BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

OPTIMIZATION OF DDOS ATTACK MITIGATION BASED ON MACHINE LEARNING

OPTIMALIZACE POTLAČENÍ DOS ÚTOKŮ S VYUŽITÍM STROJOVÉHO UČENÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **FILIP BANÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN KUČERA**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154949

Institut:	Department of Computer Systems (DCSY)
Student:	**Banák Filip**
Programme:	Information Technology
Title:	**Optimization of DDoS Attack Mitigation based on Machine Learning**
Category:	Networking
Academic year:	2023/24

Assignment:

1. Get acquainted with the problem of Denial of Service (DoS) attacks and ways to mitigate them using machine learning.
2. Study the available tools for measuring and analyzing network traffic in CESNET network infrastructure.
3. Design your method or optimizations of an existing approach to mitigate DoS attacks.
4. Experimentally implement the selected method using appropriate tools.
5. Perform experiments using an available data set and evaluate the achieved results.
6. Discuss the achieved results and the possibilities for further improvements.

Literature:

- According to the instructions of the supervisor and the consultant.

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:	Man Jakub, Ing. (CESNET)
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	13.2.2024

Abstract

DDoS attacks using the TCP protocol are still amongst the most common. This thesis aims to take advantage of information present in TCP SYN messages to improve DDoS attack detection success rate. TCP SYN fingerprints are proposed as an additional data source to consider when computing features for DDoS detection. A combination of an existing feature extraction and aggregation system with an existing autoencoder-based anomaly detector is significantly optimized and extended to make use of SYN fingerprints. The experimental results show decent DDoS detection improvements on relevant datasets. The detector is 16 and 95 times faster to train and execute respectively. The extraction and aggregation system is 23 times faster.

Abstrakt

Útoky DDoS, ktoré využívajú protokol TCP patria stále medzi tie najbežnejšie. Táto práca cieli zlepšiť úspešnosť detekcie DDoS útokov využitím informácií dostupných v TCP SYN správach. Odtlačky TCP SYN správ sú navrhnuté ako dodatočný zdroj dát pri počítaní charakteristík na vyhodnotie prítomnosti DDoS útoku. Kombinácia existujúceho systému na extrakciu a agregáciu charakteristík s existujúcim detektorom anomálií založeným na autokodéroch je zoptimalizovaná a rozšírená na využitie SYN odtlačkov. Experimentálne výsledky ukazujú obstojné zlepšenie detekcie DDoS útokov na relevantných dátových sadách. Detektor sa trénuje a testuje respektívne 16-krát a 95-krát rýchlejšie. Systém na extrakciu a agregáciu je 23-krát rýchlejší.

Keywords

DDoS, machine learning, autoencoder, KitNET, Windower, TCP SYN fingerprints

Kľúčové slová

DDoS, strojové učenie, autoenkodér, KitNET, Windower, TCP SYN odtlačky

Reference

BANÁK, Filip. *Optimization of DDoS Attack Mitigation based on Machine Learning.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Kučera

Rozšírený abstrakt

DDoS útoky využívajúce protokol HTTP tvorili 37 % všetkých DDoS útokov v sieti Cloudflare v prvom štvrťroku 2024. Väčšina HTTP premávky je stále založená na protokole TCP. Druhý najbežnejší typ DDoS útoku na sieťovej vrstve bol TCP SYN flood. [6]

Veľká časť všetkých DDoS útokov využíva protokol TCP a každý pokus o naviazanie TCP spojenia sa začína správou SYN. Správy SYN nesú špecifické informácie o operačnom systéme a aplikácii, z ktorej pochádzajú. Jedným z cieľov tejto práce bolo zhodnotiť informácie, ktoré tieto správy nesú, ako dodatočný zdroj dát pri počítaní charakteristík na vyhodnotenie prítomnosti DDoS útoku.

Odtlačkovanie TCP je metóda, ktorá extrahuje hodnoty určitých polí z hlavičiek protokolov IP a TCP, špecificky z TCP správ SYN a SYN+ACK. Kontrolu nad hodnotami týchto polí má operačný systém a aplikácia, ktorá TCP správu posiela. Na odtlačkoch sa teda odrážajú rozhodnutia programátora aplikácie, napríklad aký programovací jazyk použil, aké použil aplikáčné rozhranie operačného systému pre sieťovú komunikáciu alebo knižnicu ktorá nad nim ponúka abstrakciu, aké nezvyčajné nastavenia použil a ďalšie. Dopad, ktorý jednotlivé aplikácie, bežné alebo zákerné, majú na informácie prítomné v TCP správach sa v odtlačku môže prejaviť. Z týchto dôvodov majú odtlačky TCP potenciál rozlíšiť medzi bežnou a DDoS premávkou. Z pohľadu ochrany dôvernej siete pred DDoS útokmi sú postatné iba odtlačky TCP SYN správ.

Tento potenciál bol overený experimentom, v ktorom boli nazbierané a analyzované odtlačky SYN z rôznych bežných a zákerných aplikácií a z vybraných dátových sád. Ukázalo sa, že zákerné aplikácie často generujú odtlačky, ktoré sa od odtlačkov z bežných aplikácií významne líšia, i keď nie vždy. Datová sada so zachýtom skutočného TCP DDoS útoku ukázala, že útočník tiež generoval významne sa líšiace odtlačky v porovnaní s bežnými.

Poznatky z analýzy SYN odtlačkov z aplikácií a dátových sád viedli na návrh rozšírenia existujúceho systému na počítanie a agregáciu charakteristík, zvaného Windower. Windower počíta štatistické charakteristiky na vyhodnotenie nejakým modelom strojového učenia. Rožšírenie ktoré využíva odtlačky SYN bolo implementované. Informácie o odtlačkoch SYN sú transformované na dodatočné charakteristiky. Toto rozšírenie vedie na obstojné zlepšenie deteckie DDoS útokov na relevantných dátových sadách. Rozšírenie pomohlo detegovať útočníkov v prípadoch, v ktorých bez rozšírenia útočníci zostali zcela nedetegovaní. Prístup k integrácií SYN odtlačkov bol diskutovaný a boli navrhnuté ďalšie nápady a prístupy.

Ďalším cieľom tejto práce bolo lepšie prispôsobiť kombináciu systému Windower s modelom strojového učenia na detekciu anomálií založeného na autoenkodéroch, zvaného KitNET, pre detekciu DDoS útokov v reálnom čase. Detekcia DDoS útokov v reálnom čase predstavuje dve hlavné výzvy. Jednou je dostatočný výpočetný výkon systému, aby stíhal zpracovávať sieťovú premávku v reálnom čase. Druhá výzva spočíva v schopnosti systému sa prispôsobovať na zmeny charakteristík sieťovej premávky v čase.

Bola navrhnutá a implementovaná modifikácia modelu KitNET. Upravená verzia lepšie spĺňa požiadavky na detekciu DDoS útokov v reálnom čase a tvorí tak lepšiu kombináciu so systémom Windower. Implementácia bola zoptimalizovaná, čo viedlo na 16-násobné zrýchlenie trénovania a 95-násobné zrýchlenie testovania. Upravená verzia ale zachováva úspešnosť detekcie DDoS útokov pôvodnej verzie. Úprava umožnuje preskúmať možnosť model trénovať inkrementálne v nasadení a prispôsobovať sa tak meniacim sa charakteristikám premávky chránenej siete, pôvodná verzia tomuto nevyhovovala.

Podpriemerný výkon systému Windower bol zprofilovaný a významné súčasti ako komponent rozboru paketov a akumulátor štatistík boli zoptimalizované a preprogramované.

Toto viedlo na ďalšie významné zrýchlenie. Pôvodnej verzii trvalo 27 minút, aby spracovala určitú dátovú sadu, zatiaľ čo novej verzii to trvá iba 70 sekúnd, čo je 23-násobné zlepšenie.

Výsledky majú praktické uplatnenie a budú integrované do systému DDoS Protector, ktorý vyvíja združenie CESNET v rámci projektu bezpečnostného výzkumu Ministerstva vnútra Českej republiky.

Optimization of DDoS Attack Mitigation based on Machine Learning

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Jan Kučera. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Banák
April 25, 2024

Acknowledgements

The author wishes to thank Mr. Ing. Jan Kučera for his thoughtful and considerate attention to this work.

Contents

Chapter 1

Introduction

In the Cloudflare network, in the first quarter of 2024, HTTP DDoS attacks accounted for 37 % of all DDoS attacks. HTTP traffic is still largely based on the TCP protocol. The second most common network-layer DDoS attacks were TCP SYN floods. [6]

A big portion of all DDoS attacks rely on TCP and every TCP connection attempt starts with a SYN message. SYN messages carry specific information about the operating system and application they originated from. One of the goals of this thesis was to evaluate the information carried in these messages as another data source to extract evaluation features from, to improve DDoS detection. The impact individual applications, malicious or legitimate, have on the information carried in SYN messages can be represented in a SYN fingerprint that has the potential to discriminate between benign and DDoS traffic. To make use of SYN fingerprints, an addition to the Windower, an already existing feature extraction and aggregation system, was proposed and implemented. Windower computes statistical features for evaluation by some machine-learning model. SYN fingerprint data is transformed into additional features. The addition leads to positive improvements in DDoS detection on relevant datasets.

A modification of KitNET, an autoencoder based anomaly detection algorithm, was proposed and implemented. The modified version better fits the requirements of real-time DDoS detection and makes a better pairing for the Windower. The modified version shows no regressions in DDoS detection success-rate in combination with the Windower. The implementation was optimized, leading to 16 and 95 times faster training and execution times respectively. Major components of the Windower system were reprogrammed and optimized leading to further significant performance improvements. Whereas before it took 27 minutes to process a specific dataset, now it only takes 70 seconds, which is 23 times faster.

Chapter 2 touches briefly on DDoS attacks, the basics of machine-learning model training and evaluation, autoencoders, and various approaches to DDoS detection. It continues by describing the Windower and KitNET systems. Real-time DDoS detection requirements, the motivation for the KitNET modification and its evaluation, along with the various Windower optimizations and their evaluation can be found in Chapter 3. Chapter 4 introduces TCP fingerprinting and the reasons why it can help malicious hosts discrimination. It shows the effect common and malicious applications have on SYN fingerprints and observations made on selected datasets. Followed by the Windower modification, describing the new additional SYN fingerprint based features added to the system. DDoS detection success rate with the added features is evaluated and the approach is further discussed. Chapter 5 concludes this thesis by summarizing the proposed ideas and achieved results.

Chapter 2

ML-based DDoS Attack Detection

This chapter describes Distributed Denial of Service (DDoS) attacks [10] and the use of machine learning for detection of such attacks [36] [28] [2] [29] [1], followed by an introduction for a branch of machine learning models called neural networks, specifically autoencoders. A system for aggregation of statistical features of network traffic, called the Windower [15] is also introduced. Finally an anomaly detection model called KitNET [26] is discussed.

DDoS attacks can be detected by collecting statistics about network traffic, using the Windower, and feeding them into a machine learning model suited for discrimination of statistics computed from benign or anomalous network traffic (including DDoS traffic), such as the KitNET model.

2.1 Distributed Denial of Service attack

Parts of this section are adapted from [5].

Distributed Denial of Service (DDoS) attack [10] is a malicious attempt to disrupt the normal network traffic of a targeted server, service or network by overwhelming the target or the surrounding infrastructure with a flood of Internet traffic. The goal of DDoS attacks is to render the target unavailable to its intended users.

The attack is distributed because it utilizes many compromised computer systems — bots — as sources of attack traffic. The network of compromised systems is called a botnet. Computers in a botnet have been infected with malware, allowing them to be controlled remotely by an attacker.

In general, DDoS attacks can be categorized as:

- Volumetric attacks — this category aims to create congestion by consuming all available bandwidth between the target and the larger Internet. Large amounts of data are sent to a target by using a form of amplification or another means of creating massive traffic, such as requests from aforementioned botnets. For example, a DNS amplification attack [7]. In a DNS amplification attack, the attacker exploits functionality of public servers. Sending small requests to these servers specifically crafted such that those servers will reply not to the bot which sent the request, but to the target of the attack. These replies are much bigger in contrast to the small requests, amplifying the size of the malicious traffic headed for the target, congesting the connection links between.

- Protocol attacks — the Internet is built on common layers of technologies and communication protocols, such as TCP and UDP, which are built on the idea of openness. This gives attackers the same level ground as everybody else, because the operation of the protocol, as well as their possible vulnerabilities, can be gleaned easily from publicly accessible documents and then used in a DDoS attack. For example, a SYN flood attack [9]. This kind of attack is based on repeatedly initiating a connection handshake sequence but never finishing. For each such attempt, the target allocates resources from a limited pool. Not finishing this process leaves the allocated resources occupied waiting for messages that will never come. Thus blocking these resources for legitimate users. These identified vulnerabilities are sometimes accounted for using additions to the affected protocols such as, in this particular case, TCP SYN Cookies or TCP Reset Cookies [13].

- Application layer attacks — these attacks require more application-level knowledge but not necessarily in-depth knowledge. For example, if one understands the basics of the HTTP protocol POST, one can launch a low-and-slow POST operation by posting one out of thousands of characters at a time to an HTTP server before the session times out. Or one can perform an HTTP GET flood knowing that the server might not have enough resources to handle the burst of GET requests. While a single request is computationally cheap to execute on the bot side, it is usually much more expensive for the target to respond to. [8] Usually, the protocol-level attack is very obvious because it takes a lot more traffic to exhaust the network services, whereas the application-level attack requires a much lower volume of traffic and might be able to disguise itself until somebody familiar with the application is able to diagnose the problem.

2.2 Machine Learning

Machine-learning (ML) methods (or models) can be divided into two subsets, classical and deep-learning. Classical ML models include, for example, Random Forests, K-nearest neighbours, Support Vector Machine and Decision Tree models among others. On the other hand, deep-learning methods are based on neural networks with multiple layers, they include models such as the multi-layer perceptron and convolutional neural networks. [23] [17]

All machine-learning models need to be trained before they can be used for the desired task, in the case of DDoS detection, binary classification or anomaly detection. Machine-learning models are trained either in a supervised or an unsupervised manner [17].

Supervised learning makes use of input objects, or instances, and a desired output value to train a model. The training data is used to build a function to predict the label from the input data.

On the other hand, in unsupervised learning, the model is trained exclusively from unlabelled data. The training data is used to build a profile, or an image, of how that data looks, so that the trained model can tell if its input does or does not fit the learned profile.

To evaluate how a trained model performs, many different metrics are used. In the case of binary classification or anomaly detection, the result of classification of some number of instances can be expressed using four essential metrics, True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). Each input instance belongs in one of two classes, negative and positive (which, for example, may represent benign/malign, regular/anomalous, ...). If the model assigns the right class to the classified instance,

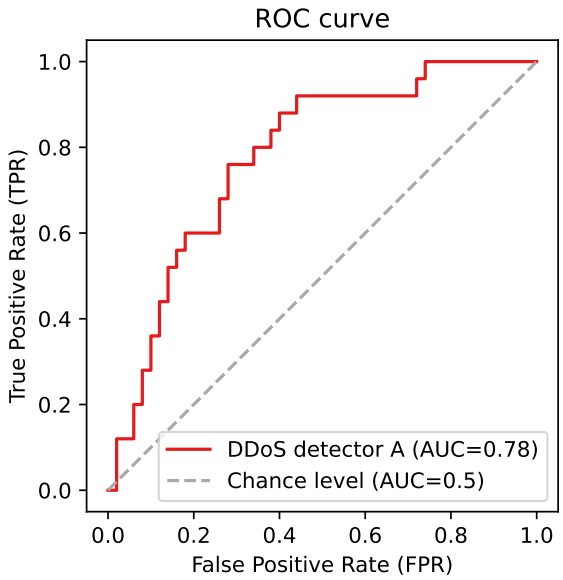

Figure 2.1: An example ROC curve.

that classification counts towards the TP or TN metric, depending on the actual class of the instance. In contrast, if an instance is assigned the wrong class by the model, that classification counts towards the FP or FN metric, depending on which class the model assigned. These four metrics are usually reported together in a table called the confusion matrix, which is structured like the following:

	Predicted positive	**Predicted negative**
Positive	TP	FN
Negative	FP	TN

Some models can assign a class to an instance based on comparison of model output to a threshold value. Varying the threshold value varies these essential metrics, thus to illustrate the performance of such a model at varying threshold values, the Receiver Operating Characteristic (ROC) curve is used. This curve is a plot of the True Positive Rate (TPR) against the False Positive Rate (FPR). The TPR representing the probability of detection, computed as $\frac{TP}{TP+FN}$, and the FPR representing the probability of a false alarm, computed as $\frac{FP}{FP+TN}$. An example ROC curve is depicted in Figure 2.1. Also present in the legend of the figure is a metric usually coupled with ROC curves, their Area Under the Curve (AUC). It ranges from 0 to 1, 1 meaning the classification was perfect. But both of these metrics paint only a part of the whole picture. Depending on the use case, other metrics need to be taken into consideration too.

This work is focused solely on models of the multi-layer perceptron (feedforward artificial neural network) type.

The following parts of this section were adapted from [17] and [37].

2.2.1 Artificial Neuron

Artificial neural networks, depicted in Figure 2.2, are composed from layers of artificial neurons, depicted in Figure 2.3. An artificial neuron is a mathematical function in-

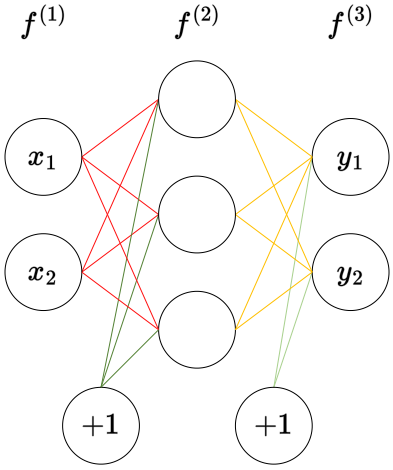

Figure 2.2: 3-layered artificial neural network. $f^{(1)}$ is the input layer. $f^{(2)}$ is a hidden layer with inputs x_1 to x_m and bias vector b_1. $f^{(3)}$ is the output layer with outputs y_1 to y_m and bias vector b_2.

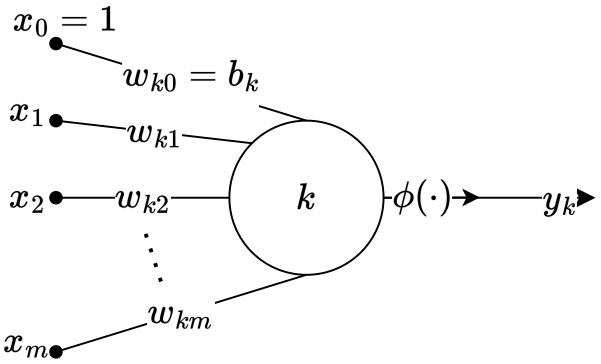

Figure 2.3: Artificial neuron k with m inputs x_1 to x_m, bias input b_k, and the activation function ϕ.

spired by the biological neuron. An artificial neuron takes in at least one input value. Each input value has a weight associated with it. Inputs are multiplied by their weight coefficients and their summation with a bias input is fed into a non-linear function called the *activation function*. Activation functions usually take on the shape of a sigmoid or a step. For example the standard logistic function graphed in Figure 2.4.

Neuron k in some layer l has $m+1$ inputs valued x_0 to x_m and weights w_{k0} to w_{km}. x_0 has a fixed value of 1, used together with weight w_{k0} to form the bias input $b_k = x_0 w_{k0}$, which is part of the layer bias vector $\vec{b}_l$, since each neuron in layer l creates one value of this vector. Inputs x_1 to x_m are outputs of the previous layer. The output of the neuron k is given by the function

$$y_k = \phi(\sum_{j=1}^{m} w_{kj} x_j + b_k),$$

where ϕ is the activation function.

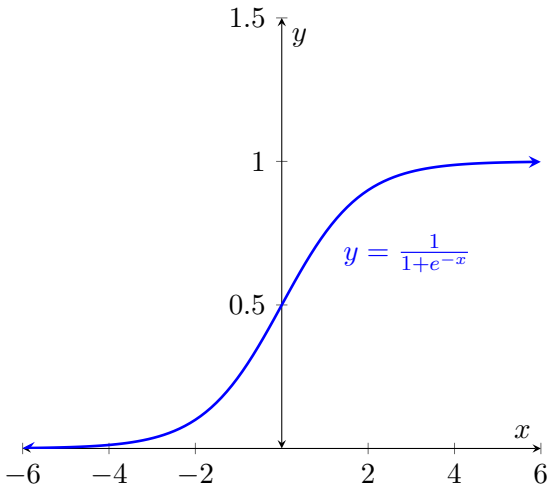

Figure 2.4: An example of an activation function — the standard logistic function.

2.2.2 Feedforward Neural Network

Feedforward neural network, depicted in Figure 2.2, is a deep-learning model. The goal of a feedforward network is to approximate some function f^*. A feedforward network defines a mapping $y = f(x, \theta)$ and learns the value of the parameters θ that result in the best function approximation. The parameters are the weights of each neuron, or in other words the pair of input weights and bias vector of each layer.

The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. During neural network training, we drive $f(x)$ to match $f^*(x)$.

The first layer is called the input layer, the last layer the output layer, and the layers between are called hidden layers. Each hidden layer of the network is typically vector valued — consisting of *units* or *neurons*. The units of the input layer just take on the value of the network input without computation, so $f^{(1)}(x) = x$.

2.2.3 Autoencoder

An autoencoder, depicted in Figure 2.5, is a feedforward neural network that is trained, in an unsupervised manner, to attempt to copy its input vector $\vec{x} \in \langle 0, 1 \rangle^d$ to its output vector $\vec{z} \in \langle 0, 1 \rangle^d$. Internally, it has a hidden layer that describes a code used to represent the input as $\vec{y} \in \langle 0, 1 \rangle^{d'}$. The network may be viewed as consisting of two parts. First, an encoder, a deterministic mapping $\vec{y} = f_\theta(\vec{x}) = s(W\vec{x} + \vec{b})$ parametrized by $\theta = \{W, \vec{b}\}$, where $s(\mathbf{x}) = (s(x_1), \ldots, s(x_d))^T$ and $s(x) = \frac{1}{1+e^{-x}}$ is the standard logistic function (see Figure 2.4). W is a $d' \times d$ weight matrix and $\vec{b}$ is a bias vector. Second, a decoder, which takes the resulting latent representation $\vec{y}$ and maps it back to a "reconstructed" vector $\vec{z} \in \langle 0, 1 \rangle^d$ in input space $\vec{z} = g_{\theta'}(\vec{y}) = s(W'\vec{y} + \vec{b}')$ with $\theta' = \{W', \vec{b}'\}$. The weight matrix W' of the reverse mapping may optionally be constrained by $W' = W^T$, in which case the autoencoder is said to have *tied weights*.

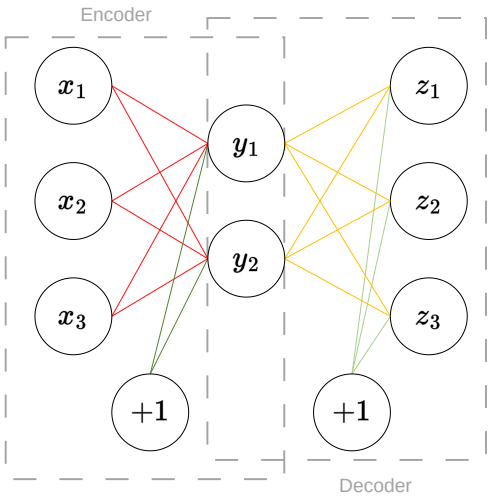

Figure 2.5: An undercomplete autoencoder with a single hidden layer.

If an autoencoder succeeds in simply learning to set $g_{\theta'}(f_\theta(\vec{x})) = \vec{x}$ everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually, they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data. Autoencoders may be thought of as being a special case of feedforward networks and may be trained with all the same techniques.

The idea being that training the autoencoder to perform the input copying task will result in $\vec{y}$ taking on useful properties. One way to obtain useful features from the autoencoder is to constrain $\vec{y}$ to have a smaller dimension than $\vec{x}$. An autoencoder whose code dimension is less than the input dimension is called *undercomplete*. Let $\beta \in (0, 1\rangle$ denote the hidden layer compression ratio. For example, if the dimension of $\vec{x}$ is $d = 3$ and the compression ratio $\beta = 0.65$ then the dimension of $\vec{y}$ is $d' = \lceil \beta d \rceil = 2$, this example is depicted in Figure 2.5.

Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data. The learning process is described simply as minimizing a loss function:

$$L(\vec{x}, \vec{z}) = L(\vec{x}, g_{\theta'}(f_\theta(\vec{x})))$$

Where L is a loss function penalizing $g_{\theta'}(f_\theta(\vec{x}))$ for being dissimilar from x (*reconstruction error*) such as the traditional *squared error*:

$$L(\vec{x}, \vec{z}) = \|\vec{x} - \vec{z}\|_2^2$$

An alternative loss, suggested by the interpretation of $\vec{x}$ and $\vec{z}$ as either bit vectors or vectors of bit probabilities (Bernoullis) is the reconstruction cross-entropy:

$$L(\vec{x}, \vec{z}) = -\sum_{k=1}^{d} \left[x_k \log z_k + (1 - x_k) \log(1 - z_k) \right]$$

This optimization will typically be carried out by stochastic gradient descent [17].

2.3 DDoS Attack Detection

Nowadays, there are generally two types of DDoS detection systems — signature-based and anomaly based. [36] [1]

There are also other ways to mitigate DDoS attacks, for example, the TCP SYN flood attack can be mitigated using the TCP Reset Cookies method [13].

Signature-based DDoS detection systems rely on a database of DDoS signatures. For example, one of those signatures might be a rule stating that the TCP source port is equal to the TCP sequence number. Malicious traffic is detected by matching these signatures. The biggest downside of such systems is the need to keep the signature database up-to-date. That means gathering DDoS attack information and searching for patterns in attacks that may or may not exist. The second disadvantage is the specificity of signatures. They do not generalize to unknown attacks. On the other hand, they may ease mitigation of attacks, since instead of using rules to mitigate traffic from hundreds of IP addresses, one mitigation rule derived from the signature is enough to mitigate the whole attack.

Anomaly-based systems usually make use of machine-learning models for DDoS detection and have been extensively studied, showing good results [28] [29] [2]. Anomaly-based systems use the statistics of network traffic such as packet header information, packet size, and packet rates to detect anomalies in the network, DDoS attacks are such anomalies. Although anomaly-based detection mechanisms do not require databases of anomalies to compare, they have difficulty in detecting attacks similar to normal traffic, such as low-rate DDoS attacks.

The advantage of at least some of these systems which take advantage of machine-learning, is the possibility to detect attacks they have not been trained on (including unknown attacks). Some systems do not even require malicious traffic for training, just legitimate traffic, further generalizing attack detection. It is thus possible to detect unknown attacks, but with no guarantees.

A major disadvantage of machine-learning systems, specifically those which are based on deep-learning models, are their computational requirements. The computations in these systems are expensive and can limit their use for DDoS detection, especially real-time detection. But these computations can be executed on graphical processing units (GPUs) much more efficiently than on central processing units (CPUs).

Data for training and testing is absolutely necessary. This data usually comes from public network intrusion detection datasets that include DDoS attacks. Some of these datasets capture real network intrusion situations which occurred on some real network, others were synthesized in laboratory-like environments. But in both cases, raw captures of network traffic are provided along with information and labels to distinguish between benign and malicious traffic in the network capture files.

In supervised learning, the traffic is processed into model input objects, which would be vectors of values extracted from the captured packets, or statistical data computed on these values, along with a label stating that the data comes from benign or malicious traffic.

In unsupervised learning, the input objects would be the same vectors but always processed from packets originating from either only benign or only malicious traffic. The traffic class used for unsupervised learning is commonly the benign traffic class. The idea being that, usually, benign traffic has different characteristics from DDoS attacks. In addition, one can make an assumption that benign traffic does not differ drastically from other benign traffic — although that may not always be the case, if one is trying to protect a network

from DDoS attacks, it is safe to assume one knows what the normal traffic characteristics of that network look like and could provide benign data for training. Also, if the characteristics of benign traffic do change over time, the model can be re-trained, or in some cases even trained incrementally, on the new benign traffic. But it is difficult to make any such assumptions about the DDoS traffic, and availability of DDoS traffic data is low. For these reasons, the benign traffic class is, in general, preferred over the malicious class for unsupervised learning.

2.4 Windower: Traffic Feature Aggregator

Windower [15] [14] is a feature extraction and aggregation method for real-time network-based intrusion (particularly DDoS) detection. It employs stream data mining and sliding window principles to compute statistical information directly from network packets. It summarizes several such windows and computes inter-window statistics to increase detection reliability. Summarized statistics are fed into an ML-based attack discriminator. If an attack is recognized, the traffic from the attacking source can be mitigated using simple access control list rules.

Windower extracts the following information from IP packets:

- Time stamp — for computation of temporal statistics,
- Source and destination IP address — for identification of communicating hosts,
- Transport layer protocol,
- Source and destination port,
- Size of headers,
- Size of useful data.

Temporal statistics are computed from this information using stream data mining and sliding window principles. Temporal statistics describe the continuous network traffic in time intervals — windows. The traffic is divided into a sequence of windows of fixed size w — the sliding window technique. See Figure 2.6 for an illustration. In each window, statistics are aggregated independently for each unique source IP address that happens to communicate in the window. Each observed IP address has a set of windows associated with it, which have statistics computed from packets originating from this source address only. So if in one time window, enough packets from two different source IP addresses are observed, then two windows with temporal statistics will be created, one for each address. Effectively, as if the traffic was separated to flows based on the source IP addresses and the sliding window was applied on each of these traffic flows independently, but at the same time interval boundaries.

Each packet belongs only to a single window. Each window computes statistics on packets that are observed in the time interval $\langle wn, w(n+1)\rangle$, where n is the window identifier starting at 0. For each address, multiple windows are stored to enable deeper communication analysis.

Statistics from a single window are not informative enough to act on. For example, in the case of fast bursts of legitimate traffic, which would produce a significant amount of packets in a short time, the window containing these legitimate but short bursts could

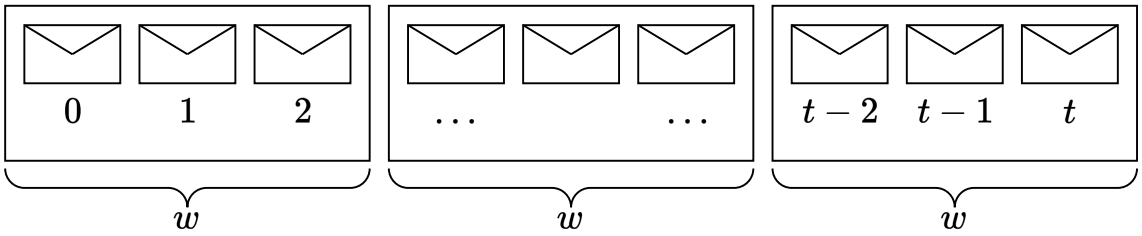

Figure 2.6: Network traffic packets divided into time intervals using the sliding window technique.

have statistic values similar to a window with a DDoS attack in the same short time. The difference being that, the DDoS attack would continue through the next windows while the legitimate bursts would not.

These windows include statistics such as timestamps, counts, minimum, maximum, mean, standard deviation, entropy values and more. Each is represented by a single value and updated incrementally with each observed packet. If an address has accumulated enough windows, all these windows are summarized into a single vector of statistics (or features). The summary and inter-window statistics included in the feature vector are listed in the following table (avg(s) — average(s), std(s) — standard deviation(s)):

1:	avg number of packets
2:	avg number of bytes
3:	approximate packets per second rate — pps
4:	approximate bytes per second rate — bps
5:	avg of avg times between observed packets
6:	avg std of times between observed packets
7:	minimum packet size
8:	maximum packet size
9:	avg of packet size averages
10:	avg std of packet sizes
11:	avg TCP traffic ratio
12:	avg UDP traffic ratio
13:	avg ICMP traffic ratio
14:	avg count of unique source ports
15:	avg entropy of source ports
16:	avg of avg amounts of packets from socket-to-socket transmissions
17:	avg fragmented packet ratio
18:	avg of header to whole packet ratio avgs
19:	std of total amounts of packets
20:	std of total amounts of bytes
21:	std of packet size avgs
22:	std of stds of packet sizes
23:	std of avg times between observed packets
24:	std of amounts of unique source ports
25:	std of source port entropy
26:	std of avg amounts of packets from socket-to-socket transmissions
27:	std of fragmented packet ratios
28:	std of avgs of header to whole packet ratio
29:	std of ratios of the dominant transport layer protocol
30:	approximate host activity inside the windows
31:	approximate window activity across the window ID interval

Socket-to-socket communication is identified by the tuple (source IP address, source port, destination IP address, destination port). The idea is that the DDoS attacks should cause different enough patterns in the statistics than legitimate traffic to be recognized. For statistics of a window to be useful, the window must observe at least some minimum number of packets, otherwise the window is not added to the set of windows of the corresponding source IP address. Similarly, the feature vector needs to be computed from some minimum amount of windows to be useful. The final feature vector computed from several windows associated with a single source IP address is then fed into a machine-learning model to discriminate the source IP address as either benign or malign, or provide some number as a measure as to how much the model "thinks" the IP address is malicious. The measure can be compared to a threshold to make the final decision. One machine-learning model suited for this task is for example the KitNET anomaly detector, see Section 2.5.

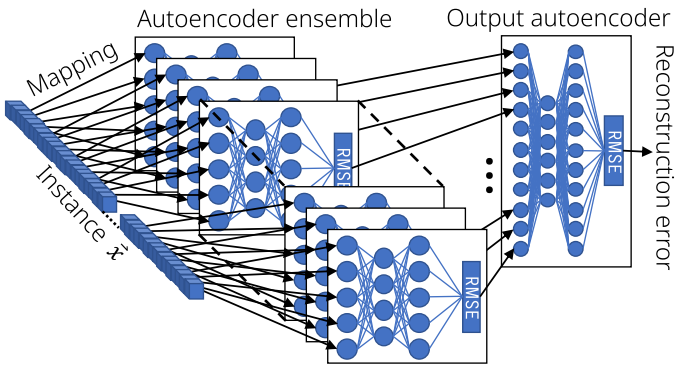

Figure 2.7: Trained KitNET (modified image adopted from [26]).

2.5 KitNET Anomaly Detector

KitNET is a generic algorithm for real-time anomaly detection based on autoencoders. KitNET is a part of the network intrusion detection system (NIDS) Kitsune [26].

2.5.1 Architecture

KitNET can be decomposed into two components, the feature mapper and the anomaly detector. The whole architecture of a trained KitNET model is depicted in Figure 2.7.

- Feature mapper (FM) — this component is responsible for decomposition of the input instance $\vec{x}$ into k smaller instances (sub-instances, denoted as $\mathbf{v}$) with at most m features. This component needs to learn a mapping from $\vec{x}$ to $\mathbf{v}$.
- Anomaly detector (AD) — responsible for processing $\mathbf{v}$ into the reconstruction error and to decide the occurence of an anomaly based on this reconstruction error. The anomaly detector is further decomposed into an ensemble of k autoencoders responsible for processing the sub-instances $\mathbf{v}$, and the output autoencoder processing their output into a final reconstruction error.

KitNET is parametrized by the single parameter m, maximum number of features in a sub-instance, which also translates to the maximum width (number of units) of input layers of autoencoders in the ensemble.

The reconstruction error of the autoencoders in KitNET is computed as the root of the mean of the squared errors — the root mean squared error (RMSE) — between the input instance $\vec{x} \in \mathbb{R}^n$ and the autoencoder reconstructed instance $\vec{z} \in \mathbb{R}^n$, where n is the input instance vector dimensionality. The input instance $\vec{x}$ can be, for example, an instance of a feature vector produced by the Windower system (see Section 2.4).

$$\mathrm{RMSE}(\vec{x}, \vec{z}) = \sqrt{\frac{\sum_{i=1}^{n} (x_i - z_i)^2}{n}}$$

2.5.2 Operation

Anomaly detection has two phases. The flow of KitNET phases is depicted in Figure 2.8. The training phase, in which KitNET is trained on instances computed from benign data.

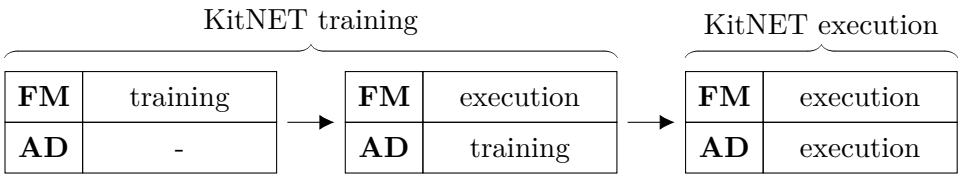

Figure 2.8: Flow of KitNET phases, including phases of its components.

The execution phase, in which KitNET detects anomalies based on the value of the reconstruction error. An anomaly is detected in the execution phase, if the reconstruction error crosses a threshold. For example, one of the ways to choose the value of the threshold, is the maximum reconstruction error from the training phase.

Both the feature mapper and the anomaly detector have two phases of operation. Before the KitNET model transitions into the execution phase and before it is able to compute the reconstruction error to detect anomalies, the training phase needs to finish. In the training phase, the feature mapper and the anomaly detector need to be trained, in this order, on instances originating from benign data.

The feature mapper needs to be trained first, because the architecture of the autoencoder ensemble depends on the mapping $\mathbf{v}$ that the feature mapper learns. This component finds the mapping by incrementally clustering the features of the input instance domain using agglomerative hierarchical clustering, for details refer to the Kitsune publication [26]. The autoencoder ensemble is constructed on the transition of the feature mapper component from the training phase into the execution phase. An autoencoder is created for each sub-group of the mapping the feature mapper learned, thus creating an ensemble of k autoencoders. Each sub-group of features contains at most m features, thus the maximum width of an input layer of each autoencoder in the ensemble is m. Each sub-instance is always mapped to a single autoencoder, always the same one. The output autoencoder is created with its input layer width being k, the number of autoencoders in the ensemble. In total, there are $k + 1$ autoencoders.

Then, the anomaly detector can be trained. Autoencoders in the ensemble are trained to produce a minimal reconstruction error from the sub-instances, and the reconstruction errors of all ensemble autoencoders are forwarded to the output autoencoder, which is also trained to minimize its reconstruction error.

After the anomaly detector is trained, the KitNET model transitions into the execution phase. In this phase, the feature mapper uses the learned mapping to create sub-instances from the input instance and forwards them to the autoencoder ensemble in the anomaly detector. The ensemble processes the sub-instances into reconstruction errors and forwards those to the output autoencoder, which computes the final reconstruction error, used for anomaly detection.

Chapter 3

Real-time DDoS Detection

This chapter discusses Distributed Denial of Service (DDoS) attack detection in a real-time setting, and what challenges are associated with this type of detection using the Windower feature extraction system, described in Section 2.4, together with the KitNET anomaly detection model, described in Section 2.5.

A modification of the KitNET model is proposed, that better fits the nature of real-time DDoS detection and incremental learning. The modification was optimized and implemented, the implementation was analysed from the computation and detection success rate perspectives.

Further, the Windower implementation was profiled and optimized resulting in significant processing speed improvements.

3.1 Real-time Detection Requirements

Detecting DDoS attacks in real-time, means having deployed a detection system that is processing the incoming traffic of the protected network in real-time, this poses some challenges. The system needs to be performant enough to keep up with the network traffic, and make use of some method to continuously adapt to the possibly changing network characteristics, such as incremental learning for example. This section discusses these challenges.

3.1.1 Computation Performance

One of the issues of such systems is the computation performance requirement. The system must be efficient enough to process all this traffic without getting overwhelmed. Especially if the input unit of the data for the system are packets, as is the case for the Windower. If the network traffic is too overwhelming for the system, one of the ways to compensate for this is to employ packet sampling. The system will only process one out of n packets, but this may degrade detection results.

Some systems use network flow data as an input unit instead. An instance of flow data contains statistical information about packets aggregated based on the five-tuple (source IP address, source port, destination IP address, destination port, transport layer protocol). Flow data is collected by some other system and sent to the DDoS detector. This relaxes the computational performance requirements in contrast to processing each packet. The disadvantage being that the detection system does not have access to all the information it would have, if it was processing packets, and the system has to be designed to work specifically with this type of data.

3.1.2 Incremental Learning

Another problem of real-time detection is the temporal discrepancy between the data the machine-learning model was trained on in contrast to the data which it is being executed on. The model is trained on up-to-date data, but in deployment, the characteristics of the traffic may change enough from what the characteristics of the training traffic looked like, to render the system results incorrect. The model is said to be invalidated due to *concept drift*.

For example, requests from users may increase in number or size, or a new service may be introduced into the network that has an effect on the traffic characteristics, and so on. This issue can be avoided by training the model on up-to-date data from scratch again whenever one knows the characteristics changed or even periodically, or training the model incrementally throughout deployment. Incremental learning is an attractive option. It is possible to build upon what the model had already learned and shape this knowledge throughout deployment to be always up-to-date.

3.2 KitNET Optimization

The Windower was designed with real-time detection in mind and the only factor that has an influence on this type of deployment is the implementation performance. The same holds true for the KitNET model. But the incremental learning technique is not completely compatible with KitNET's architecture. A modification to the architecture is proposed and implemented, called SimpleKitNET, in which this incompatibility is addressed. The modification has some implications on computation performance, which are evaluated with a set of implementations. Further, DDoS detection success rate of SimpleKitNET is contrasted to that of the original KitNET.

The anomaly detector component of the KitNET system (see Section 2.5) is suited for incremental learning. The anomaly detector can be trained incrementally at any point (between executions) in the KitNET execution phase. The anomaly detector architecture depends on the mapping produced by the feature mapper component in the KitNET training phase. If the system is to be trained incrementally, because of the possible network characteristic changes, it would not make sense to use the feature map that was learned from the training data with old network characteristics. Even if the feature map could be rebuilt at any point, which poses its own challenges, a change in the feature map would mean a rebuild of at least one autoencoder in the anomaly detector component. That would result in an abrupt change in the output of the replaced autoencoders and an inconsistent anomaly detector state, where some autoencoders have been trained to the same extent, while others are completely untrained, which could render the detection results false.

Instead of rebuilding the feature map, the feature mapper component can be removed completely. This change would mean that instead of k (the number of sub-instances created from the input instance using the feature map) autoencoders in the ensemble layer forwarding their reconstruction errors to one output autoencoder with input layer width k, the whole anomaly detector component would be composed of a single bigger autoencoder with input layer width n — the dimensionality of the feature vector input instance. Thus the KitNET training and execution phases would correspond to the training and execution phases of the anomaly detector, now being the only KitNET component, a single autoencoder.

3.2.1 Computation Cost Impact

The reason for the inclusion of the feature mapper component in the first place was to decrease computational complexity. Let $\beta \in (0, 1\rangle$ denote the autoencoder hidden layer compression ratio (Section 2.2.3), with n being the input instance dimensionality. The complexity of executing a single three-layer undercomplete autoencoder is

$$O(n \cdot \beta n + \beta n \cdot n) = O(n^2).$$

The encoder computes the dot product between an n-dimensional vector and an $n \times \beta n$ matrix. The reverse holds true for the decoder. The complexity of executing the unmodified KitNET architecture is

$$O(km^2 + k^2) = O(k^2).$$

There are k autoencoders in the ensemble layer with their input layer width being at most m, and a single output autoencoder with its layer width being k. Since m is a constant parameter, the complexity of the ensemble layer scales linearly with n — the more features the input instance has, the more autoencoders will be created in the ensemble to process more groups in the feature map. The complexity of the output layer depends on how many groups are in the feature map. In the best case scenario $k = \frac{n}{m}$.

Trade-offs

KitNET was designed to run in resource constrained environments, such as a Raspberry Pi machine. While the Windower was designed to be, at least, as memory efficient as possible, constrained environments were not the primary target. While not in the scope or focus of this work, with the vision of exploring real-time, or online, incremental learning in the future, as an attractive combination of the Windower and SimpleKitNET systems, the trade-off between computation cost and the possibility to learn incrementally was made.

The modified architecture is denoted as SimpleKitNET, or the simple version, while the unmodified is denoted as KitNET, or the original version.

3.2.2 Implementation Performance

To find out how much does the simple version affect computation performance, benchmarks were performed. Input data used for benchmarking were feature vectors produced by the feature extraction framework in Kitsune [26], specifically from Mirai botnet traffic, available online as the example dataset for KitNET at [25].

To improve the performance of the modification, a few versions were implemented. All version are implemented in the Python3 programming language, including the original KitNET. One version is implemented with the NumPy library [18], other version uses just-in-time compilation possible by applying the Numba library [21] on the NumPy version, and one more version using the Keras framework [4] (running with the JAX backend [3]), which also performs just-in-time compilation.

The implementation performance was measured separately for the training phase and the execution phase. The dataset consists of 100000 instances, of which 55000 were used for training and 45000 for execution, each instance consisting of 115 features. Each phase ran 10 times, and the best result of all runs was chosen as the used metric. All implementations ran on CPU only, no GPU acceleration, no parallelization was explicitly requested either, but some NumPy and JAX operations can decide by themselves to run on multiple CPU cores. The original KitNET version was parametrized with the default value

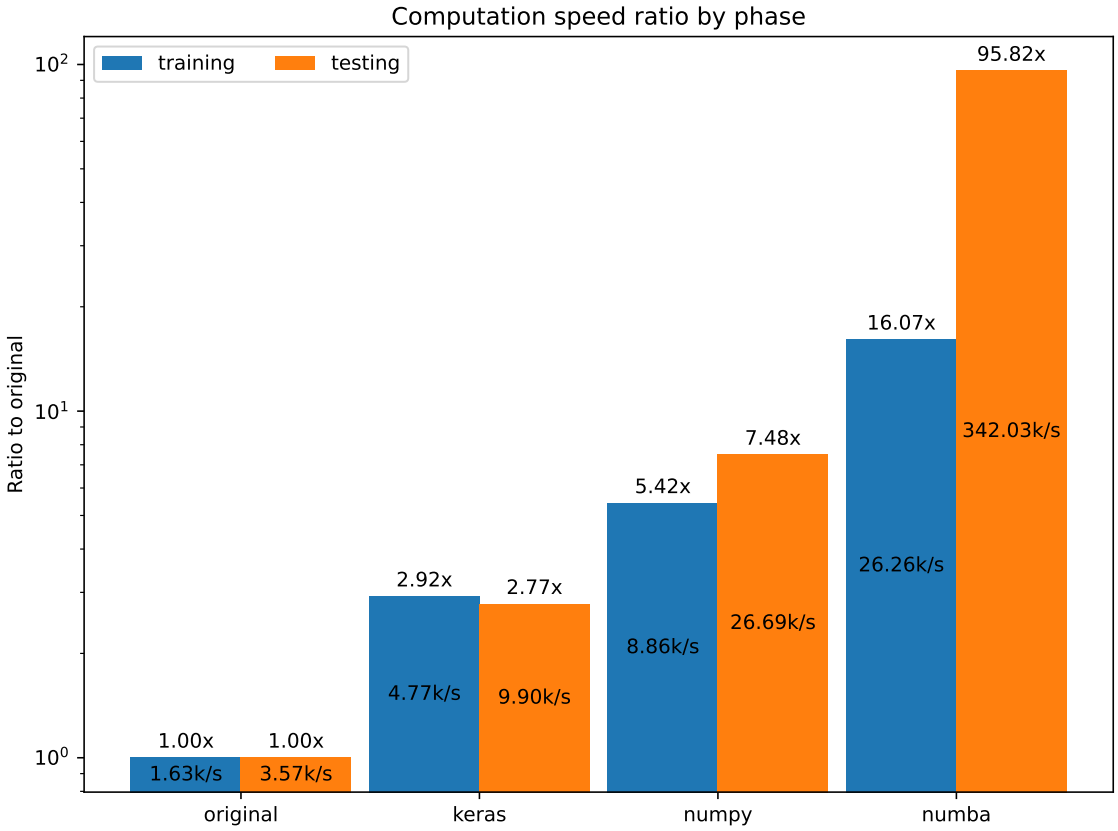

Figure 3.1: SimpleKitNet implementation speed benchmark results by model phase. Measured in absolute speed of processed kilo-instances per second, and its ratio to the original KitNET. Includes the original KitNET model ($m = 10$), and three SimpleKitNET model implementations. All SimpleKitNET implementations are faster than the original in both phases. Feature vectors were of dimensionality 115. The hidden layer compression ratio was $\beta = 0.75$.

of $m = 10$. All versions had the value of the hidden layer compression ratio parameter $\beta = 0.75$. The results are depicted in Figure 3.1 and summarized in the following table, where *k/s* stands for kilo-instances per second.

Model	Training Speed	Ratio to Original	Execution Speed	Ratio to Original
Original	1.63 k/s	1.00 x	3.57 k/s	1.00 x
Keras	4.77 k/s	2.92 x	9.90 k/s	2.77 x
NumPy	8.86 k/s	5.42 x	26.69 k/s	7.48 x
Numba	26.26 k/s	16.07 x	342.03 k/s	95.82 x

Interestingly enough, even though at least the NumPy SimpleKitNET version should perform worse than the original NumPy KitNET, the results show the exact opposite. In the training phase, all implementations of the simple version have shown to be more than twice as performant. The Numba implementation being 16 times faster. In the execution phase, again, all simple version implementations were at least two times faster. The Numba

implementation was 95 times faster. Overall, the Numba implementation performed better than the original unmodified KitNET and all other SimpleKitNET implementations.

Some of the performance difference between the Keras and Numba implementations, although both are just-in-time compiled, can be explained by the fact that the Keras implementation uses automatic differentiation for loss function minimization, while the Numba implementation optimizes using pre-derived equations. The NumPy SimpleKitNET and KitNET also use pre-derived equations.

The high computation cost of the original KitNET version in contrast to the NumPy SimpleKitNET version in practice, even though it is faster in theory, is likely due to foreign function interface (FFI) overhead caused by the implementation libraries. The original version is faster from the pure computational perspective, but in practice leads to more FFI overhead. If both versions were to be implemented completely in a compiled language such as C or C++, the original KitNET version should be faster than the modification, this fact is supported by the results in [26].

3.2.3 DDoS Detection Success Rate

To verify that the modified version does not cause DDoS detection success rate degradation, experiments on four datasets were conducted, to compare it against the original version. All three implementations of the simple version were used for testing, adding to the credibility of the results (if all three perform within the margin of error relatively to each other). The datasets used for testing were the same as in [15], prepared according to reproduction instructions in the accompanying repository [16].

Results are summarized as Receiver Operating Characteristic (ROC) curves (described in Section 2.2), depicted in Figure 3.2. The ROC curves show no regression in detection results, when comparing SimpleKitNET to the original KitNET, because both models produced the exact same classification results, within the margin of error, at varying thresholds. In addition, all SimpleKitNET implementations show the same results, as expected. Proving, the modification to the original model, does not degrade DDoS detection success rate in combination with the Windower.

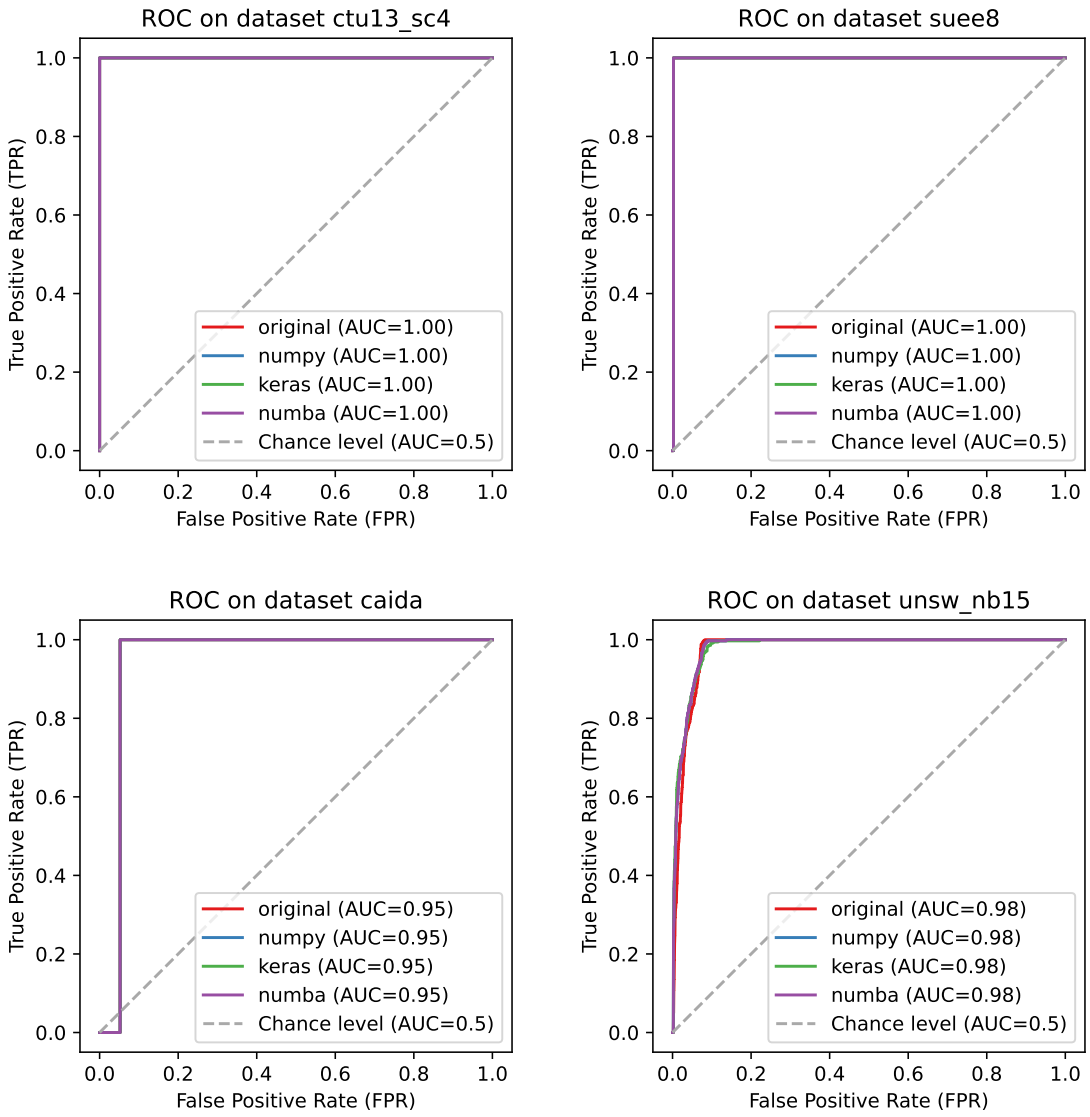

Figure 3.2: ROC curves (Section 2.2) of DDoS detection results from different datasets. Detection performed on Windower feature vectors using the original KitNET version ($m = 10$), and all SimpleKitNET implementations. The hidden layer compression ratio was $\beta = 0.75$. The SimpleKitNET model shows results identical to the original version, no regressions.

3.3 Windower Optimization

Windower was programmed in the Python3 programming language. A sensible choice from the perspective of data analysis and quick iteration times on modifications, possible due to the incredible data-science libraries available in the language ecosystem.

But experimenting with additions to the Windower (Section 4.5) showed that it is very slow. On a modern machine, processing a 950 MB traffic capture file with 5.5 million packets into final feature vectors for evaluation took 28 minutes, processing 3.26 kpkts/s on average. Even when taking into account that Python is a dynamic interpreted language, and the used, most popular of its interpreters, CPython, does not have support for just-in-time compilation, the processing speed is still very low.

Windower can be decomposed into few major components:

- Packet parser — extracts all needed values from raw packet data,
- Windowing controller — provides the Windower API, handles IP address tracking and the traffic sliding-window,
- Statistical logger — continuously updates statistics in tracked windows with every packet, computes features in the final vectors from saved windows.

The most glaring computation performance issues in the Windower were profiled and optimized, enhancing the performance by a large margin. Specifically the packet parser and the statistical logger were reprogrammed.

3.3.1 Packet Parser

Windower input is in the form of raw captured network traffic — packets. To compute statistics from values contained in the layered protocol structure of these packets, those values need to be extracted by parsing, or dissection.

To achieve this, Windower made use of the Scapy library. Scapy provides a fully-featured packet dissector that extracts all header fields from the raw packet data, for easy access. While full-featured, the library was not created for high-speed network processing and was programmed completely in Python. Profiling revealed that Windower spends around 50 % of the time just parsing packets.

To speed-up the dissection, a new parser was written from scratch in the C programming language and integrated into the Windower as a Python native extension. This new parser will be referred to as *CParser*. While CParser does not support real-time traffic capture, but only reads traffic capture files, it could be easily extended to do so, because it makes use of the libpcap library. For the purposes of this work, this suffices, as only datasets in the form of captures were being used.

CParser supports common link-layer protocols used in capture files, specifically Ethernet, Raw IP, and Linux Cooked Mode Capture version 1 [35]. Further, as the Windower only works with IP packets, the supported network-layer protocols are IP versions 4 and 6. TCP, UDP, and SCTP protocols are supported on the transport-layer. Support for ICMP versions 4 and 6 is also present. If the protocol above IP is not supported, it does not mean that the parser can not continue, only that it will be reflected in some of the values computed for the Windower. For example port numbers will be missing or the header length will be included in the total payload size instead of the total header size.

Since part of this work focuses on processing TCP SYN fingerprints into features to enhance the Windower DDoS detection capability, see Chapter 4, CParser also extracts and provides all necessary information included in these fingerprints.

All of the values from raw packets needed by the Windower are provided to it in a simple record of values for each packet. To efficiently interface with the CParser from Python, it needs to be compiled into a Python native extension. There are many ways to achieve this, the most simple by using a specialized library. There are many of these, but surface-level analysis has shown they produce modules with effectively the same Foreign Function Interface (FFI) overhead (the overhead necessary to call C code from Python, for example, converting Python objects into C compatible ones and vice-versa). The utilized library that best fit the specific use case of CParser was CFFI [31].

Implementation performance of CParser was evaluated. The scenario consists of parsing each packet in a capture file one-by-one, nothing more. The results are depicted in Figure 3.3 and summarized in the following table, where *kpkts/s* stands for kilo-packets per second.

Parser	Processing Speed	Ratio to Scapy
Scapy	4.73 kpkts/s	1.00 x
CParser	258.06 kpkts/s	54.61 x

Even with the FFI overhead, CParser is 54 times faster than Scapy. The results can vary, depending on the particular capture file used. From own experimentation, CParser is anywhere between 48 to 62 times faster than Scapy.

3.3.2 Statistical Logger

After the parser extracts the needed values for statistics computation, they are passed to the statistical logger. The logger keeps track of statistics of the current window for each observed source IP address and updates them with each passed instance of values extracted from a raw packet. When the windowing controller signals to end the window, the backlog of finished windows of each observed source IP address is expanded with the statistics from the current window. At any time the windowing controller can request to export final feature vectors, then more statistic are computed from the backlogs of windows.

Window data was represented and the statistics computed using data structures and algorithms provided by the NumPy library [18]. While NumPy provides many algorithms conveniently packed in a powerful API, these procedures were optimized for vectorized processing of many values stored in big multi-dimensional arrays. Because of the way the library is designed and implemented, calls to NumPy incur Foreign Function Interface (FFI) overhead, because many of its procedures implemented in compiled languages like C and Fortran. The computation penalties of FFI are usually negligible when compared to the computation time savings they allow by interfacing with the fast compiled code, but only when they process large arrays where the vectorization makes sense.

Most logger operations work with scalars or short lists of scalars, but many operations need to be executed with each incoming packet. Thus, for each packet the FFI overhead was incurred many times over, but due to the small size of the data, for no real benefit. Profiling showed that around 50 % of the time was spent in the logger (the remaining time spent mostly in the packet parser, see Section 3.3.1).

To optimize computation performance, the logger was reprogrammed. The dependency on NumPy was removed, and all associated data structures and opearations were replaced with ones from the Python standard library or custom, implemented in pure Python code.

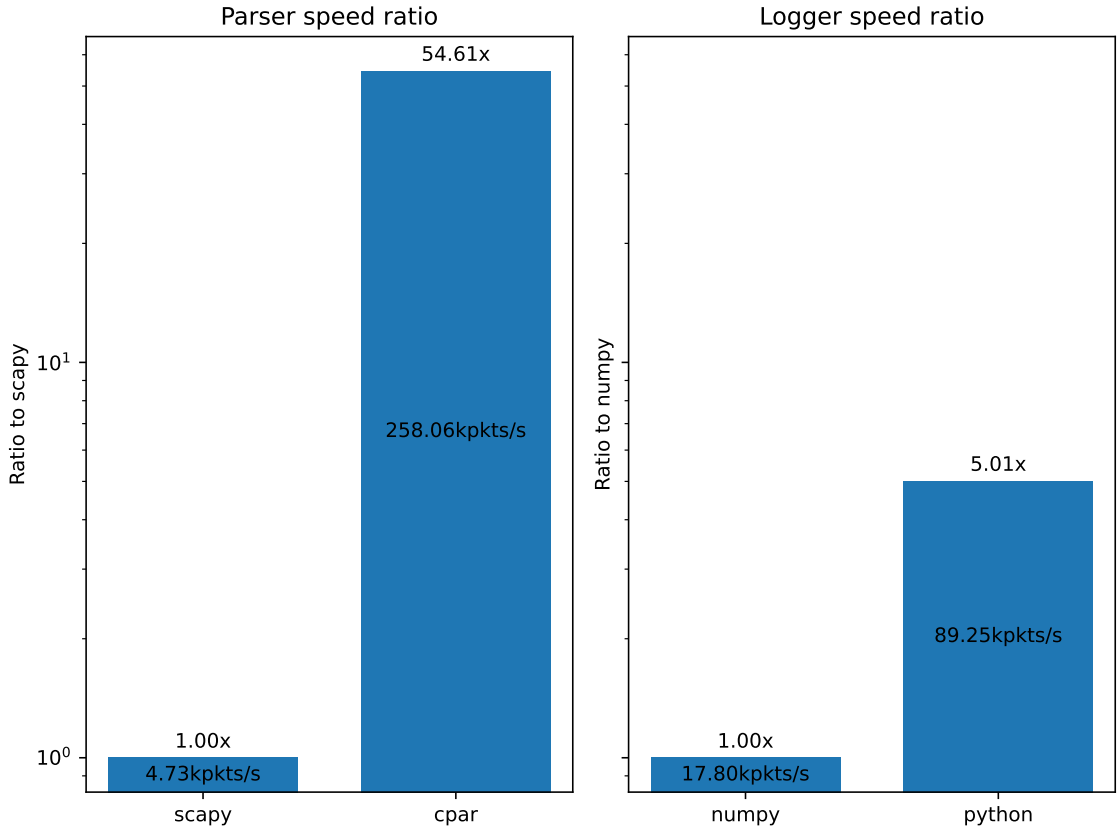

Figure 3.3: Results from parser and logger implementation benchmarks. Measured in absolute speed of processed kilo-packets per second, and its ratio to the original implementation. The logger benchmark ran with the CParser parser implementation.

Performance of the new implementation was evaluated. The scenario consisted of processing a packet capture file into final vectors of statistical features for further evaluation. The CParser parser implementation was used (Section 3.3.1). The results are depicted in Figure 3.3 and summarized in the following table, where *kpkts/s* stands for kilo-packets per second.

Logger	Processing Speed	Ratio to NumPy
NumPy	17.80 kpkts/s	1.00 x
Python	89.25 kpkts/s	5.01 x

The pure Python reimplementation is 5 times faster than the NumPy one. Depending on the particular capture file used the results can vary, from own experimentation, pure Python is anywhere between 5 to 6 times faster than NumPy.

3.3.3 Combined Performance

Before optimizations, on a modern machine, processing a 950 MB traffic capture file with 5.5 million packets into vectors with final statistical features for further evaluation took 28 minutes, processing 3.26 kpkts/s on average. After optimizing both the packet parser and

statistical logger, the same task took just 70 seconds, processing 77.52 kpkts/s on average. The optimized version is 23 times faster than the original. These results are summarized in the following table.

Version	Processing Speed	Ratio to Original
Original	3.26 kpkts/s	1.00 x
Optimized	77.52 kpkts/s	23.77 x

3.4 Summary

This chapter proposed and implemented a modified and optimized version of the KitNET model that better fits the nature of real-time DDoS detection but does not decrease the detection success rate — SimpleKitNET. Further, two major components of the Windower system were profiled and optimized resulting in significant increases in processing speed.

Chapter 4

SYN Fingerprints

In this chapter, Transmission Control Protocol (TCP) fingerprinting is introduced. An experiment was conducted to show how TCP fingerprints from legitimate and malicious applications differ. Additionally, analyses on TCP fingerprints from labelled and unlabelled network captures are performed. An addition to the Windower (Section 2.4), that works alongside it and makes use of TCP SYN fingerprints, is proposed, implemented and evaluated. The results show the addition leads to mostly better DDoS detection results. Before all of that, the relevant parts of the TCP protocol are briefly described.

4.1 Transmission Control Protocol

TCP [11] is connection-oriented transport-layer protocol that provides a reliable, in-order, byte-stream service to applications.

To establish a connection between two peers, a procedure called the three-way handshake needs to happen. The basic handshake is depicted in Figure 4.1. The messages involved in the basic handshake are SYN, ACK and SYN+ACK. These messages are recognized by the value of the control bits field (flags) in the TCP header. The handshake begins by peer A sending a SYN to peer B, then peer B sends a SYN+ACK to peer A acknowledging peer A's SYN. Finally, peer A sends an ACK message to peer B acknowledging peer B's SYN.

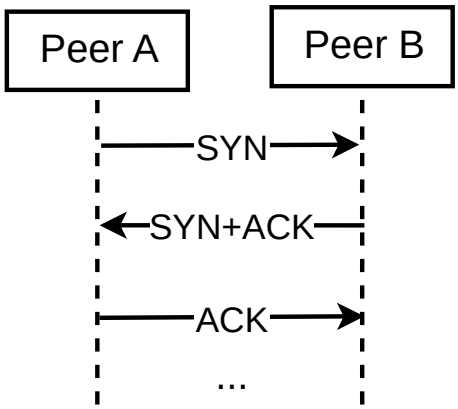

Figure 4.1: Basic TCP three-way handshake.

The SYN message is always the first message when a connection is being established, it has special meaning and it can carry additional options specific to the SYN. This behaviour is made use of in a technique called TCP fingerprinting (Section 4.2).

4.2 TCP Fingerprinting

TCP fingerprinting is a technique that extracts values of specific protocol header fields in packets which use the Transmission Control Protocol (TCP), specifically the SYN and SYN+ACK messages (Section 4.1). Since the TCP protocol is practically always layered above the Internet Protocol (IP), IP header fields can also be included in the fingerprint. The operating system (OS) and application which created a TCP packet, had to decide on these values.

Suppose these values were decided on deterministically, based only on the operating system and application pair that wants to send a TCP packet. In such a scenario, if a host running an operating system and an application sends a TCP packet, these values will be the same as the ones from a TCP packet originating from another host running the same operating system and the same application. Since they will be the same, one can infer that the packets originate from hosts running the same operating system and application. Thus, the set of these specific extracted values can be thought of as a fingerprint for the specific operating system and application pair.

In reality though, depending on which fields are extracted from the protocol headers in the packet, their values depend not only on the operating system and application pair, but also their versions and their configuration, and the libraries and system application programming interfaces used to implement network communication in the application. In addition, the software implementations creating TCP packets might make use of stochastic algorithms, possibly introducing randomness into the fingerprint. In addition, a database of fingerprints, mapping them to the operating systems and applications that produced them, needs to be kept up-to-date if there is a need to not only infer operating system and application similarity but also find out what application and OS produced the fingerprint. Moreover, malicious applications can obtain the legitimate fingerprint of the OS they are running, either from a database or maybe a quick packet capture, and try to imitate the legitimate fingerprint of the system. All these factors influence the usability of TCP fingerprints for operating system and application similarity and exact type detection.

There are two types of TCP fingerprinting methods:

- Passive fingerprinting — this method passively observes a monitored network, gathering fingerprints from observed TCP packets. An example of a passive fingerprinting tool is *p0f* [40].
- Active fingerprinting — this method gathers fingerprints by actively probing network hosts for TCP responses, from which the fingerprints are extracted. One example of an active fingerprinting tool is *Nmap* [34]. Active fingerprinting is more powerful than passive fingerprinting, since it does not have to accept only whatever communication happens, but can use custom probes to initiate communication useful for the purposes of fingerprinting.

Despite all the challenges, research has shown that even passive TCP fingerprinting is useful for operating system detection [19] [22] [24].

4.3 Malicious Hosts Discrimination

Since packet-based DDoS detection systems, like the Windower, already need to parse packets passing into the protected network, passive fingerprinting could potentially be a method to enhance the detection capabilities.

Although passive TCP fingerprinting is used for operating system (OS) detection, it is not as useful for discriminating individual applications. This is caused by many factors. For example, programming network applications using high-level built-in libraries of the programming language or high-level external libraries, which are abstractions built upon the networking application programming interfaces (APIs) of the underlying operating systems (usually the BSD sockets API), leads to fingerprints not identifying the application itself but rather the library or the language used to program the application, since that is the part of the application that has direct influence on the packet protocol fields used for fingerprinting. This is usually the case if the application is written in higher-level languages like Python, Java, and Go, or when using a lower-level language like C or C++ and the common OS APIs, but with all the default options (or customized options that are not reflected in the fingerprint).

A TCP fingerprint can identify some unique network application, or some small subset of network applications, only if the application goes out of its way and uses either a special lower-level networking API or customizes the default settings of whatever API it uses, in such a way that is reflected in the values of the fields used to create the fingerprint. An example of a special lower-level networking API in the Linux operating system are *raw sockets*, which provide complete control over the IP and TCP header fields to the application. An example of customizing the default settings of more common higher-level APIs in the Linux operating system are options of classic BSD sockets, for instance, the TCP window size can be changed with this method.

In a real-time DDoS detection setting, considering passive TCP fingerprinting, the gathered fingerprints could potentially provide helpful additional information to discriminate between malicious and benign source hosts. Considering a scenario where a trusted network is being protected, SYN fingerprints are the most useful, because SYN+ACK fingerprints would not carry any information about the potential malicious hosts connecting from outside the network.

4.4 Common and Malicious Network Applications

In this section, an experiment was conducted in order to build a better picture of the TCP fingerprints produced by individual legitimate and malicious network applications. Some common and malicious network applications were fingerprinted on modern operating systems. In addition, analyses of TCP fingerprints from labelled and unlabelled network captures were conducted.

4.4.1 Applications Analyses

Setup

The applications were fingerprinted on the Windows 11 Pro, Arch Linux with kernel version 6.5.9, Oracle Linux Server with kernel version 4.18.0 operating systems. The Windows 11 and Arch Linux machines were casual personal computers. The Oracle Linux Server system

was a development server, the graphical applications were excluded from the experiments on this machine. All experiments were performed on IPv4 only.

The common applications include web browsers with the major web engines, and command line network applications, specifically:

- Web browsers — Firefox, Librewolf, Qutebrowser,
- Command line utilities — ssh, curl, wget, telnet.

The malicious (or possibly malicious) applications include network penetration testing tools, packet crafting tools and DDoS tools, all of which can be used to generate DDoS attacks, specifically:

- Pentesting & packet crafting — hping, mausezahn,
- DDoS — slowloris [38], slowloris-ng [39], slowhttptest [32], Low Orbit Ion Canon [30].

Fingerprints were generated while the listed applications were used to initiate casual TCP connections. All common applications ran with their default settings, the malicious applications also ran with their default settings, but all operation modes were tested where possible. These applications were fingerprinted using the p0f [40] tool. This tool generated fingerprints from the TCP SYN packets sent by these tools. The fingerprints will be denoted as `ver:ittl:olen:mss:wsize,scale:olayout:quirks:pclass`, where individual fields are delimited by the colon `:`, meaning the following:

- ver — the Internet Protocol (IP) version,
- ittl — an informed guess of the initial value of the Time To Live (TTL) IPv4 field, or hop limit IPv6 field,
- olen — length of IPv4 options, always zero for IPv6,
- mss — maximum segment size, if specified in TCP options,
- wsize,scale — TCP window size. Can be expressed as a fixed value, but many operating systems set it to a multiple of MSS or MTU, or a multiple of some random integer. And the window scaling factor, if specified in TCP options.
- olayout — comma-delimited layout and ordering of TCP options, if any,
- quirks — comma-delimited properties and quirks observed in IP or TCP headers (not enumerated here for brevity),
- pclass — payload size classification: 0 for zero, + for non-zero. Abbreviated to pc, in the results.

Results

Results from the *Windows 11 Pro* desktop will be discussed next. In this operating system, all of the compatible tools produced the same fingerprint. Tools incompatible with Windows were hping, mausezahn and slowhttptest. Additionally, one more signature was produced by Low Orbit Ion Canon (LOIC) in a specific operation mode. The following table contains all of the unique generated fingerprints:

App	ittl	olen	mss	wsize,scale	olayout	quirks	pc
Legit	128+0	0	1460	mss*44,8	mss,nop,ws,nop,nop,sok	df,id+	0
LOIC	128+0	0	1460	16,0	mss,nop,ws,nop,nop,sok	df,id+	0

The legitimate fingerprint is common among all of the tools, *legitimate and malicious*. The second fingerprint is the additional fingerprint from the LOIC tool. Whereas the TCP window size was a specific multiple of the TCP maximum segment size in the common fingerprint, this fingerprint shows that the LOIC application explicitly set window size to a different, very low value, 16. It is not a multiple of MSS, nor is MSS a multiple of this value, they are likely not related in this case. Looking at the Windows networking APIs, it seems this can be achieved either by using raw sockets, or using options of common socket APIs.

Unique fingerprints from the Linux 4.18.0 development server are presented in the following table:

App	ittl	olen	mss	wsize,scale	olayout	quirks	pc
Legitimate	64+0	0	1460	mss*20,7	mss,sok,ts,nop,ws	df,id+	0
hping	64+0	0	0	512,0	-	-	0
hping	64+0	0	0	512,0	-	ack+	0
mausezahn	255+0	0	1452	10000,5	mss,sok,ts,nop,ws	-	0
slowhttptest	64+0	0	1460	1152,0	mss,sok,ts,nop,ws	df,id+	0

Web browsers and LOIC were omitted from the experiment on this machine. The legitimate fingerprint is common among all of the *legitimate* programs and shared with slowloris, slowloris-ng and slowhttptest. The hping fingerprints differ in 4 fields in contrast to the legitimate one. The mausezahn fingerprint also differs in 4 fields from the legitimate. The last one is an additional one from slowhttptest, differing in the field consisting of the window size and scale.

Hping and mausezahn are flexible packet generators, both make use of raw sockets, thus it is not a surprise that their fingerprints differ. One of the slowhttptest operation modes configures the window size socket option, one of its fingerprints differs from the legitimate for this reason. The slowloris, slowloris-ng applications are programmed in the Python programming language with its high-level socket library, as expected then, their fingerprint does not differ from the first.

Lastly, unique fingerprints from the Linux 6.5.9 desktop are summarized in the following table:

App	ittl	olen	mss	wsize,scale	olayout	quirks	pc
Legitimate	64+0	0	1460	mss*44,7	mss,sok,ts,nop,ws	df,id+	0
hping	64+0	0	0	512,0	-	-	0
hping	64+0	0	0	512,0	-	ack+	0
mausezahn	255+0	0	1452	10000,5	mss,sok,ts,nop,ws	-	0
slowhttptest	64+0	0	1460	1152,0	mss,sok,ts,nop,ws	df,id+	0
LOIC	64+0	0	1460	1152,0	mss,sok,ts,nop,ws	-	0
LOIC	64+0	0	1460	mss*44,7	mss,sok,ts,nop,ws	-	0

The legitimate fingerprint is common among all of the *legitimate* tools and shared with slowloris, slowloris-ng and slowhttptest, just like in the Linux 4.18.0 case, but it differs in window size from the Linux 4.18.0 system. Otherwise the same observations were made as on the Linux 4.18.0 system, except this time the LOIC tool was included. LOIC produced one fingerprint very similar to the legitimate one, differing only in the quirks field, and another fingerprint in a specific operation mode, which explicitly lowered the window size, similarly to the Windows experiment.

Conclusion

Although the legitimate fingerprints from all of these systems were not unique to just the legitimate applications, the results show that some malicious applications make use of raw sockets and socket options which is reflected in their TCP fingerprints. Thus, the fingerprints have potential to help discriminate between malicious and legitimate source hosts.

4.4.2 Dataset Analyses

Inspection of a traffic capture produced by real DDoS malware, available at [33] as malware capture 135 – 1, showed that the produced TCP SYN flood DDoS traffic had unusual fingerprint characteristics. It made use of raw sockets, which was evident from the fingerprints, it made use of 128 unique time-to-live values, had a specific control bits layout and included 896 bytes of payload.

Upon inspection of TCP SYN traffic from a capture from the CESNET — ACOnet link, the most common fingerprint was used for SYN flooding and scanning only, and had the characteristics of a RAW socket.

The potential to discriminate between malicious and legitimate source hosts the SYN fingerprint showed in Section 4.4.1 was thus more credible.

4.5 Windower Modification

This section proposes additional statistical features to be added to the Windower system. These features were implemented and evaluated from the DDoS detection success rate perspective. The results show more success with the new features.

4.5.1 Additional Features

7 new source IP address scoped features were added to the Windower feature vector output:

- Count of unique fingerprints,
- Range of Maximum Segment Size (MSS) values,
- Range of Window Size values (WS),
- Range of Window Scale (scale) values,
- Range of Time-To-Live (TTL) values,
- Option count average,
- Quirk count average.

CParser (Section 3.3.1) was extended to extract all of the values present in the p0f fingerprint (Section 4.4.1) with a few modifications. The ittl field is not an informed guess of the initial TTL value but rather the TTL value present upon packet observation. The guess is important for operating system detection, but the actual value is more useful for the purposes of DDoS detection. The olen field supports IPv6, in that case the value represents the length of all extension headers up to the transport-layer protocol.

These new features are computed in the same way the other features are. The fingerprint statistical logger component was added to the Windower. It is controlled by the windowing controller (Section 3.3, Section 2.4), just like the packet statistical logger. The sliding-window is the same across both of the loggers.

For each observed source IP address in the current window, a window-data record is created that keeps track of the minimum and maximum values of the MSS, WS, scale and TTL fields. It also tracks the count of unique fingerprints using the HyperLogLog [12] (HLL) probabilistic structure, and a running average of option counts and quirk counts. Once the window ends the record is appended to the window-data backlog of the corresponding source IP address.

Once the windowing controller requests to export the final features of some source IP address for evaluation, the associated window-data backlog is retrieved and the contained records are summarized. The minimums and maximums are compared to each other to choose a maximum and minimum across all records. HLL structures are merged into a single HLL structure and its cardinality estimation serves as the unique fingerprint count. The averages are averaged across all records.

At the same time the windowing controller also requests the final features for the same source IP address from the packet statistical logger and features from these loggers are merged into the final statistical features output vector, that is subsequently evaluated by the SimpleKitNET model (Section 3.2). If there was no information about SYN fingerprints in the record backlog, the new features are set to default values representing the lack of information. Thus, the addition does not limit the Windower in respect to what type of traffic it is limited to.

The features representing ranges were inspired by the variation of the TTL value in a DDoS capture described in Section 4.4.2 and the varying values of the relevant fields from application analyses in Section 4.4.1. The count of unique fingerprints was included in the same spirit. The quirk and option counts were added also due to the application analyses.

4.5.2 DDoS Detection Success Rate

The effect on DDoS detection by inclusion of the new SYN fingerprint related features was evaluated. Experiments on three datasets were conducted. The datasets used for testing were the same as in [15], prepared according to reproduction instructions in the accompanying repository [16]. The CTU-13 scenario 4 dataset was excluded, because it does not contain DDoS attacks utilizing the TCP protocol. Results are presented as confusion matrix tables (Section 2.2), where the positive class is the malign class and the negative class is the benign class.

Dataset SUEE8 – 2017

The malicious traffic in the SUEE8 – 2017 dataset [20] consists of slow-and-low DDoS attacks targeting HTTP servers. Attacks are generated by slowloris [38], slowloris-ng [39], and

slowhttptest [32]. Both slowloris and slowloris-ng are programmed in Python and both produced the same fingerprint which is just that of the Python socket abstraction library, as explained in Section 4.3. Since that fingerprint was common in benign data, the detection results could not be improved and the results were exactly the same.

Dataset UNSW-NB15

The test part of the subset of the UNSW-NB15 dataset [27], was modified to exclude packets originating from or targeted to the few addresses that were used as malicious traffic sources even if the label of these packets was benign. Upon manual inspection, the traffic was suspicious and it was likely not benign since these few addresses were the only ones used to generate malicious traffic.

The malicious traffic consists almost entirely of TCP traffic from many different DoS attacks. The following tables show the confusion matrix from evaluation without the new features and with the new features.

Without	Predicted positive	Predicted negative
Positive	0	344
Negative	0	7357

With	Predicted positive	Predicted negative
Positive	38	306
Negative	0	7357

Without the new features, no vectors from malicious host data were detected as malicious, all of them were wrongly classified as negative, but no vectors from benign host data were wrongly classified as malign. Thus, even though no attackers were detected correctly, no legitimate hosts were denied wrongly either.

In contrast, *with* the new features, 38 out of the 344 vectors from malicious hosts were correctly classified as malign. All four attackers were detected at least once, although they still went undetected most of the time. Even though in this case the attackers went mostly undetected, it was early in the scenario when they were classified correctly for the first time. Depending on the policy of how to act upon an address classified as malicious, this could lead to a significant improvement. If for example, the address would get blocked and stay blocked for a significant amount of time, over 97 % of the malicious traffic would get blocked. On the other hand, if the address would be blocked only until it is classified as benign, the results would provide a much smaller benefit.

Dataset CAIDA

This dataset is a mix of traces collected from high-speed monitors on a commercial backbone link and traces from a real DDoS attack. Manual inspection revealed that 50 % of the attack is a SYN flood. The following tables show the confusion matrix from evaluation without the new features and with the new features.

Without	Predicted positive	Predicted negative
Positive	1920	0
Negative	50	908

With	Predicted positive	Predicted negative
Positive	1913	7
Negative	50	908

Without the new features, all vectors from malicious host data were correctly detected as malicious.

In contrast, *with* the new features, 7 out of the 1920 vectors from malicious hosts that were previously correctly classified as malign are now classified as benign, although all attackers were still detected at least once. Only the vectors belonging to one specific attacker were affected. Classifications of benign vectors stayed unaffected. It is possible that the small regression is a result of the fact that the training data is just considered passive, so it may contain some malicious traffic that just so happened to be the cause of the regression. Potentially, the dataset may also be affected by a time window temporal bias and artefacts introduced by mixing the data from two different computer networks as stated in the reproduction instructions [16].

4.5.3 Discussion

The addition of features created from SYN fingerprints to the Windower caused previously completely undetected malicious hosts to be detected at least some of the time. This seems to further confirm that SYN fingerprints are a valuable data source to consider when detecting DDoS attacks. Due to the small number of viable datasets to evaluate and their issues, it is hard to draw a conclusion as to how much value the SYN fingerprints offer. The question could be answered better if more datasets of good quality were to be evaluated along with different features.

There is also an interesting alternative method of integrating SYN fingerprint features to the Windower. Instead of aggregating fingerprint features based on the source IP address, one could aggregate the features by the fingerprint itself. The data from which the features would be computed would be scoped to a single fingerprint. Thus, these features could be, for example, port entropy or IP address entropy and others, per fingerprint. A separate model would be trained to classify these feature vectors and decide if the fingerprint is malicious or not. This approach also has the benefit that it would be signature-based and could thus potentially ease mitigation by decreasing the number of mitigation rules, as already touched on in Section 2.3.

Another modification to try could be a combination of the implemented approach and this proposed alternative. The idea is to extend the fingerprint logger to not only store and compute features scoped to a single source IP address, but also hold a history of windows scoped globally. The globally scoped records would store information about all the fingerprints observed in the individual windows regardless of the source IP address. The information from the global history could then be possibly used to compute more features for the address scoped feature vectors. In order to correlate the two, either, the address scoped history would carry information about which fingerprints were observed, or the globally scoped history would carry information about which source IP addresses used each

fingerprint. One of the features could be for example, the number of hosts using the same fingerprints.

4.6 Summary

In this chapter, TCP fingerprinting was introduced as an additional resource to consider when detecting DDoS attacks. Analyses on SYN fingerprints from different legitimate and malicious applications showed that in some cases, malicious applications produce significantly different fingerprints compared to the legitimate applications. This was further confirmed by analysing labelled and unlabelled datasets. Based on those findings, additional features for the Windower system were proposed, implemented and evaluated. The additions resulted in improved detection of malicious hosts while detection of legitimate hosts was completely unaffected.

Chapter 5

Conclusion

This thesis proposed TCP SYN message fingerprints as an additional resource to extract evaluation features from to improve DDoS detection. The reasons why such fingerprints can aid in malicious hosts detection were explained. The effects legitimate and malicious applications have on SYN fingerprints were explored and summarized. The observations made on individual applications and selected datasets inspired an addition to the Windower, a packet feature extraction and aggregation system that computes feature vectors for evaluation by some machine-learning model. The addition was implemented and the new SYN fingerprint based features it adds to evaluation lead to decent DDoS detection improvements on relevant datasets. The approach of integrating SYN fingerprints was discussed and further ideas were provided.

A modification of KitNET, an autoencoder based anomaly detector, that better fits the nature of real-time DDoS detection was proposed, implemented, optimized and evaluated. The modified version makes a better pairing for the Windower, provides major training and execution time improvements, and does not degrade the DDoS detection success rate in any way. It is 16 and 95 times faster to train and execute respectively.

The subpar performance of the Windower system was profiled and major components, such as the packet parser and statistical logger, were optimized and reimplemented leading to significant performance improvements. Combined, the optimized Windower is 23 times faster. Whereas before it took 27 minutes to process a specific dataset, now it only takes 70 seconds.

These results are of practical use and will be integrated into the DDoS Protector system, developed by CESNET as part of a security research project of the Ministry of Interior of the Czech Republic.

An interesting idea for future work is signature-based mitigation of DDoS attacks that make use of TCP. The signatures would be created on-demand from SYN fingerprints, possibly in the form of BPF filters. Information about network traffic would be aggregated by fingerprints. The data from which features for evaluation would be computed would be scoped to a single fingerprint and a discrimination model would thus classify the fingerprint. This approach has the potential to ease mitigation by decreasing the number of mitigation rules in contrast to blocking large amounts of IP addresses. It is an orthogonal approach that could be used alongside aggregation by IP addresses.

Fingerprinting is not limited to TCP SYN messages. For example, SSL/TLS protocols also carry specific information in their connection setup procedures. Their fingerprints could also potentially improve DDoS detection in a similar spirit by discriminating between benign and malign applications on the layer of application protocols.

Bibliography

[1] Ahmad, Z.; Shahid Khan, A.; Wai Shiang, C.; Abdullah, J. and Ahmad, F. Network Intrusion Detection System: A Systematic Study of Machine Learning and Deep Learning Approaches. *Transactions on Emerging Telecommunications Technologies*, 2021, vol. 32, no. 1, p. e4150. ISSN 2161-3915.

[2] Ahmad Najar, A. and Manohar Naik, S. Applying Supervised Machine Learning Techniques to Detect DDoS Attacks. In: *2022 2nd Asian Conference on Innovation in Technology (ASIANCON)*. August 2022, p. 1–7.

[3] Bradbury, J.; Frostig, R.; Hawkins, P.; Johnson, M. J.; Leary, C. et al. *JAX: Composable Transformations of Python+NumPy Programs* online. 2018. Available at: http://github.com/google/jax. [cit. 2024-04-24].

[4] Chollet, F. et al. *Keras* online. 2015. Available at: https://keras.io. [cit. 2024-04-24].

[5] Chou, E. and Groves, R. *Distributed Denial of Service (DDoS): Practical Detection and Defense*. 1st ed. Sebastopol, CA: O'Reilly Media, 2018. ISBN 978-1-4920-2617-4.

[6] Cloudflare, Inc.. *DDoS Attack Trends for 2024 Q1* online. Available at: https://radar.cloudflare.com/reports/ddos-2024-q1. [cit. 2024-04-24].

[7] Cludflare, Inc.. *DNS Amplification DDoS Attack* online. Available at: https://www.cloudflare.com/learning/ddos/dns-amplification-ddos-attack. [cit. 2024-04-24].

[8] Cludflare, Inc.. *HTTP Flood DDoS Attack* online. Available at: https://www.cloudflare.com/learning/ddos/http-flood-ddos-attack. [cit. 2024-04-24].

[9] Cludflare, Inc.. *SYN Flood DDoS Attack* online. Available at: https://www.cloudflare.com/learning/ddos/syn-flood-ddos-attack. [cit. 2024-04-24].

[10] Cludflare, Inc.. *What Is a Distributed Denial-of-Service (DDoS) Attack?* online. Available at: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack. [cit. 2024-04-24].

[11] Eddy, W. *Transmission Control Protocol (TCP)* RFC 9293. RFC Editor, august 2022. Available at: https://doi.org/10.17487/RFC9293.

[12] Flajolet, P.; Fusy, É.; Gandouet, O. and Meunier, F. HyperLogLog: The Analysis of a near-Optimal Cardinality Estimation Algorithm. *Discrete Mathematics & Theoretical Computer Science*, june 2007, p. 137–156.

[13] Goldschmidt, P. *Heuristické metody pro potlačení DDoS útoků zneužívajících protokol TCP*. Brno, CZ, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[14] Goldschmidt, P. *Mitigation of DoS Attacks Using Machine Learning*. Brno, CZ, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology.

[15] Goldschmidt, P. and Kučera, J. Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning. In: *NOMS 2024 IEEE/IFIP Network Operations and Management Symposium*. 2023.

[16] Goldschmidt, P. and Kučera, J. *Windower* online. February 2024. Available at: https://github.com/xGoldy/Windower. [cit. 2024-03-16].

[17] Goodfellow, I.; Bengio, Y. and Courville, A. *Deep Learning*. Cambridge, Massachusetts: The MIT Press, 2016. Adaptive Computation and Machine Learning. ISBN 978-0-262-03561-3.

[18] Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P. et al. Array Programming with NumPy. *Nature*. Springer Science and Business Media LLC, september 2020, vol. 585, no. 7825, p. 357–362.

[19] Hulák, M.; Bartoš, V. and Čejka, T. Evaluation of Passive OS Fingerprinting Methods Using TCP/IP Fields. In: *2023 8th International Conference on Smart and Sustainable Technologies (SpliTech)*. June 2023, p. 1–4.

[20] Institute of Distributed Systems, Ulm University. *SUEE 2017 Dataset* online. Available at: https://github.com/vs-uulm/2017-SUEE-data-set. [cit. 2024-04-24].

[21] Lam, S. K.; Pitrou, A. and Seibert, S. Numba: A LLVM-based Python JIT Compiler. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY, USA: Association for Computing Machinery, November 2015, p. 1–6. LLVM '15. ISBN 978-1-4503-4005-2.

[22] Laštovička, M.; Husák, M.; Velan, P.; Jirsík, T. and Čeleda, P. Passive Operating System Fingerprinting Revisited: Evaluation and Current Challenges. *Computer Networks*, june 2023, vol. 229, p. 109782. ISSN 1389-1286.

[23] LeCun, Y.; Bengio, Y. and Hinton, G. Deep Learning. *Nature*, may 2015, vol. 521, no. 7553, p. 436–444. ISSN 0028-0836, 1476-4687.

[24] Matoušek., P.; Ryšavý., O.; Grégr., M. and Vymlátil., M. Towards Identification of Operating Systems from the Internet Traffic - IPFIX Monitoring with Fingerprinting and Clustering. In: *Proceedings of the 5th International Conference on Data Communication Networking (ICETE 2014) - DCNET*. SciTePress / INSTICC, 2014, p. 21–27. ISBN 978-989-758-042-0.

[25] Mirsky, Y. *KitNET-py* online. March 2024. Available at: https://github.com/ymirsky/KitNET-py. [cit. 2024-03-11].

[26] Mirsky, Y.; Doitshman, T.; Elovici, Y. and Shabtai, A. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. *ArXiv*, february 2018, abs/1802.09089.

[27] Moustafa, N. and Slay, J. UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set). In: *2015 Military Communications and Information Systems Conference (MilCIS)*. November 2015, p. 1–6.

[28] Najafimehr, M.; Zarifzadeh, S. and Mostafavi, S. DDoS Attacks and Machine-Learning-Based Detection Methods: A Survey and Taxonomy. *Engineering Reports*, 2023, vol. 5, no. 12, p. e12697. ISSN 2577-8196.

[29] Naveen Bindra and Manu Sood. Detecting DDoS Attacks Using Machine Learning Techniques and Contemporary Intrusion Detection Dataset. *Automatic Control and Computer Sciences*, september 2019, vol. 53, no. 5, p. 419–428. ISSN 1558-108X.

[30] Praetox Technologies et al. *Low Orbit Ion Cannon* online. 2009. Available at: https://github.com/NewEraCracker/LOIC. [cit. 2024-04-24].

[31] Rigo, A.; Fijałkowski, M. and contributors. *CFFI* online. 2012. Available at: https://github.com/python-cffi/cffi. [cit. 2024-04-24].

[32] Shekyan, S. et al. *Slowhttptest* online. 2011. Available at: https://github.com/shekyan/slowhttptest. [cit. 2024-04-24].

[33] Stratosphere. *Stratosphere Laboratory Datasets* online. 2015. Available at: https://www.stratosphereips.org/datasets-overview. [cit. 2024-04-24].

[34] The Nmap Project. *Nmap* online. Available at: https://nmap.org. [cit. 2024-03-16].

[35] The Tcpdump Group et al. *Linux Cooked Mode Capture Version 1* online. Available at: https://www.tcpdump.org/linktypes/LINKTYPE_LINUX_SLL.html. [cit. 2024-04-23].

[36] Thomas, T.; P. Vijayaraghavan, A. and Emmanuel, S. Machine Learning and Cybersecurity. In: Thomas, T.; P. Vijayaraghavan, A. and Emmanuel, S., ed. *Machine Learning Approaches in Cyber Security Analytics*. Singapore: Springer, 2020, p. 37–47. ISBN 9789811517068.

[37] Vincent, P.; Larochelle, H.; Bengio, Y. and Manzagol, P.-A. Extracting and Composing Robust Features with Denoising Autoencoders. In: *Proceedings of the 25th International Conference on Machine Learning - ICML '08*. New York, NY, USA: Association for Computing Machinery, 2008, p. 1096–1103. ICML '08. ISBN 978-1-60558-205-4.

[38] Yaltirakli, G. *Slowloris* online. 2015. Available at: https://github.com/gkbrk/slowloris. [cit. 2024-04-24].

[39] Yaltirakli, G.; Erb, B. et al. *Slowloris-Ng* online. 2017. Available at: https://github.com/vs-uulm/slowloris-ng. [cit. 2024-04-24].

[40] Zalewski, M. *P0f V3* online. 2000. Available at: https://lcamtuf.coredump.cx/p0f3. [cit. 2024-03-16].