handwritten digits recognition. Retrieved from [95].

Other layers that can be found in CNNs include fully-connected layers and ReLU layers. ReLU layers perform element-wise ReLU operations to remove negative values that may be created as the convolution's result. Fully-connected layers are typically placed at the end of the network to classify the output produced by the convolutional, pooling, and ReLU layers. Neurons in this layer are fully connected and behave exactly like neurons in feedforward neural networks described before.

CNNs for Sequence Processing

Although CNNs have been initially designed and used to process image data primarily, it was soon discovered that their use could be much broader. Data representable in the form of the image (set of 2D objects) may be fed to regular CNNs to learn patterns in them in a relatively efficient way. Alternatively, one-dimensional CNNs that operate on 1D data may also be constructed.

CNNs have been successfully used in traditional sequence modeling tasks such as text classification [122]. [4] further empirically evaluated that CNNs perform similarly or may even outperform RNNs in several sequence modeling problems in a way more efficient manner. For these reasons, the usage of CNNs may be considered in the field of DDoS detection and mitigation as well.

3.6.7 Attention Architectures

The current state-of-the-art sequence-to-sequence models for tasks such as natural language processing or computer vision include neural networks based on an attention mechanism. It allows the identification of relevant data with respect to the presented query. In contrast to RNNs, which need to process the whole input sequentially, attention allows the input to be processed at once in parallel, significantly improving both training and operation times. Attention also allows working with longer sequences than LSTM or GRU cells can. Although its widespread adoption in the text and speech processing areas, the author is not aware of any evidence of their usage in DDoS detection or mitigation at the time of writing this document (December 2020).

3.6.8 Learning

Neural networks typically learn in a supervised manner. Firstly, an output estimate for a given input is computed. A Loss function L is then computed for this estimation based on a ground truth label. L is then used to determine an error E the network made during the estimation process. The goal of learning is to minimize these errors by computing gradients ∇E. This process, alongside a network parameter update, is performed by an optimizer, such as the gradient descent algorithm. The process is repeated for each sample within the training dataset over several training epochs. Network learning, alongside its mathematical definition, is further examined in Appendix A.

3.6.9 Artificial Neural Networks for DDoS Mitigation

As one of the most popular machine learning methods nowadays, ANNs have been used in numerous research papers to detect and mitigate DDoS attacks. For example, [96] used knowledge of the tools commonly used by attackers to design a set of three neural networks with one hidden layer that could detect DDoS using TCP, UDP, and ICMP protocol. This was achieved by extracting 3-5 features from each packet's protocol header and feeding them into the network corresponding to the analyzed protocol. The system achieved real-time accuracy of 98% for both known and unknown types of attacks manually generated by publicly-available DDoS tools using a simulated botnet.

An influential publication DeepDefense [119] from 2017 suggested using deep neural network with both backward and forward recurrent layers to enhance DDoS detection capabilities. Instead of per-packet classification, the proposed system utilized per-window classification. In this case, the last packet of the window was classified by also supplying T previous packets (authors suggest 100), so the RNN with LSTM cells can make better predictions. The paper states that DeepDefense outperform Random Forest and generalizes better on bigger and unknown datasets. In [65], the authors of DeepDefense then implemented it to SDN environment and used to deflect real-time generated attack.

Other approaches, such as [123] and [27], utilize DNNs to perform the selection of the most important features automatically. These methods are more convenient but bring relatively no innovation nor improved accuracy when compared to methods described in previous paragraphs. Many other, either feedforward, recurrent, shallow or deep network implementation exists. They typically process statistical and packet features and are mostly employed as cloud IDS or a part of the SDN network.

3.6.10 Summary

Neural networks are undoubtedly one of the machine learning methods with the best generalization abilities. Their architectures and usage are still a subject to extensive scientific experimentation and research. For these reasons, they could be a perfect match for DDoS traffic classification. Nevertheless, they are not applicable in all cases due to high classification latency (further discussed in Section 3.7) and unrepresentability of their internal states, unlike decision tree and random forest algorithms.

3.7 Conclusion

As discussed in previous sections, almost all machine-learning methods can be used to successfully detect and, in a way, prevent DDoS attacks. However, for our purpose of

Algorithm	Prediction
Naïve Bayes	$O(p)$
K-means	$O(pn_{clusts})$
K-nearest Neighbors	$O(k \log(n))$
Support Vector Machines	$O(pn_{sv})$
Decision Tree	$O(p)$
Random Forest	$O(pn_{trees})$
Feedforward Neural Network	$O(pn_{l_1} + n_{l_1}n_{l_2} + \cdots + n_{l_{m-1}}n_{l_m})$

Table 3.2: ML methods theoretical predictions complexities comparison. Legend:

- k – number of K neighbors
- n – number of training samples
- n_{clusts} – number of clusters
- n_{l_i} – number of neurons in layer i
- n_{trees} – number of decision trees
- n_{sv} – number of support vectors
- m – number of NN layers
- p – number of features

real-time detection, robust models with the ability to make a decision without a significant delay are required. As shown in Table 3.2, the theoretical prediction complexities of various algorithms differ. Empirical measurements of several classification algorithms such as [88] also confirmed that even usage of a simple 3-layer neural network with fully-connected layers classifies samples significantly slower than algorithms such as SVM or Random forest. In practice, the classification time is mostly influenced by the ML method implementation and various optimizations, but classification times taking longer than several dozens of milliseconds are generally unacceptable for our purposes.

According to these findings, we conclude that the ML method for our purposes will have to be chosen very carefully, and several of them may be tried during the process. At first, the requirements on the classification accuracy of a non-trivial network traffic analysis problem need to be satisfied. However, at the same time, restrictions on the method's performance also need to be considered. Despite neural networks' superior capabilities, other alternative methods such as Random Forest or Support Vector Machines may need to be experimented with in order to meet these criteria. The issue of choosing the method will further be elaborated on in the following chapter discussing the proposed mitigation mechanism design.

Chapter 4

Machine Learning-Based System for DDoS Detection and Mitigation

This chapter will build upon the theoretical foundations of DDoS attacks and their way of ML mitigation discussed in previous chapters. Based on this knowledge, a machine learning system capable of detecting and mitigating DDoS attacks in real-time will be proposed. The system was designed in a way that it addresses various flaws of other current research in the field, aiming to provide accurate decisions during an ongoing attack in order to lower or completely mitigate its impact.

The following sections will firstly elaborate on current research imperfections and discuss constraints put on the proposed system before the design phase. The chapter will further look at the whole ML pipeline, present its pros and cons, and finally, will examine available datasets usable for our purposes.

4.1 Existing Research Shortcomings

A considerable part of the ML DDoS detection research in recent years works with statistical features of network flows. These include already cited papers from the previous chapter like [112] (2017), [39] (2017), or [27] (2021), but also others like [97] (2020), [107] (2020), and many more. This source of statistics makes sense because architecture for collecting information about flows is typically included on most of the Internet Service Providers' and larger campus networks by design. The flow information collection is done via NetFlow exporters and collectors.

The definition of a network traffic flow is rather loose and not standardized – *an artificial logical equivalent to a call or connection* (RFC 2722 [12]) or as *a set of IP packets passing an observation point in the network during a certain time interval.* (RFC [85]). However, the defacto standard for network monitoring – NetFlow, defines a flow as a 7-tuple consisting of the source IP address, destination IP address, source port number, destination port number, layer 3 protocol type (e.g. TCP, UDP), ToS (type of service) byte, and input logical interface. Therefore, a significant share of today's DDoS detection research is centered around identifying malicious traffic grouped by these values.

At this point, one may see that the phenomenon described in the previous paragraph may become somewhat problematic. If a perpetrator manages to utilize these NetFlow properties, an attack with a small number of packets per flow, or ideally, one packet-flows, may be conducted. This can be achieved by traffic with ever-changing source ports, IP

addresses, ToS fields, or managing to change the routing of a packet, so it enters the network via a different interface. For this reason, machine learning algorithms would be unable to make an accurate decision due to the insufficient amount of traffic collected from each flow.

Even though there are still various popular types of DDoS where threat actors cannot control the source port (such as amplification attacks), many other attacks are performable with these properties in mind. Attackers have already taken advantage of this fact and create more and more sophisticated tools to trick similar mechanisms. As an example, an infamous tool Low Orbit Ion Cannon (LOIC)[1] popular between 2010 and 2015 did not provide port randomization and instead used a single or a limited set of ports and a single non-spoofed IP address to send traffic from. On the other hand, its newer improvements High Orbit Ion Cannon (HOIC) and XOIC also use a single IP address, but with incrementing port numbers according to our tests. Therefore, each packet creates its unique NetFlow entry, and so estimators based on flow data would struggle to make a correct decision.

Another problem with NetFlow flows for real-time DDoS mitigation is that they are not exported from NetFlow exporters instantly. Firstly, a flow has to timeout before it is marked as completed. This may take up to several seconds based on the exporter settings. Also, completed flows are not sent to the NetFlow collector at once but rather in batches with several others to improve performance. With batching enabled, few more seconds until the flow statistics reach the NetFlow collector are added. Only there can they be retrieved by machine learning methods and processed.

As apparent, reliance on network flows can be too restricting and may significantly decrease mitigation capabilities against certain types of attacks in the real world. For this reason, this work will aim to design a system not reliant on network flows, which should be able to provide a better generalization of the problem and thus be used against a wider variety of DDoS attacks in practice.

4.2 Design Considerations and Constraints

As outlined in the previous section, the machine learning system proposed in this thesis aims to provide DDoS mitigation functionality independent of the NetFlow flow data. In order to achieve such a goal, the generalization of the classification problem needs to be moved one level higher. This section will discuss how it is achieved, as well as what advantages and disadvantages it hides. Expected requirements on the system's performance will also be outlined.

4.2.1 Generalized Traffic Statistics Estimation

In order to make the system more robust and allow working on a higher level of abstraction, we need to redefine how is the traffic coming from clients grouped and analyzed. We aim to design a system resilient to changes in most of the variables defining a NetFlow flow so that the traffic may still be grouped regardless of the packet's content sent by the same host. The common grouping value unique for each host on the network is indeed its source IP address. Therefore, we do not try to separate legitimate (benign) flows from malicious DDoS flows but rather look at the client's communication as a whole. The input for the machine learning method is thus grouped based on the client's IP address, and so the

[1] https://en.wikipedia.org/wiki/Low_Orbit_Ion_Cannon

method's expected output will be whether the particular IP address produced malicious traffic or not.

Machine learning methods working with flows look on the flow as a whole – from its start to its end since only flows marked as completed are exported from a NetFlow exporter to a collector. However, this principle does not apply to per-IP collected statistics because we cannot tell whether the IP address will send more data or not. Also, waiting for a host to stop communicating (such as with a timeout) and classifying its data afterward is counterproductive since we want to reveal an ongoing attack and mitigate malicious traffic as soon as possible. For these reasons, we want a processing technique able to process a stream of data in real-time and produce relevant statistics so that our machine learning model will be able to make a proper decision.

Despite the fact that the per-IP principle generalizes much better than per-flow classification, it is important to add that it is not entirely bulletproof. Recall that a per-flow system may be fooled by spoofing IP addresses, randomizing source or destination ports, randomizing the ToS field, or tampering with a packet routing. Although the per-IP system addresses most of these issues, the attacker has one and the only way to deny the system to collect enough data – IP spoofing[2]. This way, malicious traffic can again be spread across multiple statistics entries or intermixed into communication of other IP hosts, creating a hard time for the machine learning method to make an accurate conclusion.

Regardless of the previously mentioned setback, note that IP address spoofing is not necessarily performable in all situations. Randomization of source or destination ports or a ToS field can be performed by any client by design. Although there are various mechanisms that may mark such communication as suspicious, it does not necessarily need to indicate any threat activity, and such traffic is typically forwarded within the Internet without a problem. On the other hand, IP address spoofing is considered harmful in a vast majority of cases, and some Internet Service Providers (ISPs) and bigger network operators are actively taking precautions to prevent such behavior from happening.

The process of spoofed IP addresses filtering is defined in RFC 2827, currently marked as Best Current Practice (BCP) #38. The document briefly explains the problem of IP address spoofing, its connection to DDoS attacks and suggests a solution for filtering inbound traffic on network edge routers. Supposing these routers would restrict transit traffic originating from a downstream network to known and intentionally advertised prefix(es), the problem of source address spoofing would be virtually eliminated [37]. In other words, all traffic that does not match its source network prefix should be dropped before it is forwarded on the Internet. As efficient as this technique is, it would require adoption on the majority of Internet providers, which cannot be generally relied on. Even if traffic filtering was adopted, an adversary might still forge packets that match the source network prefix and thus partially circumvent this principle.

Another thing to take into account in per-IP systems is NAT for IPv4. Network Address Translation (NAT) is a process of translating a set of private IP addresses to one or more public IP addresses and alternatively changing a source port (Port Address Translation (PAT)). When the packet leaves an internal network, its source IP is translated into a public IP address according to the translation table. When a packet with a spoofed IP address arrives on the NAT-enabled interface, one of three scenarios typically occurs:

1. Reverse path filtering, as explained in the previous paragraph, is applied and the packet is dropped.

[2] A process of crafting a packet with a different source IP address as the original IP of the sender.

2. The packet is treated like any other LAN packet, thus being translated by NAT.

3. Filtering nor NAT are not applied, and so the packet is forwarded to the Internet as it is.

As discussed previously, reverse path filtering cannot be relied on. Some routers may treat a packet with a spoofed IP address as any other packet and thus perform NAT-ing, which efficiently denies distribution of DDoS traffic across multiple IP addresses. This is indeed beneficial for the proposed method because attack traffic will be grouped under the same IP, allowing it to be analyzed in a ML model together. Nevertheless, the described behavior is also not provided on the majority of routers by default [71], and so packets with forged IP addresses may still reach the Internet in some cases. Therefore, we consider IP address spoofing as a relevant drawback with a possibility to affect the proposed method negatively, but the extent of its use is significantly limited compared to other perpetrator's possibilities against flow-based detection systems.

The second consideration about NAT is the possibility of mixing legitimate and malicious traffic under one IP address. Suppose there are more hosts in a single network communicating simultaneously, and PAT is employed. In that case, their resulting IP addresses will be the same, with only a source port changed after the translation. This process may create a problem for our ML method since both legitimate and malicious packets could be included in the traffic statistics for a given IP. When considering high-volume attacks based on flooding a large number of packets, the problem of merging multiple data streams under one IP address is negligible. This supposition lies in the fact that tremendous amounts of malicious traffic would significantly outweigh legitimate traffic, and so its share upon the whole computed statistics for a given IP would be minimal. However, flow merging can considerably affect low-volume attacks such as slow HTTP GET/POST DoS since they can easily become statistically insignificant when mixed with other legitimate traffic. For these reasons, we suspect that the presented method will be limited for detection and mitigation of high-volume DDoS threats, but experiments with low-volume attacks will still be conducted.

4.2.2 Real-Time Performance

During the real-time DDoS mitigation, performance is one of the key aspects. Throughput of the network cannot be significantly degraded during the ongoing mitigation, so the security device does not create a bottleneck on the network. For this reason, it is computationally infeasible to analyze each incoming packet in the ML method and wait with its forwarding until a decision is made. This would limit the throughput so significantly that DoS situation for the majority of the clients could still occur because their packet would be timeouted while waiting for the ML algorithm to draw a conclusion.

For these reasons, the mitigation itself needs to be performed in a way that the packet forwarding engine can do a quick lookup whether to forward a packet based on single, easily computable rules so that the incoming data are processed with minimal latency. This principle applies on both forwarded and dropped traffic. Therefore, our machine learning method cannot aim at executing a deep per-packet inspection, but rather analyze data that describe the traffic as a whole – such as metadata, various statistical features, etc.

Therefore, the method needs to be designed to predict malicious activity based on the obtained statistical information about the network data, grouped by IP addresses. However, in order to compute these statistics, some samples of the traffic need to be collected

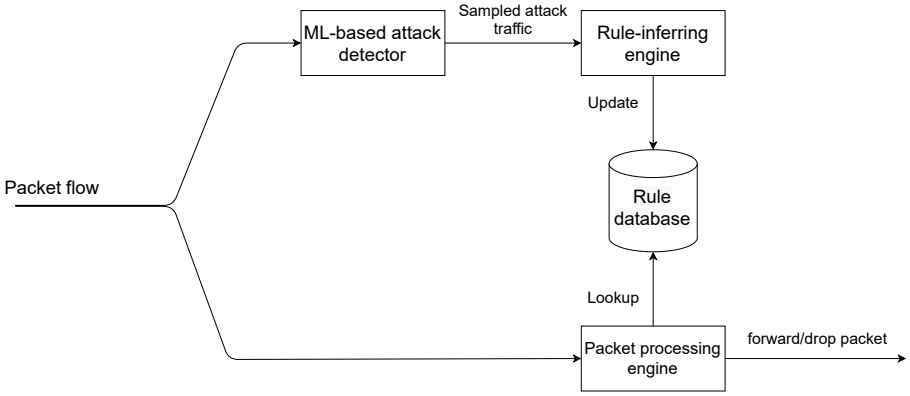


Figure 4.1: ML-based DDoS mitigation system high-level overview.

first. As it may be seen, we will not be able to decide whether the particular packet is malicious or not with ML, but we will have to focus on inferring quick lookup rules that will identify the malicious traffic. Another consequence of this design is that the ML-based decision making cannot be truly performed real-time, but some time to sample enough data, compute their statistics and make the actual prediction will be necessary. By relaxing these time requirements on the machine learning method itself, we admit that 100% mitigation accuracy cannot be achieved by design. This lies in the fact that some data need to be collected first and data that are not explicitly dropped must be implicitly forwarded to its destination. Nevertheless, if we are able to infer rules that are generic enough, a rule for malicious traffic detected for IP A might be used to drop packets from other IPs B, C, D, if they resemble the same packet structure, without the need of evaluating them in ML method.

4.2.3 System Proposal

Based on the problem definition and terms clarification in the previous subsection, a high-level concept of the system is proposed in Figure 4.1. ML-based attack detection mechanism will receive network data from the input, extract their statistics and save them to internal structures. If a sufficient amount of data is collected for a particular IP, it feeds these statistics into the underlying ML model and generates a prediction. If such a prediction detects a malicious activity for a given IP address, some of its data are sampled, and these samples are passed to the Rule-inferring engine. Based on a rule-inferring setup (e.g., signature creation), a rule to drop packets with specific characteristics will be generated. Such rule is then saved to Rule database, which is used for quick lookups from Packet processing engine, determining the following action for a concrete packet. According to this lookup, the packet is dropped or forwarded.

When mapping this high-level overview on the current CESNET's DDoS protector architecture, we may consider it to be the Packet processing engine with an API to control its rule database. As there are also several solutions for rule-inferring already available, such as [56], this work will discuss the design, implementation, and evaluation of only the ML-based Attack Detector block hereafter if not explicitly stated otherwise.

The ML detection block (Figure 4.2) is composed of three main logical parts – Feature extractor, Statistics logger, and Machine learning manager. Feature extractor processes packets and retrieves relevant features from them. It is currently implemented in software

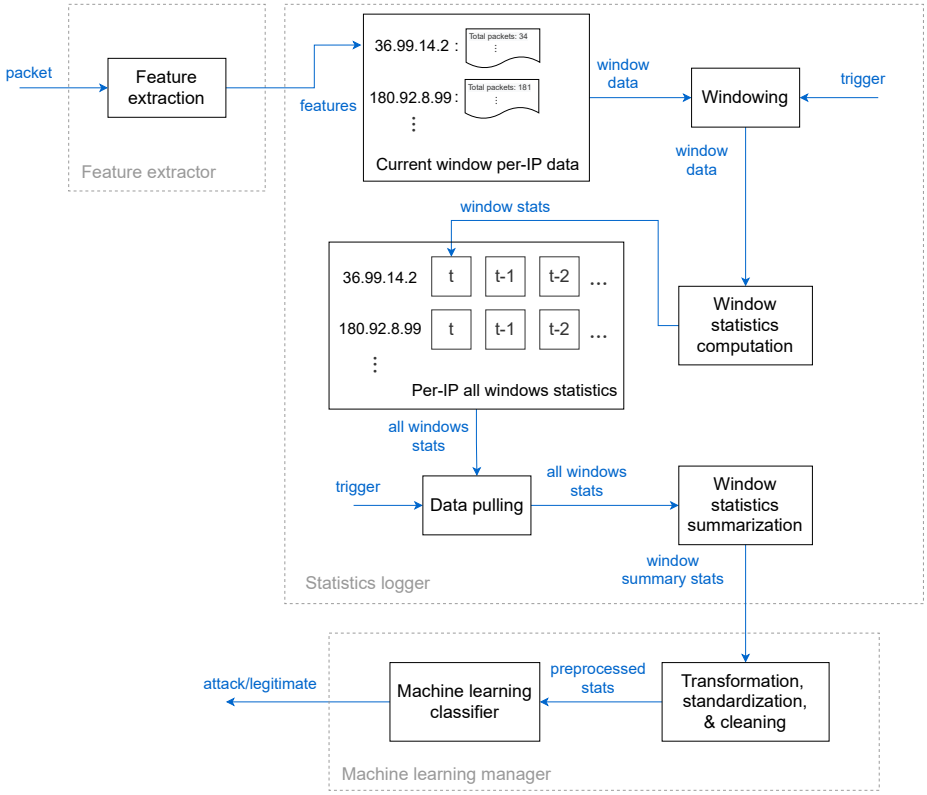


Figure 4.2: Detailed machine learning pipeline of the DDoS detection system.

for demonstration purposes, but its functionality can be entirely replaced with a hardware-accelerated solution such as FPGA.

Statistics logger is supposed to store extracted features and compute summary statistics upon them. It also exports these statistics in a format suitable for ML processing. Finally, the Machine learning manager module is responsible for preprocessing the computed statistics and managing the machine learning process as such model creation, training, and evaluation. All of these concepts are described later in this chapter.

4.3 Feature Engineering

With requirements, constraints, and leading design ideas of the system already clarified, let us look at the first step of any machine learning project – the data. As the system will be working with network packets, we will aim to retrieve relevant features from these data and transform them into a form useful for the machine learning model. This section will hence discuss what features are extracted and how are various statistics based on them computed. Derived statistical features are then used as the input for preprocessing module and the machine learning classifier, as further examined in Section 4.4.

4.3.1 Packet Feature Extraction

In order to achieve as generic use-case as possible, the proposed system aims to mainly utilize traffic metadata instead of the actual packet contents. This metadata is then used to compute various statistical features used for classification, as described in Subsection 4.3.2.

Feature name	Data type
Timestamp	Integer
Source IP	String
Destination IP	String
L4 protocol	Integer
Source port	Integer
Destination port	Integer
Headers length	Integer
Payload length	Integer

Table 4.1: List of extracted features from each packet.

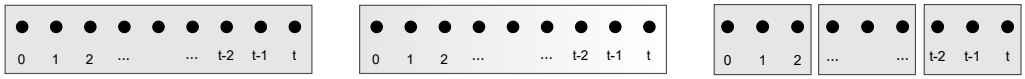
The extracted features have been chosen based on the needs of computed statistics and their ease of retrieval, which can be done in hardware with minimal additional overhead. These include the packet arrival timestamp that allows the computation of timing statistics. Since we need to know the communicating hosts, source and destination IPs are retrieved as well. Furthermore, some statistics also require layer 4 port numbers, and thus the transport layer protocol and port extraction are also performed. The complete list of extracted features from each packet is displayed in Table 4.1.

Timestamps are extracted as 64-bit unsigned integers, representing the number of nanoseconds elapsed since the Unix epoch (1 January 1970). IP addresses are extracted as strings to ease their manipulation without extra conversions later in the pipeline. However, byte representation is also possible if optimization of the extractor's memory usage would be desired. L4 protocol is represented by its protocol number assigned by IANA [55], ICMP for IPv4 and ICMP for IPv6 both represented by same value of 1 for simplicity. Port numbers are extracted as integers as they are directly extracted from L4 headers. Headers without port numbers (ICMP) leave their port field values at 0. The packet length is not extracted directly, but each packet is instead split into header length and payload length, which allow for computation of more advanced statistics, as described in the following subsections.

By design, data are only extracted from IPv4 and IPv6 packets. Other types of traffic would not provide any relevant information for our purposes and would unnecessarily diminish the performance. For this reason, non-IP traffic should not be passed to the extractor at all. Nevertheless, if such scenario occurs, a NULL-like value must be returned.

4.3.2 From Packet Features to Traffic Statistical Features

Extracted packet features in their raw form are still not much of a use for machine learning methods. They provide a little informational value on their own, and there are too many of them as well. For these reasons, several data mining algorithms can be applied to obtain information useful for learning and classification. Extracted features are then further processed in the Statistics logger module (Figure 4.2). At this point, we want to control which part of the continuous packet flow contributes to computed statistics and other data mining patterns. Several data mining techniques for streams could be considered; however, the best fitting for our purpose is the window model. The window model divides a theoretically infinite data stream into sequences of a specified length and computes statistics upon them independently.



(a) Landmark. (b) Damped. (c) Sliding.

Figure 4.3: Window models visualization. The intensity of the background color represents an object's weight.

Window Models

A few variants of windowing principles can be used throughout multiple use-cases. These include landmark windowing (Figure 4.3a), damped windowing (Figure 4.3b), sliding windowing (Figure 4.3c), and a few others. Landmark window clusters (groups) features from a starting time-point (landmark) up to the current time-point. When a new landmark starts, all items from the previous one are removed. As we are interested in the history of the host's communication and want to see how this communication changed over time, this model is not particularly useful for our use case.

In a damped window model, each object is associated with a weight depending on its arrival time. The first arrived object is assigned the highest possible weight, decreasing over time according to some aging function [69]. This windowing principle could be useful if we were interested at the start of the host's communication and less interested in its activity later on. Although this principle may be interesting in some networking applications, we aim to describe the overall host's communication over time, and so putting less weight upon the samples received later on could cause detection algorithms to perform poorly.

The sliding window model splits the stream by windows of constant length w and groups objects in the same window together. Each object belongs only to one window, and each window contains objects from the interval $\big[wn, w(n+1)\big)$, where n is the window's identifier starting from 0. The window hence "slides" from one part of the stream to the other as time progresses. When we relax the condition of disjunct windows and allow an object to exist in more windows simultaneously, a concept of overlapping sliding windows is received. In contrast to damped variant, all objects within the same window are of equal importance.

As we want to monitor the client's activity over time and compute statistics upon them, a non-overlapping sliding window model is a perfect choice. Therefore, we will group the incoming data by their IP address, whereas each IP address will have an associated list of windows, in which the statistics for a particular IP will be stored. Storing only a single (last) window is insufficient because we want to look at the communication process in more detail. For this reason, several historical windows will need to be kept. This section will further discuss which statistics are computed and how is the whole windowing system managed.

Collected Statistics

As already indicated, values extracted from each packet (Table 4.1) are not saved into windowing structures as they come. Memory conservation and relevant traffic description are achieved by statistical features like count, mean, min, max, standard deviation, and others computed instead. Each of them represented by a single value, achieving an acceptable memory usage even when millions of clients are being processed at the same time.

The Statistics logger module uses 18 unique features (Table 4.2) computed and logged in each time window for every communicating IP address. This is represented by the "Current window per-IP data" component in Figure 4.2. However, the classifier would still not be

Feature identifier	Description
window_id	Window identifier
pkts_total	Total number of packets
bytes_total	Sum of bytes of all packets
tstamp_start	Timestamp of the first packet
tstamp_end	Timestamp of the last packet
pkt_arrivals_avg	Average time between packet arrivals
pkt_arrivals_std	Standard deviation between packet arrivals
pkt_size_min	Minimum packet size
pkt_size_max	Maximum packet size
pkt_size_avg	Average of packet sizes
pkt_size_std	Standard deviation of packet sizes
tcp_pkt_count	Number of TCP packets
udp_pkt_count	Number of UDP packets
icmp_pkt_count	Number of ICMP packets
port_src_unique	Number of unique source ports
port_src_entropy	Source port entropy
conn_pkts_avg	Average number of socket-to-socket transfers
hdrs_payload_ratio_avg	Average of header to whole packet size ratio

Table 4.2: List of stored window features for each IP address.

able to correctly predict whether the malicious traffic is present or not. If only 1 window (such as a 1-second timeframe) was analyzed, quick traffic bursts of legitimate traffic could produce a large number of packets in a short duration, possibly having similar characteristics as DDoS attacks in such a short period. Therefore, statistics of a single window are not suitable on their own. However, we may compute advanced characteristics better describing the traffic if we combine several of these windows together. For this reason, values in each time window are not fed into the ML model directly but represent only auxiliary values used to compute more complex statistics.

When several of these windows are combined, a total number of 32 features is produced (Table 4.3). They can be divided into two logical groups based on their way of computation as Window summary statistics and Inter-window statistics. Whereas Window summary statistics were mainly inspired by existing research like [77], and [18], Inter-window statistics were formed based on the author's domain knowledge of DDoS characteristics.

Window summary statistics summarize the contents of all combined windows. Note that statistics with the same names as in Table 4.2 (except `pkt_size_min` and `pkt_size_max`) are not a sum but an average of the underlying statistical value over all windows. This should theoretically allow the classifier to provide relevant results even if it was trained with a different number of summarized windows than it is estimating because the statistics over all windows are averaged. Instead of storing flat counts (like TCP, UDP, and ICMP traffic), we compute their traffic shares, and again, provide an average of their values over all windows. These groups of statistics aim to capture common characteristics of DDoS in a short period of time. The rationale behind their usage is explained in the following points:

- `pkts_total`, `bytes_total`, `pkt_rate`, `byte_rate` – Volumetric attacks will have these values significantly higher than regular traffic over an extended period of time. Slow attacks may resemble patterns of regular traffic or be even lower.

Feature identifier	Description
Window summary statistics	
`src_ip`	IP address for the corresponding statistics
`window_count`	Number of summarized windows
`window_span`	Difference between the last and the first window ID
`pkts_total`	Total number of packets
`bytes_total`	Sum of bytes of all packets
`pkt_rate`	Estimate of pps value
`byte_rate`	Estimate of bps value
`pkt_arrivals_avg`	Average time between packet arrivals
`pkt_arrivals_std`	Standard deviation between packet arrivals
`pkt_size_min`	Overall minimum (not avg of min) packet size
`pkt_size_max`	Overall maximum (not avg of max) packet size
`pkt_size_avg`	Average of packet sizes
`pkt_size_std`	Standard deviation of packet sizes
`proto_tcp_share`	TCP traffic share
`proto_udp_share`	UDP traffic share
`proto_icmp_share`	ICMP traffic share
`port_src_unique`	Number of unique source ports
`port_src_entropy`	Source port entropy
`conn_pkts_avg`	Average number of socket-to-socket transfers
`hdrs_payload_ratio_avg`	Average of header to whole packet size ratio
Inter-window statistics	
`pkts_total_std`	Std of a total number of packets
`bytes_total_std`	Std of a total number of bytes
`pkt_size_avg_std`	Std of packet size averages
`pkt_size_std_std`	Std of packet size stds
`pkt_arrivals_avg_std`	Std of a packet average time between packet arrivals
`port_src_unique_std`	Std of number of unique source port number
`port_src_entropy_std`	Std of source port entropy values
`conn_pkts_avg_std`	Std of number of packets per connection averages
`hdrs_payload_ratio_avg_std`	Std of header to whole packet ratios
`dominant_proto_ratio_std`	Std of ratio of the dominant L4 protocol
`intrawindow_activity_ratio`	Host activity estimate within the summarized windows
`interwindow_activity_ratio`	Host activity estimate during the summarized period

Table 4.3: Complete list of summary statistics over several windows for a single IP. *Std = standard deviation.

- `pkt_rate`, `byte_rate` – $pps_e = \frac{n_{pkts}}{t}$, $bps_e = \frac{n_{bytes}}{t}$ are packets per second (pps) and bytes per second (bps) estimates. n_{pkts}, and n_{bytes} are the total number of packets/bytes collected over the whole summarized window, whereas $t = t_e - t_s$, t_e being a timestamp of the client's last communication in the last window and t_s a timestamp of the client's first communication in the first window. Note that these are only estimates because a situation when the client's data are not sampled, or it

does not reach the sufficient number of packets to be included in the window statistics (discussed later), may occur.

- `pkt_arrivals_avg`, `pkt_arrivals_std`, `pkt_size_std` – DDoS attacks are typically performed by bots running malicious packet-crafting software that produces packets at a specific rate and sends them to the victim. The packets are often the same, are produced one after another very shortly, and thus arrive in regular intervals. These features aim to detect this behavior, where we expect much smaller values for an attack than for regular traffic.

- `proto_tcp_share`, `proto_udp_share`, `proto_icmp_share` – TCP is the most dominant L4 protocol on the Internet nowadays. According to our measurements from April 2021 on CESNET's backbone network, TCP has a share of 78.6% and UDP 20.03% of all traffic. A regular client will, therefore, utilize these protocols with similar share ratios. Significant deviations from them – like 99% of UDP or ICMP share may signalize UDP/ICMP flood attack. However, these features are mutually collinear, so some machine learning methods like Linear or Quadratic Discriminant Analyses may not be able to provide easily interpretable results.

- `port_src_unique`, `port_src_unique` – Malicious software for DDoS packet-crafting often utilizes port randomization techniques. These two features aim to detect an exceptionally large number of ports and port entropy on a similar principle as presented back in Section 2.3.1.

- `conn_pkts_avg` – An average number of packets sent per connections. This feature partially relates to the previous point, as attackers often randomize source ports and thus send a very small number of packets from them. Recall that socket-to-socket communication is defined by a (Source IP, Source port, Destination IP, Destination Port) 4-tuple. If an attacker does port randomization, this value will be significantly smaller than for regular users.

- `hdrs_payload_ratio_avg` – Despite its name, this feature describes the ratio between the header and the whole packet size. Some attacks (such as SYN flood) typically send only headers without any payload to maximize possible packet throughput by not wasting bandwidth on the unnecessary payload. On the other hand, other attacks aim to exhaust the target's bandwidth by creating packets with large, mostly junk payloads. Therefore, this value is expected to be either very small (big payloads) or close to 1 (no payloads) in the case of the attack.

The second logical group, as the name suggests, is computed based on properties between different windows. Most of these features utilize a standard deviation between all summarized windows with the rationale that attackers tend to produce malicious traffic with predictable metadata patterns. Even if the packets' size is randomized and duration between their sending nondeterministic, the randomization patterns will eventually produce relatively similar statistics in each analyzed window if we look at the traffic from a longer perspective. The standard deviation of these statistics should indicate whether the packets are generated from a legitimate client (high variance between windows) or a malicious source (lower variance). Another two, activity ratio statistics are also computed:

- `intrawindow_activity_ratio` – estimates the host's activity within the summarized window: $\frac{t_e - t_s}{l_w}$, where t_e is the host's communication end timestamp, t_s is the com-

munication start timestamp, and l_w is the length of the window. This gives us an approximation of how long the host was active within the given window. As already mentioned, malicious software generating DDoS will typically send packets in small deterministic intervals. Therefore, its intra-window activity should be close to 1, as new packets are flowing from the given IP constantly. Whereas legitimate hosts can hit the start of the window a few times, their burst-based communication should not achieve values close to 1 in a longer-term. This value is computed for each window, and an average of them is provided.

- `interwindow_activity_ratio` – estimates the host's activity between all summarized windows: $\frac{n_w}{id_e - id_s}$, where n_w is the number of summarized windows, id_e is the ID of the last, and id_s the ID of the first summarized window. If a host is communicating continuously, this value is 1. However, if the host does not send enough data or its data are not sampled at all, no window entry for it will be created, so there may be gaps in the IDs of windows to be summarized. Gaps will cause the inter-window activity to be less than 1. Again, attackers continuously sending data will have their ratio as 1 almost exclusively (except for slow attacks), whereas burst-based traffic of legitimate traffic should achieve smaller values of this indicator.

4.3.3 Statistical Features Computation

The previous subsection discussed *which* features are calculated as well as the rationale behind their use. This part will cover *how* they are calculated in an online scenario with an endless data flow.

As it can be seen in Table 4.2, the system collects 5 counts, 4 averages, 2 standard deviations, 2 unique counts, and 1 entropy estimation for every IP in each window. As the system may need to process tens of gigabits of traffic per second, it is computationally infeasible to store extracted features for all packets in each time window. As a result, we need an effective way to compute these values without storing them in the memory. For this purpose, stream data mining and processing algorithms will be used.

The most frequent feature – counts, can be computed easily. For each packet that matches a certain condition, a corresponding counter is incremented by one. For example, if the packet contains a TCP header, increment the counter for TCP segments. However, computing other statistics like unique elements or entropies becomes a little more tricky.

The commonly-known form of the mean (Eq. 4.1) and standard deviation (Eq. 4.2) require to process of the whole dataset before producing a result. This fact becomes problematic for our case since we cannot save the data for later processing. Therefore, stream (known as running or moving) algorithm variants need to be used. Later on, this part will also describe how the system computes stream entropy and the number of unique elements.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.1}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{4.2}$$

Streaming Mean Computation

The computation of the streaming mean is relatively straightforward and can be derived from its standard form in Eq. 4.1. Assume μ as $\overline{x}$. Then:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}\left(\sum_{i=1}^{n-1} x_i + x_n\right) = \frac{1}{n}\big((n-1)\overline{x}_{n-1} + x_n\big) = \frac{n\overline{x}_{n-1} - \overline{x}_{n-1} + x_n}{n}$$

Therefore:

$$\overline{x_n} = \overline{x}_{n-1} + \frac{x_n - \overline{x}_{n-1}}{n} \tag{4.3}$$

Given the above computation and the result in Eq. 4.3, we are able to compute the running mean by keeping a counter of how many elements were processed and the previously computed mean (x_{n-1}).

Streaming Variance Computation

There are several ways of computing streaming variance. Nevertheless, some of them are numerically unstable. Welford's algorithm [113] provides a numerically stable way of online variance computation in a single pass. This is achieved by keeping an auxiliary value s updated for each new element (Eq. 4.4) along with running mean $\overline{x}$ (Eq. 4.3). After all the elements are processed, streaming variance is computed according to Eq. 4.5. Streaming standard deviation is then a square root of the variance, as usual: $\sigma = \sqrt{s^2}$

$$s_n = s_{n-1} + (x_n - \overline{x}_n) * (x_n - \overline{x}_{n-1}) \tag{4.4}$$

$$s^2 = \frac{s_n}{n-1} \tag{4.5}$$

Number of Unique Elements

Determining the number of unique elements (cardinality estimation) is needed in two places. These are counting the number of unique ports and the number of unique connections in order to compute the average number of packets per connection. A typical way of doing this would be to implement a set-like structure either by hashing or a tree. This principle would be sufficient for most cases, but in the data-mining world, there is one significant disadvantage – memory requirements.

Since we are working with data streams, we cannot predict how many elements will be processed in advance. Will it be a hundred or a million? In a regular set structure, each processed element would need to be saved, thus consuming additional computer memory after each add operation. This behavior is highly undesirable because memory requirements to maintain such structure could become unbearably high, eventually leading to significant performance degradation or even program crashes. This issue is addressed by the use of probabilistic data structures.

Probabilistic data structures are data structures with a probabilistic component, which is used to reduce time or space trade-offs. They cannot give a definite answer but rather provide an approximation within some maximum error range [102]. For the purpose of cardinality estimation, the HyperLogLog probabilistic algorithm is used.

HyperLogLog's fundamental idea is based on the observation that the cardinality of the multiset of uniformly distributed values can be estimated by calculating the maximum number of leading zeros of each number in a set. Simulation of the uniform distribution is achieved by hashing each element and logging its result to one of the multiset subsets (buckets). An estimate of distinct elements is then calculated as 2^N, where N represents a harmonic mean of the maximum values of observed leading zeros of each bucket [40] [38].

The standard error of HyperLogLog can be controlled by the number of buckets and their width. In most available libraries, the bucket's width is determined automatically, and only the parameter n is accepted. Parameter n defines the number of buckets as $m = 2^n$, and the maximum error is then typically about $\sigma = 1.04\sqrt{m}$ [38]. This allows us to estimate cardinalities beyond 10^9 with a typical accuracy of 2% while using 1.5 kB of memory. In our case, setting $n = 9$, and thus achieving a standard error of 4.6%, is totally acceptable.

Entropy Computation

Source port entropy is computed using a mechanism of sampling alongside a standard way of Shannon's entropy computation specified in Eq. 4.6. Online data stream sampling can be done by a number of techniques, the proposed system using Reservoir sampling.

$$H(X) = \sum_{n}^{i=1} P(x_i) \log P(x_i) \tag{4.6}$$

Suppose the objective is to maintain a random sample of n elements without replacement from a stream of N elements, where N is not known a priori. Let the elements be $a_1, a_2, \ldots a_N$. In Reservoir sampling, the first n elements of the stream are deterministically included in the sample. For $t \geq n$ when a_{t+1} arrives, it is included in the sample with the probability $\frac{n}{t+1}$. If an element is selected to be included in the sample, a randomly chosen element from the current sample is replaced. This way, the resulting sample is equally likely to be any of the subsets of size n composed of stream elements $a_1, a_2, \ldots a_N$ [64].

If a sufficient number of samples with respect to the window length was given, this technique should provide a sufficiently accurate estimate of the source port entropy corresponding to a particular IP address. Accuracy for large streams can further be improved by utilizing specialized techniques for sample entropy [91] or with techniques designed to compute entropy directly from streams [22].

4.4 Machine Learning Pipeline

With respect to various considerations regarding the system design outlined in Section 4.2, the following pages will present a view of the whole system pipeline. As already discussed, the ML-based detector will accept (sampled) network packets, compute, store, and group its relevant statistics, and provide them for further processing after a sufficient amount of them was collected. The provided statistics will further have to be preprocessed and finally fed to the ML model, determining whether they are malicious or not (Figure 4.2). The following section will examine the functionality behind each of these pipeline components in more detail.

4.4.1 Feature Management

The process of how features are extracted and computed has already been described in Section 4. This subsection will now put these statistics into the ML pipeline and explain how they are fetched for further analysis and processing.

As shown in Figure 4.2, the extracted packet features are firstly processed in the "Current window per-IP data" component of the Statistics logger. This component produces per-IP window features shown back in Table 4.2. Nevertheless, the pipeline also needs to control the window creation.

Although this could be done internally, a better approach is to let an external process handle windowing by issuing external signals (triggers), marking the end of the current window, as well as starting the new one. This gives more control to the process using the Logger and allows for easier multithreading, where the management thread is in charge of windowing, and worker threads handle packet logging and statistics computation.

Upon each window end, we need to determine whether the gathered statistics for a particular IP address are usable. In this context, usability is considered a simple check against the threshold of minimum packets per a given window. Consider a scenario when only a single packet in a window was received. Processing such a window would make statistics like averages, standard deviations, unique counts, min, max, and others worthless because 1 sample is simply not enough to generate a representative statistic of the whole window. Windows with such small values could potentially skew the statistics of the whole communication, so we want to filter them out prior. Specifying a minimum threshold, say 10 packets per second, will ensure that windows with fewer data will not be included in the statistics and thus not skew it.

After the window statistics are filtered out, those who satisfy the minimum packets conditions are recomputed, and *Window summary statistics* from Table 4.3 are obtained. They are then saved to the "Per-IP all windows statistics" component in Figure 4.2. As already mentioned in Section 4.3.2, we want to look at more than one window in order to provide relevant classification results. For this reason, a data-pulling mechanism needs to be designed.

Retrieval of all window statistics needs to be controlled similarly to window statistics for the current window. A threshold of the minimum number of windows is specified, and statistics are only retrieved if this minimum was reached. Therefore, a "Data pulling" component is suggested, which lets the caller know if there are any available IP addresses with a sufficient number of windows to be processed. If yes, the caller may utilize its interface to retrieve those window statistics.

After all of the window statistics for a particular IP are pulled, they are summarized, primarily by computing averages of all processed windows. Inter-window statistics (Table 4.3) are computed as well. Finally, after all these steps, statistical features for a given IP are returned for further processing and classification.

4.4.2 Data Analysis

Although not a direct part of the ML pipeline itself, data analysis is necessary for most machine learning projects. Data analysis aims to reveal relationships between data and thus provide valuable intelligence for the machine learning engineer. In general, there are three types of data analyses according to the number of variables considered: univariate, bivariate, and multivariate.

Univariate data analysis aims to describe only one variable. Univariate statistical descriptors used in this project include the mean, median, variance, kurtosis, and others. The visual univariate analysis utilizes boxplots, empirical cumulative distribution functions, histograms, and kernel density estimates.

Bivariate data analysis examines the relationship between two variables. These can be two features, or a feature and a target variable. It is often beneficial to add target variable y to traditional univariate plots like boxplots, histograms, or kernel density functions, which will then display the distribution of the variable with respect to y. The proposed system also utilizes this feature. Another possibility is to display bivariate relationships are scatterplots.

Multivariate data analysis compares multiple variables to each other. This creates 3D (for 3 variables), or generally ND plots. Due to low interpretability by humans, this type of analysis is not used.

4.4.3 Data Preprocessing

Data preprocessing is a set of data filtering and transformation techniques applied to the data before passing them to the machine learning method. Preprocessing is used to improve the quality of data, which simplifies their processing and enhances the learning and classification capabilities of the ML algorithm. Typical actions performed in these steps include data cleaning, encoding categorical and non-numeric types, data standardization, and alternatively performing feature projection or selection. This subsection will further present some of these techniques performed in the proposed system's pipeline to achieve maximum performance.

Cleaning

Data cleaning generally includes handling the entries with missing values and optionally removing redundant features or entries with outliers[3]. These practices are typically crucial for noisy data or data coming from measurements prone to error. Missing values and outliers removal are not relevant for the proposed system since all features are generated by the Statistics logger module. Therefore, we do not expect any missing values on the input. Outliers are also not a problem since we know that they originate from a valid source and thus are not a product of an invalid measurement or other error.

Nevertheless, not all statistical features from the Statistics logger's output are usable for classification. Therefore, the cleaning phase drops `src_ip`, `window_count`, and `window_count` columns before further processing described later in this section. The `src_ip` column is used for feature grouping (Section 4.4.4), whereas `window_count` and `window_span` are only informational and are not used for any specific purpose in the current version of the system.

Variable Encoding

Variable encoding procedures are used for non-numeric values or numeric categorical values. In these cases, a qualitative[4] variable type has to be converted to a quantitative type, so that machine learning algorithms will be able to process it. There are many types of encodings based on the use-case, one of the most popular being dummy encoding. Nevertheless,

[3] An observation that lies an abnormal distance from other values in a random population sample.

[4] A property that cannot be numerically measured.

as our input features are produced internally, they have been intentionally chosen to be representable by quantitative numerical types, and thus no variable encodings are required.

Standardization and Normalization

Some machine learning models such as neural networks or SVMs require their input features to be in the same interval. In some cases, unscaled input variables can result in a slow, unstable learning process or even causing learning to fail. These problems can be avoided by input features scaling. Scaling can be done by the standardization or normalization techniques applied as a part of the preprocessing pipeline.

Normalization is the process of rescaling the original data into the specified interval, most typically $[0, 1]$. Normalizing the value x is done via the *MinMax* function shown in Eq. 4.7, where x_{min} and x_{max} represent the minimum and maximum values of the given feature across the whole training dataset.

$$MinMax(x) = \frac{x - x_{min}}{x_{min} - x_{max}} \tag{4.7}$$

Standardization (whitening) means rescaling the distribution so that the mean of observed values is 0 and the standard deviation is 1. Standardization assumes that the scaled data fit the Gaussian distribution. If this condition is not met, the reliability of results or the ML model's learning ability may be negatively affected. Standardization is achieved by the *Standard* function according to Eq. 4.8. Similar to *MinMax*, statistical features mean μ_x and standard deviation σ_x have to be computed across the training dataset in prior.

$$Standard(x) = \frac{x - \mu_x}{\sigma_x} \tag{4.8}$$

The proposed system implemented and experimented with both of these techniques. The currently preferred one is *MinMax*, although no significant differences between models' performance have been discovered, as further discussed in Chapter 6.

Dimensionality Reduction

One of the last steps of ML preprocessing is dimensionality reduction. This set of procedures is used when working with very high-dimensional data in order to prevent the curse of dimensionality[5]. In machine learning, the curse of dimensionality is closely related to the peaking phenomenon. This phenomenon states that for a fixed number of training samples N, an initial improvement of the classifier's predictive power is achieved by increasing the number of dimensions, but increases beyond a critical value result in predictive power to deteriorate instead [105]. This issue is tackled by techniques of feature projection and feature selection.

Feature projection algorithms like Principal Component Analysis (PCA) or Linear Discriminant Analysis (LDA) aim to transform the data from high-dimensional space into a space of fewer dimensions. For example, PCA performs linear transformation, but other non-linear transformations are available as well.

Instead of projecting features from one space to the other, the process of *feature selection* selects a subset of the most relevant features without transformation. The selection is primarily beneficial for the model's interpretability, as we exactly know which features were

[5]With an increase of dimensionality, the volume of the space increases so fast that the data are becoming sparse, causing numerous problems.

selected and how they contribute to the overall model decision-making. This is unavailable in projection techniques since the methods transform the state space, causing it to lose information about the particular dimensions.

The selection process may be performed by numerous means. The most straightforward principle depends on removing features with very low variance. More complex, already mentioned feature selection principle with decision trees (Section 3.5) works with Gini impurity or information gain (entropy) principles. In this case, features with the highest decrease in the Gini index or entropy are selected as the most relevant features providing the most information gain. Several other, like sequential feature selection and recursive feature elimination, can also be used.

Due to the fact that the ML system uses only 32 features (Table 4.3), dimensionality reduction techniques are not used as a part of the pipeline by default. However, the system was designed so their functionality can be easily plugged in and turned on and off as required, as further discussed in Chapter 5.

4.4.4 Model Training

After extracting, analyzing, and preprocessing the data, they can finally be fed to the machine learning model. After the model is trained, we typically want to evaluate its performance on yet-unseen data. For this reason, the dataset is typically split into two separate parts – train data and test data. Train data are used to fit the model parameters and test data to evaluate the model's performance, such as by computing prediction accuracy. See Chapter 6 for further information about model evaluation.

In our case, one additional action with the data must be performed before the training. In general, ML methods expect that training samples are not mutually dependent. This assumption is almost always wrong, but we can improve classification capabilities if we have information about their dependence. Recall the functionality of the Statistics Logger module from 4.4.1. After a sufficient amount of windows for a particular IP address are collected, they can be pulled from the structure. Such pulled windows are then summarized, additional statistics are computed, and so a new dataset sample is created. However, the given IP address will probably communicate longer, creating new windows containing its particular statistics. Again, after sufficient numbers of these windows are collected, they may be pulled, processed, and another dataset sample created. This can become somewhat problematic because we obtain several highly dependent dataset samples (coming from the same IP).

If such highly dependent samples were present in both training and the test dataset, the ML method would perform excellently during the evaluation phase but might fail miserably in the real world. This is based on the fact that if a yet-unseen but highly dependent element is classified, a ML algorithm should have an easy time because it resembles very similar characteristics to the data used during the training. This phenomenon could significantly skew the method evaluation results and thus needs to be avoided.

The above-mentioned situation could be solved by defining a common feature for each strongly dependent sample and grouping them based on it. Such a common grouping feature is the IP address, which is prepended to each created sample (Table 4.3). The dataset splitting mechanism then needs to respect these groups and ensure that samples from the same group will not be split between train and test dataset subsets. By enforcing this splitting policy, more accurate and unskewed results of dataset evaluation may be obtained.

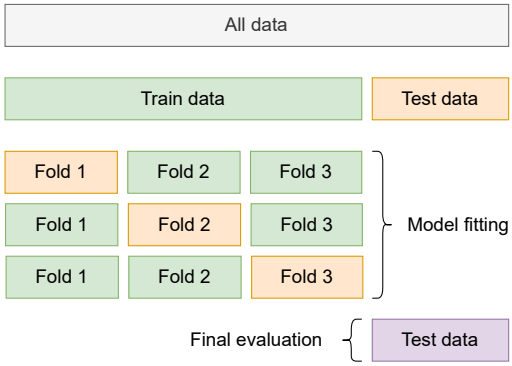


Figure 4.4: 3-fold cross validation with additional test split.

4.4.5 Model Selection

When the dataset is fully prepared, and its elements are properly grouped, we may finally select the most appropriate machine learning model. This process is known as model selection. The model can be chosen according to different criteria. In our case, the concerning parameters are accuracy, low false-positives ratio, and computational complexity.

In an ideal ("data-rich") scenario, the dataset would not be split into two but rather into three disjunct subsets – training, validation, and test. The candidate models would be fit on the training set, evaluated and selected on the validation set, and the performance of the final model reported on the test set [46].

Nevertheless, splitting the dataset into three parts is often too data-demanding, which is a problem in most ML projects. For this reason, approximation techniques like probabilistic measures and resampling methods are used for the model selection. Probabilistic methods are only applicable for simpler linear models, so resampling methods are typically preferred.

This project compares and selects models using Cross-validation (CV), a popular method based on resampling. This procedure splits the dataset into k subsets (called k-fold CV), performs the training on $k-1$ of them, and evaluates the fitted model on the remaining one. The process is then repeated for every combination of $k-1$ subsets. The final model's evaluation result is computed as an average over the results of all combinations. Similar to regular train-test splitting, the technique determines the model's ability to predict yet-unseen data during the training. However, since the performance evaluation is run multiple times across different dataset subsets, the final performance estimation is generally more accurate and thus better represents real-world scenarios while being very efficient with the data it uses. In some cases, the available data is split before the CV is applied (Figure 4.4). This creates another test dataset subset, which may be used for final evaluation after the best model with its hyperparameters is selected based on the CV procedure.

In the system, cross-validation is used for model performance comparison as well as model evaluation like validation and learning curves plotting.

4.4.6 Hyper-Parameters Optimization

The process of hyper-parameter fine-tuning is typically interleaved with or as a part of the model selection process. Hyper-parameters are parameters of the model that are not directly learned within estimators. Typical examples are the number of neighbors in K-nearest neighbors or the number of neurons in the hidden layer of a neural network. During

the optimization procedure, we aim to find the values for these parameters that perform the best upon the analyzed dataset.

An approach to solve this problem is to define a parameter space to be explored and use the Cross-validation technique, as explained in Subsection 4.4.5. In such a scenario, the model is trained with combinations of hyper-parameters from the parameter space, and its performance is estimated with CV. The combination of parameters with the best score is chosen as the best model configuration.

There are numerous ways of how to search the parameter space and thus find the best combination. This project uses the technique called *GridSearch*, which exhaustively generates candidates from a grid of parameter values and evaluates the model for each such combination. This method provides the best results but might be computationally costly because the whole state space needs to be explored in a "brute-force" manner. Other techniques like *RandomizedSearch* or various heuristics also exist. These do not explore the whole parameter space but only its subset, resulting in a much faster search process at the cost of typically finding a local maximum instead of the global one.

4.5 DDoS Datasets

Despite all design considerations discussed throughout this chapter, the machine learning system would not be able to provide relevant decisions without quality balanced datasets. Such datasets are required to resemble patterns of both legitimate and attack traffic realistically. Additionally, datasets in their raw PCAP form are required since the project utilizes a custom feature extractor. As discussed further in this section, obtaining such datasets proved to be quite problematic. The section further looks at various publicly available datasets, briefly describes them, and explains why they were or were not used in the project. Modifications and transformations of used datasets will also be outlined. The provided datasets list is not exhaustive but covers most of the relevant public datasets to this day, so its usage as a reference for future research is possible.

4.5.1 Current State of Public Datasets

In general, the field of DDoS detection and mitigation suffers from a severe lack of availability of quality datasets resembling the behavior of modern computer networks and threat actors. Currently, there is no standardized dataset used for testing state-of-the-art detection techniques. This fact has led various research works to utilize various types of non-standard, sometimes exotic datasets. These are often outdated, irrelevant to the DDoS field (e.g., datasets for intrusion detection), or publicly unavailable. They are either self-generated or captured from internal networks and kept in secret.

This phenomenon is mainly caused by privacy issues because packet payloads and IP addresses cannot be publicly shared. Another reason is the fact that DDoS traffic is not trivial to capture due to high packet rates and overall overload of network equipment during an active attack. On top of these setbacks, one has to think about a competitive struggle between big players in the industry. There is no doubt that the market leaders like Akamai, Imperva, or Cloudflare have their own DDoS traffic from past attacks captured but are unwilling to make it public as the data are precious commodities nowadays. As far as the author of this document is aware, CESNET does not have such DDoS attack capture available.

Historically, a to-go variant for DDoS and generally intrusion detection was the KDD-99 dataset [110] created at The Fifth International Conference on Knowledge Discovery and Data Mining 1999. The dataset is provided as a file with 41 high-level features describing the user's session activity like the number of accessed files, opened shells, or bytes transferred between the communicating parties. Its cleaned and improved variant, NSL-KDD [104], was, and respectively still is, a popular choice for testing intrusion detection systems. Although designed for intrusion detection, it also contains several DoS-labeled sessions and thus was widely used for DDoS detection ML projects as well. However, very few features from the dataset are actually usable for DoS purposes, and thus the dataset was mainly used because nothing "better" was available at that time. As reported by [80], KDD'99 was used in 149 research articles across 65 journals between 2010 and 2015. In some of the recent DDoS detection-related research, NSL-KDD is still being used [31]. Several other datasets for DDoS methods validation created between 1998 and 2014 have also been mentioned in [7], but most of them are not publicly available to this day anymore.

In recent years, several popular datasets for IDS and D(DoS) detection have been published by the Canadian Institute of Cybersecurity (CIC) in cooperation with the University of New Brunswick. These include datasets ISCXIDS2012 [15], CIC-IDS2017 [17], CIC DoS dataset 2017 [16], CSE-CIC-IDS2018 [18], and CIC-DDoS2019 [19]. They have become relatively popular in the community, and most of the published papers nowadays use one of them for training and evaluation purposes. However, several problems have emerged during their testing with our system, as described later in the section. Therefore, finding a non-outdated, well-documented dataset of raw PCAP data with acceptable quality is a relatively challenging task nowadays.

4.5.2 Attack Traffic Datasets

Due to the higher number of available datasets with questionable quality, the project did not restrict itself to a single dataset only. Instead, several datasets were downloaded, tested, and intermixed to produce the final result. This process ensures better generalization capabilities of the trained model as well as more accurate results with respect to real-world scenarios. Most of the datasets are publicly available online downloadable by anyone. Two datasets are available after their owner's approval, and one is a private dataset of legitimate traffic captured on CESNET's networks.

CAIDA "DDoS Attack 2007" Dataset

CAIDA "DDoS Attack 2007" Dataset [20] is currently one of the most popular DDoS public datasets. It contains approximately one hour of anonymized traffic traces from a real DDoS attack on August 4, 2007 (20:50:08 UTC to 21:56:16 UTC). The data consist of a volumetric attack type, which attempts to block access to the target server by consuming computing resources of the server and underlying network infrastructure.

The one-hour trace is split up into 5-minute PCAP files. The total size of the dataset is 21 GB. CAIDA states that it contains only communication between the attackers and the server, with non-attack traffic removed as much as possible. The dataset is only available after approval from CAIDA through a data access request form.

After downloading, the dataset was separately split into attacker's requests (to-victim) and responses (from-victim). For our purpose, only the attacker's traffic was the point of interest. Therefore, all 5-minute traffic captures were merged into a single PCAP file and

processed like that. After these filters were applied, the modified dataset contained 85 M of packets, 8736 unique IP addresses, and span 36.1 minutes.

Although the dataset is quite old, it contains precious information about real DDoS attacks and their characteristics. Nevertheless, its older age may cause some characteristics not to correspond to attacks in modern network environments. These primarily include timing, as modern networks can transfer data at much higher rates with lower round-trip times. According to the protocol hierarchy analysis, the dataset is composed of 91.94% of ICMP traffic, above 8% of TCP traffic and a only a few UDP segments. For these reasons, it may be need to be combined with other, fresher datasets to resemble as many real-world characteristics as possible.

CIC-DDoS2019

CIC-DDoS2019 Evaluation Dataset [19] looks very good on the paper at first glance. It is supposed to contain various DDoS attacks like UDP flood, SYN flood, WebDDoS, NetBIOS, and many others. The dataset is split into two days, each containing a series of raw PCAP files and an associated CSV with per-flow features already extracted.

Although the CSV variant seems to be comprehensive and well-labeled, working with the PCAP version is not straightforward at all. Firstly, time-interval captures were merged into one file with the `mergecap` utility. According to the dataset's webpage and CSV analysis, the attacks should come from `172.16.0.5`, so the merged data were filtered to only include packets from the given source IP with `tcpdump`.

Despite limiting the capture to 1 source, malicious and legitimate traffic could still not be clearly distinguished. This is because the given IP sends various types of traffic, clearly not limited to attacks only. The other crucial factor is that the PCAP variant of the dataset does not explicitly specify demarcations between the attacks. Although these times are specified on the web, a manual analysis has revealed that provided timestamps do not correspond to PCAP timestamps. At last, the amount of traffic (224 M for the first day, 55 M for the second) is so enormous that manual analysis was also hardly performable.

Regardless of all the previously mentioned problems, a limited packet analysis was made. During the process, it was discovered that the SYN Flood attack, which should begin at 11:28 and last until 17:35, starts at 16:28:42.58 UTC and only lasts until 16:43:58.06 UTC. Therefore, there is probably a 5-hour shift between actual and declared timestamps. Nevertheless, since all other attacks are UDP-based, distinguishing their starts and ends is not always possible. Due to these reasons, attacks were not extracted one by one, but the whole file captures were used. This approach causes that some legitimate traffic will be unconditionally included in the final file. This effect was reduced by additionally cleansing the data by removing around 46 k of TCP traffic, mostly HTTP and SSH, which are indeed not part of these attacks. The resulting files thus contain only UDP traffic and packets from the SYN Flood, but guarantee that no legitimate traffic is included within the capture cannot be made.

The purpose of previous paragraphs was not to comprehensively review all dataset flaws but to provide an overview of its state and question its quality and credibility. Despite all of these setbacks, the dataset was extracted with the proposed logging mechanism and marked as attacking traffic. However, it will provide sub-optimal results due to the reviewed problems. Experiments with it will be further described in Chapter 6.

CSE-CIC-IDS2018

CSE-CIC-IDS2018 [18] is a dataset focused on anomaly and intrusion detection. It contains different attack scenarios like Brute-force, Hearthbleed, Botnet, DoS, DDoS, and others distributed across 10 days. Legitimate synthetic traffic based on profiles is also provided. The dataset was generated on a topology of almost 500 computers.

Similar to CIC-DDoS2019, the dataset also have contains several flaws. This project used datasets from days 02-15 (Slowloris & Goldeneye), 02-16 (SlowHTTPtest & HULK), 02-20 (LOIC-HTTP & LOIC-UDP), and 02-21 (HOIC & LOIC-UDP). Data within these days are provided as PCAP captures from particular computers, whereas attack targets are specified very ambiguously. Fistly, merging of all the captures within one file with `mergecap` was tried. This returned various errors probably caused by non-compliant PCAP creation software or capture file corruption. Therefore, `tcpdump -r filename.pcap -w filename.pcap.fixed` command was used to read the files and save them with `tcpdump`, which erroneous PCAP captures to be trimmed. These files were then finally merged with `mergecap`.

The next step was to separate attacking and legitimate traffic from the single merged file. According to the dataset's webpage, there should be 10 attacking IP addresses in 02-20 and 02-21. Therefore, a filter to extract all traffic from these addresses was applied via `tcpdump`. While this process managed to extract 208 M of packets for these IP addresses in 02-21, none in 02-20 were extracted.

Manual analysis of traffic from supposedly attacking hosts in 02-21 shows that the first 203 M corresponded to the first attack that day (DDoS-LOIC UDP) between 10:09 and 10:43. Timestamps were again shifted by 3 hours from UTC as in CIC-DDoS2019, but otherwise correct. This attack was extracted perfectly – it contains all of the packets with no other traffic, as found out by `tshark` analysis. However, the second attack was apparently conducted from different IP addresses or is not present because a brief inspection of the remaining 102 M of packets did not reveal possible attacking IPs. Therefore, only 1 attack from this day was extracted.

In the 02-20 file, despite the web stating that 10 IP addresses should be attacking, no traffic from them was found at all. 02-15 and 02-16 files were also not used due to an enormous amount of PCAP errors, which rendered these captures unusable.

Despite all the setbacks, extracted traffic from 02-21 was marked as attacking and used within the project. The one extracted attack has a solid quality, but several other attacks could not be used due to mislabeling of attackers' IP addresses or PCAP errors.

CIC-IDS2017

Intrusion Detection Evaluation Dataset (CIC-IDS2017) [17] from the Canadian Institute of Cybersecurity wants to address various flaws from IDS datasets created by 2017. These include a lack of diversity and volume, anonymization of packet data, and general inability to address current attack trends. The dataset comprises 5 days of traffic containing abstraction behavior of 25 legitimate users, Brute-force, DoS and DDoS attacks, Web and Database attacks, infiltration, and port scanning. In our case, legitimate traffic and DDoS attacks in the captures from Monday, Wednesday, and Friday are the main point of interest.

As Monday is declared to contain legitimate traffic only, we used this capture file without any modifications. It contains 11 M packets, including ARPs and control traffic like ICMPv6 messages. IP, IPv6, and ICMP traffic could potentially be filtered with a particular `tcpdump`

command. However, since the ML system is supposed to ignore undesirable input, the file can safely be processed as is.

The capture from Wednesday is declared to contain Slowloris, Slowhttptest, Hulk, and Goldeneye attacks from a single IP address. In order to extract these attacks, a `tcpdump` filter with source and destination IPs was used. The first attack (Slowloris) is supposed to start at 9:47. As found out, the times are probably specified in local times as the start of the attack was found to be 12:48:46 UTC based on the manual analysis. Timestamps thus approximately correspond to the times declared on the webpage with a 3-hour shift. However, the last attack – DoS GoldenEye actually ended 3 minutes earlier.

Friday's capture is described to contain ARES Botnet traffic, port scanning, and DDoS LOIT traffic between 15:56 and 16:16. The DDoS is declared to come from 3 distinct machines. Nevertheless, their IPs are merged into one – 172.16.0.1 due to the incomprehensible use of NAT within the network environment. The victim machine is also under NAT with a private IP 192.168.10.50. With this knowledge, traffic between attackers and the victim was filtered.

Similarly to Wednesday, Friday's timestamps were again shifted 3 hours from UTC, but otherwise correct. Manual analysis has revealed that the port scan started at 17:51 UTC sent around 162 k packets to various ports, looking more like TCP SYN Flood with randomized destination ports. HTTP DDoS attack was started at 18:56:31 UTC and lasted until 16:16:12 UTC with around 926 k packets.

Although an acceptable quality, the problem with this dataset is that all attacks come from a single IP. Therefore, they will not produce many dataset entries and could not be used in both train and test dataset subsets due to entries grouping (Subsection 4.4.4).

4.5.3 Legitimate Traffic Datasets

As discovered, datasets of legitimate traffic are much easier to come by than datasets containing DDoS attacks. These are available in several places, either as unrestricted public or public on request. This subsection will present a few datasets of legitimate traffic that were used within the project.

CAIDA Anonymized Internet Traces Dataset

CAIDA Anonymized Internet Traces Dataset [21] contains traces collected from high-speed monitors on a commercial backbone link. Data are provided by CAIDA to encourage research on the characteristics of Internet traffic like including application breakdown, security events, geographic and topological distribution, flow volume, and duration. Data are collected regularly since 2008. Similar to CAIDA's "DDoS Attack 2007" Dataset, data are also available only after a data access request is made.

This project utilizes anonymized traces from the 2016 Equinix Chicago passive monitor. In this case, the captured traffic is split into 1-minute traffic intervals. Each file has around 1 GB of size when compressed, containing between 20 and 40 million packets with anonymized IPs and trimmed L4 payloads. 3 such files from January 2016 and 3 files from March 2016 have been merged, creating a small sample of legitimate traffic. Many more files could have been used, but legitimate traffic from other sources was also used, so additional samples from this dataset were not needed.

CESNET ⟷ ACONET Traffic Capture

More samples of legitimate traffic were retrieved from a private capture between CESNET and ACONET networks. The capture has been made on June 18 2018, lasts 29 m and 26 s, contains over 509 M packets, and its uncompressed size is 393 GB. IP addresses are anonymized and payloads encrypted to preserve privacy. Since there have been no reported attacks at the time of capture, we suppose all the traffic is legitimate, although this statement cannot be interpreted as a ground truth.

CIC Synthetic Data

Some datasets from CIC contain legitimate synthetic traffic generated using user profiles. Although it is generally non-trivial to distinguish legitimate and malicious traffic in these datasets, some dataset parts are marked as "legitimate only" and thus can be safely used as legitimate traffic samples. These include June 11, 2010, in ISCXIDS2012, and July 3, 2017, in CIC-IDS2017. Therefore, synthetic traffic from CIC datasets was also extracted to complement real legitimate traffic from CAIDA and CESNET-ACONET captures, although real-world traffic will always be preferred.

4.5.4 Other Datasets

This subsection will briefly describe other available DDoS datasets that were not used within this project but may be helpful in other circumstances or as future work.

DARPA 2009 Intrusion Detection Dataset

DARPA 2009 Intrusion Detection Dataset [68] looks ideal for our purposes. It is a synthetically generated dataset with HTTP, SMTP, and DNS legitimate background traffic to emulate hosts' behavior on the Internet. It is composed of over 7000 PCAP files with around 6.5 TB of the total size. However, it is available only by request via the IMPACT project [53]. The requests are only available for researchers in the United States and several other approved locations, the Czech republic not being on the list. Therefore, this dataset was unable to be obtained.

CIC DoS Dataset 2017

CIC DoS dataset 2017 [16] is supposed to contain Slow HTTP attacks generated using different tools. Nevertheless, similar to other CIC datasets, it is highly disorganized and hard to interpret. The dataset is composed of a 4.4 GB PCAP file and the text document `attacks.txt` describing the attacks. The file contains 26 lines in the following form:

```
slowread to  74.55.1.4  after 11:02 662 minutes
```

As it may be seen, an example line from the file specifies the type of attack, the target, and the starting timestamp. However, there is no information about attack source IP addresses, and the timestamp is highly ambiguous as there is no ending of the attack specified. On top of that, the dataset's webpage states that attack traffic is intermixed with legitimate one, but there is no information about how such traffic can be identified. Supposed attack traffic (with a destination IP defined by the file) was extracted with a custom script. After the extraction, the file containing packets only with destination IPs specified

by `attacks.txt` contained 788k packets with a size of only 95 MB. Although manual inspection has shown signs of the attack, it was still unclear whether such extracted data contain only attack traffic or some regular was still present. However, only-attack traffic could not be extracted due to the lack of information, so dataset usage was abandoned.

CIC ISCXIDS2012

ISCXIDS2012 [15] was the first dataset from the Canadian Institute of Cybersecurity, which utilized models and profiles to facilitate the reproduction of certain real-world behaviors on the network [101]. The dataset is split into 7 days of over 83GB of raw packet data. Each day contains normal activity, and some contain intrusion data like brute-force, network infiltration, and DDoS. However, its interpretability is not optimal again.

Due to labeling by flows using XML files, a custom XML parser distinguishing attacking and legitimate IPs was written. However, the dataset uses a very limited scope of source IP addresses, which are even usually the same for both attack and legitimate traffic. Therefore, attack traffic cannot be easily extracted from the PCAP by IP addresses, but per-flow extraction needs to be made. The dataset contains only slightly above 40 thousand DDoS flows, out of which some were found to be mislabeled by manual analysis. Therefore, we conclude that writing a custom per-flow packet extractor is undesirable due to low possible benefits for our use case. Although attack traffic was not used, legitimate synthetic packets from the 11th of June 2010 were extracted for complement already-collected legitimate samples, as mentioned in Subsection 4.5.3.

NDSec-1

NDSec-1 [72] is a synthetic dataset created by incorporating traces and log files of various cyber-attacks performed at Fulda University in Germany. It was created in 2016 to benchmark existing intrusion detection systems and support research in new detection techniques. Attacks were performed using state-of-the-art tools in three distinct attack scenarios [6].

NDSec-1 covers a set of classic and novel attack vectors encapsulated within simple but realistic scenarios that can be adopted to most network environments easily. The dataset includes raw network traces, including payload along with Syslog and Windows logs. Data are labeled by bidirectional flows as either legitimate or malicious with additional labels specifying an attack category (such as DoS, brute-force, probe) and optionally additional information about a particular service being attacked.

Although the dataset contains several DoS labeled flows, it was not used within this project. Similar to CIC ISCXIDS2012, labeling is done by flows, and so malicious DoS packets would need to be extracted with a custom script. Since only 2330 flows are considered a malicious DDoS [60], no attempt to extract them was made due to low possible gains by such an action.

CSV Datasets

The following datasets could not be used because they provide only comma-separated-values data (often only on a per-flow basis) with an insufficient amount of information for our purposes:

- KDD-99 [110]
- NSL-KDD [14]

- Boğaziçi University Distributed Denial of Service Dataset [33]

There are also a few datasets on a ML community webpage `www.kaggle.com` such as [58]. However, these are either in unusable for our purposes (CSVs) or provide absolutely no descriptive information, so their usage and interpretability would be highly questionable.

4.5.5 A Note on Generating Custom Datasets

As discussed throughout this section, out of all examined datasets, only a few suffice. The lack of information, a different purpose, or incorrect or flow-based only labeling caused that a majority of public datasets could have been used only partially or not at all. Since the proposed mechanism does not need the attack and legitimate data to be intermixed, generating an own DDoS dataset may also be considered. Although not a direct part of the project, this subsection will briefly suggest how such a dataset could be generated. This could be helpful for further development or as future work.

As already indicated, there is enough synthetic and even real legitimate traffic available. The biggest problem is the attack data. When generating them, one must ensure that their properties are realistic with respect to modern computer networks and other technologies. This can be ensured by using commonly-used operating systems, real-world networking equipment, and software commonly used by threat actors.

Popular choices for generating simple volumetric DDoS attacks are High Orbit Ion Cannon (HOIC), its predecessor LOIC, and updated variant XOIC. These provide a simple graphical interface, allowing to launch of a D(DoS) attack for almost anyone. Depending on the type of attack, various Linux command-line tools like `hping3` or `Scapy` can also be used to craft custom packets by more advanced attackers. Slow DDoS attacks may be generated by utilities like `HULK`, `Slowloris`, `RUDY`, `Tor's Hammer`, and many more. It may also be desired to simulate amplification attacks with services like DNS, NTP, and ICMP (Smurf attack) to cover a wider range of attack vectors.

When creating a custom dataset, note that it is important to generate longer-lasting attacks from multiple IP addresses. According to the extraction mechanism functionality, a single dataset sample is created for several time windows for 1 IP address. Therefore, a quality dataset would contain at least several minutes of traffic from 10 or more IP addresses. Of course, more traffic with more IP addresses would create more dataset entries.

Chapter 5

Implementation and Usage

After outlining the motivation, presenting theoretical background, and discussing design concepts, this chapter will take a look at selected parts of the system's implementation. The project's purpose was not to implement a method instantly usable in the production but rather to ensure the functionality of the designed ML system proposal. Therefore, no big emphasis was put on the system's performance, although several memory optimizations have been made. Instead, the focus was shifted on flexibility so that future experiments can be performed without the need for significant code changes.

Due to excellent support for data processing and various machine learning libraries, Python was chosen as the implementation language. Machine learning functionality is provided by Scientific Learn (Scikit) [82] and XGBoost [24] libraries.

The whole system was split into a series of 5 independent scripts, which resemble the machine learning pipeline described throughout Chapter 4. All of these scripts are highly configurable with an attached YAML file, allowing for replication, modification, and fine-tuning of the pipeline's functionality without the need to touch the code. For demonstration purposes, a single-command run script is also provided. It executes the whole pipeline with a single command and presents the results to the user. The following chapter will now examine the above-mentioned concepts in more detail.

5.1 Configurability

Most of the machine learning projects in the development phase are typically hard-coded solutions fit for a certain purpose according to the dataset they use. In this project, the dataset will always have the same structure regardless of the input traffic. This is because input packets are processed by the custom extraction mechanism usable for any PCAP or PCAPNG files. One may thus supplement different capture files, creating a completely new dataset with the structure the pipeline already understands. This fact allows for much greater flexibility, as users unaware of machine learning principles may still use the model by plugging different PCAP files at the extractor's input and observing the results.

For this reason, the project was developed to be as flexible and configurable as possible. The aim was to provide a solution, which can be used, evaluated, and modified by anyone with none to minimal machine learning knowledge. This is achieved with a YAML configuration file and various command-line options for each script. Options for each script can be viewed with `-h` or `--help` options.

```
libraryName:
    modelName1:
        param1: value
        ...
        paramN: value

    modelName2:
        param: value
        ...
```

Figure 5.1: Models' hyperparameters configuration syntax.

Furthermore, all scripts comprising the system are configurable by a single file in `src/config/config.yml`. Its default version is shown in Appendix B. Top-level keys define the configured module, while the keys on the second level specify a particular module's setting. Every script module can be configured with the configuration, so one script may have (and typically has) numerous configuration keys. Configuration keys used by the particular script are included in its help message.

For example, the user may have control over how are the IP addresses logged, such as by specifying the length of a time window, the minimum number of packets per window, and the minimum number of time windows to log the host. When such a dataset in the form of a CSV file is created, aspects like feature exploration and data preprocessing techniques can also be configured. The user may further choose a particular machine learning model for the training and evaluation. All configuration options can be viewed in `src/config/config.yml` or Appendix B.

The goal of most machine learning projects is to find a suitable model and its hyperparameters best for the solved task. As mentioned in the previous paragraph, the model can be chosen simply by changing the system's configuration file. The model hyperparameters can be provided in a similar manner through a YAML configuration. For this purpose, the `src/config/models.yml` file is provided with the syntax as displayed in Figure 5.1. The top-level key is defined by library name, currently either `scikit` for Scientific Learn and `xgboost` for XGBoost. The second-level keys specify a particular model from that library, and the third-level keys its particular parameters. Several libraries and models can be placed within a single file, while the system always picks only the relevant configuration according to the specified model according to `config.yml`.

To facilitate the creation of such hyperparameters configuration files, the system script `model_manager.py` allows exporting such configuration using the `--params-save` option after performing grid search, as further elaborated on in Subsection 5.2.4.

5.2 Pipeline Scripts

The machine learning pipeline is not implemented as a simple monolithic program but rather split into 5 independent scripts: `dataset_creator`, `dataset_editor`, `dataset_explorer`, `model_manager`, and `mitigator`. Each script implements a part of the pipeline, so the output of one is typically used as an input for another. Their typical pipelined usage is depicted in Figure 5.2. The section will briefly describe each of these scripts' features, functionality, and submodules.

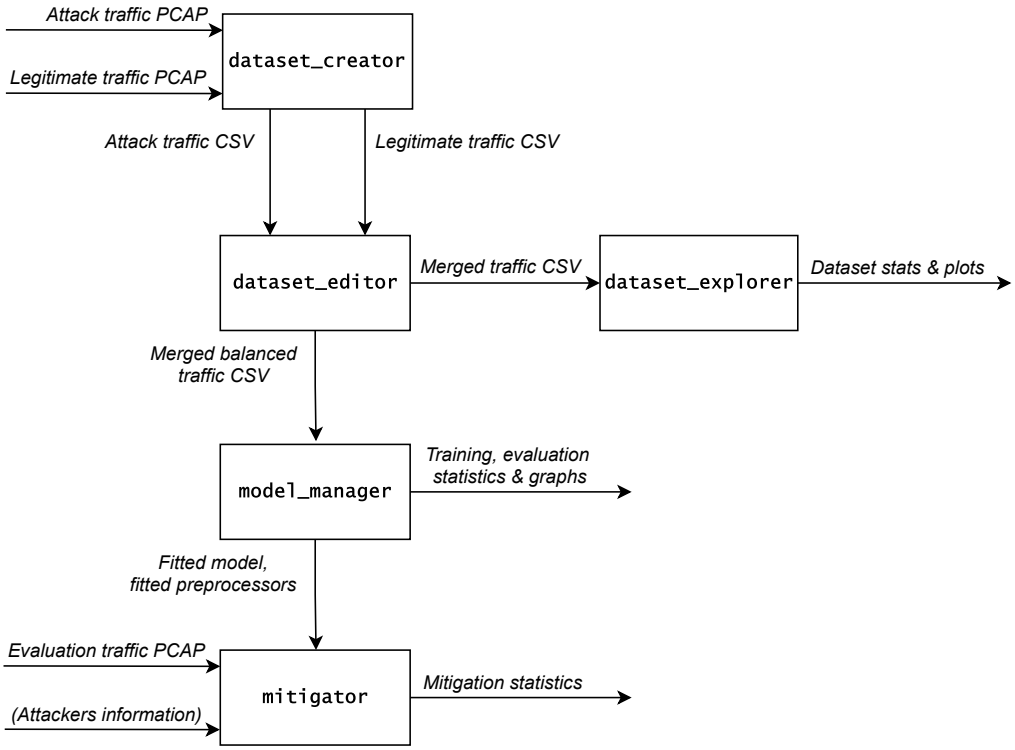


Figure 5.2: Implementation of the system pipeline through scripts.

5.2.1 Dataset Creator

The *Dataset Creator* stands at the start of the pipeline. It is used to transform an input PCAP file into CSV samples according to its configuration and supplied command-line arguments. It thus incorporates the Extractor and Statistical logger modules to retrieve relevant features from PCAPs and compute their statistics using the windowing mechanism, as explained in Section 4.3. The script can accept a single PCAP file, which has to be marked with command-line argument "`-p`" for positive (attack) traffic or "`-n`" for negative (benign) traffic. Both options can also be provided at once. In this case, the script merges positive and negative traffic into a single output file.

Packet Processing

Packet processing includes reading each packet from the input, extracting its features, and logging them to the Statistical logger. This functionality is provided by the `packet_handler` module inside the `dataset_creator` package. Packet reading and feature extraction are achieved with `Scapy`, a packet manipulation library for Python.

Extracted features are processed within the Statistical logger's `log` method. In addition to managing feature extraction and logging, the Packet handler module also handles the logger's windowing. As briefly depicted in Subsection 4.4.1. The windowing is performed by checking the time of each processed packet and keeping the value of the last started window. If the difference between the last started window and the currently processed packet is greater than the specified window's length, a window length value is added to the last started window variable, and the `end_window()` logger's method is called.

Statistical Logger

According to design matters considered in Section 4.3 and Subsection 4.4.1, the primary purpose of the Statistical logger is to log processed packets in the windowing structures and compute their relevant statistical features. For this purpose, a data structure containing window statistics (Table 4.2), a list of source port samples, and two HyperLogLog structures are stored for each IP address. HyperLogLogs are used for logging the number of unique source ports and connections. In addition, another data structure containing the last packet arrival timestamp and two auxiliary values for running variance computation, as defined by Subsection 4.3.3, is also needed.

Based on the current design, a statistical entry is created for every communicating IP with enough data sent in each window. When enough of these entries are collected, the particular IP's data can be pulled by an external call. During the pull, summary statistics are computed and returned, whereas processed window statistics for the given IP are removed. However, if an IP host does not communicate for extended time periods, its statistics may get stuck in the dictionary structure, never be retrieved, and thus never be deleted. This phenomenon may cause undesired memory demands. This issue can be addressed by data structures with a limited number of entries. Therefore, window statistics storage is implemented as `TTLCache` from the `cachetools` library. Time-to-live (TTL) cache is a regular dictionary structure with a limited number of entries expiring after a specific time has elapsed. Using this structure thus guarantees that old window statistics entries will be removed if they are not pulled within the TTL timeout.

To signalize which IPs are ready to be pulled, the Logger maintains a list of "ready" IP addresses, which have communicated with enough traffic in at least N windows, so a sufficient amount of statistics was collected. Logger's API can then be used to retrieve this list. When a pull for a particular IP is made, its entry from the list is removed. However, the code using Logger may not wish to retrieve statistics for some particular IP, and a similar situation as for window statistics – memory overflow, may occur. Therefore, this ready list is implemented via `LRUCache` from the `cachetools` library. This structure is similar to `TTLCache` with a limited number of entries. However, they do not automatically timeout, but the least recently used entry is removed when the structure is full and a new entry is being added. This process also helps to control the memory consumption of the module.

Although the module is not implemented with respect to performance, another huge memory optimization is achieved by logging the statistics in the form of `numpy` arrays. These structures provide very efficient storage with C-like datatypes and padding. With all these considerations in place, the module's memory usage can be efficiently limited, thus not exceeding the memory on regular systems or various system limits. However, packet extraction, logging, and statistics computation in Python are very inefficient and thus not suitable for real-time traffic processing at all.

5.2.2 Dataset Editor

The purpose of the *Dataset Editor* script is to modify the dataset in order to prepare it for further processing. Its primary goals are to remove redundant fields obtained from PCAP processing and merge two independent datasets obtained from the Dataset Creator. As already mentioned during the design discussion, features `src_ip`, `window_count`, `window_span` are not used for classification purposes but are removed during the preprocessing phase. Since `src_ip` is used for the feature grouping, it cannot be removed before

the dataset is split to test and train parts, so only `window_count` and `window_span` features are dropped in this state by default. Additional features may be dropped by changing the `cleaning` configuration key.

Datasets can be merged using the `-m` option. In this situation, one has to consider a scenario when IP addresses from both datasets are the same. In this case, merging the datasets without changing these IPs could create somewhat undesirable behavior since they would be considered as coming from the same source and thus never be split between train and test datasets. This would not be a problem if only a negligible percentage of IPs were affected. However, if a significant portion of IPs would be the same in both datasets, the resulting dataset splits could miss some important data features, possibly affecting overall performance. To mitigate this issue, option `-u` (`--unique-ips`) is provided. If the option is present, conflicting IPs are not merged as they are, but the IP with fewer samples is changed to a random one, thus preventing the above-mentioned phenomenon from occurring.

An additional important feature of the Dataset Editor is the ability to balance datasets. A balanced dataset is a dataset with the same number of elements in all classes. Keeping the balance is vital to prevent the model from being biased towards one of the classes during the training. In practice, datasets created by extracting PCAP files are almost always unbalanced since input PCAPs have a different amount of source IP addresses, number of packets, and timing properties. Therefore, the option `-b` (`--balance`) is provided, which ensures that the number of elements corresponding to attack and benign traffic is exactly the same. By default, this is achieved by undersampling – dropping data from the class with more elements. However, oversampling methods like duplicating the rows multiple times or creating synthetic data are also supported. The balancing behavior can be changed by modifying the `resampling` configuration top-level key.

Additional features provided by the script are dataset trimming (`-t`) and shuffling (`-s`). Trimming functionality trims the dataset (after merging and balancing) to the specified number of elements. This is achieved by undersampling. Shuffling simply randomly shuffles the dataset entries.

5.2.3 Dataset Explorer

`Dataset Explorer` script's functionality corresponds to design concerns mentioned in Subsection 4.4.2. The module provides an easy interface to perform exploratory data analysis upon the dataset retrieved from the Dataset Editor. The script may be used to print dataset information (`-i`), print or save feature statistics to file `-s`, plot various graphs according to design (`-p`), and determine the feature importance (`-f`).

Graph plotting may be controlled by the configuration's file key `feature_plotter`. There, a user may specify the plots' target directory, file format, types of graphs to be plotted, and even specify feature pairs to be plotted as multivariate scatter plots. Since plotting all graphs may take significant time, types of created graphs can be controlled by toggling `True/False` values in various second-level keys.

As it is often desired to know how to interpret the model, determining feature importance is also provided as a part of the script. By default, a "direct" method using the Random Forest algorithm is applied. This method uses tree-based models to determine feature importance directly from the model according to information gain or Gini impurity reduction, similarly to feature selection principles.

Another commonly used technique for this purpose is the method of Permutation importance. This algorithm utilizes an already-fitted classifier and a test dataset. Firstly, a

classification's performance baseline upon the dataset with the given classifier is established. Then, iteratively for each feature, one at a time, its state space is randomly permuted. Classifications are then performed on a dataset with such permuted features. The permuted feature for which the classifier achieved the biggest deviation from the baseline is considered important because its modification influenced the model the most.

5.2.4 Model Manager

The *Model Manager* script incorporates element grouping (explained in Subsection 4.4.4), data preprocessing like standardization and dimensionality reduction, and various actions with the machine learning model itself. These actions include training of the concrete model (`-t`), comparison of models' performances (`-C`), estimation of model parameters using grid search (`-e`), and evaluation of the already fitted model (`-l`).

All of the mentioned actions are highly configurable using the configuration file. The trained model can be selected using the `model_source` and `model_type` subkeys of the `model` configuration top-level key. Model source specifies the model's source library – `scikit` or `xgboost`, whereas model type specifies the concrete model like "bayes". Path to model hyperparameters can be specified by the `models_cfg_file` subkey. In case the file does not exist or the desired model is not included within it, default model parameters from the underlying library are used. The training and evaluation process of the model can further be plotted with the `-P` option, configured via the `model_plotter` top-level key.

A trained model instance can also be dumped to a binary file (`-d`) and later loaded with the `-l` option or within the Mitigator script. The ability to perform feature selection or projection and dump the fitted transformation objects to the file is also provided. Another important thing that needs to be saved is data statistics for feature standardization. By default, the statistics are computed within each run upon the training set and applied to the whole dataset afterward. However, one may want to use statistics that have been already precomputed or simply save the statistics for the model deployment. Saving of the computed statistics can be achieved with the `-g` option, which creates a YAML with average, max, mean, and standard deviation computed for all dataset features.

Hyperparameter estimation is achieved by a grid search technique. Grid search simply takes all hyperparameter combinations from the specified hyperparameter space, trains the model with them, and performs evaluation using cross-validation. In the end, the best parameters are printed to standard output or saved to the hyperparameter configuration file with the `-p` (`--params-save`) option.

Model comparison is performed via 5-fold cross-validation. Models to compare can be specified in the `comparison_models` subkey within the `model` top-level key. Each compared model then provides statistics about its training (fitting) time, scoring time, accuracy, F1-score, precision, and recall by default. An example of such output is shown in Figure 5.3. Displayed statistical values can further be customized with the `score_metrics` subkey.

5.2.5 Mitigator

The *Mitigator* script simulates the deployment of the model in the real world. Its inputs are a dumped model and saved feature statistics YAML file generated by the Model Manager. Alternatively, the script can also accept dumped feature selector and projector files from Model Manager if feature dimensionality functionality is required.

The primary purpose of the Mitigator is to check how the model would perform in practice. During the evaluation in Model Manager, the model performance statistics are

```
----------------------------   CROSS VALIDATION RESULTS   ----------------------------
Model                 fit_time score_time accuracy accur_std f1      precision recall
scikit.bayes            0.0163   0.0089     0.9947   0.0014    0.9947  0.9932    0.9962
scikit.mlp              1.5644   0.0145     0.9991   0.0006    0.9991  0.9986    0.9996
scikit.svm              0.0868   0.0269     0.9987   0.0011    0.9987  0.9978    0.9996
scikit.random_forest    0.1405   0.0152     0.9991   0.0011    0.9991  0.9996    0.9986
xgboost.xgboost        73.2325   0.1826     0.9994   0.0004    0.9994  0.9994    0.9994
```

Figure 5.3: Output of model comparison using cross-validation in the Model manager script.

computed upon already computed statistical features. In this case, one such entry can consist of hundreds of packets scattered across multiple seconds. Labeling such entries as attacking or legitimate provides information about the classifier's performance, but not about how many packets were actually dropped and how many of them were forwarded. For this reason, the Mitigator script works with packets and logs statistics related to them instead. This approach provides a more accurate view of the mitigation progress itself.

The Mitigator uses a Statistics Logger instance in the same way as during a regular dataset creation. However, instead of creating new dataset entries, the statistical features are sent for preprocessing and to the classifier to make a prediction. If this prediction signifies that an attacking IP is present, the IP is added to a denylist (implemented as `LRUCache`), which simulates that traffic from a particular IP is being denied. Traffic from both legitimate and attack sources is logged. In the end, statistics of how many packets were allowed and denied, alongside the success rate and others, are printed.

The script is primarily supposed to read the data from PCAP files (offline) with `-r`. In addition, the ability to read packets and perform classifications from a specified interface (online mode) with `-i` is also included. This functionality is provided by a `Scapy` sniffer, which has been implemented to run in a separate thread. In this case, the program enters an infinite loop and is stopped by entering a `Ctrl+C` into the command line. The emitted Interrupt signal is caught, sniffer stopped, and statistics printed to the standard output. However, keep in mind that despite the included online functionality, the program is not designed to work in online mode due to high processing inefficiencies, leading to significant decreases in network throughput. Instead, it is provided for demonstration purposes only.

In order to print mitigation statistics in the end, the script needs to know the classification ground truth. Such information can be supplied as a list of attackers with the `-e` (`--evaluate`) option. The list of attackers is supposed to be in the form of `IP1\n IP2\n`, etc.

5.3 Running the Pipeline

So far, this chapter has briefly presented how the pipeline is implemented and outlined the most important command-line options and configuration parameters of each script. Let us now apply this knowledge and take a look at an example of how such a ML pipeline could be run. Firstly, a step-by-step example of its usage will be introduced (Subsection 5.3.1). However, this process may be too complex for some users at first, so a single-command demonstration solution is provided in Subsection 5.3.2.

```
1  dataset_creator.py -c config.yml -p traffic_attack.pcap attack.csv
2  dataset_creator.py -c config.yml -n traffic_legit.pcap legit.csv
3  dataset_editor.py -c config.yml -m attack.csv legit.csv -u -b -s dataset.csv
4  dataset_explorer.py -c config.yml -i -s stats.txt -p dataset.csv
5  model_manager.py -c config.yml -C dataset.csv
6  model_manager.py -c config.yml -v -e -p estimation_params.yml dataset.csv
7  model_manager.py -c config.yml -t -s -P -g std_params.yml -d model.bin
     dataset.csv
8  mitigator.py -c config.yml -e attackers.txt -E export_stats.txt
     -s std_params.yml -r traffic_verify.pcap model.bin
```

Figure 5.4: Sequence of commands for manual pipeline run.

5.3.1 Manual Example

Suppose we have filtered our legitimate and attack traffic into files `traffic_legit.pcap` and `traffic_attack.pcap` placed in the same path as the scripts. Already prepared configuration file is located in relative path `config.yml`. We have a Python ≥ 3.9 with all the required modules and libraries installed. The process of executing the pipeline manually then is depicted in Figure 5.4.

As the first step, PCAP files are converted into CSVs by extracting relevant features, performing windowing, and computing relevant statistics (lines 1-2). Next, the created CSVs are merged into a single dataset file while keeping the IPs unique, balancing the dataset, and shuffling it at the end. The resulting dataset is saved as `dataset.csv`. With the data prepared, the exploratory analysis is performed by querying the basic information about the dataset, calculating features statistics to `stats.txt`, and plotting various graphs according to the supplied configuration (line 4).

As we gathered enough information about the dataset, several machine learning models are chosen and written to the `comparison_models` subkey of the `model` top-level key in the system configuration file. At this point, cross-validation is run to determine the model we will use (line 5). According to the output analysis, we determine that a simple decision tree model has the best decision-making times with acceptable accuracy. Therefore, we decide to use this model for future classification. We change the `model`'s `model_source` subkey to `scikit` and `model_type` to `tree`. The next step is to determine the best hyperparameter values for *min_samples_leaf* and *max_depth* parameters. We thus update the `model`'s `estimation_params.yml` subkey with the hyperparameters' names and the values to be tried. After this, a grid search to estimate the best parameters in a verbose mode is run, and its results are saved to the `models_params.yml` file (line 6).

After the best parameters are estimated, we can finally train the model with them. Since we will also be interested in using the model later, the trained model is saved to the file `model.bin` and features statistics for standardization to `std_params.yml`. Advanced statistics about model evaluation alongside various plots are also requested (line 7).

In the end, we want to simulate a deployment of the trained model against a validation PCAP dataset `traffic_verify.pcap`. Suppose we have specified attacking IP addresses from the given dataset in the `attackers.txt` file. Mitigator script can thus be run to perform such evaluation upon the saved model and standardization data while exporting the mitigation information into the `export_stats.txt` file (line 8).

By executing the commands according to Figure 5.4, we have managed to successfully extract statistics from training PCAP files, preprocessed them, chose the model and determined its best hyperparameters, fit such model, and verified its functionality in a simulated model deployment scenario.

5.3.2 Automated Demonstration

In order to facilitate the pipeline execution for demonstration purposes, the `run.py` script is provided. The script utilizes files in its default locations to perform a quick overview of the main system parts' functionality described in Section 5.2. When run without arguments, the pipeline execution starts from line 4 in Figure 5.4. This process prints statistics and plots feature graphs, compares multiple models, estimates parameters, trains and saves the model into the file, and finally performs an evaluation with the Mitigator.

Lines 1-3 are not executed by default since dataset creation may take a significant amount of time, so its inclusion within the demonstration may be inappropriate. If the user wishes to execute this part of the pipeline as well, the option `-f` (`--full`) can be used. In this case, own `traffic_attack.pcap` and `traffic_legit.pcap` files used for the extraction and `traffic_verify.pcap` along with `attackets.txt` for validation have to be supplied. These are not included in the final submission due to their immense size and limited space conditions.

Note that some filenames from Figure 5.4 were shortened in order to fit the figure's width and thus may not need to correspond to the filenames used within the system's code. Please consult `run.py`'s lines 26-54 for the naming of demonstration files within the project.

Chapter 6

Evaluation

Ability to evaluate the performance of the machine learning model and alternatively interpret its decisions is crucial for the practical usability of the system. Depending on the usage, different criteria may be put on the model. Therefore, interpretation of the "best" model may also differ across various scenarios. When choosing the final model or the final ML pipeline as a whole, numerically expressible metrics may not be the only relevant factor, but other requirements such as fitting time, estimation time, interpretability, scalability, and others may also need to be taken into account.

When evaluating the system as a whole, considering only the model is not sufficient. The process of how the data are retrieved and preprocessed must be incorporated into the evaluation process as well. This is especially important in our case since feature extraction and statistics computation is very flexible due to all possible configuration options.

In this chapter, a discussion on all of the above-mentioned issues will be conducted. Firstly, relevant criteria for the system evaluation will be specified (Section 6.2). This will be followed by the introduction of standard numerical evaluation metrics for ML models (Section 6.1), discussion about hyperparameters during dataset creation (Section **??**), and finally, the evaluation of the model based on various experiments with available datasets (Section 6.4). Section 6.5 will then summarize these results and discuss the suggested form of the ML pipeline concerning performance and other relevant factors.

6.1 Model Evaluation Metrics

As briefly outlined at the start of this chapter, there are various evaluation metrics and criteria to determine the performance of a ML system. This section will focus on the most commonly used numerical and visual evaluation metrics for binary classification problems. For these purposes, terms True positives, True negatives, False positives, and False negatives need to be firstly defined:

- *True positives* (tp) – outcomes where the model correctly predicts the positive class.
- *True negatives* (tn) – outcomes where the model correctly predicts the negative class.
- *False positives* (fp) – outcomes where the model incorrectly predicts the positive class.
- *False negatives* (fn) – outcomes where the model incorrectly predicts the negative class.

With respect to these definitions, other model classification metrics can now be defined.

6.1.1 Numerical Metrics

Numerical metrics for model evaluation aim to describe the model's performance by a single numerical value. These are typically in the range $[0, 1]$, with 0 as the worst score, whereas 1 represents the perfect score. Metrics presented in this section include accuracy, precision, recall, f-score, and Matthews correlation coefficient.

Accuracy (Eq. 6.1) represents the portion of successful classifications out of all that were made. In order words, it gives a probability estimate of a correct prediction. This metrics works only with balanced datasets since high class imbalance may provide a false sense of high accuracy. This happens if the model classifies most elements of significantly smaller classes wrongly. However, high accuracy is achieved regardless due to the influence of other populous classes. This issue can be tackled by the *Balanced accuracy* metric.

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn} \tag{6.1}$$

Precision (Eq. 6.2) is defined as a ratio of correctly classified elements in the positive class to all positively classified elements. Therefore, the metric describes the ability of the estimator not to make mistakes when classifying an object as positive. For this reason, it is a helpful indicator in scenarios with false positives being one of the main points of interest.

$$Precision = \frac{tp}{tp + fp} \tag{6.2}$$

Recall (Eq. 6.3) describes a relationship between correctly classified elements in the positive class and all positive class elements. Intuitively, it is an ability of the estimator to recognize objects that should be classified positively.

$$Recall = \frac{tp}{tp + fn} \tag{6.3}$$

Precision and recall metrics are rarely used separately because they have little informative value on their own. In fact, precision and recall are often in an inverse relationship, where it is possible to increase one at the cost of reducing the other [13]. For this reason, they are typically compared for a fixed level at the other measure (e.g., recall at a precision level of 0.9) or both combined into a single measure such as the F-score.

F-score (Eq. 6.4) in the ML context typically represents *F1-score*, which the harmonic mean[1] of the precision and recall. By combining these metrics, F-score provides a comprehensive evaluation measure for ML systems, for which simple accuracy is not descriptive enough. These include systems with imbalanced datasets or systems with differing costs of false positives and false nagatives. A more generic F-score variant – F_β also exists, which applies additional weights, valuing precision or recall more than the other.

$$F_1 = \frac{2}{precision^{-1} + recall^{-1}} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{6.4}$$

Application of recall, precision, and f-score is sometimes argued to provide a biased estimate, and should not be used without understanding these biases [83]. Instead, [25] suggests to use *Matthews correlation coefficient* (MCC) (Eq. 6.5), as it is more reliable

[1] $H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} = \left(\frac{\sum_{i=1}^{n} x_i^{-1}}{n} \right)^{-1}$

$$\begin{matrix} tp & fn \\ fp & tn \end{matrix}$$

Figure 6.1: Confusion matrix layout for binary classification problem.

statistical rate which produces a high score only if the prediction obtained good results in all of the four possible classification results: tp, tn, fp, and fn.

$$MCC = \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}} \tag{6.5}$$

6.1.2 Performance Visualization

In addition to numerical metrics, the model's performance can also be visualized. Performance or behavior visualization allows presenting numerous important quantities in a single, easily interpretable plot. Techniques presented in this subsection include confusion matrix, ROC, AUC, validation, and learning curves.

The *confusion matrix* (Figure 6.1) is used to visualize true positives, true negatives, false positives, and false negatives values. It is a matrix of NxN elements (2x2 for binary classification), in which rows represent actual class instances, whereas columns represent predicted class instances.

The *receiver operating characteristics* (ROC) curve depicts a relationship between the true positive rate (TPR) and the false positive rate (FPR). TPR is a rate of correct positive results among all positive samples, computed the same way as precision in Eq 6.2. Therefore, $tpr = precision$. On the other hand, FPR defines a rate of incorrect positive results among all negative samples (Eq. 6.6). Plotting these values against each other (FPR $= x$, TPR $= y$) shows relative trade-offs between true positives (benefits) and false positives (costs). An ideal classifier would yield a point in the coordinate $(0, 1)$, representing no false negatives nor false positives. This scenario is rather unlikely in practice, so the model hyperparameters are tweaked to reach the desired performance instead.

$$fpr = \frac{fp}{fp + tn} \tag{6.6}$$

The *area under (the) curve* (AUC) is a value representing the area under ROC curve ($\int_0^1 ROC(x)\, dx$). AUC can be interpreted as the probability that the model ranks a random positive example higher than a random negative one [36]. AUC value of 1.0 would then represent a model with 100% correct classifications.

Although estimation of the model's parameters should be achieved by techniques like Grid search (Subsection 4.4.6), it may be sometimes helpful to measure the influence of a single hyperparameter on both training and test scores. Plotting such a relationship can be done with the *validation curve*, typically used to determine whether the model is overfitting or underfitting at specific parameter values. Axis x on the plot is thus a hyperparameter space, whereas y represents a specified evaluation metric, typically accuracy for classifiers.

Lastly, the *learning curve* can be used to show the validation and training scores of an estimator for varying numbers of training samples. This tool can determine the benefit of adding more training data or whether the model suffers from a variance or a bias error.

6.2 Evaluation Criteria

In the case of an online ML DDoS mitigation system, the most crucial feature is the ability to determine and mitigate the attack traffic. However, this process needs to be fast, so network throughput will not be significantly impacted, and so the model's decisions could affect the mitigation process as soon as possible. For this reason, we aim to pick only high-performance models with as low classification time as possible. Since the model can be trained offline and deployed later, the estimator's fitting time is of no importance.

With timing requirements clear, let us have a look at what a high-performance model represents. Typically, one of the key determining factors of the model performance is its accuracy. Since the proposed system generates a dataset by itself, inbalance will not be a concern. Therefore, the accuracy will not be biased. However, this metric only describes an ability to estimate the correct class in general. In the context of DDoS mitigation and generally in cybersecurity, more strict standards are typically in place.

The purpose of attack mitigation is to block the attacker's messages so the victim's infrastructure and end-users will not be affected. Therefore, ongoing mitigation must affect an end-user as little as possible. Ideally, it would be completely transparent. With this in mind, we require the model to work with a small ratio of false positives (higher precision), even at the cost of slightly more false negatives (lower recall). In other words, we do not mind if the model misses a few attacking IP addresses that much, but misclassifying legitimate users and denying their traffic is highly undesired.

Concrete values of precision and recall should be specified according to the network environment and additional requirements on the system. However, it is reasonable to demand the precision of at least 0.95 with recall above 0.8. One way to tweak these values is to modify the model's hyperparameters. Some models may also have the ability to return classes' probabilities instead of directly estimated class labels. In these scenarios, the threshold can be manually adjusted to limit the number of false positives. Visualization tools such as a ROC curve may be helpful for this purpose.

6.3 Statistics Computation Parameters

Before quantified the system's performance, the parameters of the used data should be specified first. Recall that the Statistical logger module uses a concept of time windows to group packets' features and compute statistics upon them independently. It is thus crucial to set these parameters carefully so that the attack detection can be achieved with high precision in a reasonable amount of time. Referring to Appendix B, top-level key `logger`, the following statistics logging parameters can be configured:

- `window_length` – length of the window in seconds
- `history_min` – minimum number of collected windows for a given IP
- `history_size` – maximum number of history elements in memory
- `history_timeout` – validity of history entries duration in seconds
- `packets_min` – minimum number of packets in the window to log it
- `samples_size` – number of samples to collect for entropy estimation

Parameter history_size

Although `history_size` is mostly a memory optimization parameter, it may affect the computation of some history entries upon massive loads of traffic. In such a case, so many history entries are created within each window so that the memory will not be sufficient, causing some valid entries (least recently used due to implementation) to be dropped.

During the experiments, the default parameter of 0 was used. This setting allowed around 5 GB of memory to be available, so almost 48 M history entries would be required in order for valid entries removal due to full memory to happen. This corresponds to almost 8 M IP addresses with 6 history entries communicating simultaneously in the worst case. Although this scenario is possible in backbone network deployments, we can ignore this parameter as it cannot influence the produced statistics.

Parameter history_timeout

Parameter `history_timeout` defines for how long are the collected statistics valid in history. Invalid history entries are not used for summary statistics computation and are deleted instead. This parameter thus specifies how old statistics for a communicating host are we willing to accept in order to determine whether it generates malicious traffic or not.

DDoS attacks typically produce a continuous stream of data, so in their case, the timeout could be equal to `window_length` × `history_min`. However, regular clients typically communicate in bursts, so some reserve needs to be held if a client does not manage to communicate with at least `packets_min` packets within a particular window, so its traffic will not be logged at all.

A reasonable parameter value could be 10-20 times of `window_length` × `history_min` to not keep the old statistics in the memory for too long and not miss any possible burst-based attacks. The experiments were performed with this value set to 240 seconds, as we are not generally limited by the memory in offline mode (disk swapping is not such a performance issue) but want to keep the experiments realistic.

Parameter samples_size

Modifying the `samples_size` parameter only influences the precision of source port entropy estimation. Smaller value provides lesser memory consumption, whereas bigger value a better estimate. In the experiments in this chapter, the value of 40 was predominantly used. This was chosen based on tests with various parameter values and standard deviation computation across 5 such runs upon both legitimate and attack dataset subsets. Legitimate dataset normalized entropies reached a standard deviation of 0.05 at the threshold of 25 samples, whereas 0.1 std was achieved at 50 samples for attack traffic. Value 40 was thus chosen as a compromise, providing a reasonable entropy estimate and relatively low memory consumption.

Parameters window_length and history_min

Parameters `window_length`, `history_min`, and `packets_min` are closely related, as they define the minimum length of the host's communication and the number of packets it is supposed to send. They also influence the generated statistics the most. Generally, we want to detect an attack as soon as possible but also achieve high detection precision and recall. As one may notice, there is a trade-off between these two – the sooner we want to detect an attack, the fewer data we can collect, and thus the detection performance is worse.

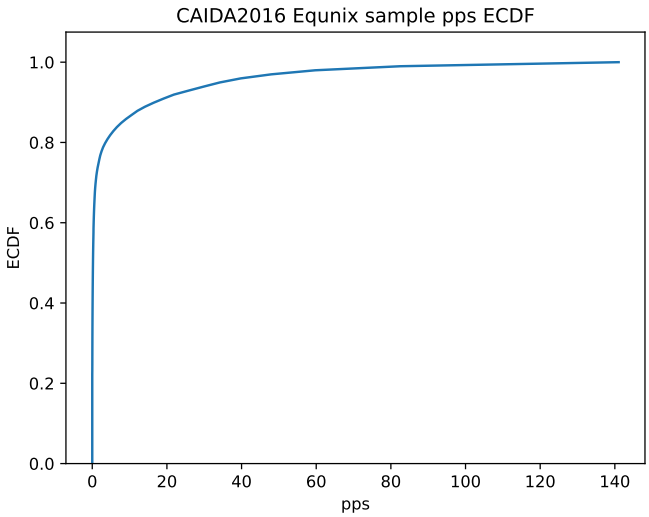


Figure 6.2: Packets-per-second empirical cumulative distribution function for CAIDA 2016 Internet Traces sample.

According to the project requirements, the reaction to the attack needs to be done within 10 seconds of its start. This includes attack detection, perpetrator's IP addresses determination, mitigation rule creation, and its application in the database. Since rule creation requires to sample some of the attacker's data and rule inferring and database update taking some time as well, this leaves us with a space of 6-8 seconds to detect attacking IP addresses. For this reason, the experiments use `window_length` of 1 or 2 seconds and `history_min` of 3 to 6 depending on the previous parameter.

Parameter packets_min

The last not-yet discussed parameter is `packets_min`. It defines the minimum number of packets that needs to be sent by the client in order to log its data. By tweaking it, we aim to limit the number of classified IP addresses, so entries from clients sending low amounts of traffic are not considered. Statistical data such as mean or standard deviation may be significantly skewed if computed on a small set of data. Therefore, by setting this number sufficiently high, the system makes sure that only clients with relevant statistical data will be classified. This ensures statistical features of finer quality, leading to better classification results. As a positive side-effect, the state space of analyzed entries is significantly limited, the model needs to perform a lesser number of classifications, leading to improved packet throughput and better system reaction times.

The value of this parameter was chosen according to packets-per-seconds (PPS) analysis of the several legitimate captures with respect to the `window_length` parameter (Figure 6.2). Based on these results, a PPS value of 10 corresponds to roughly 0.89-percentile (CAIDA Traces) to 0.93-percentile (CESNET $\longleftrightarrow$ ACONET capture) of all the legitimate clients over the length of their active communication. Therefore, by specifying a minimum of 10 packets per a 1-second time window, we filter out approximately 90% of all legitimate clients' traffic uninteresting for our purposes. Since (D)DoS attacks typically produce much larger traffic volumes, the ability to detect them will not be negatively affected.

Parameter name	Values
`window_length`	1, 2
`history_min`	4, 6, 8
`history_size`	0 (unlimited)
`history_timeout`	240
`packets_min`	10, 15, 20, 30, 40
`samples_size`	40, 50

Table 6.1: Parameter values tried throughout experiments.

Parameters Configuration Summary

In order to bring a little diversity and perform various experiments not only for the machine learning model itself but for statistics computation as well, the data were extracted by various parameter settings, as suggested by Table 6.1. However, not all combinations from the given table were tried, but the state space was rather reduced only to a subset of the most relevant combinations.

The limited scope of experiments with various dataset creation settings is caused by the complexity of computation. Processing 1 M of packets standardly takes between 7 (`cider.liberouter.org`) to 25 minutes (`pinot.liberouter.org`), so processing files with tens to hundreds of millions of packets gets computationally very expensive. Achieved classification results with various parameter combinations are discussed in Section 6.4.

6.4 Classification Performance

This subsection covers several picked experiments with the system, provides their analysis, results, and a brief commentary. Experiments cover the detection performance of both volumetric and slow (D)DoS attacks, comparison of various machine learning models and data modifications and transformations. As outlined in Table 6.1, various parameters were used during the data extraction process. Statistical logging mechanism was then further applied on datasets from Section 4.5 with various parameters combinations.

All the described datasets were created by `dataset_creator.py` script and merged using `dataset_editor.py`. Further data processing steps are described in their particular subsection. All datasets have 4 different variants according to the Statistical logger's configuration during the data extraction process. These include:

1. *window_length = 1, history_min = 6, packets_min = 10, samples_size = 40*
2. *window_length = 1, history_min = 6, packets_min = 15, samples_size = 40*
3. *window_length = 1, history_min = 4, packets_min = 15, samples_size = 40*
4. *window_length = 1, history_min = 4, packets_min = 20, samples_size = 40*

Most of the described experiments were performed with the first dataset variant containing at least 10 packets per window with minimum 6 windows to generate a single data sample. If not explicitly stated otherwise, this variant is used. CSV datasets created with this configuration are displayed in Table 6.2. Note that PCAP files extracted from CIC datasets use a significantly limited number of IP addresses, and thus the windowing system in Statistics logger produced a relatively small amount of CSV entries.

Dataset name	Samples
CAIDA Internet Traces	338310
CESNET ⟷ ACONET	361666
CIC-IDS2017 Benign	14649
CIC-ISCXIDS2012	96025

(a) Benign datasets (Subsection 4.5.3).

Dataset name	Samples
CAIDA "DDoS Attack 2007"	394200
CIC-DDoS2019	1877
CSE-CIC-IDS2018	3363
CIC-IDS2017 (all)	586
CIC-IDS217 (Slow DoS)	361

(b) Attack datasets (Subsection 4.5.2).

Table 6.2: Dataset sizes for 1-sec windows, 10-packets per window minimum, 6-windows size and 40-sized samples.

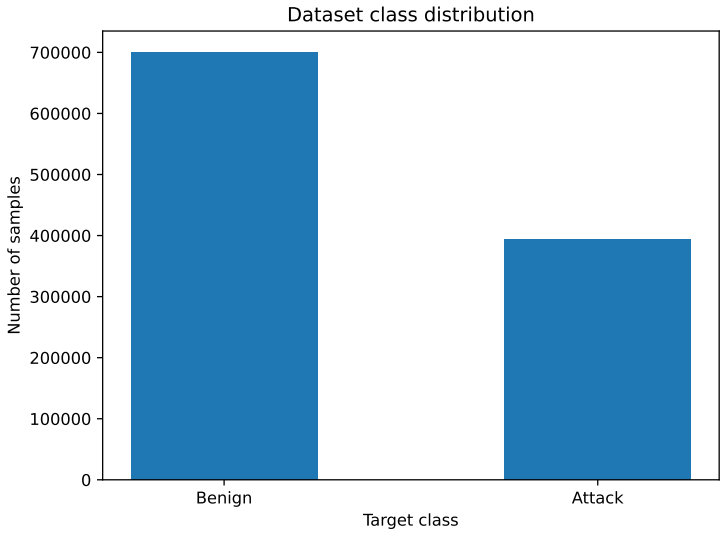


Figure 6.3: Distribution of classes in dataset extracted from real data.

The following section will firstly present experimental results with available real (non-synthetic) datasets (Subsection 6.4.1) and mixed datasets (Subsection 6.4.2).

6.4.1 Real Data

First performed experiments were performed with real (non-synthetic) data. These include merged CESNET ⟷ ACONET capture, CAIDA anonymized traces from Subsection 4.5.3 as legitimate data and CAIDA DDoS Attack 2007 from Subsection 4.5.2 as attack traffic.

Initial analysis of the merged dataset has shown rather significant data imbalance (Figure 6.3). For this reason, the dataset was balanced for processing in ML methods by randomly sampling values from benign class, until the same number of entries in both classes was reached. This process yielded a dataset with over 788 k of elements. In addition, all the feature graphs are plotted based on a balanced sample of 50 000 elements to keep the file sizes low.

Feature Correlation

As the first analysis step, the feature correlation heatmap was plotted (Figure 6.4). According to this analysis, it may be seen that `pkts_total`, `bytes_total`, `pkt_rate`, and `byte_rate` are almost perfectly positively correlated. This makes sense, as `pkts_total` and `bytes_total` are an average over all summarized windows, whereas `pkt_rate` and `byte_rate` and an average over the whole time the client communicated (even including inter-windows not logged in the system). However, when there are no window gaps between the communication, the values will be approximately the same. Another highly correlated features with them are `pkts_total_std` and `bytes_total_std`. This is most likely caused by the burst communication character, so the number of packets/bytes between different windows is highly unbalanced.

Other understandable correlations like between the maximum size of the seen packet and standard deviation of its sizes are shown. These have a strong positive correlation. Medium positive correlations between the client's inter-window and intra-window activity signify that suppositions about their possible usefulness were fulfilled – if the client communicates continuously throughout a more extended period, it may be an attacker. However, this cannot be considered ground truth because regular clients can achieve similar characteristics using data streaming services.

Nevertheless, the analysis has also shown that the maximum size of the packet and TCP protocol share a relatively strong negative correlation with the target. In contrast, ICMP protocol share has a perfect positive correlation. These findings are rather intriguing since the mentioned features could provide some degree of information in combination with others but should not correlate with the target so strongly. Therefore, the legitimate data either do not contain enough ICMP traffic or ICMP share in attack traffic is so predominant that it may cause a classifier to become biased and generalize poorly upon out-of-dataset data.

Correlation analysis has provided valuable insights, leading to the conclusion that columns with the total number of packets and bytes may be dropped, as they have the same relationship with all other features and the target variable. Protocol shares may also become relatively problematic, as they correlate relatively strongly with the target, and according to expert knowledge, this correlation is rather unfounded. Dropping these features will also be tried in order to increase the generalization capabilities of the final model.

Feature Analysis

Another step to better understand the data is feature analysis. Generally, we aim to know more about features whose values cannot be estimated by expert knowledge or which help us to get to know the unique properties of the processed dataset. This part includes a few picked plots (Figure 6.5) alongside their short descriptions.

Figures 6.5a and 6.5b confirm our thoughts from the previous section. Attack data are primarily composed of ICMP traffic, and real benign data contain almost none of it. For this reason, classifiers will probably have an easy time because the two classes are separable by a simple linear decision boundary with this feature.

Figure 6.5c also aligns with the statement described earlier and confirms our hypothesis that attackers send large amounts of packets continuously. Therefore, there is a minimal inactivity period between the window start and the host's first packet and its last packet and the window end. A very similar distribution was achieved for the intra-window activity ratio as well.

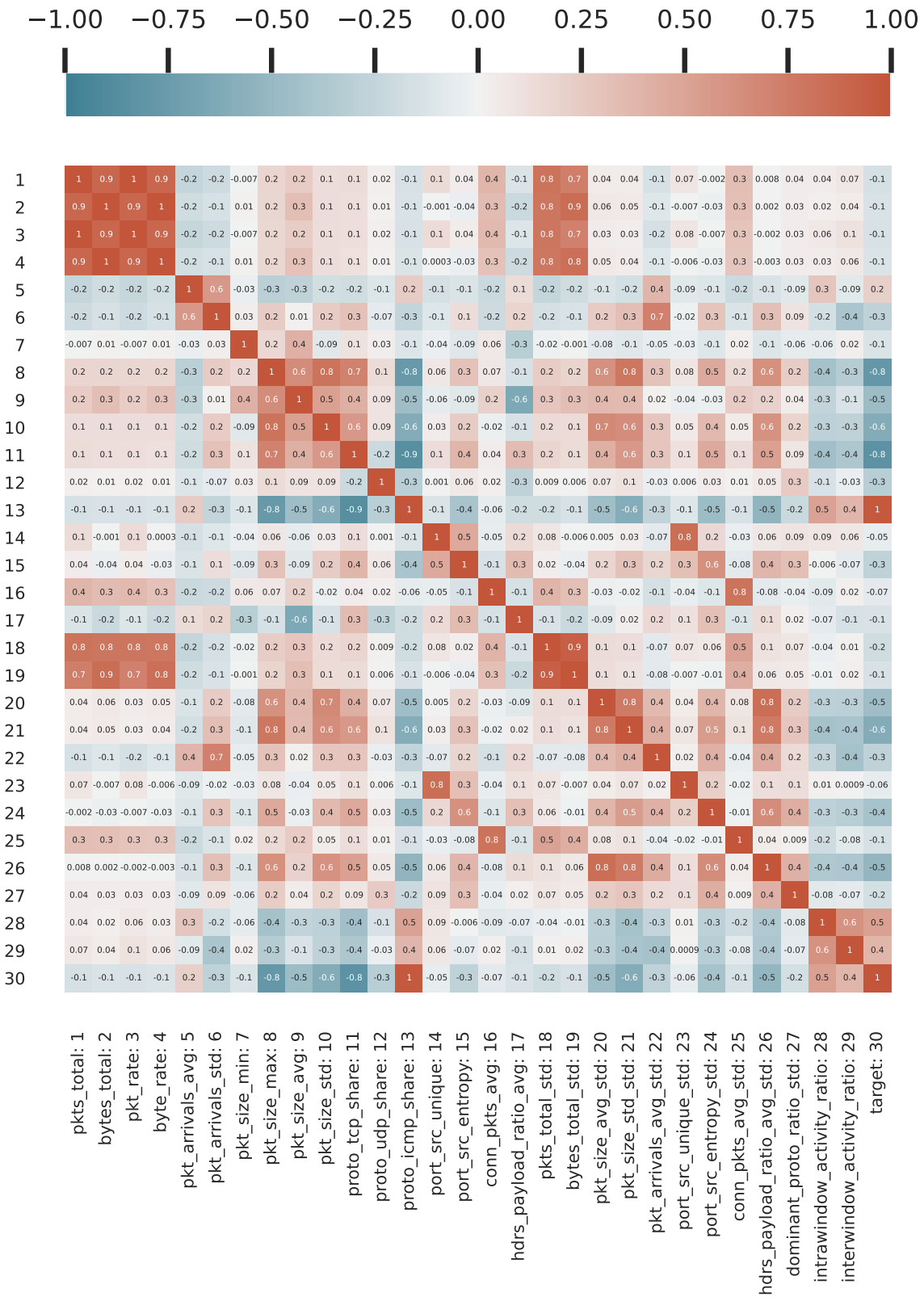
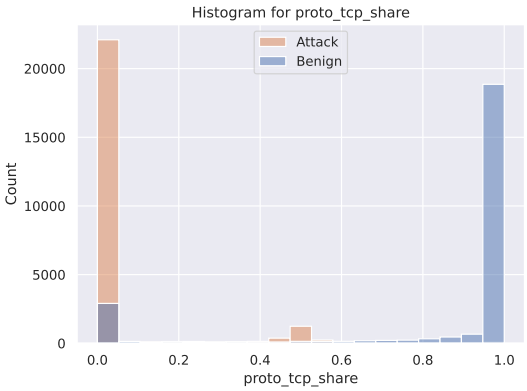
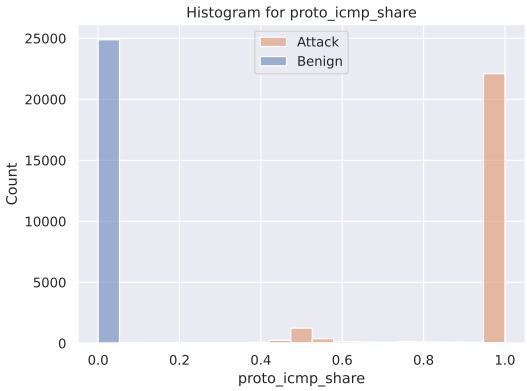


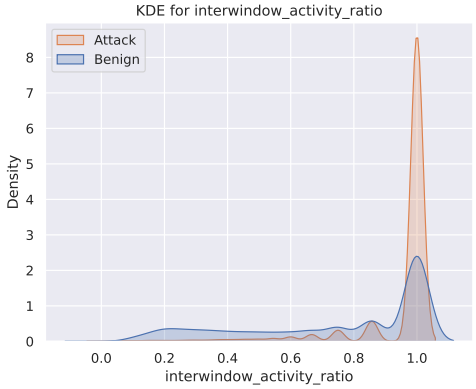
Figure 6.4: Real dataset features correlation.



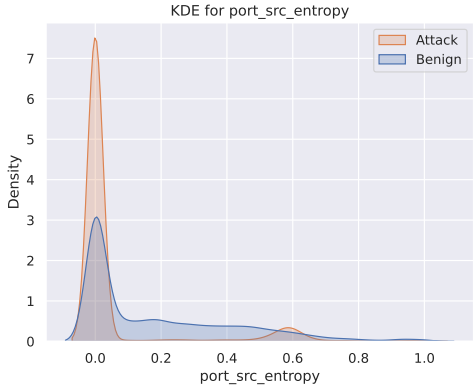
(a) TCP Protocol share.



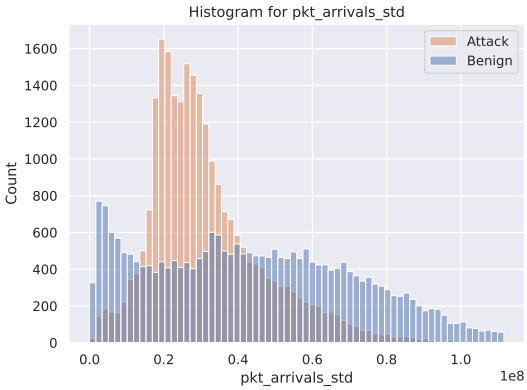
(b) ICMP Protocol share.



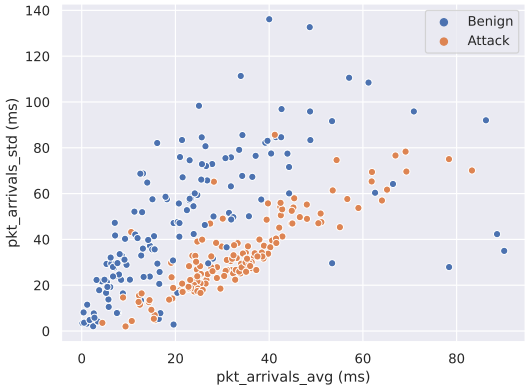
(c) Inter-window activity ratio.



(d) Source port entropy density.



(e) Packet arrivals standard deviation density.



(f) Relationship between packet arrivals average and standard deviation.

Figure 6.5: Real dataset feature analysis.

Figure 6.5d is a direct consequence of ICMP traffic share predominance. Since ICMP does not use port numbers, the resulting port is 0, and thus its entropy is also 0.

An interesting insight is displayed in Figure 6.5e. As it may be seen, the attack traffic does not actually have its packet arrivals standard deviations close to 0 but is somewhat

Model	fit_time	score_time	accuracy	accur_std	f1	precision	recall
scikit.adaboost	184.4754	1.7544	0.9996	0.0	0.9996	0.9996	0.9995
scikit.bayes	1.2891	0.2852	0.9640	0.001	0.9652	0.9358	0.9965
scikit.extra_trees	83.2605	2.2573	0.9998	0.0	0.9998	1.0	0.9996
scikit.grad_boosting	680.1072	0.3994	0.9998	0.0	0.9998	0.9999	0.9996
scikit.logreg	18.3962	0.1693	0.9987	0.0	0.9987	0.998	0.9994
scikit.lda	26.5794	0.2798	0.9972	0.0001	0.9972	0.9986	0.9958
scikit.mlp	149.9141	1.4796	0.9991	0.0004	0.9991	0.9993	0.9990
scikit.nearest_centroid	1.1646	0.2108	0.9779	0.0003	0.9774	0.9979	0.9577
scikit.svm	862.7121	29.1655	0.9992	0.0	0.9992	0.9988	0.9995
scikit.tree	27.1975	0.1733	0.9997	0.0	0.9997	0.9997	0.9997
scikit.random_forest	232.4759	1.4291	0.9998	0.0	0.9998	1.0	0.9997
xgboost.xgboost	435.8562	0.9845	0.9999	0.0	0.9999	1.0	0.9997

Figure 6.6: Model comparison with cross-validation upon the real dataset containing 788 k samples. Trained on Fujitsu Esprimo Q920.

Codename	Full name	Codename	Full name
scikit.adaboost	AdaBoost	scikit.mlp	Multilayer Perceptron*
scikit.bayes	Naive Bayes	scikit.nearest_centroid	Nearest Centroid Classifier
scikit.extra_trees	Extra trees	scikit.svm	Support Vector Machines
grad_boosting	Gradient Boosting	scikit.tree	Decision Tree
scikit.logreg	Logistic Regression	scikit.random_forest	Random Forest
scikit.lda	Linear Disc. Analysis	xgboost.xgboost	XGBoost

Table 6.3: Explanation of machine learning model codenames.
*Regular neural network with 1 hidden layer.

centered around 30 ms, which is quite a lot for DDoS traffic. However, the attack capture is from 2007, so networks and packets rates were significantly lower back then.

Figure 6.5f displays a relationship between packet arrivals averages and standard deviations. According to seen data, a relatively accurate quadratic or even linear boundary can be drawn to distinguish both classes. This fact aligns with our previous prediction that IP addresses with bigger arrivals deviation would most likely resemble legitimate clients, as packets are not generated periodically by some malicious software.

Classification Results

After reviewing the dataset and obtaining an idea of its characteristics, we may try to fit a machine learning model onto them and review its performance. As stated in previous subsections, the ICMP share should be the most important factor, as the data are almost perfectly linearly separable according to its value. All available models with their default parameter settings were reviewed with cross-validation (Figure 6.6). Dataset columns with values not within the range $[0, 1]$ were standardized with the $MinMax$ function. Model codenames in Figure 6.6 used throughout this section are explained in Table 6.3.

As it may be seen, all models have achieved outstanding performance. This was probably caused by the task character, as the two classes were so easily separable. Performance evaluation was performed on Fujitsu Esprimo Q920. The "weaker" workstation was chosen instead of the server on purpose, so fitting and scoring times are amplified, and timing differences between them thus becoming more apparent. The k-nearest neighbors model could not be used because the dataset is so extensive that the model consumes a vast amount of memory, causing the operating system to terminate the process.

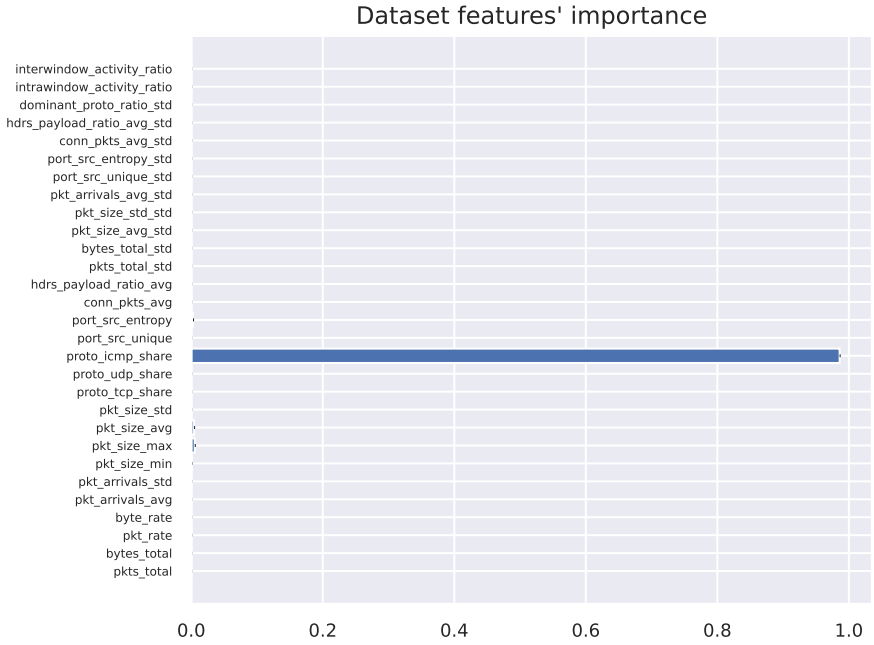


Figure 6.7: Decision Tree Gini features' importance for real dataset.

At this point, performing hyperparameter tuning or other data preprocessing techniques would have no purpose since all the models classify the data so well already. To confirm our assumptions, a process to determine feature importance (Figure 6.7) was executed. Importance was estimated directly from a tree classifier according to the mean decrease in impurity (Gini importance) technique. As it may be seen, a decision tree is able to decide almost exclusively based only on a single feature – the share of ICMP traffic, with almost perfect accuracy.

Since a bigger share of ICMP than usual may signify a network anomaly, deciding only according to it is nonsense because it definitely cannot generalize on different kinds of traffic. This result was somewhat expected after the first dataset analysis. Therefore, some less-generalizing features need to be removed in order to remove these skewed results.

Enhanced Dataset Features for Better Generalization

In order to achieve better generalization, we may try to remove irrelevant and obviously skewed features, which may affect the generalization performance negatively. These features include:

- `pkts_total` and `bytes_total` – irrelevant as found out with the correlation analysis
- `proto_tcp_share`, `proto_udp_share`, `proto_icmp_share` – provide skewed results according to the processed dataset

Similar to the whole dataset's case, cross-validation was run upon such modified data as well (Figure 6.8). As apparent, removing features allowing linear-separability caused less sophistical models (Naive Bayes, Nearest Centroid) to decrease their performance, especially the precision. However, more complex models such as neural network or ensemble

Model	fit_time	score_time	accuracy	accur_std	f1	precision	recall
scikit.adaboost	167.3584	1.5803	0.9983	0.0001	0.9983	0.9978	0.9988
scikit.bayes	1.4161	0.4063	0.9401	0.0004	0.9427	0.9039	0.9849
scikit.extra_trees	144.1588	3.4363	0.9996	0.0001	0.9996	0.9996	0.9995
scikit.grad_boosting	627.6410	0.4925	0.9993	0.0001	0.9993	0.9991	0.9995
scikit.logreg	14.3863	0.2113	0.9798	0.0005	0.9801	0.9675	0.9930
scikit.lda	11.6721	0.2315	0.9498	0.0006	0.9519	0.9132	0.9940
scikit.mlp	292.2500	1.6201	0.9983	0.0002	0.9983	0.9973	0.9994
scikit.nearest_centroid	1.1571	0.2366	0.8534	0.0004	0.8642	0.8048	0.9329
scikit.svm	3664.7791	215.5575	0.9945	0.0001	0.9946	0.9899	0.9993
scikit.tree	28.5166	0.2055	0.9993	0.0001	0.9993	0.9993	0.9993
scikit.random_forest	319.4423	1.9953	0.9996	0.0001	0.9996	0.9996	0.9996
xgboost.xgboost	441.6553	0.8398	0.9997	0.0001	0.9997	0.9996	0.9997

Figure 6.8: Model comparison with cross-validation upon the feature-reduced dataset containing 788 k samples. Trained on Fujitsu Esprimo Q920.

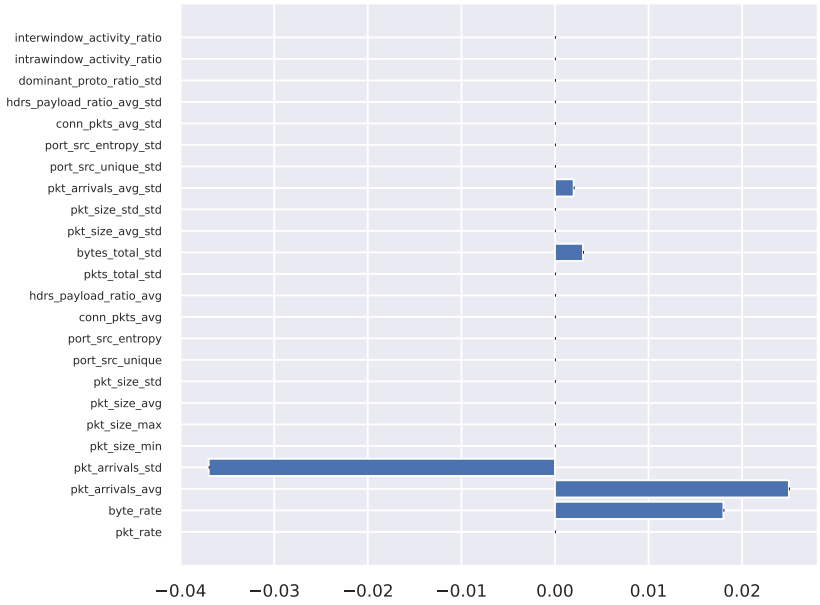


Figure 6.9: Feature importance for a neural network in feature-reduced dataset. Based on feature permutation importance technique.

techniques could still achieve excellent results. To our surprise, the performance of a single decision tree classifier also did not drop significantly. Feature importance analysis was thus performed again.

As apparent, more sophisticated methods have managed to find other important features with which they could successfully classify the given traffic. Whereas more sophisticated models were able to find many complex relationships such as the neural network (Figure 6.9) or random forest (Figure 6.10). Simple models such as the decision tree managed to provide similar performance with only 3 used features: `pkt_size_max` (0.84), `pkt_size_avg` $(0, 11)$, and `port_source_entropy` (0.02).

Reacting to these results, a closer look at the features has confirmed that the majority of attack traffic is composed of small packets, so almost perfect performance can be achieved almost purely based on them. This behavior is, indeed, undesired, as it biases towards

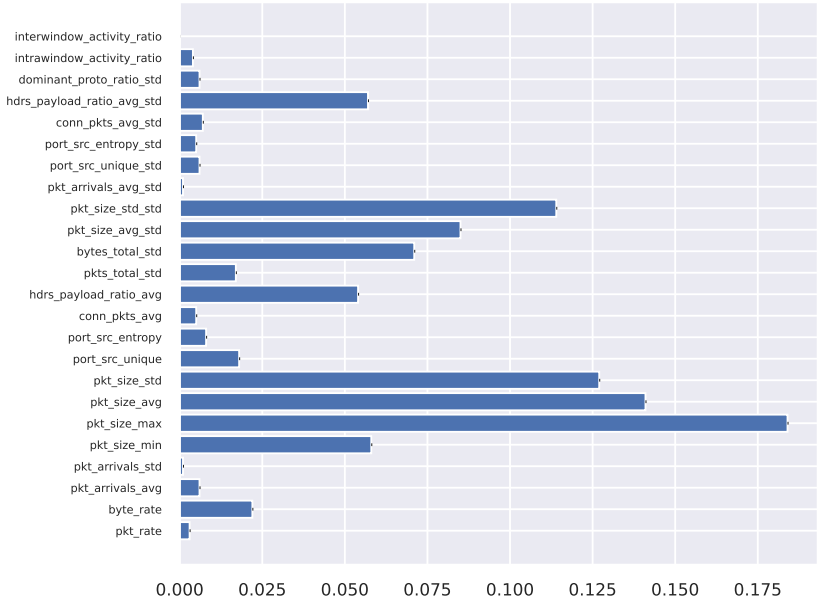


Figure 6.10: Feature importance for a random forest in feature-reduced dataset. Based on feature permutation importance technique.

a concrete dataset heavily and would thus provide very poor classification performance in general. Therefore, as the last attempt to increase generalization abilities and make the estimator's performance "more realistic", features `pkt_size_min`, `pkt_size_max`, and `pkt_size_avg` were dropped in addition to those mentioned at the start of this subsection. Cross-validation with default hyperparameters was again run, as shown in Figure 6.11.

Apparently, more sophisticated methods were still able to achieve astonishing results despite removing 8 out of 29 features used for classifications. All of the left features are mostly based on standard deviations, ratios, and rates, so they should generalize rather well on other types of traffic as well. The 6 most important features determined by the permutation feature importance technique for the XGBoost and neural network models are listed in Table 6.4. As it may be seen, two models working on different principles are affected by completely different features. Although due to the character of the permutation importance method, one has to keep in mind that this result is only an importance estimate. Figure 6.12 displays empirical feature (permutation) importance for a single decision tree. Gini impurity importance signified `pkt_size_std_std` to be the most significant with the value of 0.7684 and `hdrs_payload_ratio_avg` as the second with 0.1716.

Real Data Experiments Summary

This subsection has presented several of the picked experiments' results performed upon the real dataset. The tests were executed on the data extracted with the configuration of 1-second windows, 10-packet minimum per window, 6-window entry minimum, and 40 entropy samples. Cross-validation was run on data extracted with other configurations as well (e.g., 15-packet per window minimum with 6 windows or 20-packet minimum with 4 windows). Obtained results were very similar, all achieving over 0.995 f-score with high accuracy as well for more complex models. However, such high scores are nothing unusual.

Model	fit_time	score_time	accuracy	accur_std	f1	precision	recall
scikit.adaboost	150.98	1.5022	0.9948	0.0005	0.9948	0.9940	0.9955
scikit.bayes	0.7383	0.2699	0.9290	0.0009	0.9324	0.8902	0.9788
scikit.extra_trees	156.3544	3.5291	0.9994	0.0001	0.9994	0.9993	0.9994
scikit.grad_boosting	581.0599	0.4709	0.9988	0.0001	0.9988	0.9983	0.9994
scikit.logreg	15.3653	0.2006	0.9482	0.0004	0.9502	0.9149	0.9883
scikit.lda	11.3532	0.2339	0.8899	0.0006	0.9000	0.8248	0.9902
scikit.mlp	317.4023	1.2292	0.9984	0.0002	0.9985	0.9974	0.9995
scikit.nearest_centroid	0.7684	0.2428	0.8085	0.0006	0.8263	0.7559	0.9113
scikit.svm	2896.6739	259.16	0.9948	0.0001	0.9948	0.9904	0.9992
scikit.tree	29.4183	0.2507	0.9989	0.0001	0.9989	0.9989	0.9989
scikit.random_forest	428.4336	2.1473	0.9994	0.0001	0.9994	0.9993	0.9995
xgboost.xgboost	443.5139	0.6792	0.9994	0.0000	0.9994	0.9993	0.9996

Figure 6.11: Model comparison with cross-validation upon a feature-decimated real dataset containing 788 k samples. Trained on Fujitsu Esprimo Q920.

Rank	**XGBoost**		**Neural Network (MLP)**	
	Feature	Value	Feature	Value
1	hdrs_payload_ratio_avg	0.273	byte_rate	0.024
2	pkt_size_std_std	0.224	pkt_arrivals_std	−0.014
3	pkt_size_std	0.015	bytes_total_std	0.012
4	dominant_proto_ratio_std	0.012	pkt_arrivals_avg_std	0.002
5	port_src_entropy	0.008	pkt_arrivals_avg_std	0.001
6	conn_pkts_avg	0.001	pkt_size_std_std	0.000

Table 6.4: Permutation feature importance for various models upon feature-decimated real dataset.

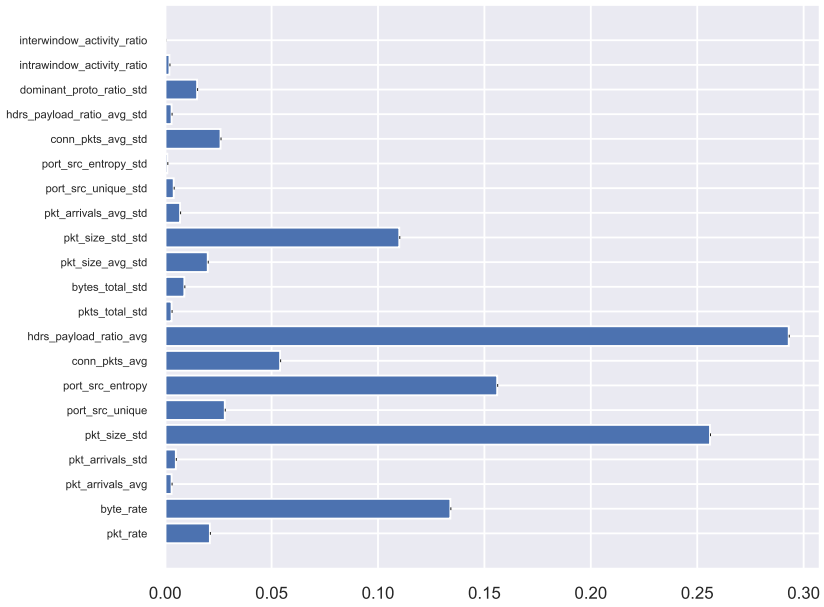
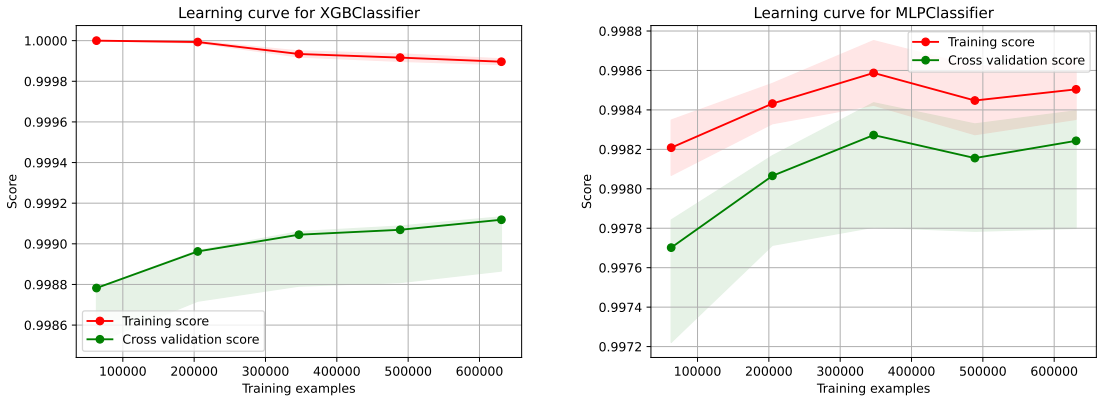


Figure 6.12: Feature importance for decision tree classifier based on feature permutation importance upon a feature-decimated real dataset.



(a) Learning curve for an XGBoost model. (b) Learning curve for a neural network model.

Figure 6.13: Effect of number of dataset samples on models' performance.

Similar research papers working with the CAIDA2007 DDoS attack dataset such as [108], [9], or [90] also declared similar accuracy with completely different classification approaches.

Since most of the models provide such perfect results, hyperparameter tuning was not performed, as it would be time-consuming with none to minimal benefits and may even lead to undesired overfitting in some cases.

As evident from the results, even a single decision tree model achieved almost perfect accuracy, f1-score, and other relevant metrics above 0.998 for all cases. These scores were obtained even after dropping most of the "bad" features, which were either unsuitable for generalization or suffered from being skewed based on the used dataset. The dataset itself was indeed not perfect – most of the data was ICMP, packets were mostly small, and the average number of packets/bytes sent by a single host was also relatively low for a DDoS scenario. For this reason, the dataset will be combined with synthetic data and experiments performed once again, as further described in Subsection 6.4.2.

Since the experimental results were obtained from a regular workstation PC, fitting and scoring times (in seconds) were relatively high. However, this fact amplified the models' performance, crucial for our online classification scenario. Therefore, despite some models performing with solid results, such as SVMs, random forest, or extra trees, their usage for our purposes is significantly limited due to their scoring times. As already noted, fitting times are not critical as there are no limits on them. These times can be shortened by not processing the dataset as a whole but only its representative sample of a limited size. However, significant dataset trimming was not performed because additional data may still increase the model's performance very slightly (Figure 6.13).

6.4.2 Mixed Data

The last subsection has performed system evaluation on real data. As discovered, the real DDoS dataset is mainly composed of ICMP traffic and thus is highly biased towards L4 protocol share features. Therefore, this subsection will mix the given dataset with synthetic data from CIC datasets and perform the evaluation process once again.

CIC datasets are mostly comprised of high-volume traffic with both TCP and UDP protocols. However, due to limited number of source addresses, a number of created dataset entries is rather low (Table 6.2). For this reason, all datasets cannot be simply merged

together because a high imbalance between them would cause the real datasets to dominate other entries, and thus a bias towards ICMP traffic would still be a threat. Therefore, real datasets were undersampled to include exactly as many entries as synthetic CIC data, so biasing should be significantly limited. Slow DoS attacks were not used in this series of experiments due to their specificity, as briefly outlined in the last paragraph of 4.2.1. The final balanced dataset used in the presented results thus contains 21860 entries of benign and attack traffic from both real and synthetic sources.

Feature Correlation

Similar to analysis in Subsection 6.4.1, a feature correlation analysis was performed first, as depicted in Figure 6.14. As in the real dataset, features describing the total number of bytes, the total number of packets, and their rates are correlated very strongly among themselves. In fact, pairs (`pkts_total`, `pkt_rate`) and (`bytes_total`, `byte_rate`) correlate exactly the same with all other features. Therefore, one of each can be considered redundant and dropped as in the previous case. As rates commonly generalize better than sums, `pkts_total` and `bytes_total` features can be safely dropped.

As can be observed from the correlation heatmap, many strong correlations are very similar to the real dataset. Since half of the mixed dataset is comprised of the previously analyzed one, strong correlations not ultimately negated by the synthetic dataset will still be present. An example of such negated relationship can be seen between the `proto_icmp_share` and `target`. In the real dataset, the correlation between them was 1.0. After supplying synthetic data, its value dropped to 0.6 since synthetic data contained no ICMP attacks.

Characteristics describing flat packet size (`packet_size_min`, `packet_size_max`, and `packet_size_avg`) now correlate with the target rather weakly. This is an optimistic finding because they correlated rather strongly in the real dataset, thus significantly decreasing the model's generalization. However, with such weak correlations like these, the features will probably not need to be dropped, as their impact on the model decision-making would not be that big.

A definitely unhealthy correlation can be found between TCP protocol share and the target. This relationship was strong even in the real dataset at the value of -0.8. Since the synthetic traffic did not provide enough TCP attacking data, the negative correlation strengthened. A strong negative correlation like this signifies a low probability of the attack if the traffic is of the TCP type. Although most of the TCP traffic on the Internet is indeed not malicious, the feature cannot be used as an ultimate deciding factor during traffic classification on its own.

By analyzing other correlations between features and the target, we may see that all others correlate expectedly according to expert knowledge and initial suppositions presented in Subsection 4.3.2. This is a relatively good sign, as the skewed real dataset was partially healed by merging it with the synthetically generated traffic. However, correlations between the target and protocols' shares will probably still need to be dropped.

Feature Analysis

After reviewing the features' correlation, some more exciting feature distributions and relationships are listed in Figure 6.15. Most of the features are the same as for the real dataset – primarily to illustrate the effects of dataset mixing, but also because they are relatively significant in this case as well.

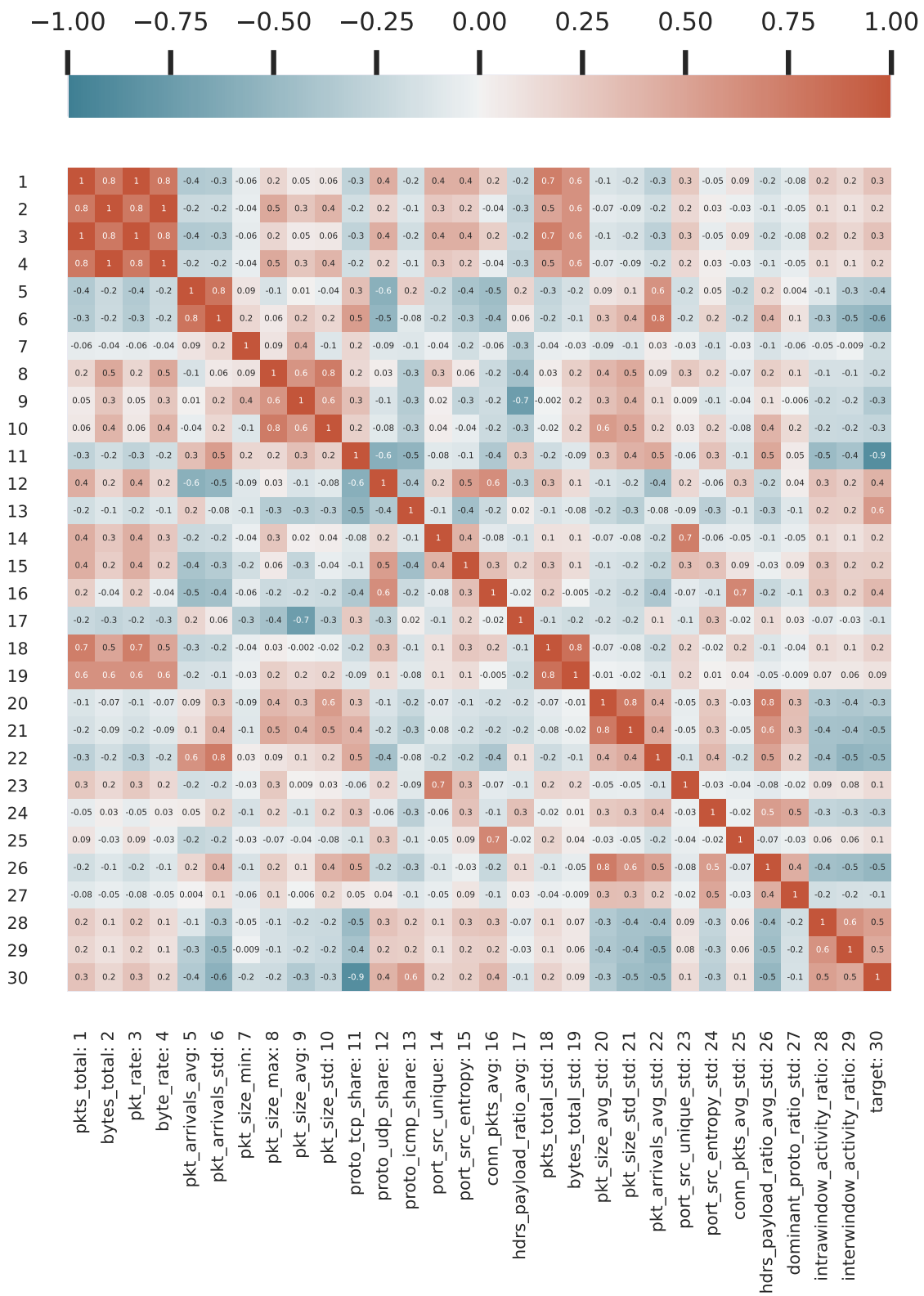
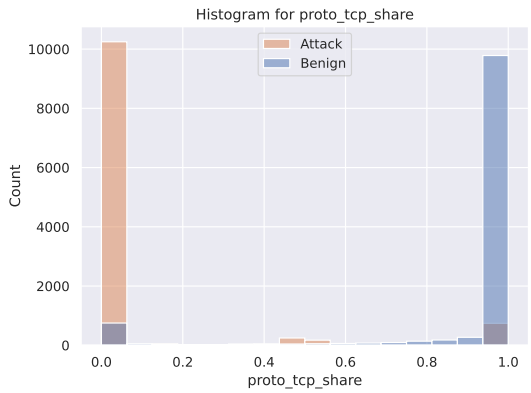
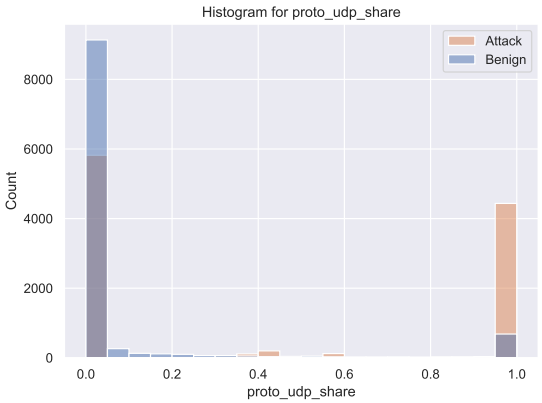


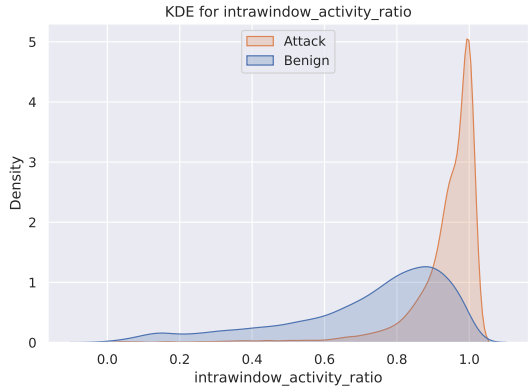
Figure 6.14: Mixed dataset features correlation.



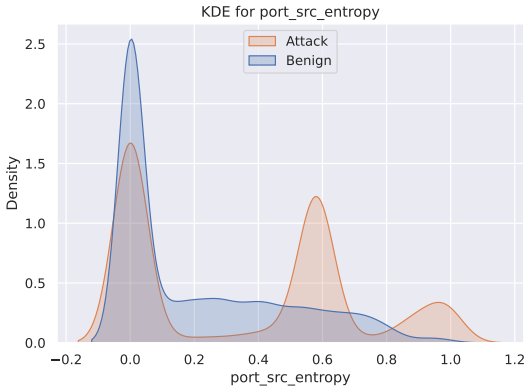
(a) TCP Protocol share.



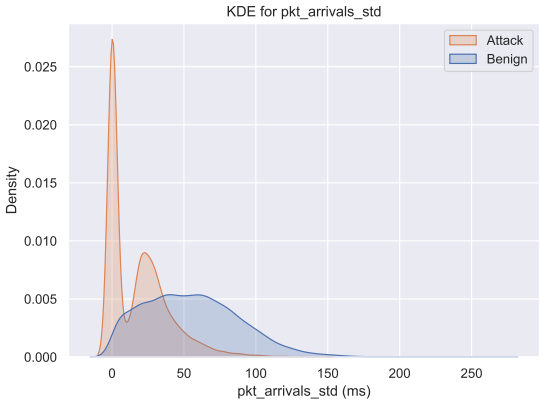
(b) UDP Protocol share.



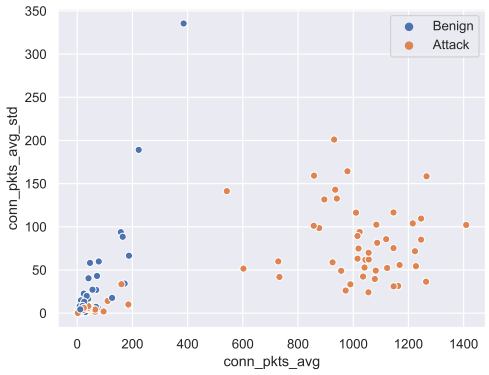
(c) Intra-window activity ratio density.



(d) Source port entropy density.



(e) Packet arrivals standard deviation density.



(f) Average number of packets per connections vs. inter-window standard deviation.

Figure 6.15: Real dataset feature analysis.

Figure 6.15a and Figure 6.15b illustrate TCP and UDP protocol shares across the attack and benign traffic. Compared to the real's dataset TCP share histogram (Figure 6.5a), it looks almost the same. Although there is some legitimate traffic at 0.0 share and some attack traffic at 1.0 share, their amount is so negligible that classification can be done

almost linearly with relatively good accuracy. On the other hand, UDP share ratio 1.0 is mostly associated with attack traffic, and a ratio of 0.0 dominates the legitimate class. However, there is a significant portion of attack traffic at a 0.0 ratio as well.

Inter-window and intra-window ratios (Figure 6.15c) distributions are almost identical to each other and very similar to the real dataset's case. Mixing the dataset thus has not affected this feature as much, but its density was somewhat increased due to volumetric attack characteristics behavior.

Source port entropy density (Figure 6.15d) and packet arrivals standard deviation (Figure 6.15e) clearly show the effects of mixing the datasets when compared to their equivalents in Figure 6.5. In the original dataset, attack traffic port entropy was predominant at 0.0 value due to ICMP traffic with no ports. Although port entropy of 0.0 is still predominant in attack traffic (due to the real dataset still comprising a significant portion), other 2 peaks around 0.6 and 1.0 for attack traffic were added. This is a sign of synthetic volumetric attacks, which often contain randomized ever-changing port numbers. The packet arrivals standard deviation was also updated by introducing a big peak at 0.0, again caused by high-volume attacking traffic.

An interesting relationship was found between the average number of packets per connection and their standard deviation between various windows (Figure 6.15f). These results actually counter our initial suppositions, which expected bigger numbers of packets per connection to be typical behavior of legitimate traffic. However, such connections with larger traffic amounts are almost exclusively related to attacking traffic. This can be explained in two scenarios: the attacker does not utilize port randomization, and so tremendous amounts of traffic may arrive from the same port, thus under a single connection. The second explanation lies within the real's dataset ICMP traffic. Since ICMP does not use port numbers, all of its traffic is labeled to come from a port 0 and thus may create an illusion that all data come from a single connection.

Classification Results

The evaluation of the classification performance was firstly performed by cross-validation upon all data without dropping any features. Since more sophisticated models again achieved over 0.998 performance similar to the real dataset, the specified features with lower generalization abilities or redundancy were again dropped. These include `pkts_total`, `bytes_total`, `proto_tcp_share`, `proto_udp_share`, and `proto_icmp_share`. Classification results upon mixed datasets with these features dropped are then displayed in Figure 6.16.

As apparent, the results of simple models (Bayes, nearest centroid) dropped significantly, but more complex models are still performing extraordinary well even without parameter tuning. Feature importance for a decision tree and random forest models is given in Table 6.5. As seen, these tree-based techniques were still able to pick the most relevant features based on packet sizes, which do not correlate with the target very strongly. As these are less generalizing for the real world, they were again dropped as in the real's dataset case. Although the dropping affected simpler methods like Bayes, LDA, logistic regression, and Nearest centroid, which achieved between 0.6266 and 0.8772 of F-score, more sophisticated methods like Random Forest, XGBoost, or Extra trees did not change their performance at all. To our surprise, a simple decision tree also achieved a 0.9921 F-score with the best scoring times overall.

Model	fit_time	score_time	accuracy	accur_std	f1	precision	recall
scikit.adaboost	3.2352	0.082	0.9946	0.0004	0.9946	0.996	0.9932
scikit.bayes	0.0526	0.0205	0.7231	0.0025	0.6279	0.9573	0.4672
scikit.extra_trees	1.9194	0.1248	0.9979	0.0009	0.9979	0.9989	0.9970
scikit.grad_boosting	14.1202	0.0258	0.9974	0.0007	0.9974	0.9981	0.9968
scikit.logreg	0.4203	0.0179	0.9366	0.0045	0.9369	0.9316	0.9423
scikit.lda	0.2157	0.0170	0.9263	0.0051	0.9267	0.9219	0.9315
scikit.mlp	28.3703	0.0290	0.9952	0.0018	0.9952	0.9947	0.9957
scikit.nearest_centroid	0.0342	0.0140	0.7939	0.0074	0.8068	0.7592	0.8609
scikit.svm	6.8902	2.6329	0.9704	0.0023	0.9705	0.9677	0.9734
scikit.tree	0.614	0.0147	0.9946	0.0014	0.9946	0.9944	0.9949
scikit.random_forest	6.1454	0.1095	0.9979	0.0009	0.9979	0.9989	0.9970
xgboost.xgboost	10.3377	0.0284	0.9985	0.0007	0.9985	0.9985	0.9984

Figure 6.16: Model comparison with cross-validation upon a feature-reduced mixed dataset containing 21 860 samples. Trained on Lenovo Yoga 460.

Rank	**Decision Tree**		**Random Forest**	
	Feature	Value	Feature	Value
1	pkt_size_max	0.609	pkt_size_max	0.154
2	pkt_rate	0.206	hdrs_payload_ratio_avg	0.096
3	hdrs_payload_ratio_avg	0.097	pkt_size_std_std	0.092
4	pkt_size_std	0.019	pkt_size_min	0.081
5	pkt_size_avg	0.011	pkt_size_avg_std	0.078
6	port_src_unique	0.009	hdrs_payload_ratio_avg	0.078

Table 6.5: Gini feature importances upon feature-reduced mixed dataset.

Minimum Number of Features

After failing to intentionally lower the models' performance by dropping the most important biasing features, we will now look onto the minimum number of required features for successful classification results. For this purpose, 3 models: neural network, random forest, and XGBoost, were chosen as the best performing estimators based on their times and classification performance. Additionally, a decision tree model was added as the one with the best scoring times and solid performance for comparison. Cross-validation classification performance expressed with an f-score with regard to the number of features is shown in Figure 6.17. Features were obtained using sequential forward feature selection based on random forest with 100 estimators. An f-score was used as the target scoring parameter. Features were selected in the following order:

1. hdrs_payload_ratio_avg
2. pkt_size_std
3. port_src_unique
4. conn_pkts_avg
5. dominant_proto_ratio_std
6. pkt_arrivals_avg
7. bytes_total_std
8. hdrs_payload_ratio_avg_std
9. pkt_size_std_std
10. port_src_entropy
11. byte_rate
12. intrawindow_activity_ratio

According to the results, it is apparent that a single feature – hdrs_payload_ratio_avg is able to provide enough information so the models can classify attacking and nonattacking hosts with an astonishing f-score higher than 0.9. With another feature added, the score for tree-based models jumps to 0.98. Four to five features have then practically achieved

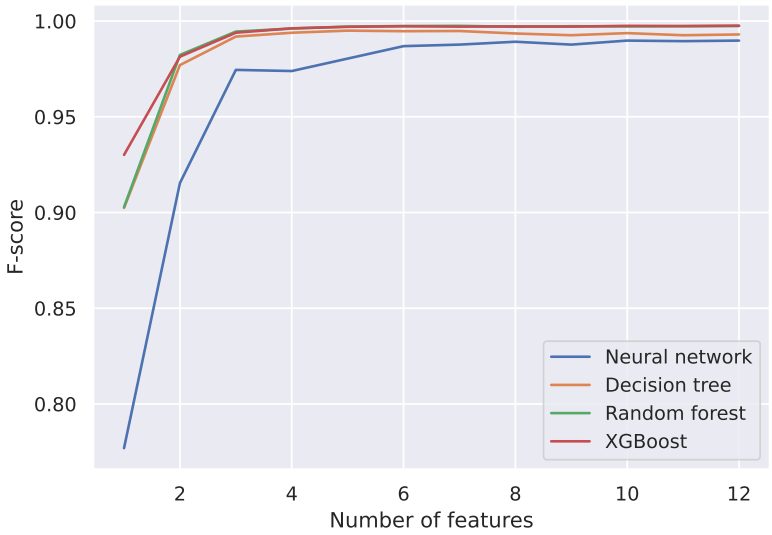


Figure 6.17: F-score classification performance based on the number of features.

the same performance of > 0.995 as they with all the available data. Adding additional features increases the score only very slightly. As it can be seen, the neural network model initially performed much worse than the tree-based models. This is most probably caused by hyperparameters that used their default values from the Scientific learn library. Tuning them for the specific task would undoubtedly increase a prediction score to be comparable with other models.

Although achieving such a good performance with a single feature was surprising, the features selected by the model make perfect sense in order to determine attacking hosts in practice. As all the features represent rates, standard deviations, and other statistical characteristics gathered throughout multiple windows, their generalization in real-world scenarios should be acceptable. However, the used dataset will still need to be slightly updated or modified, so a single feature will not be able to achieve such good results.

Decreasing Data-Collection Time

All the previously performed experiments used data summarized across 6 seconds (6 windows of 1 second) of the host's traffic. However, with such astonishing results achieved previously, it may be considered to lower this time to make the detection process faster. Intuitively, collecting data over a shorter period of time should lead to lowered performance, as fewer samples for statistics computation would be collected. In such cases, sudden traffic bursts or legitimate flash events may produce numerous false positives. This theory was tested with the configuration of 1-second windows, 4-window blocks, and 15 or 20 minimum packets per window, as outlined at the start of Section 6.4. The minimum number of packets per window was slightly increased to lower the amount classifications since smaller window blocks will now produce classifiable data more often.

As usual, the evaluation was performed with cross-validation upon various models, with results displayed in Figure 6.18. Biasing and redundant features such as protocol rates and packet sizes were again dropped dropped prior to the evaluation process. According

Model	fit_time	score_time	accuracy	accur_std	f1	precision	recall
scikit.adaboost	3.2352	0.0820	0.9946	0.0004	0.9946	0.9960	0.9932
scikit.bayes	0.0526	0.0205	0.7231	0.0025	0.6279	0.9573	0.4672
scikit.extra_trees	1.9194	0.1248	0.9979	0.0009	0.9979	0.9989	0.9970
scikit.grad_boosting	14.1202	0.0258	0.9974	0.0007	0.9974	0.9981	0.9968
scikit.logreg	0.4203	0.0179	0.9366	0.0045	0.9369	0.9316	0.9423
scikit.lda	0.2157	0.0170	0.9263	0.0051	0.9267	0.9219	0.9315
scikit.mlp	28.3703	0.0290	0.9952	0.0018	0.9952	0.9947	0.9957
scikit.nearest_centroid	0.0342	0.0140	0.7939	0.0074	0.8068	0.7592	0.8609
scikit.svm	6.8902	2.6329	0.9704	0.0023	0.9705	0.9677	0.9734
scikit.tree	0.6140	0.0147	0.9946	0.0014	0.9946	0.9944	0.9949
scikit.random_forest	6.1454	0.1095	0.9979	0.0009	0.9979	0.9989	0.9970
xgboost.xgboost	10.3377	0.0284	0.9985	0.0007	0.9985	0.9985	0.9984

Figure 6.18: Model comparison with cross-validation upon a feature-decimated mixed dataset with 4-second traffic blocks. Trained on Lenovo Yoga 460.

to achieved results, it may be stated that attacking and benign traffic may be recognized with high precision even if only 4 seconds are given to collect the data. This fact would make the detection mechanism especially competent during real-time mitigation in various environments.

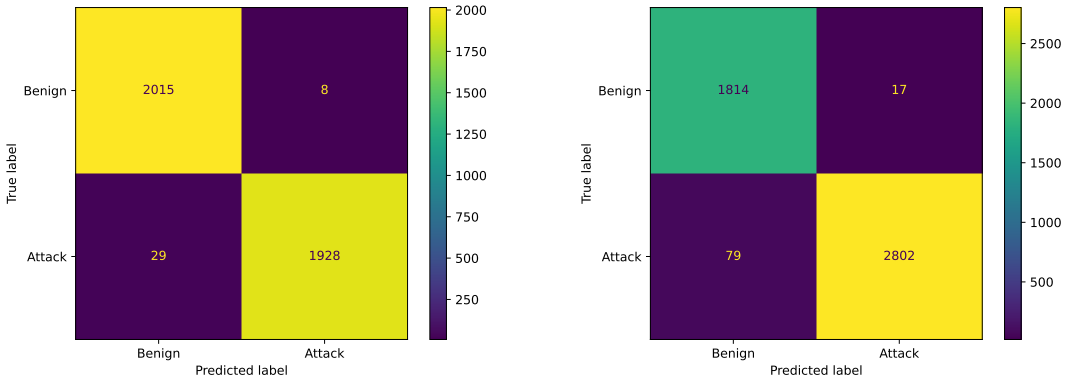
Slow DoS Attacks

Slow DoS attacks were excluded from all the experiments performed so far. This was done due to the reason that the system was primarily designed against volumetric DDoS attacks. Therefore, the proposed system may not be able to detect them based on the various specific characteristics they have. This series of tests will try to answer whether such attacks can be detected.

Currently, our biggest problem with slow DoS attacks is the data. The only available slow DoS data were extracted from CIC-IDS2017 Dataset and contain 361 entries (Table 6.2). On top of that, all the attacking traffic comes from a single IP address. The grouping mechanism used during train/test data splitting would then not separate these samples but keep them as a whole in either train or test dataset subset. For this reason, IP addresses in the slow DoS dataset were randomized. This removes the above-mentioned problem but causes that statistically dependent samples (from the same IP) will be present in both test and train datasets. IP randomization may slightly skew the results towards higher scores, but it is the only possible option that can be done without obtaining more data.

After IP randomization, the data was appended to the existing mixed dataset used throughout the previous tests. In order to balance this new dataset, 361 additional synthetic samples of legitimate traffic were also added. As usual, columns `pkts_total`, `bytes_total`, `proto_tcp_share`, `proto_udp_share`, and `proto_icmp_share` to prevent bias of badly generalizable features. As the slow DoS traffic represents only 1.6% of the whole dataset, the results will be presented upon a concrete model and its confusion matrix. This allows us to see the exact number of classifications instead of a single score, which would not represent the required information due to the small slow DoS sample size.

As the best performing model in most of the previous tests, XGBoost was chosen for the task. Its confusion matrix on the dataset not containing any slow DoS packets is displayed in Figure 6.19a. Figure 6.19b then represents the classification results after slow DoS attack data were added. As it may be seen, adding several Slow DoS samples caused the false



(a) Dataset *without* DoS confusion matrix.

(b) Dataset *with* DoS confusion matrix.

Figure 6.19: Comparison of DoS and non-DoS datasets performance.

negatives to increase by 0.9% and false positives by 0.1%. Matthews correlation coefficient for slow DoS traffic was 0.9578, whereas a value of 0.9815 was computed for non-slow DoS traffic.

6.4.3 Experiments Summary

The experiments performed throughout this section aimed to evaluate the performance of the machine learning system as a whole, regardless of the particular model or its hyper-parameters. According to achieved results, it may be concluded that almost any machine learning model performs exceptionally well when inappropriate data are given to its input. This might be seen in Figure 6.6, where most of the models achieved an f-score of > 0.998. Only trivial models like Naive Bayes or nearest centroid achieved a lesser score than 0.99.

As shown during the analysis, the real attack dataset was primarily comprised of ICMP messages, and thus an almost linear decision boundary could be drawn to determine whether the IP is an attacker or not (Figure 6.5b). Features utilizing this ability, alongside packet sizes, were then dropped to increase trained models' generalization capabilities. This process has fairly reduced the score of trivial methods, but more complex ones were affected only minimally (Figure 6.11), and an f-score of > 0.998 was kept.

Real dataset flaws were partially healed by mixing them with synthetic traffic from CIC datasets. This fact has significantly reduced the number of available training samples (from 788 k to around 22 k), but this seemed not to affect the models' ability to learn. The mixed dataset partially removed some unhealthy features correlations, but few strong correlations, like `proto_tcp_share` with the target, remained (Figure 6.14). Therefore, features leading to bias were again dropped, and the experiments repeated.

Using the mixed dataset with dropped biasing columns lowered the overall f-score from the initial 0.999 to 0.996. As revealed by later experiments, a single feature – `hdrs_payload_ratio_avg` was typically enough for models to achieve over 0.9 accuracy and f-score results. With 4 and more features, the performance improvements were relatively negligible and, in some cases, even dropped (Figure 6.17).

The end of the section then elaborated on detecting an attack within a shorter time period of only 4 seconds, which achieved similar performance to 6-second captures. This indicates that the attacking sources can be theoretically detected within 4 seconds of the

mitigation process started. Experiments with lower values were not performed, but a 3-second time block may provide similarly good performance as well.

Although the findings regarding Slow DoS attacks have revealed that the performance was worsened by 0.9% in false negatives and by 0.1% in false positives, the model could pick the most important traits of the slow DoS attack and thus successfully detect its originating IP addresses. However, this experiment is a little biased by design because not enough diverse samples were available for both evaluation and training.

In general, the best performing models were XGBoost, Random forest, Gradient boosting, and Extra trees. The neural network model with Adaboost performed slightly worse, followed by a Decision tree. However, no hyperparameter tuning for these models was performed, so it is possible that they would have very similar results in the end.

Best scoring times were achieved by Logistic regression, LDA, Naive Bayes, Nearest Centroid, and Decision trees. However, all mentioned methods except Decision trees were rather sensitive for dropping biasing features and their f1-score typically more or less decreased. On the other hand, the Decision tree model has kept its high precision and perfect estimation times throughout all of the experiments.

The overall winner of the comparison would be XGBoost, providing an excellent trade-off between classification performance and scoring times. Similar results to XGBoost were also achieved by the Gradient Boosting algorithm. Nevertheless, XGBoost often performed slightly better, is a popular choice in the machine learning community nowadays, can be visualized, and most importantly, also offers implementation in C++, which can even accept dumped trained models from its Python API. Therefore, real-world deployment and experiments continuation is suggested with the XGBoost algorithm.

6.5 Final System Considerations and Remarks

The previous section has performed several experiments comparing various machine learning models in different scenarios. At the end of the section, the XGBoost algorithm was chosen as the suggested model to use with the system due to its classification performance and speed of the execution. This section will discuss the tuning of such model, present a simulated model deployment and discuss a possible future work.

6.5.1 Hyperparameter Tuning

As discussed throughout the previous subsection, hyperparameter tuning was mostly not performed initially due to models having exceptionally high scores already. These assumptions were proven true by several performed tests showing that hyperparameters have little to no impact on the overall classification performance if the scores are so high. The only parameters intentionally set to very low or very high values have the actual potential to change the result – typically to worse, as shown in Figure 6.20. In this setup, the hyperparameter `max_depth` was set to 1, which indeed produced too simple models suffering from higher variance and thus achieving the lower classification score.

6.5.2 Simulated Model Deployment

The simulation of model deployment can be done with the provided script `mitigator.py`. The script computes the number of allowed and denied packets, as it would be in the real network where all packets (even from attackers) have to be allowed at first before a sufficient

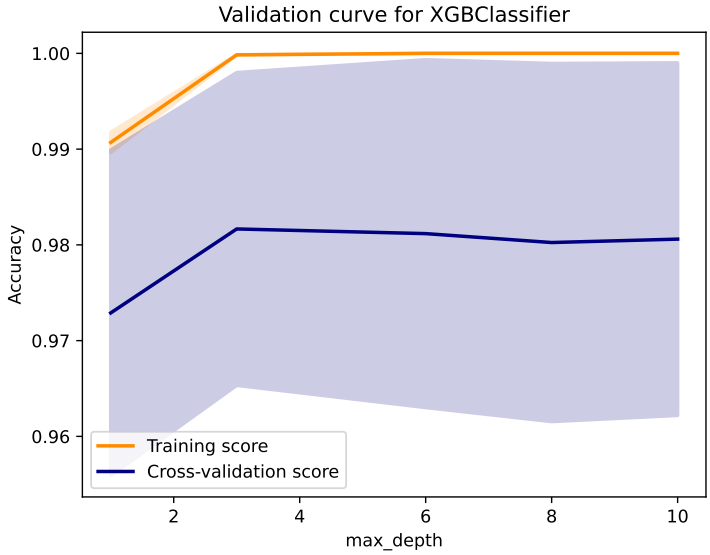


Figure 6.20: Influence of max_depth parameter for XGBoost performance.

amount of statistics is collected. Therefore, the method evaluates the "true" accuracy, as it does not evaluate the results by IP addresses but by packets. A test scenario with 5 attackers and 9 legitimate hosts was set up. Attacking hosts were extracted from CAIDA 2007 DDoS dataset and legitimate hosts from CAIDA 2016 Anonymized Internet traces. All entries corresponding to given IPs were removed from the training dataset to prevent mutually dependent samples from skewing the results.

The script was then run with the XGBoost model configured. According to the results, 4 of 5 attackers were detected successfully, and no false positives were produced. The method made a total number of 321 classifications, with 4 classifications being incorrect. The total number of denied packets for attackers was 40 914 out of 43 520, achieving accuracy for successfully denied packets of 0.940. No benign packets were denied, so the overall accuracy for benign clients was 1.00. This evaluation process can be replicated by running the whole ML pipeline with `python run.py` command.

6.5.3 Future Work

The most urgent issue to address in future work would be the data. As outlined numerous times throughout the document, the data with DDoS traffic in their raw PCAP form are very hard to come by. Although few such datasets exist, their quality is often unsuitable for the needs of the proposed mechanism. For this reason, the system was evaluated on sub-optimal data, causing models to get biased towards features with smaller generalization capabilities. Although this issue was tried to be solved by dropping such features, the quality of fitted models may be questionable due to the given data. A well-defined, modern, and diverse dataset resembling DDoS traffic characteristics is thus needed to prevent such biases and make the method usable in real-world scenarios. A discussion regarding dataset generation is further briefly conducted in Subsection 4.5.5.

Additional focus may be put on the data extraction and overall system pipeline configuration. Most of the experiments were performed upon dataset collected throughout 6

seconds of the client's communication. Nevertheless, tests in Subsection 6.4.2 have shown that the method works reliably even for 4-second traffic captures. Collecting statistics over a lesser number of windows (such as 3, 2, or even 1 second) was not performed, but it might be interesting to see models' behavior in such conditions as well.

Numerous other possibilities to "play" with the system also exist. One may try to implement other statistical features or other ways of stream data mining techniques. Although the window model proved to be efficient, other stream-processing algorithms may also be tried. Stream processing can also be offloaded to solutions like Apache Flink[2], which would allow distributed processing as well as collecting data from multiple network sources at once.

Lastly, for the method to be usable in practice, the overall pipeline needs to be accelerated in lower-level technologies such as compiled languages or FPGA to allow real-time packet processing without significantly decreasing the network throughput. The current implementation is done in Python, whereas almost no emphasis was put on the processing performance. Offloading feature extraction and statistics computation into FPGA would surely provide a significant performance boost. Data processing and utilization of machine learning models can also be reimplemented in lower-level languages such as C++, providing numerous machine learning frameworks working with much bigger overall performance.

[2]Stream-processing and batch-processing framework for stateful computations over data streams. Homepage: https://flink.apache.org/.

Chapter 7

Conclusions

The primary goal of this thesis was to design, implement, and evaluate a method for the mitigation of DDoS attacks using machine learning. Instead of implementing such a method in a hard-coded manner, this work aimed to create a highly configurable set of tools and scripts for that purpose. The leading idea behind the project was to provide a simple interface for anyone with minimal machine learning knowledge to use.

The ease of usability is achieved by 6 programs and supplementary scripts, which provide the complete functionality of a typical ML pipeline, including custom feature extraction, data preprocessing, and model training and evaluation. Typical ML tasks like dataset exploratory analysis and model deployment simulation are also highly automated. These programs allow to easily create own datasets from supplied PCAP files and evaluate various models with specified hyperparameters. A fitted model can then be exported and used for further experiments or deployment in the production environment.

Based on the analysis of several dozens of existing researches within a field, an attack detection mechanism based on the statistical features and packet metadata was proposed. In total, 8 features are extracted from every received packet. These are then processed within the windowed computational model, which computes relevant stream statistics upon them. Such statistics are summarized for several windows, and a 32-element feature vector is produced for classification. Evaluation of several publicly available datasets indicates the system's accuracy of over 99%, with an ability to detect an ongoing attack within the first 4 seconds of its start.

Although several real and synthetic datasets were used within the project, the data quality was still suboptimal. In general, there is a huge shortage of quality DDoS datasets in the raw PCAP format. Existing datasets are mainly available as CSV files with already pre-extracted features. However, these could not be utilized due to the specific needs of the proposed extraction mechanism. Therefore, several available PCAP datasets were used, but they were often corrupted, mislabeled, had a low attack diversity or very vague specification.

The above-mentioned issues with datasets were combated by merging the data from 7 different sources. This process has solved some issues of individual datasets, but many flaws and biases towards certain features were still retained. Therefore, the creation of a well-defined, modern, and diverse dataset resembling real DDoS traffic characteristics should be the top priority for future research. Additional future work, such as the need for system acceleration or possibilities of distributed computations, is briefly discussed in Subsection 6.5.3.

Bibliography

[1] *Biological Neuron Structure*. Pixabay, 2014. [Online; accessed 18-August-2020]. Available at: https://pixabay.com/vectors/neuron-nerve-cell-axon-dendrite-296581/.

[2] *Neural Networks Part 1: Setting up the Architecture*. Stanford University, 2020. In *CS231n: Convolutional Neural Networks for Visual Recognition*. [Online; accessed 19-August-2020]. Available at: https://cs231n.github.io/neural-networks-1/.

[3] Akamai Technologies. *Why Akamai Cloud Security for DDoS Protection?* 2021. [Online; accessed 13-March-2021]. Available at: https://www.akamai.com/uk/en/products/security/ddos-protection-service.jsp.

[4] Bai, S., Kolter, J. Z. and Koltun, V. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *CoRR*. arXiv. 2018.

[5] Barton, A. College Calculus with Analytic Geometry. (2nd edition.). *The Mathematical Gazette*. Cambridge University Press. 1972. DOI: 10.2307/3617003.

[6] Beer, F., Hofer, T., Karimi, D. and Bühler, U. A new Attack Composition for Network Security. Bonn: Gesellschaft für Informatik e.V. 2017, p. 11–20.

[7] Behal, S. and Kumar, K. Trends in Validation of DDoS Research. *Procedia Computer Science*. 2016, vol. 85, p. 7–15. ISSN 1877-0509. International Conference on Computational Modelling and Security (CMS 2016).

[8] Bengio, Y. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*. January 2009, vol. 2. DOI: 10.1561/2200000006.

[9] Bhaya, W. and EbadyManaa, M. DDoS attack detection approach using an efficient cluster analysis in large data scale. In: *2017 Annual Conference on New Trends in Information Communications Technology Applications (NTICT)*. 2017, p. 168–173.

[10] Boyd, S. and Vandenberghe, L. *Convex Optimization*. The Edinburgh Building, Cambridge, CB2 8RU, UK: Cambridge University Press, 2004. ISBN 978-0-521-83378-3.

[11] Breiman, L. *Bias, Variance, and Arcing Classifiers*. University of California.

[12] Brownlee, N., Mills, C. and Ruth, G. *Traffic Flow Measurement: Architecture*. October 1999. In Request for Comments: 2722.

[13] Buckland, M. and Gey, F. The relationship between Recall and Precision. *Journal of the American Society for Information Science*. January 1994, vol. 45, no. 1, p. 12–19.

[14] Canadian Institute for Cybersecurity. *NSL-KDD dataset*. 2009. Dataset for Intrusion detection. [Online; accessed 16-July 2021]. Available at: https://www.unb.ca/cic/datasets/nsl.html.

[15] Canadian Institute for Cybersecurity. *Intrusion detection evaluation dataset (ISCXIDS2012)*. 2012. Dataset for Intrusion detection. [Online; accessed 16-July 2021]. Available at: https://www.unb.ca/cic/datasets/ids.html.

[16] Canadian Institute for Cybersecurity. *CIC DoS dataset (2017)*. 2017. Dataset for Intrusion detection. [Online; accessed 16-July 2021]. Available at: https://www.unb.ca/cic/datasets/dos-dataset.html.

[17] Canadian Institute for Cybersecurity. *Intrusion Detection Evaluation Dataset (CIC-IDS2017)*. 2017. Dataset for Intrusion detection. [Online; accessed 16-July 2021]. Available at: https://www.unb.ca/cic/datasets/ids-2017.html.

[18] Canadian Institute for Cybersecurity. *CSE-CIC-IDS2018 on AWS*. 2018. Dataset for Intrusion detection. [Online; accessed 13-July 2021]. Available at: https://www.unb.ca/cic/datasets/ids-2018.html.

[19] Canadian Institute for Cybersecurity. *DDoS Evaluation Dataset (CIC-DDoS2019)*. 2019. Dataset for Intrusion detection. [Online; accessed 16-July 2021]. Available at: https://www.unb.ca/cic/datasets/ddos-2019.html.

[20] Center for Applied Internet Data Analysis. *The CAIDA UCSD „DDoS Attack 2007“ Dataset*. February 2010. [Online; accessed 18-Jun 2021]. Available at: https://www.caida.org/catalog/datasets/ddos-20070804_dataset/.

[21] Center for Applied Internet Data Analysis. *The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019)*. April 2018. [Online; accessed 18-Jun 2021]. Available at: https://www.caida.org/catalog/datasets/passive_dataset/.

[22] Chakrabarti, A., Cormode, G. and McGregor, A. *A near-optimal algorithm for computing the entropy of a stream*. January 2007.

[23] Chen, L., Zhang, Y., Zhao, Q., Geng, G. and Yan, Z. Detection of DNS DDoS Attacks with Random Forest Algorithm on Spark. Elsevier B.V. 2018, vol. 134, p. 310–315.

[24] Chen, T. and Guestrin, C. XGBoost: A Scalable Tree Boosting System. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, p. 785–794. KDD '16. ISBN 978-1-4503-4232-2.

[25] Chicco, D. and Jurman, G. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*. Jan 2020, vol. 21, no. 1, p. 6. ISSN 1471-2164.

[26] Chung, J., Gulcehre, C., Cho, K. and Bengio, Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv. December 2014.

[27] Cil, A. E., Yildiz, K. and Buldu, A. Detection of DDoS attacks with feed forward based deep neural network model. *Expert Systems with Applications*. 2021, vol. 169. DOI: https://doi.org/10.1016/j.eswa.2020.114520. ISSN 0957-4174.

[28] Cisco Systems. *Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper*. 70 West Tasman Dr., San Jose, CA 95134 USA, Jan 2018. Updated on March 9, 2020. Available at: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[29] Cloudflare, Inc.. *Comprehensive DDoS Protection*. 2021. [Online; accessed 13-March-2021]. Available at: https://www.cloudflare.com/ddos/.

[30] Cybenko, G. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*. Dec 1989, vol. 2, no. 4, p. 303–314. DOI: 10.1007/BF02551274. ISSN 1435-568X.

[31] Das, S., Mahfouz, A. M., Venugopal, D. and Shiva, S. DDoS Intrusion Detection Through Machine Learning Ensemble. In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2019, p. 471–477.

[32] Dong, S. and Sarem, M. DDoS Attack Detection Method Based on Improved KNN With the Degree of DDoS Attack in Software-Defined Networks. *IEEE Access*. 2020, vol. 8, p. 5039–5048.

[33] Erhan, D. and Anarim, E. Boğaziçi University Distributed Denial of Service Dataset. *Data in Brief*. August 2020, vol. 32.

[34] Fadlil, A., Riadi, I. and Aji, S. DDoS Attacks Classification using Numeric Attribute-based Gaussian Naive Bayes. *International Journal of Advanced Computer Science and Applications*. The Science and Information Organization. 2017, vol. 8, no. 8. DOI: 10.14569/IJACSA.2017.080806.

[35] Farnaaz, N. and Jabbar, M. Random Forest Modeling for Network Intrusion Detection System. *Procedia Computer Science*. Elsevier B.V. 2016, vol. 89, p. 213–217.

[36] Fawcett, T. An introduction to ROC analysis. *Pattern Recognition Letters*. 2006, vol. 27, no. 8, p. 861–874. ISSN 0167-8655. ROC Analysis in Pattern Recognition.

[37] Ferguson, P. and Senie, D. *Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing*. RFC Editor, May 2000. Request For Comments 2827.

[38] Flajolet, P., Fusy, É., Gandouet, O. and Meunier, F. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. Discrete Mathematics and Theoretical Computer Science. June 2007, DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), p. 137–156. DMTCS Proceedings.

[39] Gavrilis, D., Dermatas, E., Alrashdan, W. K., Wang, D. and Katkar, V. D. *Denial of Services Attack Detection using Random Forest Classifier with Information Gain*. 2017.

[40] Goldschmidt, P. *Heuristic Methods for the Mitigation of DDoS Attacks That Abuse TCP Protocol*. May 2019. Bachelor thesis. Faculty of Information Technology, Brno University of Technology. Available at: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=197664.

[41] Goldschmidt, P. TCP Reset Cookies – a Heuristic Method for TCP SYN Flood Mitigation. In: *Excel@FIT 2019*. Apr 2019.

[42] Goldschmidt, P. Adaptive SYN Flood Mitigation Based on Attack Vector Detection and Mitigation Process Monitoring. In: *Excel@FIT 2020*. May 2020.

[43] Graupe, D. *Principles of Artificial Neural Networks*. 3rd ed. World Scientific Publishing Co. Pte. Ltd., 2013. Advanced Series in Circuits and Systems.

[44] Gupta, B. B. and Badve, O. P. Taxonomy of DoS and DDoS attacks and desirable defense mechanism in a Cloud computing environment. *Neural Computing and Applications*. Dec 2017, vol. 28, no. 12, p. 3655–3682. DOI: 10.1007/s00521-016-2317-5.

[45] Haris, S. H. C., Ahmad, R. B. and Ghani, M. A. H. A. Detecting TCP SYN Flood Attack Based on Anomaly Detection. In: *Second International Conference on Network Applications, Protocols and Services*. September 2010, p. 240–244. DOI: 10.1109/NETAPPS.2010.50. ISBN 978-0-7695-4177-8.

[46] Hastie, T., Tibshirani, R. and Griedman, J. *The Elements of Statistical Learning: Data Mining Inference, and Prediction*. 2ndth ed. Springer, January 2016. ISBN 978-0387848570.

[47] Hochreiter, S. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*. April 1998, vol. 6, p. 107–116. DOI: 10.1142/S0218488598000094.

[48] Hochreiter, S., Bengio, Y., Frasconi, P. and Schmidhuber, J. Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. In: *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, March 2001.

[49] Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural Computation*. Cambridge, MA, USA: MIT Press. November 1997, vol. 9, no. 8, p. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. ISSN 0899-7667. Available at: https://doi.org/10.1162/neco.1997.9.8.1735.

[50] Hoffmann, L. D. and Bradley, G. L. *Calculus For Business, Economics, and the Social and Life Sciences*. 10th ed. Avenue of the Americas, New York, NY 10020: McGraw-Hill, 2010. ISBN 978–0–07–353231–8.

[51] Hornik, K. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*. 1991, vol. 4, no. 2, p. 251 – 257. ISSN 0893-6080.

[52] Hsieh, C.-J., Chang, K.-W., Lin, C.-J., Keerthi, S. S. and Sundararajan, S. A Dual Coordinate Descent Method for Large-Scale Linear SVM. In: *Proceedings of the 25th International Conference on Machine Learning*. Association for Computing Machinery, 2008, p. 408–415. ICML '08. ISBN 9781605582054.

[53] IMPACT. *DARPA 2009 Intrusion Detection Dataset – Dataset Details*. 2009. Online; accessed 17-July 2021]. Available at: https://www.impactcybertrust.org/dataset_view?idDataset=742.

[54] Imperva, Inc.. *DDoS Protection*. 2021. [Online; accessed 13-March-2021]. Available at: https://www.imperva.com/products/ddos-protection-services/.

[55] Internet Assigned Numbers Authority. *Protocol Numbers*. Last Updated: 2021-02-26. [Online; accessed 12-July 2021]. Available at: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[56] Jacko, D. *Inference of DDoS Mitigation Rules*. Brno, 2021. Master's thesis. Brno University of Technology. (Slovak).

[57] Jia, B., Huang, X., Liu, R. and Ma, Y. A DDoS Attack Detection Method Based on Hybrid Heterogeneous Multiclassifier Ensemble Learning. *Journal of Electrical and Computer Engineering*. March 2017, vol. 2017, p. 1–9. DOI: 10.1155/2017/4975343.

[58] Kaggle contributors. *DDoS Botnet Attack on IOT Devices*. 2020. Online; accessed 17-July 2021]. Available at: https://www.kaggle.com/siddharthm1698/ddos-botnet-attack-on-iot-devices.

[59] Kekely, L., Cabal, J., Puš, V. and Kořenek, J. Multi Buses: Theory and Practical Considerations of Data Bus Width Scaling in FPGAs. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, p. 49–56. DOI: 10.1109/DSD51259.2020.00020.

[60] Kiourkoulis, S. *DDoS Datasets: Use of Machine Learning to Analyse Intrusion Detection Performance*. 971 87 Luleå, Sweden, 2020. Master's thesis. Luleå University of Technology. Supervisor: Dr. Ali Ismail Awad.

[61] Kokila RT, Thamarai Selvi, S. and Govindarajan, K. DDoS detection and analysis in SDN-based environment using support vector machine classifier. In: *2014 Sixth International Conference on Advanced Computing (ICoAC)*. IEEE, 2014, p. 205–210.

[62] Krebs, B. *Study: Attack on KrebsOnSecurity Cost IoT Device Owners $323K*. May 2018. Available at: https://krebsonsecurity.com/2018/05/study-attack-on-krebsonsecurity-cost-iot-device-owners-323k.

[63] Kupreev, O., Badovskaya, E. and Gutnikov, A. *DDoS attacks in Q4 2020*. Kaspersky Lab, February 2021. [Online; accessed 27-February-2021]. Available at: https://securelist.com/ddos-attacks-in-q4-2020/100650/.

[64] Lahiri, B. and Tirthapura, S. Stream Sampling. In: *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, p. 2838–2842. ISBN 978-0-387-39940-9.

[65] Li, C., Wu, Y., Yuan, X., Sun, Z., Wang, W. et al. Detection and defense of DDoS attack–based on deep learning in OpenFlow-based SDN. *International Journal of Communication Systems*. vol. 31, no. 5, p. e3497.

[66] Li, F.-F. et al. *Convolutional Neural Networks (CNNs / ConvNets)*. 2020. In Standford's course *CS231n Convolutional Neural Networks for Visual Recognition*. [Online; accessed 14-January-2021]. Available at: https://cs231n.github.io/convolutional-networks/.

[67] Luger, G. F. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 5th ed. Pearson Addison Wesley, 2005. ISBN 0 321 26318 9.

[68] Manaf Gharaibeh. *DARPA 2009 Intrusion Detection Dataset*. July 2016. Online; accessed 17-July 2021]. Available at: http://www.darpa2009.netsec.colostate.edu/.

[69] Mansalis, S., Ntoutsi, E., Pelekis, N. and Theodoridis, Y. An evaluation of data stream clustering algorithms. *Statistical Analysis and Data Mining: The ASA Data Science Journal*. June 2018, vol. 11. DOI: 10.1002/sam.11380.

[70] Mitchell, T. M. *Machine Learning*. 1st ed. USA: McGraw-Hill, Inc., 1997. ISBN 0070428077.

[71] Mutually Agreed Norms for Routing Security. *MANRS Implementation Guide – Anti-Spoofing*. January 2017. [Online; accessed 11-Jul 2021]. Available at: https://www.manrs.org/isps/guide/antispoofing/.

[72] Network and Data Security Group Hochschule Fulda. *NDSec-1 Dataset Website*. 2016. [Online; accessed 20-Jun 2021]. Available at: https://www2.hs-fulda.de/NDSec/NDSec-1/.

[73] Nguyen, H.-V. and Choi, Y. Proactive detection of DDoS attacks utilizing k-NN classifier in an anti-DDoS framework. *World Academy of Science, Engineering and Technology*. March 2009, vol. 39, p. 640–645.

[74] Nogueira, M., Santos, A. A. and Moura, J. M. F. Early Signals from Volumetric DDoS Attacks: An Empirical Study. arXiv. Sep 2017.

[75] Nooribakhsh, M. and Mollamotalebi, M. A review on statistical approaches for anomaly detection in DDoS attacks. *Information Security Journal: A Global Perspective*. Taylor & Francis. 2020, vol. 29, no. 3, p. 118–133. DOI: 10.1080/19393555.2020.1717019.

[76] Ohsita, Y., Ata, S. and Murata, M. Detecting distributed denial-of-service attacks by analyzing TCP SYN packets statistically. In: *IEEE Global Telecommunications Conference, 2004. GLOBECOM '04.* 2004, vol. 4, p. 2043–2049 Vol.4.

[77] Oo, T. and Phyu, T. A Statistical Approach to Classify and Identify DDoS Attacks using UCLA Dataset. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*. May 2013, vol. 5.

[78] Oo, T. T. and Phyu, T. *Statistical Anomaly Detection of DDoS Attacks Using K-Nearest Neighbour*. January 2014.

[79] Osanaiye, O., Choo, K.-K. R. and Dlodlo, M. Distributed denial of service (DDoS) resilience in cloud: Review and conceptual cloud DDoS mitigation framework. *Journal of Network and Computer Applications*. 2016, vol. 67, p. 147–165. DOI: https://doi.org/10.1016/j.jnca.2016.01.001. ISSN 1084-8045.

[80] Özgür, A. and Erdem, H. A review of KDD99 dataset usage in intrusion detection and machine learning between 2010 and 2015. *PeerJ Prepr.* April 2016, vol. 4.

[81] Pascanu, R., Gulcehre, C., Cho, K. and Bengio, Y. How to Construct Deep Recurrent Neural Networks. arXiv. December 2013.

[82] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B. et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, p. 2825–2830.

[83] Powers, D. M. W. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. *International Journal of Machine Learning Technology*. January 2008, vol. 2.

[84] Pramana, M. I. W., Purwanto, Y. and Suratman, F. Y. DDoS detection using modified K-means clustering with chain initialization over landmark window. In: *2015 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*. 2015, p. 7–11.

[85] Quittek, J., Zseby, T., Claise, B. and Zander, S. *Requirements for IP Flow Information Export (IPFIX)*. October 2004. In Request for Comments: 3917.

[86] Radware Inc.. *Global application & network security report 2015-2016*. 2016.

[87] Raileanu, L. E. and Stoffel, K. Theoretical Comparison between the Gini Index and Information Gain Criteria. *Annals of Mathematics and Artificial Intelligence*. May 2004, vol. 41, no. 1, p. 77–93.

[88] Raina, H. and Shafi, O. Analysis Of Supervised Classification Algorithms. *International Journal of Scientific & Technology Research*. September 2015, vol. 4. ISSN 2277-8616.

[89] Ramamoorthi, A., Subbulakshmi, T. and Shalinie, S. M. Real time detection and classification of DDoS attacks using enhanced SVM with string kernels. In: *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*. 2011, p. 91–96.

[90] Rejimol Robinson, R. R. and Thomas, C. Ranking of machine learning algorithms based on the performance in classifying DDoS attacks. In: *2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*. 2015, p. 185–190.

[91] Richman, J. S., Lake, D. E. and Moorman, J. Sample Entropy. In: *Numerical Computer Methods, Part E.* Academic Press, 2004, vol. 384, p. 172–184. Methods in Enzymology. ISSN 0076-6879.

[92] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*. 1958, 65 6, p. 386–408.

[93] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning representations by back-propagating errors. *Nature*. October 1986, vol. 323, no. 6088, p. 533–536. DOI: 10.1038/323533a0. ISSN 1476-4687.

[94] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Fourth editionth ed. Pearson, 2020. ISBN 978-0136042594.

[95] Saha, S. *A Comprehensive Guide to Convolutional Neural Networks – the ELI5 way*. December 2018. Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[96] Saied, A., Overill, R. E. and Radzik, T. Detection of known and unknown DDoS attacks using Artificial Neural Networks. *Neurocomputing*. 2016, vol. 172, p. 385–393. DOI: https://doi.org/10.1016/j.neucom.2015.04.101. ISSN 0925-2312.

[97] Santos, R., Souza, D., Santo, W., Ribeiro, A. and Moreno, E. Machine Learning Algorithms to Detect DDoS Attacks in SDN. *Concurrency and Computation: Practice and Experience*. June 2020, vol. 32, no. 16. DOI: https://doi.org/10.1002/cpe.5402.

[98] Shalev Shwartz, S., Singer, Y., Srebro, N. and Cotter, A. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming*. Mar 2011, vol. 127, p. 3–30.

[99] Shannon, C. E. A Mathematical Theory of Communication. *Bell System Technical Journal*. 1948, vol. 27, no. 3, p. 379–423.

[100] She, C., Wen, W., Zheng, K. and Lyu, Y. Application-Layer DDoS Detection by K-means Algorithm. In: *2016 4th International Conference on Electrical & Electronics Engineering and Computer Science (ICEEECS 2016)*. Atlantis Press, December 2016, p. 75–78. ISBN 978-94-6252-265-7.

[101] Shiravi, A., Shiravi, H., Tavallaee, M. and Ghorbani, A. A. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*. 2012, vol. 31, no. 3, p. 357–374. ISSN 0167-4048.

[102] Singh, A., Garg, S., Kaur, R., Batra, S., Kumar, N. et al. Probabilistic data structures for big data analytics: A comprehensive review. *Knowledge-Based Systems*. 2020, vol. 188, p. 104987. ISSN 0950-7051.

[103] Subbulakshmi, T. et al. Detection of DDoS attacks using Enhanced Support Vector Machines with real time generated dataset. In: *2011 Third International Conference on Advanced Computing*. 2011, p. 17–22.

[104] Tavallaee, M., Bagheri, E., Lu, W. and Ghorbani, A. A. A detailed analysis of the KDD CUP 99 data set. In: *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. 2009, p. 1–6.

[105] Theodoridis, S. and Koutroumbas, K. *Pattern Recognition, Fourth Edition*. 4thth ed. USA: Academic Press, Inc., 2008. ISBN 1597492728.

[106] Tuan, N. N., Hung, P. H., Nghia, N. D., Van Tho, N., Phan, T. V. et al. A Robust TCP-SYN Flood Mitigation Scheme Using Machine Learning Based on SDN. In: *2019 International Conference on Information and Communication Technology Convergence (ICTC)*. 2019, p. 363–368.

[107] Tuan, N. N., Hung, P. H., Nghia, N. D., Tho, N. V., Phan, T. V. et al. A DDoS Attack Mitigation Scheme in ISP Networks Using Machine Learning Based on SDN. *Electronics*. February 2020, vol. 9, no. 3. ISSN 2079-9292.

[108] Tuan, N. N., Hung, P. H., Nghia, N. D., Tho, N. V., Phan, T. V. et al. A DDoS Attack Mitigation Scheme in ISP Networks Using Machine Learning Based on SDN. *Electronics*. 2020, vol. 9, no. 3. ISSN 2079-9292. Available at: https://www.mdpi.com/2079-9292/9/3/413.

[109] Uhrig, R. E. Introduction to artificial neural networks. In: *Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics*. 1995, vol. 1, p. 33–37 vol.1.

[110] University of California, Irvine. *KDD Cup 1999 Data*. October 1999. [Online; accessed 16-July 2021]. Available at: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[111] Viswarupan, N. *K-Means Data Clustering*. Jul 2017. [Online; accessed 26-June-2020]. Available at: https://towardsdatascience.com/k-means-data-clustering-bce3335d2203.

[112] Wang, C., Zheng, J. and Li, X. Research on DDoS Attacks Detection Based on RDF-SVM. In: *2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA)*. 2017, p. 161–165.

[113] Welford, B. P. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*. Taylor & Francis. 1962, vol. 4, no. 3, p. 419–420.

[114] Wikipedia contributors. *Decision tree learning*. 2020. [Online; accessed 5-August-2020]. Available at: https://en.wikipedia.org/wiki/Decision_tree_learning.

[115] Xie, Y. and Yu, S. Monitoring the Application-Layer DDoS Attacks for Popular Websites. *IEEE/ACM Transactions on Networking*. 2009, vol. 17, no. 1, p. 15–25. DOI: 10.1109/TNET.2008.925628.

[116] Yan, Q., Yu, F. R., Gong, Q. and Li, J. Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges. *IEEE Communications Surveys Tutorials*. 2016, vol. 18, no. 1, p. 602–622. DOI: 10.1109/COMST.2015.2487361.

[117] Ye, J., Cheng, X., Zhu, J., Feng, L. and Song, L. A DDoS Attack Detection Method Based on SVM in Software Defined Network. *Security and Communication Networks*. Hindawi. Apr 2018, vol. 2018.

[118] Yiu, T. *Understanding Random Forest*. Jun 2019. [Online; accessed 7-August-2020]. Available at: https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

[119] Yuan, X., Li, C. and Li, X. DeepDefense: Identifying DDoS Attack via Deep Learning. In: *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2017, p. 1–8.

[120] Yudhana, A., Riadi, I. and Ridho, F. DDoS Classification Using Neural Network and Naïve Bayes Methods for Network Forensics. *International Journal of Advanced Computer Science and Applications*. December 2018, vol. 9, p. 177–183. DOI: 10.14569/IJACSA.2018.091125.

[121] Zakka, K. *A Complete Guide to K-Nearest-Neighbors with Applications in Python and R*. July 2016. [Online; accessed 26-June-2020]. Available at: https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor.

[122] Zhang, X., Zhao, J. J. and LeCun, Y. Character-level Convolutional Networks for Text Classification. *CoRR*. arXiv. April 2015.

[123] Zhu, M., Ye, K. and Xu, C.-Z. Network Anomaly Detection and Identification Based on Deep Learning Methods. In: Luo, M. and Zhang, L.-J., ed. *Cloud Computing – CLOUD 2018*. Springer International Publishing, 2018, p. 219–234.

Appendix A

Neural Networks Learning

Learning of the neural networks is typically done in a supervised manner. During the process, a network receives pairs of $(\overline{x}, y)$ where $\overline{x}$ is the input feature vector and y the expected output. Neural networks firstly compute an output estimation $\hat{y}$ for $\overline{x}$ (forward pass). A Loss (Error, Cost) function L is then used to compute a difference (estimation error) between $\hat{y}$ and y. Let $T = (\overline{x}_1, y_1), \ldots, (\overline{x}_n, y_n)$ be the set of all training pairs. The goal of the learning is thus to minimize $L(T, \theta)$ by modifying network parameters θ. For this purpose, a gradient ∇E is firstly computed, and its values are used by one of the optimizers, such as gradient descent, to perform an actual parameter update.

An efficient and most popular way to compute gradients in feedforward networks is to use the Backpropagation (BP) algorithm [93]. The algorithm firstly computes an error function and gradients for the network's output layer. Gradients of the current layer are then used for gradient computation in the previous layer and so on. The error is thus propagated backward from the last to the first layer. Various modifications of the algorithm are used for other network types, such as Backpropagation Through Time (BPTT) for RNNs. The following paragraphs will briefly describe the math behind BP for FNNs.

The loss function defined in the original paper and used throughout the following explanation is defined as Eq. A.1, where n_L represents the number of neurons in the output layer (size of the output vector). This function is calculated for each input pair separately and then summed with other samples to form a total training loss used to compute the gradients. Nowadays, other loss functions, such as Cross-Entropy and Hinge loss, are utilized, but the principles explained here stay mostly the same.

$$L = \frac{1}{2} \sum_{j=1}^{n_L} (\hat{y}_j - y_j)^2 \tag{A.1}$$

Network Parameters Update

In order to minimize the function L, the algorithm needs to compute its partial derivatives of all network parameters θ, called gradient. Since gradient represents the direction of the function increase, the mechanism needs to update parameters in the opposite direction, so that L will decrease. For example, the Gradient descent optimizer updates network parameters according to Eq. A.2, where T is a set of training samples, θ^t are network parameters in time t, and α is a learning constant.

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial L(T, \theta^t)}{\partial \theta} \tag{A.2}$$

Full (batch) gradient descent would require the gradient computed for every sample in the training set and perform an update from Eq. A.2 afterward. This is highly computationally inefficient, so in practice, either stochastic (SGD) or mini-batch (MGD) gradient descent variants are used. SGD does not compute the partial derivatives for each element but performs an update after each training sample. On the other hand, MGD performs parameter update only according to the gradients of the training data subset. Other more sophisticated optimizers like RMSprop, Adam, or AMSGrad also change the learning rate dynamically and utilize properties of the gradient (like steepness) to update the network's parameters even more efficiently, often resulting in faster convergence.

Gradient Computation

Although numerous computing gradient methods exist, the Backpropagation algorithm and its modifications are used in most neural network applications nowadays. Suppose:

- ${}^k w_{ji}$ be a weight between node j in k^{th} layer and node i in $k-1^{\text{th}}$ layer
- n_k be a number of neurons in the k^{th} layer
- ${}^k\varphi(x)$ be an activation function for the k^{th} layer
- ${}^k y_j$ be an output of the neuron j in k^{th} layer, ${}^k y_j = {}^k\varphi({}^k u_j)$
- ${}^k u_j$ be an internal state of the neuron j in k^{th} layer: ${}^k u_j = \sum_{i=1}^{n_{k-1}} {}^k w_{ji} {}^{k-1} y_i + {}^k\Theta_j$

The partial derivative of the particular weight can be computed using the chain rule[1] as: $\frac{\partial L}{\partial {}^k w_{ji}} = \frac{\partial L}{\partial {}^k u_j} \frac{\partial {}^k u_j}{\partial {}^k w_{ji}}$. The first term is usually called the error and is denoted by δ: ${}^k\delta_j = \frac{\partial L}{\partial {}^k u_j}$. The partial derivative of the second term is $\frac{\partial {}^k u_j}{\partial {}^k w_{ji}} = \frac{\partial}{\partial {}^k w_{ji}} \left(\sum_{l=1}^{n_{k-1}} {}^k w_{jl} {}^{k-1} y_l + {}^k\Theta_j \right) = {}^{k-1} y_i$. Therefore, the partial derivative of the loss function L with respect to a particular weight ${}^k w_{ji}$ is defined by Eq. A.3.

$$\frac{\partial L}{\partial {}^k w_{ji}} = {}^k\delta_j {}^{k-1} y_i \tag{A.3}$$

We will now calculate partial derivatives for the previously defined loss function in Eq. A.1. Firstly, partial derivatives for the output layer are calculated, and the error is then propagated back to hidden layers. The loss function can be expressed in terms of the value ${}^o u_j$ as $L = \frac{1}{2} \sum_{j=1}^{n_o} (\hat{y}_j - y_j)^2 = \frac{1}{2} \sum_{j=1}^{n_o} ({}^o\varphi({}^o u_j) - y_j)^2$, where o is the index of the last (output) network layer. Applying partial derivative with the chain rule then gives: ${}^o\delta_j = ({}^o\varphi({}^o u_j) - y_j)\, {}^o\varphi'({}^o u_j) = (\hat{y}_j - y_j)\, {}^o\varphi'({}^o u_j)$. Therefore, the loss function partial derivative of a weight for a neuron in the last layer can be expressed according to Eq. A.4.

$$\frac{\partial L}{\partial {}^o w_{ji}} = {}^o\delta_j {}^{o-1} y_i = (\hat{y}_j - y_j)\, {}^o\varphi'({}^o u_j) {}^{o-1} y_i \tag{A.4}$$

[1] Formula to compute the derivative of a composite function: $(f \circ g)' = (f' \circ g)\,.\,g'$.

$$ {}^{k}\delta_j = \frac{\partial L}{\partial {}^{k}u_j} = \sum_{l=1}^{n_{k+1}} \frac{\partial L}{\partial {}^{k+1}u_l} \frac{\partial {}^{k+1}u_l}{\partial {}^{k}u_j} \tag{A.5} $$

At this point, the computed error needs to be propagated to other non-output layers. Using the chain rule, ${}^{k}\delta_j$ for $1 \leq k < o$ can be expressed as Eq. A.5. The first term of the equation is simply a δ of the neuron in the next layer. The numerator of the second term is ${}^{k+1}u_l = \sum_{i=1}^{n_k} {}^{k+1}w_{li}\,{}^{k}y_i + {}^{k+1}\Theta_l = \sum_{i=1}^{n_k} {}^{k+1}w_{li}\,{}^{k}\varphi({}^{k}u_i) + {}^{k+1}\Theta_l$. Its partial derivative would thus result in $\frac{\partial {}^{k+1}u_l}{\partial {}^{k}u_j} = {}^{k+1}w_{lj}\,{}^{k}\varphi'({}^{k}u_j)$. Considering these calculations, the error term ${}^{k}d_j$ for hidden layers may be written as Eq. A.6, called the backpropagation formula. Finally, Eq. A.7 represents the loss function derivative of a weight in a hidden layer by substituting the error term into Eq. A.3.

$$ {}^{k}\delta_j = \sum_{l=1}^{n_{k+1}} {}^{k+1}\delta_l\,{}^{k+1}w_{lj}\,{}^{k}\varphi'({}^{k}u_j) = {}^{k}\varphi'({}^{k}u_j) \sum_{l=1}^{n_{k+1}} {}^{k+1}w_{lj}\,{}^{k+1}\delta_l \tag{A.6} $$

$$ \frac{\partial L}{\partial {}^{k}w_{ji}} = {}^{k}\delta_j\,{}^{k-1}y_i = {}^{k}\varphi'({}^{k}u_j)\,{}^{k-1}y_i \sum_{l=1}^{n_{k+1}} {}^{k+1}w_{lj}\,{}^{k+1}\delta_l \tag{A.7} $$

As it may be seen in Eq. A.7, in order to compute gradients in any hidden layer k, error terms (δ) need to be calculated for every neuron in the layer $k+1$. Backpropagation takes advantage of this property and thus allows to compute gradients very efficiently in an iterative manner. After each layer's gradients for every input-output pair are computed, they are combined to produce the total gradient $\frac{\partial L(T,\theta)}{\partial {}^{k}w_{ji}}$ for the entire set T. The total gradient is then used to update weights based on the used optimizer. This process is repeated until a predefined criterion, such as the number of training epochs, is met.

Appendix B

System's Configuration File

```
# Mitigation of DoS Attacks Using Machine Learning project global configuration file
# Author: Patrik Goldschmidt (xgolds00@stud.fit.vutbr.cz)
# Date: 14.06.2021, last rev. 19.07.2021
# Note: Non-required lines can be commented out. In this case default values will be used.
---
dataset_creator:
    # (Optional, default: 0) Number of packets to report after during processing. Disable: 0.
    report_status_packets: 1000000

cleaning:
    # Column names to drop from dataset
    drop_cols:
        - "window_count"
        - "window_span"

feature_importance:
    # Method to determine feature importance. Options: permutation|direct
    # Direct method for tree-based techniques such as adaboost or xgboost represent decrease
    # in impurity within each tree.
    method: "direct"

    # Library to load the model from. Options: scikit|xgboost
    model_source: "scikit"

    # Which particular model from the library to use. Note that all models support
    # permutation method, but only some can provide feature importance estimation directly
    # by themselves. Consult model's documentation to learn more.
    # Models supporting direct method:
    # Scikit: adaboost, extra_trees, grad_boosting, tree, random_forest
    # Xgboost: xgboost
    # Other models without direct feature importance estimation:
    # Scikit: bayes, kneighbors, logreg, lda, nearest_centroid, svm
    model_type: "adaboost"

    # Scoring to use for importance estimation. Leave commented out for default model scorer
    #scoring = "accuracy"

feature_plotter:
    # Directory to which to save plots
    plots_dir: "plots"

    # Resulting plot files extension
```

```
    # Options: ps|eps|pdf|png|svg|jpg|jpeg|tif|tiff
    file_extension: "pdf"

    # (Optional, default: True) Whether to remove outliers when plotting histograms
    hist_rem_outliers: True

    # (Optional, default: True) Plot boxplots for each feature
    plot_boxplots: False

    # (Optional, default: True) Plot Empiric distribution functions for each feature
    plot_ecdfs: False

    # (Optional, default: True) Plot histograms for each feature
    plot_histograms: True

    # (Optional, default: True) Plot Kernel density estimations for each feature
    plot_kdes: False

    # (Optional, default: True) Plot all-in-one graphs.
    # All-in-one graph is a single file containing Boxplot,ECDF, Histogram, and KDE
    plot_all_in_ones: True

    # (Optional, default: True) Plot summary graphs.
    # Summary graph contains plots for all features except target variable in one file
    plot_summaries: True

    # (Optional, default: True) Plot correlation heatmap for all variables
    plot_cor_heatmap: True

    # (Optional, default: []) Multivariate plot to show relationship between two features
    # Enter pairs in the form of the list - ['feature1', 'feature2']
    multivariate_scatter_features:
        - ['port_src_unique', 'port_src_entropy']

    # (Optional, default: False) Call custom plotting function for user-defined plots
    plot_custom: False

    # (Optional, default: 300) Number of scatterplot samples to plot
    scatter_samples: 300

feature_projection:
    # Feature projectcion method to use
    # Options:
    # fa  - Factor Analysis
    # lda - Linear Discriminant Analysis
    # pca - Principal Component Analysis
    method: "pca"

    # Desired number of components after feature projection
    n_components: 25

feature_selection:
    # Feature selection configuration. See [1] details.
    # Options:
    # varthreshold - Variance Threshold selector
    # kbest  - Select K best features according to statistical test
    # rfe  - Recursive Feature Elimination
    # model  - Select From Model Selection
    # sfs  - Sequential Feature Selector
```

```
    # [1] https://scikit-learn.org/stable/modules/feature_selection.html
    # Note: Due to the complexity and the number of options available, parameters of
    # selectors cannot be modified using this config file. The only available option is
    # "n_features_to_select" to specify the number of desired features for Kbest, RFE, model,
    # and SFS. Varthreshold option may also utilize 'threshold' option. Selectors that
    # require model use random forest with 100 estimators, which was empirically proven to
    # perform relatively solid. If more advanced configuration is desired, modify the
    # FeatureSelector's constructor in dataprocessing/feature_modification/selection.py file.
    method: "model"

    # Number of features to select. Supports all selectors except "varthreshold"
    n_features_to_select: 20

logger:
    # Length of the window in seconds
    window_length: 1

    # (Optional, default: 6) Minimum number of collected windows to process the given IP
    history_min: 6

    # (Optional, default: 0) Maximum number of elements that are stored in memory for
    # history. 0 refers to "infinity", allowing to store records to up 5GB of computer memory
    history_size: 0

    # (Optional, default: 120) Maximum number of seconds for which historical logs are valid
    history_timeout: 240

    # (Optional, default: 20) Minimum number of packets in the window to log it
    packets_min: 15

    # (Optional, default: 40) Number of samples for entropy estimation per IP per window
    samples_size: 40

mitigator:
    # (Optional, default: 0) Number of packets to report after when running in offline mode
    report_status_packets: 1000000

    # (Optional, default: 1000000) Size of the denylist (blacklist) in entries
    denylist_size: 1000000

model:
    # From which library is the model loaded. Options: scikit|xgboost
    model_source: "scikit"

    # Which model to use from the particular source
    # Scikit options : adaboost|bayes|extra_trees|grad_boosting|kneighbors|logreg|lda|
    # nearest_centroid|svm|tree|random_forest
    # XGboost options: xgboost
    model_type: "random_forest"

    # Relative path to file containing model hyperparameters
    models_cfg_file: "models.yml"

    # (Optional, default: False). If True, does not raise exception when model config is not
    # found, but uses empty config instead. This is especially useful when config with the
    # specified name is generated on-the-go in the pipelined processing
    ignore_missing_config: True

    # (Optional, default: 'all') List of models to include with -C parameter.
```

```
    # Use 'all' for all available models or list of model_source.model_type strings, such as
    # ['scikit.kneighbors', scikit.lda, scikit.random_forest]
    # K Nearest Neighbors is disabled by default due to very poor time and memory performance
    # upon larger datasets
    comparison_models:
        - "scikit.adaboost"
        - "scikit.bayes"
        - "scikit.extra_trees"
        - "scikit.grad_boosting"
        - "scikit.logreg"
        - "scikit.lda"
        - "scikit.mlp"
        - "scikit.nearest_centroid"
        - "scikit.svm"
        - "scikit.tree"
        - "scikit.random_forest"
        - "xgboost.xgboost"

    # (Optional, default: 'accuracy') Metric to tune hyper-parameter against
    estimation_metric: "accuracy"

    # Hyperparameters to estimate during the hyperparameter tuning phase. Has to include
    # names of parameters relevant to the given model + their values to try as a list
estimation_params:
    n_estimators: [1,5,10,20,40,60,80,100]

    # (Optional, default: 'default') Metrics to print for cross-validation model comparison.
    # Takes parameters from:
    # https://scikit-learn.org/stable/modules/model_evaluation.html
    # Use list to list desired metrics. "fit_time", and "score_time" to print time required
    # for classifier training and data evaluation. "default" represents:
    # ['accuracy', 'f1', 'precision', 'recall', 'fit_time', 'score_time']
    score_metrics: "default"

    # (Optional, default: 'plots') Directory to save model evaluation plots to.
    plots_dir: "plots"

model_manager:
    # Which columns to standardize when generating configuration
    # "default" stands for all columns that are not within <0, 1> range already
    # Otherwise, column names may be specified in the list, "all" or "none"
    std_cols: "default"

    # (Optional, default: 0.2). Test data portion when splitting the data. Range <0.0, 1.0>
    test_size: 0.2

model_plotter:
    # Directory to which to save plots
    plots_dir: "plots/model"

    # Resulting plot files extension
    # Options: ps|eps|pdf|png|svg|jpg|jpeg|tif|tiff
    file_extension: "pdf"

    # Plotting for validation curve
    # Syntax:
    # validation_curve:
    # model_name:
    # arbitrary_name:
```

```
    # param_name:  - Name of the parameter to validate
    # param_range_low:  - Lower bound to start testing at
    # param_range_high: - Higher bound to stop testing at
    # param_type:  - Parameter type - float/int
    # param_samples:  - Samples num to draw from [param_range_low, param_range_high]
    # scoring:  - (Optional, default: 'accuracy') Scoring metric to use.
    validation_curve:
        scikit.random_forest:
        n_estimators:
            param_range_low: 10
            param_range_high: 100
            param_type: 'int'
            param_samples: 10
        max_depth:
            param_range_low: 1
            param_range_high: 10
            param_type: 'int'
            param_samples: 10

preprocessor:
    # Additionally dropped columns during mitigation additionally to cleaning config
    extra_drop_cols:
        - "src_ip"

resampling:
    # (Optional, default: "undersampling"). Method used if resampling is performed.
    # Choices: undersampling, oversampling
    method: "undersampling"

    # (Optional, default: "random"). Resampling algorithm used. Choices:
    # undersampling: random, nearmiss, tomeklinks, editednn
    # oversampling: random, smote, adasyn
    algorithm: "random"

standardization:
    # Standardization method to use
    method: "minmax"
```