Leipzig University
Faculty of Mathematics & Computer Science
Department of Computer Science
Database Group

Comparing Anomaly-Based Network Intrusion Detection Approaches Under Practical Aspects

Bachelor's Thesis

Presented in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in **Computer Science**

Author
Daniel Helmrich

Supervisor
Martin Grimmer, M.Sc.

Second Reviewer
Prof. Dr. Erhard Rahm

Leipzig, 9 April 2021

Abstract

English

While many of the currently used network intrusion detection systems (NIDS) employ signature-based approaches, there is an increasing research interest in the examination of anomaly-based detection methods, which seem to be more suited for recognizing zero-day attacks. Nevertheless, requirements for their practical deployment, as well as objective and reproducible evaluation methods, are hereby often neglected. The following thesis defines aspects that are crucial for a practical evaluation of anomaly-based NIDS, such as the focus on modern attack types, the restriction to one-class classification methods, the exclusion of known attacks from the training phase, a low false detection rate, and consideration of the runtime efficiency. Based on those principles, a framework dedicated to developing, testing and evaluating models for the detection of network anomalies is proposed. It is applied to two datasets featuring modern traffic, namely the UNSW-NB15 and the CIC-IDS-2017 datasets, in order to compare and evaluate commonly-used network intrusion detection methods. The implemented approaches include, among others, a highly configurable network flow generator, a payload analyser, a one-hot encoder, a one-class support vector machine, and an autoencoder. The results show a significant difference between the two chosen datasets: While for the UNSW-NB15 dataset several reasonably well performing model combinations for both the autoencoder and the one-class SVM can be found, most of them yield unsatisfying results when the CIC-IDS-2017 dataset is used.

German

Obwohl viele der derzeit genutzten Systeme zur Erkennung von Netzwerkangriffen (engl. *NIDS*) signaturbasierte Ansätze verwenden, gibt es ein wachsendes Forschungsinteresse an der Untersuchung von anomaliebasierten Erkennungsmethoden, welche zur Identifikation von Zero-Day-Angriffen geeigneter erscheinen. Gleichwohl werden hierbei Bedingungen für deren praktischen Einsatz oft vernachlässigt, ebenso wie objektive und reproduzierbare Evaluationsmethoden. Die folgende Arbeit definiert Aspekte, die für eine praxisorientierte Evaluation unabdingbar sind. Dazu zählen ein Schwerpunkt auf modernen Angriffstypen, die Beschränkung auf One-Class Classification Methoden, der Ausschluss von bereits bekannten Angriffen aus dem Trainingsdatensatz, niedrige Falscherkennungsraten sowie die Berücksichtigung der Laufzeiteffizienz. Basierend auf diesen Prinzipien wird ein Rahmenkonzept vorgeschlagen, das für das Entwickeln, Testen und Evaluieren von Modellen zur Erkennung von Netzwerkanomalien bestimmt ist. Dieses wird auf zwei Datensätze mit modernem Netzwerkverkehr, namentlich auf den UNSW-NB15 und den CIC-IDS-2017 Datensatz, angewendet, um häufig genutzte NIDS-Methoden zu vergleichen und zu evaluieren. Die für diese Arbeit implementierten Ansätze beinhalten, neben anderen, einen weit konfigurierbaren Netzwerkflussgenerator, einen Nutzdatenanalysierer, einen One-Hot-Encoder, eine One-Class Support Vector Machine sowie einen Autoencoder. Die Resultate zeigen einen großen Unterschied zwischen den beiden ausgewählten Datensätzen: Während für den UNSW-NB15 Datensatz verschiedene angemessen gut funktionierende Modellkombinationen, sowohl für den Autoencoder als auch für die One-Class SVM, gefunden werden können, bringen diese für den CIC-IDS-2017 Datensatz meist unbefriedigende Ergebnisse.

Contents

1. Introduction **1**
1.1. Motivation 1
1.2. Goals 2
1.3. Thesis Structure 3

2. Background **4**
2.1. Network Traffic 4
2.1.1. Relevant Network Protocols 4
2.1.2. Network Flows 6
2.2. Network Vulnerabilities and Attacks 6
2.2.1. Definitions 6
2.2.2. Types of Network Attacks 7
2.2.3. Zero-Day Attacks 8
2.3. Anomaly-Based Network Intrusion Detection 8
2.3.1. Types of Anomalies 8
2.3.2. Components of Anomaly-Based NIDS 9
2.3.3. Comparing Learning Methods for Network Traffic Anomaly Detection 10
2.3.4. One-Class Support Vector Machines 14
2.3.5. Autoencoders 15
2.4. Evaluation of Anomaly-Based NIDS 17
2.4.1. Binary Classification 17
2.4.2. Relevant Measurements 17
2.5. Datasets for Network Intrusion Detection 19
2.5.1. Dataset Requirements 19
2.5.2. CIC-IDS-2017 and CSE-CIC-IDS-2018 20
2.5.3. UNSW-NB15 21
2.5.4. CIC DoS 21
2.5.5. Outdated Datasets 21

3. Related Work **22**
3.1. Usage of One-Class Support Vector Machines 22
3.2. Usage of Autoencoders 23
3.3. Payload Analysis 23
3.4. Comparative Experiments 25

4. Concept **26**
4.1. Overview 26
4.2. Preprocessing 26
4.3. Model Components and Training 28
4.4. Classification of Unknown Network Traffic 31
4.5. Evaluation 32
4.6. Hyperparameter Search 33

5. Implementation 35
5.1. General Overview and Utilized Technologies 35
5.2. Dataset Preprocessing 36
5.2.1. Assigning Packets to Flows 36
5.2.2. Occurring Problems 39
5.2.3. Preprocessing Result and Validation 40
5.3. Feature Extraction 42
5.3.1. Network Flow Generation 42
5.3.2. Flow-Based Payload Analysis 44
5.4. Feature Transformation 46
5.4.1. Min-Max-Scaling 46
5.4.2. Standardization 47
5.4.3. One-Hot Encoding 47
5.4.4. Principal Component Analysis 48
5.5. Decision Engines 48
5.5.1. One-Class SVM 48
5.5.2. Autoencoder 49

6. Experiments 50
6.1. Overview and Experiment Setup 50
6.2. Results 51
6.2.1. One-Class SVM 51
6.2.2. Autoencoder 52
6.2.3. Flow Feature Extractor 53
6.2.4. Payload Analysis 54
6.2.5. Unclean Training Data 54
6.2.6. Bigger Subsets for the UNSW-NB15 Dataset 54

7. Conclusion 55
7.1. Summary 55
7.2. Future Work 56

A. Bibliography i

B. List of Figures vi

C. List of Tables vii

D. Tables ix

Chapter 1 | Introduction

1.1. Motivation

As nowadays more and more of daily life and communication is connected by networks like the Internet, there is an increasing demand for protection and security. Not only private persons, but also all kinds of companies and organizations are steadily digitalizing parts of their daily routines and processes and are therefore relying on web services and data that is sent via networks. Thereby, the risk of getting exposed to security vulnerabilities is growing, and at the same time the potential loss of personal data, business secrets, or money. The German business association *Bitkom* concluded in a report that 7 out of 10 interviewed companies faced damage through digital attacks in 2017 or 2018 [3, p. 14]. Moreover, the number of attacks was reported to increase in 74% of the companies compared with the years before [3, p. 16] and in total, damage of roughly 205 billion euro was determined to be caused. This sum arose from different domains, such as blackmailing, losses in sales due to plagiarism, and costs for lawsuits, investigations, or for the acquisition of compensating measures [3, p. 23]. An even more drastic consequence had an attack on a German hospital, which reportedly caused delays in the treatment of patients and presumably resulted in the death of a woman [39].

To cope with such dangers, network intrusion detection systems (NIDS) are being developed. Their purpose is to identify malicious behaviour in networks and to raise alerts when suspicious events occur. In contrast to this exist Host Intrusion Detection Systems, which monitor the internals of a single machine.

There are two main types of NIDS, based on their underlying detection mechanism: **signature-based** and **anomaly-based NIDS**. The first can only recognize already known behaviour, whereas the latter aim to distinguish all network traffic that differs from what is considered normal. Nowadays the most used NIDS are signature-based, with the tool *snort* being a prominent open-source representative. It relies on rule sets that contain the necessary information for detecting a great number of present-day network attacks. While having success with protecting networks in real-world scenarios, a downside of this approach is the inability to detect yet unknown attacks (*zero-day attacks*), since there is no rule available for them yet.

For recognizing malicious activities without having prior knowledge of their characteristics, anomaly-based NIDS are better suited. They create a profile of normal behaviour in the network and then thereby can detect anomalies (or *outliers*) that indicate an ongoing attack. In theory, this makes zero-day attack detection possible. Albeit, a typical problem is a high false alarm rate and a prediction accuracy that underlies those of signature-based systems when encountering already known attacks. Despite such problems, the practical significance of detecting new attacks is constantly increasing, as a recent report of the German Federal Office for Information Security[1] shows: 117.4 million new malware variations could be identified within a one-year-long period between 2019 and 2020 [11]. One of the more notorious malicious programs in recent times was *Emotet*, which combines several intrusion techniques to infect and spy out victim systems, so that the attackers

[1] in German: *Bundesamt für Sicherheit in der Informationstechnik (BSI)*

are able to execute ransomware on systems that appear profitable to them. The victim's system or data thereby gets decrypted and a ransom demand is sent, which is reported to be of the order of tens of millions in some cases. Until January 2021, when *Emotet* servers were reported to be put out of operation [20], it infected various systems of organizations worldwide, among them the Lithuanian healthcare system [22] and a German university [49].

While there is a lot of recent research on this type of NIDS, most of it does not go into detail about the suitability for being used in a real-world setup. The following necessities for **evaluating anomaly-based NIDS under practical aspects** can be identified:

Independence from a specific network setup, or dataset: Most models in research are only tested on one dataset, which makes comparing results for inferring their practical significance difficult.

No reliance on the availability of an already-labelled dataset: The existence of network data together with the complete information about present attacks in it is unrealistic in a practical setup. This restriction also influences the range of choices for the learning methods that are used for the classification of network traffic. In some research papers, the models are trained in a supervised manner and rather learn the signature of the attacks. Such experiments, however, give little information about its zero-day attack detection abilities.

Detection of modern types of network attacks: The NIDS must be able to detect contemporary network attacks. Some works use outdated datasets which do not contain attacks that can be observed nowadays.

Analysis of the network traffic in real-time, with high runtime efficiency: The NIDS is required to detect network attacks as fast as possible. Therefore it must read the traffic of a network with a low delay and be able to scale adequately if the traffic volume grows.

High detection accuracy and a low false detection rate: Being a typical flaw of anomaly-based NIDS, special attention should be given to the false detection rate. A high false detection rate would put the network administrator under too much work, which renders the system unusable in practical settings.

Interpretability of the system's reasoning: The administrator should be able to understand why the system classifies network traffic as benign or as an attack. [37, p. 23]

As described later in section 3.4, it is hard to find works that take most of the mentioned requirements for a practical anomaly-based NIDS into consideration, when comparing or evaluating them. This founds the motivation of this thesis, whose goals are described in the following.

1.2. Goals

Motivated by the lack of adequate solutions, the goal of this bachelor's thesis is to provide a framework dedicated to objectively evaluating and comparing approaches to anomaly-based NIDS under practical aspects. It should take into account some of the mentioned requirements, in particular: *(i)* dataset independence, *(ii)* no reliance on the existence of class labels for the training dataset, *(iii)* a focus on modern attack types, *(iv)* the evaluated systems' runtime efficiency, and *(v)* their false detection rates (alongside other performance measurements). An important step in preparation for this is creating comparability between already-existing datasets that can be used for evaluating NIDS. Furthermore, the architecture of such systems must be characterized in a generalized way. With the usage of the developed framework, some commonly used methods for anomaly-based NIDS are then implemented and analysed.

Despite interpretability being an important issue for real-world usage, it should not be included in the scope of the thesis, and rather serve as a starting point for further work on this topic. Furthermore, domain-specific knowledge about network protocols is not incorporated to a great extent: Except for IP and TCP, most protocols that are highly relevant in practice are not handled extraordinarily in this thesis, such as HTTP, FTP, or SMTP. This aims to provide a general-applicable way for comparing NIDS methods, without relying too much on specific protocol details.

1.3. Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 starts with theoretical background information about network traffic and attacks. It then discusses traits of anomaly-based NIDS and directs particular emphasis on different learning methods for them, before going into detail about one-class support vector machines and autoencoders, two frequently used methods for the detection of network anomalies. Afterwards, the chapter lists measurements for the evaluation of such systems and selection criteria for datasets suited for it. Based on those, some of the publicly available datasets are discussed, resulting in the choice of two (CIC-IDS-2017 and UNSW-NB15), which are used in the following chapters.

Chapter 3 then summarizes the current state of research on anomaly-based NIDS employing autoencoders and one-class support vector machines, as well as on the utilization of traffic payload analysis for anomaly detection. It ends with a short overview of other works that have a similar goal as the present thesis (i.e. comparative experiments with different approaches to anomaly-based NIDS), albeit mostly with some kind of shortcoming.

Chapter 4 proposes an evaluation framework that fulfils the aforementioned goals. After defining requirements for creating comparability between different datasets, it shows a generalized architecture for anomaly-based NIDS and discusses each of the phases (training, classification and evaluation) such systems need to run through in order to compare them.

The proposed concept is then made concrete in chapter 5, where specific implementations for the components of anomaly-based NIDS (called *feature extractors*, *feature transformers*, and *decision engines*) are presented. It also goes into detail about the preprocessing of the chosen datasets and problems that occur there.

Afterwards, the described approaches are compared against each other in chapter 6. It contains a description of the experiments that were run as part of this thesis and a discussion of their results. Finally, chapter 7 ends with a summary of the thesis and recommendations for further work.

Chapter 2 | Background

2.1. Network Traffic

2.1.1. Relevant Network Protocols

Communication over networks is based on a set of commonly established protocols. With the Internet Protocol Stack, they can be grouped into five layers, which are shown in figure 2.1: the *physical layer*[2], the *link layer*, the *network layer*, the *transport layer*, and the *application layer* [35, p. 50]. Due to their practical relevance, the *Internet Protocol* (IP), the *Transmission Control Protocol* (TCP), and the *User Datagram Protocol* (UDP) are explained more deeply in the following.[3]

Figure 2.1.: The Internet Protocol Stack with examples for each layer.

The **Internet Protocol (IP)** is part of the network layer and handles the exchange of packets between two hosts [35, p. 308]. Hosts are identified by *IP addresses*. With them, it is possible to transfer data globally to other connected hosts. In practice, however, these addresses are not uniquely assigned: The actual host that is identified by a particular address depends on the subnet. The task of directing network data to the targeted host inside a subnet or to another subnet is done by routers. The network of hosts connected via the Internet Protocol is called the Internet.

There are two versions of IP in use, the older IPv4 and the more recent IPv6 which is meant to replace the former, providing a bigger set of possible addresses. The segments of an IPv4 packet

[2]Some authors omit the physical layer completely, which makes the protocol stack have only four layers. Here, the model as described in [35] is referred to. In fact, most of the technologies used in the link layer provide protocols for the physical layer, which is why a clear differentiation is not possible.

[3]As application layer protocols, such as HTTP or DNS, are not focused on later in this thesis, they are not described in this section. The same applies to protocols of the physical and the link layer, which are here only needed as a means of transporting IP packets.

Name	Length	Description
Version	4 bits	the version of the IP packet
Header Length	4 bits	the number of 32-bit segments in the packet's header
Type of Service (TOS)	8 bits	information about the type of the packet's data; used to distinguish real-time and non-real-time traffic
Datagram Length	16 bits	total length of the IP datagram
Identifiers	16 bits	used for identifying connected fragments of one IP packet
Flags	3 bits	control options that are used for fragmentation
Fragmentation Offset	13 bits	used for reassembling multiple fragments of an IP packet
Time-to-live	8 bits	a counter that gets decremented by one each time the packet passes a router in the network; to prevent endless circulation inside a network, the packet is dropped when this value reaches zero
Protocol	8 bits	uniquely identifies the transport-layer protocol of the IP packet's payload (e.g.: 6 for TCP and 17 for UDP)
Header Checksum	16 bit	computed by using 1s complement arithmetic; helps to detect bit errors in the header
Source IP address	32 bit	IP address of the sender
Destination IP address	32 bit	IP address of the recipient
Options	variable	placeholder for rarely-used options
Data / Payload	variable	the transport-layer data that is to be delivered by the IP packet

Table 2.1.: IP packet segments in IPv4

are shown in table 2.1 (cf. [35, p. 333f.]). All of the segments displayed there, except the payload, belong to the *IP header*.

The **User Datagram Protocol (UDP)** uses IP packets for transporting data in a connection-less and unreliable way (i.e. there is no guarantee that a message is delivered). Nevertheless, it provides a simple integrity verification for the sent data and the specification of **source** and **destination ports**, which allows the association of a message to a process at the respective host. Altogether, UDP adds a very small header overhead of only 8 bytes to the packet. It is therefore often used for multimedia streaming, internet telephony, and for routing protocols [35, p. 198ff.].

Another frequently-used transport layer protocol is the **Transmission Control Protocol (TCP)**. In contrast to UDP, it is considered *reliable*, which means that it ensures that messages are correctly delivered by using error detection, retransmissions and header fields for sequence and acknowledgement numbers. Furthermore, it is connection-oriented, i.e. before the actual message is sent, both communicating hosts are performing a *three-way handshake* in order to open a connection. This is done by alternately sending TCP packets between the two communicating hosts which have certain header bytes ("flags") set. More precisely, the initiating host sends a packet with the `SYN` flag set, to which the targeted host answers with a packet with the `ACK` flag, indicating that it has received the packet. The first host then sends a packet with both the `SYN` and `ACK` flag ("`SYN-ACK`"), which indicates that the connection is established on both sides. Later it can be ended with the `FIN` flag, or be reset using the `RST` flag.

TCP has a bigger header overhead than UDP and is used for many application-layer protocols, such as SMTP, HTTP, and FTP.

2.1.2. Network Flows

It is often desirable to have insight into the characteristics of a network, e.g. for ensuring its well-functioning, for describing its utilization or for detecting malicious behaviour. To achieve this, the network traffic can be monitored. There are two main strategies for doing so: (1) capturing the raw packets or (2) the identification of network flows [54, p. 2]. When **capturing the full packets**, all information of a traffic record is saved, in particular the packet's payload. Tools like `tcpdump`[4] or `Wireshark`[5] are able to do this. They store the traffic information in the so-called *PCAP* or *PCAP-NG* formats.

For **network flows** on the contrary, multiple packets are grouped. A flow aggregates IP packets that lie close together time-wise and have the same set of attributes, like source and destination port, IP addresses, and protocol type. For performance reasons, network flows might disregard a packet's payload, and instead identify other, statistical, characteristics of the whole flow (like the total numbers of packets or the overall flow duration, i.e. the time between the first and last packet of a flow). Flows can be either uni-directional or bi-directional, according to whether the packets are directed only from host A to B, or also from B to A [48, p. 3].

One of the first mechanisms of this type was *NetFlow*, which was introduced to Cisco routers in 1996. Here, the ending of a flow is based on: *(i)* the existence of TCP flags which indicate the end of a connection (`FIN` and `RST`), *(ii)* a flow timeout of 15 seconds, *(iii)* a maximum flow time of 30 minutes, and *(iv)* the size of a router's flow cache [19, p. 2]. For capturing the traffic information, it is monitored in multiple points of the network. The observed information is then transmitted via UDP to a central machine that handles the flow creation and analysis. Open-source tools such as *argus*[6], *zeek*[7] and *CICFlowMeter*[8] provide similar functionalities.

It is possible to generate network flows from raw packet data by grouping the stored packets based on their attributes. The reverse direction, i.e. determining which packets belong to a flow, is more complicated and requires the full packet information to be existent in either the network flows or as an additional data source (e.g. a PCAP file).

2.2. Network Vulnerabilities and Attacks

2.2.1. Definitions

Network vulnerabilities are "inherent weaknesses in the design, implementation and management of a networked system" [6, p. 46]. A **network attack** is a "sequence of operations that puts the security of a network or computer system at risk" [6, p. 52], by exploiting a vulnerability. It should be noted that it is not always possible to differentiate between accidental malfunction due to software bugs or human misconduct on the one side, and intentional network attacks on the other side, as they might appear similar on a technical level (e.g. when network packets are observed). Network attacks are usually conducted with the goal of compromising a system's availability, confidentiality, or integrity. They can roughly be divided into passive and active attacks, based on the behaviour of the attacking subject [18, p. 18]. An attempt at a more detailed classification is shown in the next section.

[4] `https://www.tcpdump.org/`
[5] `https://www.wireshark.org/`
[6] `https://openargus.org/argus-ids`
[7] formerly named *bro-ids*, `https://zeek.org/`
[8] `https://github.com/ahlashkari/CICFlowMeter`, [13]

2.2.2. Types of Network Attacks

By classifying network attacks and defining a taxonomy, it is aimed to find a consistent way for specifying relationships between different attacks and for deciding which attacks share similarities or are different from each other (cf. [6, p. 52]). However, there are various network attack taxonomies proposed in literature [27, p. 3], and most attacks cannot clearly be put only into one category, which complicates the idea of a hierarchical structuring. The following attack categories are selected from the taxonomies described by Hansman [26], Bhuyan et al. [7, p. 44] and Bhattacharyya et al. [6, p. 52].

Denial-of-Service (DoS) Attacks: A DoS attack tries to make a network service unavailable for its legitimate users. This can be done, for instance, by deliberately causing the victim's system to crash, e.g. by exploiting a buffer overflow vulnerability (such as done by the *Ping of Death* attack) or by overwhelming its resources (*TCP SYN flood*). In the case that the attack is performed from multiple unique IP addresses, it is called a **Distributed DoS Attack (DDoS)**. An ongoing DDoS attack therefore cannot be stopped by simply blocking a single IP address.

Password Attacks: Here, an attacker aims to obtain credentials, by trying several user-password combinations. Commonly used approaches are dictionary attacks or brute-force guessing. For instance, the attacks can target website accounts or SSH logins. A password attack often is used for gaining access to a remote system, and then can also be called a **Remote to Local Attack (R2L)** [6, p. 54].

Network-Based Attacks: Attacks in this group have in common that they are carried out foremostly on the network infrastructure, especially on its underlying protocols [26, p. 19]. Examples are **spoofing** and **session hijacking**. Attacks against services on the application layer of the TCP/IP stack are called **web application attacks**. The most famous of those are SQL injections: By injecting SQL statements in data-driven services, code with malicious intent will be involuntarily executed [26, p. 21f.].

Infection Attacks: Such attacks aim to install malicious programs on the target's system, such as **worms**, **viruses**, or **trojans**. The first-mentioned both are self-replicating programs, which can cause harm by themselves or carry out other attacks. By replicating themselves, they carry out additional infection attacks. The main difference between the two is that worms can function independently, whereas viruses need a host program to run. Trojans are trustworthy-looking programs which in reality serve a malicious use case, like the collection of personal information.

User to Root (U2R): In this scenario, the attacker already has remote access to a system, but only limited. By exploiting an additional vulnerability, they are able to gain more permissions on the system than they are supposed to have. [6, p. 54]

Information Gathering Attacks: These attacks are supposed to collect information about a system or its data, without causing any direct harm. An example is scanning a host for open ports (**probing**). Another way to obtain information are passive attacks, by **sniffing** or **eavesdropping**: Here, the network layer is attacked and afterwards, the data sent on it can be intercepted passively.

Attacks from one or more categories are often combined into a **blended attack**. As an example, *Code Red* is a worm that accomplishes a DoS attack by carrying out a buffer overflow attack [26, p. 25].

2.2.3. Zero-Day Attacks

Zero-day attacks are network attacks exploiting a vulnerability that has not been disclosed publicly yet [9, p. 1]. A study by Bilge and Dumitras concluded that zero-day attacks are far more common than usually believed: More than half of the vulnerabilities that were identified in their work were yet unknown [9, p. 3]. Public disclosure of a network vulnerability is either done by the vendor of the affected system or by individuals. Usually, the vulnerability is then incorporated into the database of the *Common Vulnerabilities and Exposures* (CVE) consortium[9]. Once a signature for a signature-based NIDS is released, the attack can then also be detected by such systems, often in a much more targeted way than by anomaly-based ones. Before such a signature is released, however, a signature-based NIDS will likely fail to recognize the attack; this makes an anomaly-based approach the more promising detection method for the time between the attack's first launch and the release of a signature matching it. Figure 2.2 shows typical events that occur after a new vulnerability is introduced.

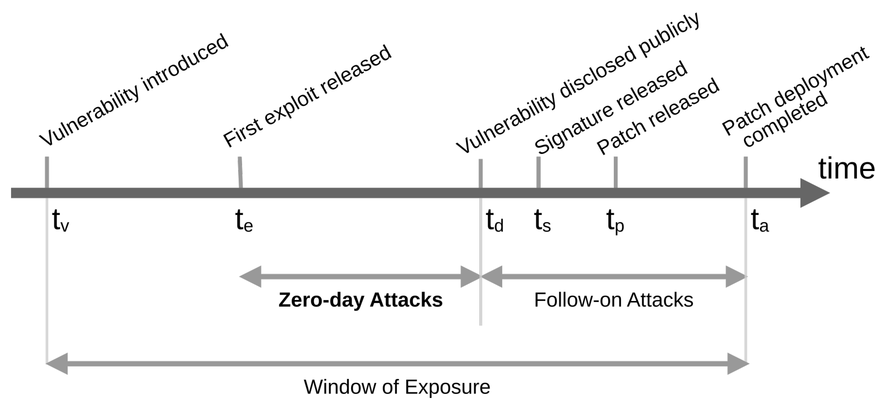

Figure 2.2.: Typical timeline of a vulnerability, adapted from [9, p. 3].
A signature-based detection mechanism is effective for $t \geq t_s$. For $t_e \leq t < t_s$ an anomaly-based approach can be expected to have more success in detecting a zero-day attack in general.

2.3. Anomaly-Based Network Intrusion Detection

2.3.1. Types of Anomalies

Network anomalies are instances of network traffic that do not conform with an acceptable notion of what is considered normal behaviour (cf. [6, p. 45]). It is important to point out that not all anomalies are caused by network attacks, but might arise due to different reasons, like a network overload or misconfiguration. The objective of an effective anomaly detection mechanism in NIDS must therefore be to only find malicious anomalies. However, as the intent of network traffic is not always accurately determinable, this goal might not be completely achievable in practice. In general, anomalies are classified into three principal groups [1, p. 22]:

Point Anomalies: This category contains anomalous instances in respect of the rest of the data. For example, a single login attempt to an online banking account can be classified as a point anomaly if the requested user name contains suspicious keywords (like `SHOW` or `return`), but normally would consist of the customer's ID number.

Contextual Anomalies: Some instances are only anomalous within their context (e.g. within a

[9] `https://cve.mitre.org`

certain time range). An example is a high traffic load in a company network at 2 a.m., which is an anomaly for this particular time, but would be normal during working hours or in the evening.

Collective Anomalies: The third category are groups of instances that collectively behave anomalous, with respect to the rest of the data. One example of this is a botnet consisting of different hosts with distinguishable IP addresses, performing a DDoS attack. Another reason for a collective anomaly could be legitimate web users with uncommon browsers that result in an anomalous HTTP user agent. This is also an example of a non-malicious, i.e. benign, anomaly.

2.3.2. Components of Anomaly-Based NIDS

As mentioned earlier, an anomaly-based NIDS aims to identify anomalies within a network in order to detect attacks. Moustafa et al. identify four components of such systems (cf. [43, p. 37f.]):

Data Source: The NIDS must be able to read traffic from one or more network devices or have similar access to their data (like reading captured PCAP files). In addition, for the system's evaluation it must be possible to read from a dataset for which the ground truth (i.e. the information whether a traffic record is an attack or not) is known (see section 2.5).

Data Preprocessing: The aim of this step is to provide a numeric representation of the input data that can be used by the subsequent decision engine. Feature creation, feature reduction, feature conversion, and feature normalisation are the main parts of this stage:

- **Feature creation** describes the generation of numeric data instances on the basis of the raw network traffic. This can imply the creation of contextual aggregations, in particular network flows (see section 2.1). The created data aims to describe certain traits (*features*) of the observed traffic.

- **Feature reduction** is the task of lowering the dimensionality of the feature space (i.e. the number of features), for example by compression or by removing irrelevant features, so that a better classification performance can be achieved. This can be achieved manually or by running a series of tests with different subsets for the features and analysing their impact.

- **Feature conversion** is the mapping of features from one data type to another type. Features can often be found in either numerical or categorical (symbolic) form, whereas many decision engines work with numerical data only. Therefore, categorical features (e.g. the protocol type of a network packet) must be converted to a numerical representation.

- **Feature normalisation** means to apply a function on the data instances for scaling their features into a confidence interval (e.g. $[0, 1]$ or $[-1, 1]$). It is used for removing a potential bias from the captured network data. Moustafa et al. name the *z-score* and linear transformation as common functions that are utilized in this step.

Decision Engine: The term *decision engine* refers to the module that is responsible for classifying traffic into benign behaviour and attacks. There are two phases that the decision engine runs through: In the training phase, a normal profile is established, which is then used in the classification phase[10] in order to recognize attacks within new network traffic. There are several approaches to designing an anomaly-based decision engine. They can be categorised into classification-based, clustering-based, deep learning-based, knowledge-based, combination-based, and statistical-based approaches.

Response Mechanism: Based on the use case of the NIDS, an attack detection should cause an alert that can be viewed by a security administrator or initiate automatic defence measures.

[10] also called testing phase

Moreover, the NIDS should provide an interface that allows reading the classifications it made.

2.3.3. Comparing Learning Methods for Network Traffic Anomaly Detection

As described in the last section, for automatically distinguishing attacks from benign traffic, a decision engine must be trained. Before delving into different approaches to it in the later sections, the following discusses which training paradigm should be followed. In literature, learning methods are often classified in supervised, unsupervised, and semi-supervised approaches [6, p. 123f.]. For anomaly detection however, usually *one-class classification (OCC)* methods are proposed, which cannot exactly be put in one of those categories. The most notable difference between the aforementioned is, that algorithms of the first three categories are usually used for two-class classification, which means that they use both attacks and benign traffic in the training phase, whereas OCC approaches only have access to benign ("normal") instances. To show that the latter is in fact the only plausible paradigm for the detection of arbitrary novel attacks, the differences between the four mentioned concepts and their applicability in practical settings are discussed in the following.

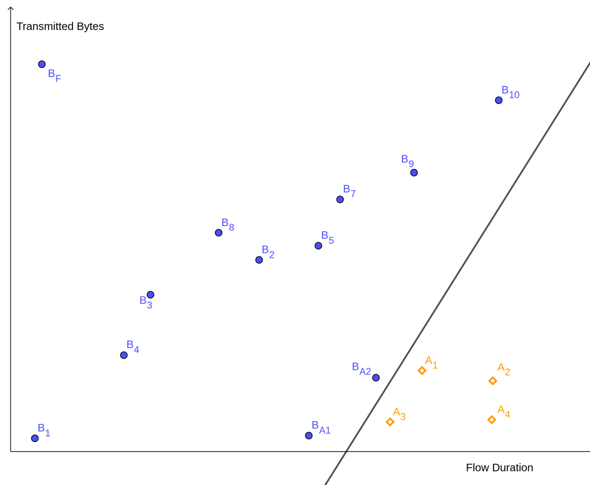

Figure 2.3.: Schematic representation of a training dataset for a supervised learning algorithm
It is known to the classifier whether a flow belongs to an attack (diamond symbol) or represents normal traffic (circle). The black line outlines a possible classifier that divides the dataset into two classes. The features depicted here base on a grouping of the observed packets into flows (cf. 2.1.2) and represent the transmitted bytes and the duration of each flow. The most benign data instances (B_1-$B_1$0) can be roughly described by a linear function; however, there are three benign outliers within the dataset: B_F origins from a transfer of a big file (which transmits a big amount of data in a short time), B_{A1} and B_{A2}, on the other hand, could result from technical anomalies, which causes the loss of packets and a delay in time. The attack flows A_1-A_4 could be distributed flooding attacks, causing a flow to be of long duration but not transmitting much data.

Supervised learning refers to machine learning algorithms that have access to the ground truth, i.e. the correct class labels, during their training phase. For NIDS, such approaches would therefore require the existence of a labelling, which determines whether a portion of network traffic belongs to an attack or is of benign nature. Creating accurate class labels is a difficult task [6, p. 193] and requires a big amount of human resources, which makes such methods infeasible from a practical point of view. Fig. 2.3 shows a supervised classifier for a two-dimensional (and for illustration purposes strongly simplified) feature space.

Unsupervised learning on the contrary does not require already-labelled data for the training phase. Instead, its goal is to find existent classes or patterns in it independently. Examples of unsupervised algorithms are k-means or hierarchical clustering and the local outlier factor. In

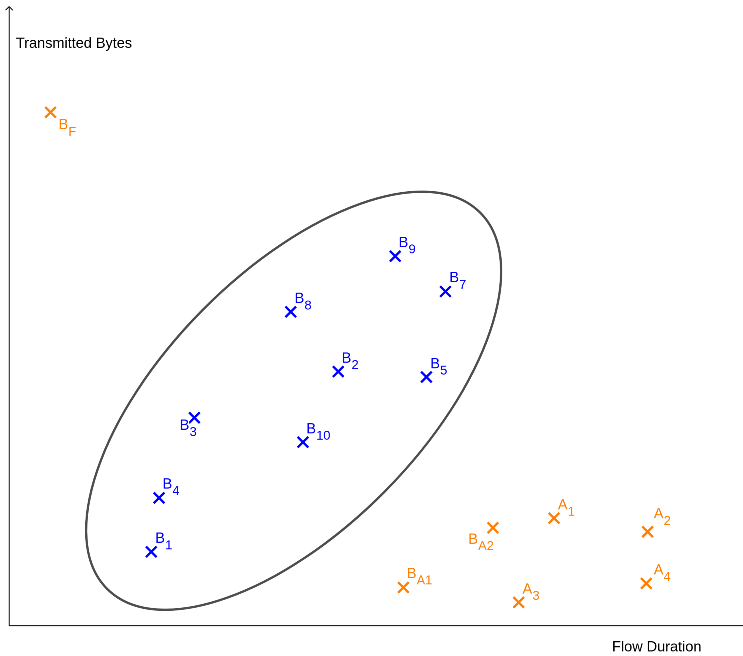

Figure 2.4.: Training dataset for an unsupervised learning algorithm
The ground truth is not known to the classifier (depicted as crosses), therefore it may classify benign outliers as attacks. The pictured boundary is only one of the possible solutions, whose determination here is not as clear as in a supervised setup: For example, an unsupervised algorithm could also label the right instances as a *benign* group, leaving only one outlier in the upper left corner.

general, these methods are designed for data that contains two or more classes, and therefore, in the scope of this thesis, the term *unsupervised* refers to such approaches. For the case of detecting patterns in a training dataset with only one class, *one-class classification* is used, which is explained below.

A problem with unsupervised methods is that they have to distinguish attacks and normal network traffic within the data autonomously, in order to detect incoming attacks. As shown in figure 2.4, this process can be error-prone if the only heuristic for recognizing attacks is their level of abnormality, in particular, if the classifier is trained with traffic data that contains attacks.

Semi-supervised learning can be seen as a mixture of both supervised and unsupervised learning. It incorporates a few labelled and much more unlabelled instances in the training process [66, p. 3], as shown in figure 2.5. This reflects practical circumstances where the labelling of a few network traffic records can be done manually. It should be noted that the term *semi-supervised* is sometimes also used for one-class classification, which is explained below [6, p. 123].

As a consequence of having access to attack instances, the paradigms described so far more easily learn the characteristics of attacks included in the dataset (especially when those are labelled as such). While this could be a desired effect for signature-based NIDS, it negatively affects the system's ability of detecting unknown attacks, as stated by Zhao et al. [65, p. 2]. As the characteristics of zero-day attacks cannot be known in advance, neither can be predestined their distribution in the feature space. Attacks that are included in the training set must therefore not necessarily be similar to novel attacks.[11] In fact, the opposite could be true, as the following example illustrates: If a classifier learns to only identify UDP flood attacks (a subtype of DoS attacks), it can be expected to not recognize a SQL injection that exploits a vulnerable login mask

[11] However, there are approaches that apply transfer learning methods for detecting zero-day attacks based on their resemblance to known attacks [65]. It can be argued though, that such methods introduce a bias, which decreases the system's performance for detecting attacks that are not similar to the observed ones.

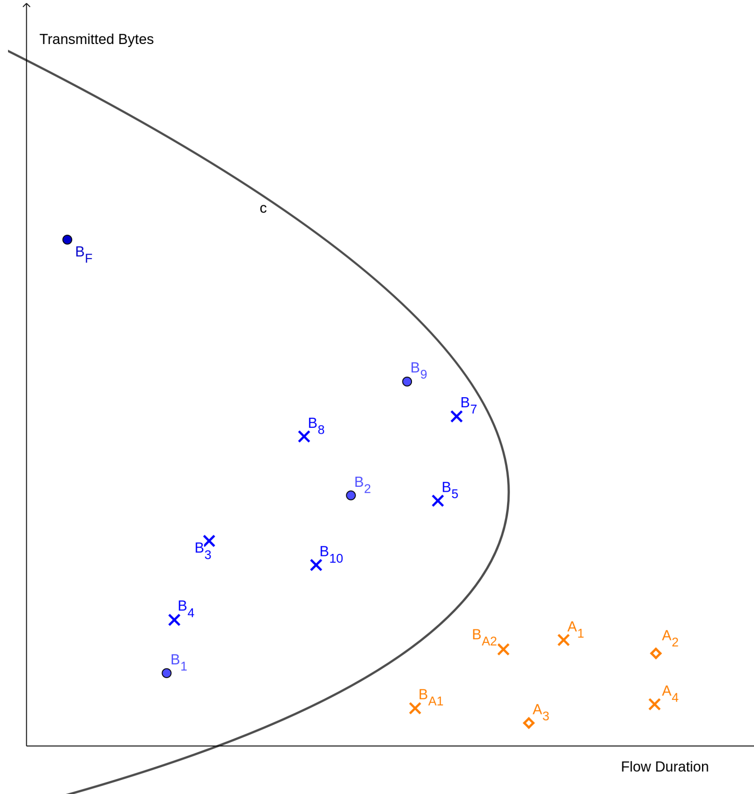

Figure 2.5.: Training dataset for a semi-supervised learning algorithm
The ground truth is only known partially: known labels are represented by circles (benign) or diamonds (attacks), unknown instances by crosses. The black line shows a possible classifier operating on such a dataset. The benign flows B_{A1} and B_{A2} are miss-classified due to their labels not being known and being close to the attacks A_3 and A_2.

on a website. UDP flood attacks can be identified by a large number of sent UDP packets, whereas the malicious part of the SQL injection could be contained in the payload of one single packet. To the classifier, it therefore appears more similar to other, benign traffic (like a normal login attempt to the website), than to the already known UDP flood attacks.

To avoid such problems, **one-class classification (OCC)** can be used. It is also referred to as *outlier detection*, *novelty detection*, *concept learning* or *data description* [56, p. 13f.]. In contrast to the former methods, OCC algorithms are designed for datasets with exactly one class: In the domain of network intrusion detection, the training set consists of benign traffic only (cf. 2.6). By imposing the condition on the training dataset to only consist of benign traffic, the classifier will, in theory, not involuntarily learn the characteristics of any attacks. One-class classification algorithms don't focus on recognizing patterns of already known attacks (as those are not present in the training phase), but rather on characterizing the behaviour of benign traffic. Attacks are later reported by finding anomalous behaviour in reference to the normal profile.

In practice, this should facilitate the dataset's creation compared to supervised methods, because benign traffic is expected to be present more frequently. Whether or not it can be assured that no attacks are included in the training set depends on the network and available resources or manpower. However, keeping in mind certain practical circumstances that result in inaccurate data, it is possible that attack traffic still gets included in the training set. In this case, the OCC algorithm should still assume that it deals with pure benign traffic. An important question is then how well the classifier performs both on unseen attacks and on such that are similar to the ones included in the training set. On the other side, the algorithm must also handle atypical, but benign instances.

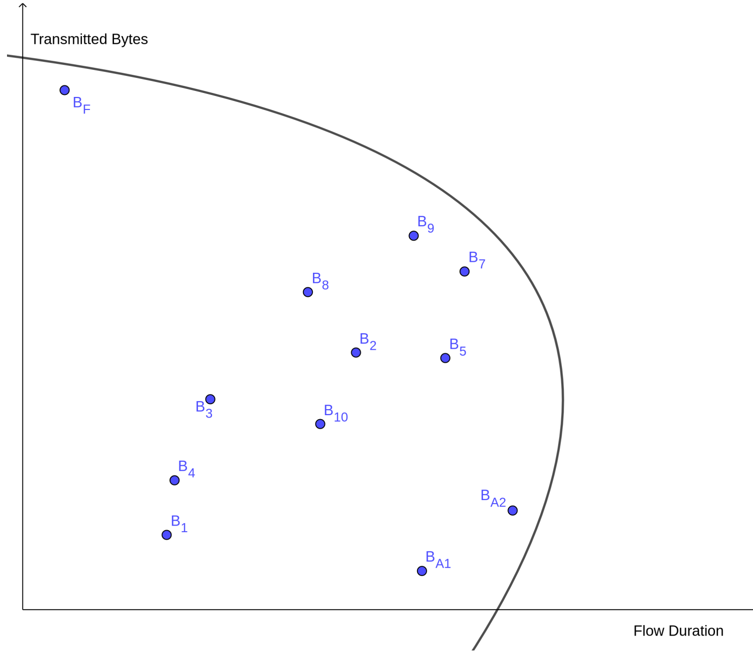

Figure 2.6.: Training dataset and possible boundary for a one-class classification learning algorithm It only consists of normal traffic, and this information is known to the classifier. All instances that are on the right side of the boundary will be determined as anomalies in the classification phase.

	Training Set		
Training Method	Benign Traffic	Attacks	Labelled
Unsupervised	Yes	Yes	No
Supervised	Yes	Yes	Yes
Semi-Supervised	Yes	Maybe	Partly
One-Class Classification	Yes	No	Only benign traffic (implicitly labelled)

Table 2.2.: Relationships between learning methods and requirements for their training dataset

Furthermore, an issue OCC methods must handle is the proper generalization of the training data. This means that it must be decided which features are best-suited for separating benign from normal traffic and how close the boundary should fit the observed instances. Another problem that OCC methods face is the curse of dimensionality [56, p. 15]. It describes the phenomenon that a classifier's accuracy might decrease in a high-dimensional feature space. This is due to the feature space volume growing exponentially in the number of features, which makes the training data more sparse and the classifier more prone to overfitting [10, p. 33ff.].

Popular OCC approaches to anomaly-based network intrusion detection are isolation forests, one-class support vector machines, and autoencoders, of which the latter two are described in the following sections.

At the end of this section, the presented training paradigms are compared regarding their conditions for the selection of a training dataset. These relationships are illustrated in table 2.2 and should be considered when choosing a dataset for the training of an attack detection model. Conversely, the structure of already-existing data influences the criteria for a suitable classification algorithm.

2.3.4. One-Class Support Vector Machines

The application of support vector machines to one-class classification problems was proposed by Tax et al. [56] under the name *Support Vector Data Description* (SVDD) and by Schölkopf et al. using a *ν-SVM* [50]. SVDD attempts to find a hypersphere as a boundary around a set of training points. Points lying on this boundary are called support vectors. The volume of the hypersphere is minimized with the constraint that all or most of the points are inside the boundary. A ν-SVM on the other hand tries to find a hyperplane that separates the points from the origin with maximum margin [63, p. 75]. The tradeoff between the fraction of training errors and data points that lie within the boundary can be influenced by a parameter, which is called ν ("nu") for a ν-SVM [45, p. 2].

As in general it cannot be expected that the training data is well-suited to fit into a hypersphere (resp. is separable by a hyperplane), both of the mentioned methods establish a mapping $\phi(x_i)$ of a data point x_i into a higher-dimensional feature space, where a better-fitting hypersphere (resp. a hyperplane with a greater margin) can be found. However, the actual calculation of this mapping is avoided; instead, the so-called *kernel-trick* is made use of. It describes the fact that only inner products of the $\phi(x_i)$-mappings must be calculated, which can be replaced by a *kernel function* that avoids cost-intensive calculations in the higher-dimensional feature space [56, p. 29].

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \tag{2.1}$$

The following kernel methods are available in the implementation of the python library `scikit-learn` [51], which is also utilized in this thesis (cf. 5.5.1):

Gaussian kernel: The Gaussian kernel function, also called *(Gaussian) radial basis function* (rbf), is defined by

$$K(x_i, x_j) = e^{-\gamma ||x_i - x_j||^2} \tag{2.2}$$

(whereby the parameter γ is sometimes expressed as the reciprocal of another parameter, e.g. by $\frac{1}{c}$ [50, p. 4] or $\frac{1}{s^2}$ [56, p. 44]). $||x_i - x_j||^2$ denotes the squared Euclidean distance between two instances. This kernel has the advantage that its mappings are always separable from the origin [50, p. 13] which is why it is widely used in practice. Furthermore, SVDD and ν-SVMs are equivalent when the Gaussian kernel is used [63, p. 75].

Polynomial kernel: This kernel implicitly applies a polynomial function of degree n to the data points and calculates their inner product as follows [56, p. 30]:

$$K(x_i, x_j) = (x_i \cdot x_j + 1)^n \tag{2.3}$$

A more generalized formula is implemented by `scikit-learn`:

$$K(x_i, x_j) = (\gamma(x_i \cdot x_j) + r)^n \tag{2.4}$$

where γ, r and n are free parameters.

Sigmoid kernel: The sigmoid kernel is defined by:

$$K(x_i, x_j) = \tanh(\gamma(x_i \cdot x_j) + r) \tag{2.5}$$

with the free parameters γ and r.

Linear kernel: The linear kernel is the most simplest kernel and defined as:

$$K(x_i, x_j) = x_i \cdot x_j \tag{2.6}$$

A characteristic of the one-class SVM is that it does not take into account the density estimate of the training data [56, p. 19]. Instead, it suffices to have an acceptable number of characteristic points as part of the data which define the boundary.

2.3.5. Autoencoders

Autoencoders are feedforward neural networks[12] with at least one hidden layer that are trained to reconstruct the input layer. While the network is trained, a self-taught encoding emerges that the network uses for transforming the input to the central hidden layer and to decode it reversely. Accordingly, the layers of an autoencoder can be divided into an encoder and decoder. Training the autoencoder is done by adjusting its weights w in order to minimize a loss function which expresses the reconstruction error. An often-used metric for that is the *mean squared error* (MSE) between the m training vectors x_i and their reconstructed outputs $f(x_i, w)$ [25, p. 132]:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} ||x_i - f(x_i, w)||^2 \tag{2.7}$$

Other metrics can take into account properties like the sparsity of the representation or robustness to noise and missing inputs, in which case the term *regularized autoencoders* is used. It should be noted that there are also *variational autoencoders* (VAE), which however do not reproduce the input layer in the output layer and are in fact generative modelling approaches [25, p. 501]. Due to this fundamental difference, they are not considered in this thesis when the term *autoencoder* is used, even if they are reported to be useful for anomaly detection [2].

Being feedforward neural networks, autoencoders inherit their common traits and functionalities, such as not having a circular architecture and being able to be trained with the back-propagation algorithm [25, p. 200]. Furthermore, usually an autoencoder employs an activation function that is applied to the input of each neuron. By using a non-linear activation function, the network is able to calculate non-linear functions as well. Typical activations functions are listed in the following.

relu: The *rectified linear unit* x^+ is a very simple non-linear function defined by

$$x^+(x) = max\{0, x\}. \tag{2.8}$$

It is a commonly chosen activation function due to being nearly linear, which makes the model's optimization easier with gradient-based methods and also lets it generalize better, according to [25, p. 170f.].

softplus: This function produces a "smoothed" version of the relu [25, p. 66]:

$$\zeta(x) = log(1 + e^x). \tag{2.9}$$

sigmoid: The logistic sigmoid function [25, p. 65] is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.10}$$

Notable characteristics of this function are that it outputs lie within the interval $(0, 1)$ and that it "saturates when its argument is very positive or very negative, meaning that the function becomes very flat and insensitive to small changes in its input" [25, p. 66].

tanh: The *hyperbolic tangent* activation function is defined by

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{2.11}$$

It is related to the sigmoid function and is in fact reported to typically perform better than it, as it is more similar to the identity function near the origin ($tanh(0) = 0$, wheras $\sigma(0) = 0.5$) and facilitates the autoencoder's training [25, p. 192].

[12] a.k.a. *multilayer perceptrons* or *deep forward networks* [25, p. 164]

A common use case for autoencoders is dimensionality reduction of a large feature space. More important for the anomaly detection domain however is the possibility to detect data that is different to the training profile based on the reconstruction error the autoencoder is committing: If the output is significantly different from the input, the assumption is made that the autoencoder was not trained with this type of input. For determining when this case occurs, a threshold on the reconstruction error can be defined [2, p. 4].

An important parameter that has to be determined for an autoencoder is its architecture, which means its number of layers and the number of neurons for each layer. Based on this, autoencoders can be grouped as follows (cf. [25, p. 500ff.]):

Undercomplete autoencoders: In this case, the learned feature representation (the *code*) has a lower dimensionality than the input. The autoencoder therefore must learn an effective representation of the input in order to reconstruct it properly. If only one hidden layer together with MSE is used, the emerging encoding resembles the Principal Component Analysis (PCA) [56, p. 78]. With an increasing number of hidden layers, which can model more complex non-linear functions, a more powerful generalization of PCA can be found by the autoencoder [25, p. 501].

Overcomplete autoencoders: Here, the learned input representation has a higher dimension, which is greater than in the input itself. A regularization for the code, e.g. regarding its sparsity, is then needed, as described previously. When only using a loss function like MSE, it is not guaranteed that the code contains anything useful about the input data distribution, as it can simply copy the input, making the learned encoding function trivial.

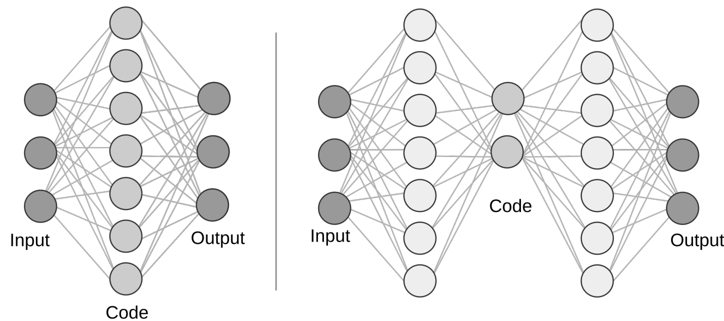

Figure 2.7.: Schematic illustration of an overcomplete (on the left side) and undercomplete autoencoder (on the right side)
The central layer respectively contains the learned representation of the input data (*code*).

Autoencoders are considered *deep* autoencoders in this thesis when they have three or more hidden layers. As opposed to this, networks with lower depth are called *shallow*. According to the universal approximator theorem, already with one additional layer in the encoder and decoder respectively (resulting in five layers total), an autoencoder can represent any decoding and encoding function arbitrarily well, given enough hidden units. An advantage of increased depth is that the computational costs for some functions and the amount of data needed can be reduced exponentially. Furthermore, the compressions observed in experiments are reported to be better with deep autoencoders compared to shallow ones. [25, p. 505f.]

2.4. Evaluation of Anomaly-Based NIDS

2.4.1. Binary Classification

The evaluation of an NIDS takes place after the decision engine's normal profile was built and trained. It is then faced with yet unseen traffic, whose true nature (i.e. the traffic labels) is only known to the evaluating system. An anomaly-based system labels all anomalous instances that are found as attacks, and all normal traffic as benign. By this equalization, the NIDS' task can be described as a binary classification problem and the following variables can then be determined for describing its performance (cf. [6, p. 236f.]):

True Positives (TP): the number of correctly classified attack instances

False Positives (FP): the number of benign traffic instances which are incorrectly classified as attacks

True Negatives (TN): the number of correctly classified benign traffic instances

False Negatives (FN): the number of attack instances that were incorrectly classified as benign traffic

A confusion matrix as in 2.3 can illustrate the connection between actual attacks and benign traffic and the classifications made by the NIDS.

	Attack Traffic	Benign Traffic
Classification as *Anomaly*	TP	FP
Classification as *Normal*	FN	TN

Table 2.3.: Confusion matrix of the underlying binary classification problem.

2.4.2. Relevant Measurements

Based on the confusion matrix variables, the following indicators for an NIDS' performance can be identified:

Recall is the portion of correctly classified attacks within all existent attacks:

$$Rc = \frac{TP}{TP + FN} \tag{2.12}$$

It is also known as *hit rate*, *detection rate*, *sensitivity* or the *true positive rate*. Recall is important if security matters highly, but on the other hand can be raised very easily by classifying all instances as an attack (which would result in a recall of 100%). Recall alone is therefore no sufficient indicator for an NIDS' performance.

Precision is the portion of correct predictions within the instances that are classified as attacks:

$$Pr = \frac{TP}{TP + FP} \tag{2.13}$$

It reflects the system's confidence of attack detection [37, p. 12], and therefore has a higher priority when efficiency is important.

The **False Detection Rate** or False Discovery Rate reflects the portion of miss-classified benign traffic within the instances that were classified as an attack.

$$FDR = \frac{FP}{TP + FP} = 1 - Pr \tag{2.14}$$

If this value is too high, a lot of false alarms are generated, which is an unwanted effect in practical setups. A significant measure here is also the absolute number of falsely classified traffic instances (i.e. the number of false positives) within a certain time range: a relatively low false-detection rate of 1% could still be manageable for a rarely-visited website, but might already require too many human resources in a university network. [29, p. 16]

The **False Positive Rate** is the portion of miss-classified benign traffic instances within all benign traffic instances.

$$FPR = \frac{FP}{TN + FP} \tag{2.15}$$

The **True Negative Rate** is the portion of correctly classified benign instances within all benign instances.

$$TNR = \frac{TN}{TN + FP} \tag{2.16}$$

Balanced Accuracy is an enhanced measurement for the model's *accuracy*. Accuracy on its own, which is a common metric defined by the portion of correctly classified instances, has the disadvantage that it does not take into account imbalances of the evaluation dataset. For example, an accuracy of 95% would be a desired value when the dataset consists equally of benign and attack traffic, but when the fraction of attacks only amounts to 5%, the classifier would be no better than a naive approach which labels all traffic as benign.

Therefore, instead of accuracy, balanced accuracy can be used. It is defined as the mean of the recall and true negative rate:

$$BA = \frac{Rc + TNR}{2} \tag{2.17}$$

F-Measure o F1-Score is the harmonic mean of recall and precision [17, p. 5]:

$$F1 = 2 \times \frac{Pr \times Rc}{Pr + Rc} \tag{2.18}$$

It is often used for expressing an NIDS' performance as a single numeric value. A potential problem is that it does not depend on the number of true negatives (i.e. correctly classified benign packets) and that it can be misleading for datasets with a big imbalance of attack traffic: Here, a classifier that always classifies traffic as attacks can yield a high f1-score[13]. Although the latter case is not very likely when evaluating an NIDS (because, usually, benign traffic is in the majority, like in the datasets used later in this thesis), it cannot be completely ruled out.

Matthew's Correlation Coefficient (MCC) is defined as (cf. [17, p. 5]):

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{2.19}$$

Its values are within $[-1, 1]$, where $MCC = 1$ describes a perfect classifier and $MCC = 0$ indicates random guessing. It is reported to not be misleading on imbalanced datasets, which makes it preferable to f1-score and accuracy and which is why it is proposed as an replacement for them [17, p. 11 5].

[13] For example, a dataset with 900 attack packets and 100 benign packets would result in a recall of 1 and a precision of 90%, yielding a f1-score of approximately 94.7%.

While f1-score being the most prominent single-score evaluation metric for the overall performance of an NIDS, it might be useful to describe the correlation between two of the described metrics, in subjection to a single modifiable parameter (e.g. a configurable threshold which affects the system's detection ability.). This is the use case for *receiver operating characteristic (ROC) curves*, which illustrate the relation between the false positive rate and the recall. Considering the importance of a low false detection rate, another possibility for visualizing the prediction performance is a precision-recall curve. Precision can be seen as the reverse of the FDR, which therefore can be concluded from it (cf. equation 2.14).

2.5. Datasets for Network Intrusion Detection

2.5.1. Dataset Requirements

There are several datasets available that are dedicated to training and evaluating network intrusion detection systems. However, it is often mentioned that many of them have characteristics that make their usage for the evaluation of practical, anomaly-based NIDS problematic [4, p. 2]. Some of the key challenges that such datasets face are listed in the following (cf. [57, p. 3]).

Realistic Network Traffic: Having real network traffic that is similar to such that can be observed in production-type network setups is necessary for evaluating this type of NIDS. Many datasets however use synthetic or emulated traffic (19 out of 34 examined NIDS datasets in a survey by Ring et al. [48, p. 7]). Emulating traffic means that the dataset authors have control over the traffic generation and capture it using a test bed, while synthetic traffic refers to datasets that were generated artificially and not captured using a network device [48, p. 5]. One reason for this is that self-generated traffic can be labelled more easily. While such datasets aim to simulate real-world scenarios, it is not evident if they actually reflect this kind of traffic.

Modernity: For evaluating its usefulness in current practical setups, the NIDS must be tested with modern, current-day attack types. In the following, the term *modern* refers to datasets that were created in 2015 or thereafter.

Providing Network Packets: Many datasets already come with extracted features based on network flows. This makes it difficult to compare the performance of one NIDS on different datasets, because the underlying feature extraction methods are not consistent: Both flow generation and feature extraction rely on different parameters, like the chosen flow timeout. Packet data (e.g. PCAP files) on the other hand assert that every dataset is presented in the same form to the NIDS.

Completness and Validity: The attack scenarios must be complete and valid, i.e. all traffic that belongs to an attack is captured and the included attacks are properly implemented. On the other hand, no traffic records that are considered "irrelevant" should be omitted in the dataset because this creates a bias and results in a non-realistic dataset.

Existent and Correct Labelling: For evaluating an NIDS, labels are needed which indicate whether a traffic record belongs to an attack or not. Furthermore, it should be ensured that the class labels actually reflect the ground truth. This can be a problem if the ground truth is not known, e.g. when real-world traffic is captured within a honeypot and it is uncertain when and where attacks occur.

Diversity: For being able to assess an NIDS it should be challenged with a variety of different attack types and traffic behaviour. This requirement does not necessarily apply if an NIDS should only be able to identify a restricted set of attacks (e.g. only DDoS attacks), but it is needed

Property Name	Optimal Value(s)	Required?
Year of Traffic Creation	$\geq$ 2015	No
Public Availability	on request, yes	Yes
Normal Traffic	yes	Yes
Attack Traffic	yes	Yes
Format	packet	Yes
Anonymity	none	No
Kind of Traffic	real	No
Labeled	indirect, yes	Yes

Table 2.4.: Relevant properties of datasets dedicated to network intrusion detection

for the evaluation system being developed in this thesis: Its objective is to find a preferably general method for zero-day attack detection. For the usage of one-class classification algorithms, adequate existence of traffic that is both benign and rich in variety is required.

Reproducibility: It should be transparent how the dataset records, i.e. the normal behaviour and attacks, and the extracted features (if existent), were created. This makes other researchers able to adjust the dataset for their needs (e.g. to incorporate new, modern attacks), to fix contingent inaccuracies or to justify its validity.

Privacy-Compliance, without Distortion through Anonymization: For making a dataset public it is important for it to not contain any personal data, or making any kind of inference to such information possible. The usage of anonymization techniques on the other hand could affect the NIDS' performance and should therefore be avoided. For the capturing of real-world traffic this, can usually not be assured due to privacy-relevant information like IP addresses or non-encrypted personal data.

Fulfilling all of those principles is a difficult task, as some contradict or complicate each other (e.g. containing realistic data but being privacy-compliant). For the selection of a suiting dataset, Ring et al. [48, p. 4] identify several dataset properties. A relevant selection of them is shown in table 2.4, together with their optimal value based on the stated requirements above. These eight requirements are not fulfilled by any of the 34 surveyed datasets by Ring et al., but three datasets have accordance with seven of them: CIC-IDS-2017 [53], UNSW-NB15 [44], and CIC DoS [31]. Those datasets are examined in the following sections. Thereafter other commonly-used, but outdated, datasets are shortly looked at, such as DARPA98 and its successors.

2.5.2. CIC-IDS-2017 and CSE-CIC-IDS-2018

The CIC-IDS-2017 dataset [53] was established at the *University of New Brunswick* and the *Canadian Institute for Cybersecurity* in 2017. The traffic was captured over a span of five workdays (Monday to Friday) inside an emulated network environment and contains 3.1 Million network flows. In addition to the labelled flows and raw packet data, it also contains 80 extracted features that can be used for machine learning purposes. The flows and their features were extracted using the tool *CICFlowmeter* (see section 2.1.2). Although the emulated character, the work especially focuses on realistic background traffic. By using scripts it simulates the behaviour of 25 users and uses a variety of protocols, such as HTTP, HTTPS, FTP, SSH, and email protocols. It contains attack types like brute force (SSH and FTP), botnets, (distributed) denial-of-service, heartbleed, infiltration, and web attacks. A characteristic that facilitates the training of one-class classification algorithms is that the *Monday* split only contains benign traffic. The dataset is publicly available.

As mentioned, the dataset provides raw packet data, but no dedicated labelling for it. The packet-

wise labelling must therefore be concluded from the flow labels, based on the packets' IP addresses, ports and timestamps.

The CSE-CIC-IDS-2018 dataset [14] is similar but created at a larger scale, resulting in a total size of approximately 400 GB raw packet data. It isn't used as much yet in other scientific works, but in general seems to be suited equally, because it has the same structure as the CIC-IDS-2017 dataset.

2.5.3. UNSW-NB15

The UNSW-NB15 dataset was published by Moustafa et al. [44] at the University of New South Wales in 2015. It aims to be a modern NIDS benchmark and consists of 2.530.044 traffic records with the following attack types: Fuzzers, Analysis Attacks, Backdoors, Denial-of-Service, Exploits, Worms, and Shellcode Attacks. For the dataset creation, a testbed was established and 100 GB of emulated traffic was captured on two days, for a total time of 31 hours. The raw traffic (in form of PCAP files) was analyzed with the tools Argus and Bro-IDS for creating network flows and extracting features from them. Additional features were generated by applying self-written algorithms. By that way, 49 features per record are available in the dataset, including the class label that indicates whether or not the package belongs to an attack. The dataset is available to the public.

The authors provide two subsets of the extracted features which can be used for training and testing intrusion detection systems. However, the training set contains attack flows, which makes the split inadequate for the anomaly-based methods in this thesis.

2.5.4. CIC DoS

The CIC DoS dataset [31] was created in 2017 for analysing application-layer based DoS attacks at the Canadian Institute for Cybersecurity. Like UNSW-NB15 and CIC-IDS-2017, its traffic is emulated. It partly consists of network traffic from a former dataset (ICSX 2012), but also newly-generated attacks. It has a total time span of 24 hours and contains 4.6 GB of network packets. Due to the limitation on DoS attacks and to some extent relying on older data, it is not further considered here.

2.5.5. Outdated Datasets

Furthermore, there are other, nowadays rather outdated, datasets, which were and still are utilized in many research papers. A popular instance of this is KDD'99. It was created in 1999 as an updated version of a former dataset, called DARPA98 and contains different attack types, which can be classified into the four groups DDoS, Probing, U2R, and R2L. Since its publishing, it was widely used in research and still is. In fact, between 2010 and 2015, its usage was yet increasing [67, p. 4] and between 2002 and 2018 it was utilized by 63.8% of NIDS-related papers [5, p. 21]. Despite this, the dataset is often criticized as it contains redundant data and there is an imbalance between the training and testing sets. To resolve these shortcomings, the NSL-KDD dataset was created as an enhancement, but it still lacks contemporary network traffic due to its age. [43, p. 38f.]

As there are enough dataset candidates that contain modern-day network traffic, those old datasets are not considered within this thesis furthermore.

Chapter 3 | Related Work

3.1. Usage of One-Class Support Vector Machines

Zhang et al. [64] used a one-class support vector machine to detect anomalies on the KDDCUP99 dataset and compared the performance to a probabilistic neural network and a conventional two-class SVM. They reported high detection rates of almost 1 for DoS and probing attacks and overall a high precision and f1-score for the one-class SVM.

Ghanem et al. [23] compared one-class SVMs that utilized linear and Gaussian kernels with two-class SVMs. They created an own synthetic dataset based on wireless and Ethernet LAN traffic with five attack types, of which three are HTTP attacks in wireless networks, one exploits the IEEE 802.11 wireless protocol itself and the last one contains port scanning attacks. The generated features for the Ethernet LAN network are not flow-based; instead, they measured certain metrics like the current throughput or the number and distributions of open ports once per second. While there are good results for the other attack types, the probing attacks only are recognized with a detection rate of 61.37% and a false positive rate of 1.15%.

Winter et al. [62] used a different approach and trained a one-class SVM with malicious traffic that was extracted from a honeypot dataset. Through hyperparameter optimization, they achieved a false detection rate of zero, while only using the source and destination ports, TCP flags and the IP protocol as features. However, different datasets are used for extracting the benign and malicious traffic respectively, which might result in a bias that is not observable in reality. Furthermore, training the one-class SVM with malicious traffic makes the assumption that attacks in the testing phase will be similar to the already-seen ones.

Multiple works point out the inefficiency of support vector machines when handling large and high-dimensional data, and propose a feature dimensionality reduction beforehand. Kuang et al. [34] successfully used KPCA (Kernel Principal Component Analysis) for that; however, the remaining model worked in a supervised manner. Tang et al. [55] used a two-stage approach and utilized the 2-dimensional encoding of a deep autoencoder as the input features for a one-class SVM. For its evaluation, they used the DDoS attacks of the CIC-IDS-2017 dataset and manually selected 13 features from it. In their experiments, they compared the approach with a one-class SVM that uses the unreduced features and reported an improved precision of 99.97% and a recall of 98.28%. Additionally, they could drastically reduce the training and testing times. Still, the plain one-class SVM yielded a precision of 96.26% and a recall of 98.21%.

3.2. Usage of Autoencoders

Besides the already-mentioned approach by Tang et al., there are a few other works that utilize autoencoders for feature reduction. For instance, in a paper by Javaid et al. [30] the features that are extracted by sparse autoencoders are then learned by a softmax regression classifier. They reported an increased F-Measure value when using the autoencoders; however, they only consisted of one hidden unit. An improvement could be to add more depth to the network, as well as using unsupervised classifiers instead of regression.

Other works proposed the usage of (deep) autoencoders for classification, as done by Hindy et al. [28]. The model there is trained with benign instances only, which are split into a training and validation set. Upon facing unknown traffic, the autoencoder recognizes attacks when an instance's reconstruction error is bigger than a given threshold. The particular threshold value, as well as the autoencoder's architecture (i.e. the number of layers and number of neurons for each layer), are hyperparameters and are determined by the random search algorithm. For the CIC-IDS-2017 dataset, the optimal parameters result in a deep autoencoder with three hidden layers, consisting of 18 nodes in the input and output layer, 15 nodes in the first and third hidden layer, and 9 nodes in the central hidden layer.

Mirsky et al. [41] applied an ensemble of autoencoders that can be deployed in a distributed manner and with low runtime requirements. Statistical features are extracted from a stream of network packets using damped incremental statistics. The features are then mapped to smaller subsets which are used as inputs to the ensemble of autoencoders. Each autoencoder has three layers and provides a reconstruction error for its subset. The actual anomaly detection module is another autoencoder whose inputs are the aggregated reconstruction errors of the ensemble autoencoders. It outputs a score that reflects the anomaly of the inspected packets.

Gharib et al. [24] proposed "AutoIDS", which uses two autoencoders that are parallelly trained on normal traffic. The first autoencoder is a sparse autoencoder that uses sparsity as a classification criterion, whereas the second uses the reconstruction error. For the classification of traffic, the sparse autoencoder is queried first, which runs faster. Only the traffic that it is not certain about, is then shown to the second autoencoder, which takes more time, but is more accurate. With this procedure, a precision of 95.59 % and a recall of 97.43% was achieved on the NSL-KDD dataset.

3.3. Payload Analysis

Utilizing the payload of the sent packets, instead of only statistical data from their headers, is proposed in several works. It promises to offer more insight into what the sender intends with the traffic. While it means to analyse significantly more data (as the payload of a packet is usually larger than its header information), it intuitively seems an appropriate instrument to distinguish anomalous packets from normal behaviour.[14]

A first set of works use the distribution of bytes or byte n-grams for comparing the packet contents. Wang et al. [60] proposed a method called **PAYL** that calculates the byte frequency and standard deviation of the application payload and compare them with the observations for the same port and a similar packet length. This takes into consideration the assumption that for certain ports and packet lengths there are characteristic payloads that can be observed under normal circumstances. Significant deviation from it, which is calculated using the Mahalanobis distance, can then identify outliers, indicating a network attack. The whole calculation can take place in linear time and

[14] Of course, analysing the payload only can offer any valuable insight if the payload is not encrypted. If the NIDS cannot decrypt the payload, it also is not able to recognize any anomalies in a reasonable way. It is therefore assumed in the following that the NIDS can read the unencrypted payload. This limits its possible applications in practice, e.g. to internal networks behind an encryption proxy.

yields convincing results on the DARPA dataset, with a false positive rate of only 0.1% for port 80 using TCP. More details about PAYL can be found in section 5.3.2, as it is used as a payload analyser within this thesis.

PAYL was later enhanced by Wang et al. with multi-centroid clustering (i.e. providing multiple byte distributions per packet length) and by incorporating the correlation between the inbound and outbound traffic (*ingress/egress correlation*). The new approach is suitable for detecting worms accurately, as they report. [59]

Perdisci et al. proposed McPAD [46]. It uses so-called 2_ν-grams for feature extraction, which represent the occurrence frequency of two bytes with the distance ν within the payload. For each value ν this results in a fixed number of 256^2 features (which takes significantly less memory than calculating n-grams for $n > 2$). A clustering algorithm is then applied for dimensionality reduction. For classification, an ensemble of one-class support vector machines is used. The method was reported to have lower false positive rates in comparison with PAYL.

Furthermore, there are many approaches utilizing **deep learning** methods for autonomously recognizing features in the network traffic's payloads. Recently, deep learning techniques have led to advances in many fields of computer science and thus there is an increasing research interest to utilize them for NIDS as well. As shown in a survey by Liu et al., 14 out of 26 recent papers adopt deep learning methods [37, p. 20].

Wang et al. [61] transform the raw network traffic data into two-dimensional images that are later used by a convolutional neural network for extracting features. CNNs are widely used for image classification and have led to significant progress in this field. With their help, it is possible to extract spatial features from the images. In the paper, two approaches are discussed: The first uses images that are generated from the whole network flow as the CNN input. With the second method, however, one image per packet is made. Furthermore, the temporal relations among the packet vectors are learned. By this, an overall accuracy of 99.69% on the ISCX2012 dataset is achieved, with a false positive rate of only 0.22%. However, the detection rates of certain attack types like R2L and U2R in the DARPA1998 dataset are not as high as in competing approaches (only 74.19% and 64.25%, respectively). The authors point out that further work is required for improving the performance on imbalanced datasets and in respect to the inclusion of traditional traffic features.

Min et al. [40] applied *Natural Language Processing* (NLP) techniques for increasing the performance of NIDS. They used a text-convolutional neural network for extracting features from the packet payloads. This is done by utilizing a byte-level word embedding method that is similar to word2vec. Normally, word2vec is used as a word representation that preserves semantic relations between words. By contrast, for the payload analysis used in the paper, the packet payloads of each network flow are concatenated and each byte in a flow is then considered as a word. The emerging embeddings of the flows are then used to extract features with a Text-CNN. Together with statistical features from each network flow (like fields of the packet headers), a Random Forest algorithm is trained to classify attacks. With this approach a high classification accuracy (99.13%) and low false positive rate (1.18%) is achieved, which both out-perform other learning algorithms that are shown as a comparison in the paper. Nevertheless, the CNN used in the paper, as well as the classification algorithm, are both trained in a supervised manner. This limits the potential for real-world usage due to the inability to detect yet unknown attacks. Also, the ISCX2012 dataset is used, which was recorded in 2012 [48, p. 9] and does not necessarily reflect up-to-date attack types.

3.4. Comparative Experiments

Most of the aforementioned works also run tests for their proposed approaches. However it is hard to compare them with each other, as *(i)* the experiment conditions differ from paper to paper and *(ii)* it can be observed that the experiments usually favour the own methods. This can indicate that the other approaches used for comparison are not optimized as well or are categorically inferior.

Nevertheless, there are various works which run objective comparative experiments to determine the performance of machine learning techniques on detecting modern network attacks, such as [21], [4], [42], [33], [58], [38], and [32]. The mentioned articles however differ from the methodology of this thesis, as they are not using one-class classification algorithms and instead train the models with already known attacks.

This reduces the set of works with suitable experiments to only a few. One of them is the already-mentioned article by Hindy et al. [28] which compares one-class SVMs with autoencoders. Çakir [12] compared different approaches to the detection of zero-day attacks, and also incorporated one-class SVMs and autoencoders, albeit mainly focusing on the outdated KDD'99 dataset. In general, it seems to be difficult to find research that takes into account all of the requirements stated in section 1.1.

Chapter 4 | Concept

4.1. Overview

This chapter proposes a sequential procedure for comparing different anomaly-based network intrusion detection approaches. Its primary focus is to provide an abstract architecture and define the theoretical foundation for the later chapters. Concrete details, such as characteristics of the utilized datasets and implemented algorithms are discussed in chapter 5.

The first and initial step of the scheme described in the following consists in **preprocessing** the chosen datasets, which creates the basis for later comparisons. Afterwards, an anomaly detection model can be trained with a certain set of parameters. During the **training**, it is assumed that only benign traffic is observed, and therefore only this type is shown to the model. This corresponds with the one-class classification approach described in section 2.3.3. Subsequently, the **classification** step tests the trained model with unlabelled traffic and lets it decide which network packets belong to an attack. These predictions are then compared with the actual labels of each packet in the **evaluation** step. Several metrics, grouped by attack categories, are calculated as a result.

During this process, it is attempted to retain a practical perspective, which specifically means that the training and classification steps should conceptually not differ from a real-world setup. The objective that should be achieved by this is to minimize the gap between the results yielded in the evaluation step and those that can be observed in more realistic setups.

4.2. Preprocessing

For training a model that can detect anomalies in network traffic, it needs some data to learn from. While there already exist many datasets which are dedicated to providing such sample data, they do not come in a consistent format. Therefore, the first step is to preprocess the chosen datasets, which here means to bring them in line with a fixed, real-world orientated data structure that is later used for the training. To be more specific, after the preprocessing it must be possible to:

1. read the dataset's network packets as a stream, e.g. by replaying PCAP files,
2. group the dataset into fixed subsets which are used for the training and classifying steps respectively, while containing the proper type of network traffic in each set,
3. provide the information whether a specific packet belongs to an attack or not, and if it belongs to an attack, to which category. This is later needed for the evaluation step.

The complexity of those preprocessing goals depends on the structure of the given dataset. For instance, the last point likely requires more work if the dataset only provides a labelling per flow, because associating the corresponding packets is not a trivial task, as later described.

The preprocessing step therefore provides the *data source* as described in section 2.3.2. However, it differs insofar from the *data pre-processing* step defined there, as there is not any feature extraction involved. Instead, a general interface for the data source is supplied, which can be accessed independently of the actually-used dataset, and without introducing any potential bias. By targeting the three described goals, the following benefits can be achieved as a result of the preprocessing:

Supply a universal and practice-oriented entry point for the model training and classification: In a real-world application, capturing network traffic as a PCAP file is straightforward due to the existence of dedicated tools and incorporation of suitable libraries into many programming languages. It therefore is more natural to test an anomaly detection approach directly on this format than on a processed version of it (like a grouping into flows). While for a system used in production it is probably more feasible to directly stream network traffic into the anomaly detection system as soon as it is observed, the underlying data structures and principles are not likely to change much. As the focus of this thesis is to create an evaluation system, and reproducibility being an important keystone for it, PCAP files as the input source seem to be more appropriate than to directly listen on network devices.

Provide the entire traffic information: Grouping network packets into flows, as it is done in many datasets, can discard information about the original network traffic. Most importantly, the payload of an IP packet, which could be important for anomaly detection, cannot directly be accessed within a flow-based grouping that only shows statistical properties of the packets (like their size, IP addresses or used protocols).

Assure comparability between different approaches: Comparing two different approaches on the same dataset only is meaningful when the training and classification subsets are not changed between both evaluations. Many datasets do in fact already provide a predefined split for training and testing (e.g. UNSW-NB15 and CIC-IDS-2017), but some approaches proposed in literature nevertheless only use a selection of those splits. Furthermore, the provided training subset does not always solely contain benign traffic, which complicates the utilization of OCC algorithms. To avoid such situations, the preprocessing step should precisely define which parts of the dataset can be used for training and testing (OCC-based) anomaly detection models.

Create comparability between different datasets: The process of reducing all datasets to a common format makes it easier to compare the performance of one anomaly detection approach on different datasets. As described in section 2.5, many datasets come with a predefined grouping, like the aggregation of network packets into flows. Such groupings already rely on some knowledge about the network packet content, e.g. about the used protocols, and therefore create a bias. As an example, the application responsible for generating the network flows in the CIC-IDS-2017 dataset, called CICFlowMeter, groups with the help of an arbitrarily set flow timeout, and additionally on the basis of TCP connection terminations [13]. While the chosen preferences might intuitively sound appropriate, it is hard to argue that they are the only valid parameters for generating such flows. Furthermore, it is not guaranteed that other datasets share the same methods.

Prevent evaluation biases: The grouping of multiple packets into one flow could later distort the evaluation: For instance, the miss-classification of 1000 attack packets in a packet-based evaluation setup would yield 1000 false negatives, but when grouped to a single flow it would only yield one false negative. Providing the label for each packet separately avoids such biases.

Provide the attack category for the evaluation phase: When evaluating a certain approach, it might be interesting for which kind of network attacks it performs well. By providing the necessary information for each packet, namely the name of the corresponding attack category, if there is any, this process is being facilitated.

4.3. Model Components and Training

After having a standardized format for the data source, it is easier to design how an anomaly detection model can be built for network traffic data in a general way. It takes a network packet stream of the training subset, which is defined in the preprocessing, as its input and learns from it the characteristics of normal behaviour (*training*). Afterwards, the model is stored in a database from where it can be loaded for later usage. The following section describes both the structure of the model and its training process.

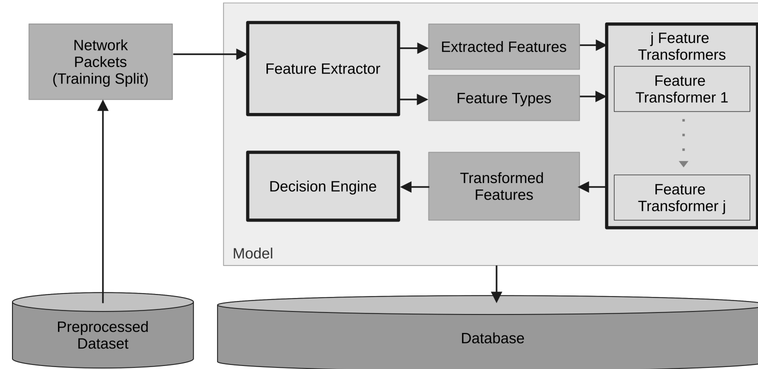

Figure 4.1.: Conceptualisation of the training phase

To achieve a generic architecture for an anomaly detection model, it is divided into three components: the **feature extractor**, any number of **feature transformers** and the **decision engine**. They are responsible for processing the network data in a pipeline, where each pipeline step has a defined input and output format. Each concrete implementation of those components is supposed to be substitutable, which means it should not rely on any specific implementation of another component. Figure 4.1 shows a schematic overview of the whole training process. The components of a model are explained in the following pages.

Feature Extractor: The feature extractor is responsible for generating a feature matrix from a given stream of network packets. It therefore carries out the *feature creation* step, as described in section 2.3.2. That is, having r vectors $p_1, ..., p_r$ with $p_i \in \{0,1\}^*$ representing the timestamp-sorted network packets in binary format[15], it maps them to n feature vectors $x_1, ..., x_n$ of a fixed length m with $x_i \in \mathbb{R}^m$. In the following, for a feature vector x_i the term *instance* is used interchangeably.

Additionally, the feature extractor provides a single vector $t = (t_1, \ldots, t_m) \in T^m$ which represents the type of each feature. T is a finite set of all possible feature types, such as *numerical* or *categorical*. Its concrete items and their meaning are no further defined and instead left to the concrete implementation. The vector t can be used to pass information about the features to the following model components (in particular feature transformers). The types can be seen as hints, instead of strict constraints.[16]

In a formal way, the function that is provided by the feature extractor can be described as:

$$e : (\{0,1\}^*)^* \to T^m \times (\mathbb{R}^m)^*, m \in \mathbb{N} \tag{4.1}$$

where the number of features M is a fixed value for a certain configuration of the extractor. Note

[15] The details about the packet format, which also includes how the timestamp of a packet is encoded, are spared here and left to the implementation.

[16] An example that employs this information is the one-hot encoder (cf. section 5.4.3), which must know which features are categorial in order to encode them.

that r and n are no fixed values for e, but rather depend on the concrete input that is fed to the feature extractor after its training is done. That is why definition 4.1 requires e to be defined for any number of packets.[17]

For a specific set of packets, the extracted features can then be represented in a $n \times m$ feature matrix, as shown in an exemplary way in Figure 4.2. It is up to the implementation of the feature extractor whether an own feature vector is created per network packet ($r = n$) or if multiple packets are accumulated and represented as one instance ($r > n$). The remaining case $r < n$ is not allowed, as it could result in problems during classification (see section 4.4).

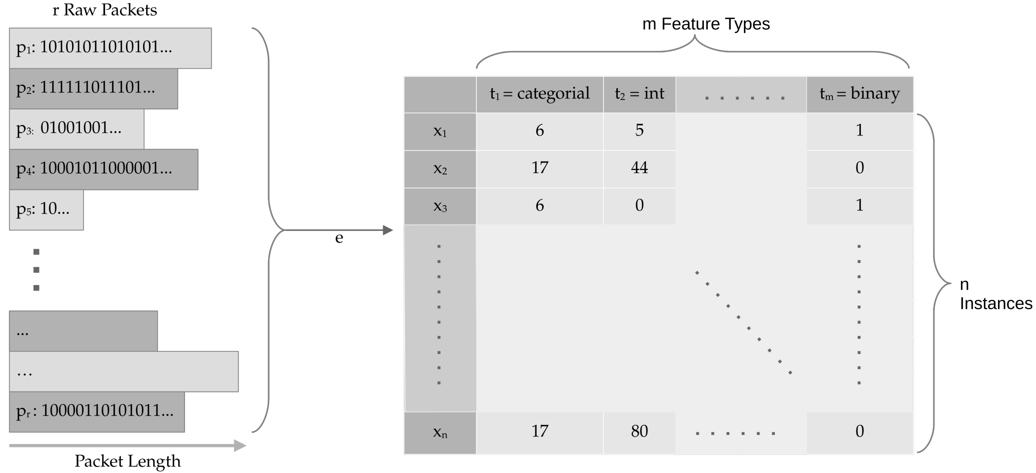

Figure 4.2.: Depiction of a trained feature extraction function e. From an arbitrary number of packets of varying length it extracts instances with a fixed number of features, together with the corresponding feature type information.

The goal of the training phase is to build and provide the function e. This is trivial in the case that e can be declared independently from the traffic in the training set. The generation of network flows as described in section 5.3.1 is an example of this. But it is also possible that the feature extractor requires an internal state, which is formed during the training. This might be a distribution for an observed statistical property, for instance the 50 most accessed IP addresses in the network traffic, or common patterns in the network packet's payload (as collected by the PAYL extractor described in section 5.3.2). That information is then used to generate features for any incoming packets. The state is stored in the database as part of the model, so that it can be used in later steps. After the function e is built, it is called on the raw traffic itself, in order to create the instances which are forwarded to the next model component.

Feature Transformer: A feature transformer, here denoted as τ, must provide a function

$$t_\tau : \mathbb{R}^{\tau_{in}} \to \mathbb{R}^{\tau_{out}}, (v_1, \ldots, v_{\tau_{in}}) \mapsto (u_1, \ldots, u_{\tau_{out}})$$

which maps a feature vector of length $\tau_{in} \in \mathbb{N}$ to a feature vector of length $\tau_{out} \in \mathbb{N}$. By this, it is able to execute *feature reduction*, *feature conversion* or *feature normalization* (cf. section 2.3.2). Unlike the feature extractor, τ applies the function t_τ to each of the extracted instances one by one. It therefore is only permitted to modify feature values, to delete some of them or to add new ones, but not to change the overall number of instances n or their order. As such operations can change the type of the feature values, the transformer must also provide a function

$$typetransform_\tau : T^{\tau_{in}} \to T^{\tau_{out}}, (t_1, \ldots, t_{\tau_{in}}) \mapsto (t_1, \ldots, t_{\tau_{out}})$$

which maps the feature types of the input instances to those of the output instances. Both τ_{in}, which depends on the input data provided by the previous step, and τ_{out}, which is either a

[17] Also, the list of feature types chosen from T intuitively should not change for a specific function e (that is, the type of the extracted features should be consistent, no matter how often e is executed). This detail is again left to the implementation to no further complicate the notations here.

transformer-specific constant, or determined during the training, must be fixed after the training of the transformer is done.

For its training, a feature transformer receives as input all n instances that were generated by the previous component (the feature extractor or another transformer), together with their feature types. This data is then used to train the transformer's internal model and to create the functions t_τ and $typetransform_\tau$. Afterwards, t_τ is called on each of the input vectors to obtain the transformed vectors which are then forwarded to the next component in the same order as received. As a model might have multiple feature transformers $\tau^1, \ldots, \tau^j$, these steps have to be repeated consecutively for each one of them, as also described in the following section. Examples of feature transforming are standardization and normalization. For preventing a bias between the features of the training and the classification step, it is important that their scaling is consistent. For example, the minimum and maximum values used for a min-max normalization must not change between the training and classification step. Therefore, any state which is needed by the transforming function used in the training phase must be stored in the database, if it depends on the training data.

Decision Engine: The purpose of the decision engine is to judge whether an attack occurs or not. The goal of its training phase is to build a function which executes this decision and can later be used in the classification step. Formally, this function can be described as

$$d : \mathbb{R}^{\tau^j_{out}} \to \{0, 1\}, x \mapsto \begin{cases} 0 & \text{iff. x is normal} \\ 1 & \text{iff. x is an anomaly} \end{cases}$$

where each instance x has τ^j_{out} features (τ^j being the last feature transformer). For building this function, it receives the output of the feature transformation step. During training, the decision engine is not required to output anything. After the training, d is stored as a part of the whole anomaly detection model in the database.

4.4. Classification of Unknown Network Traffic

The classification step relies on a trained anomaly detection model which is exposed to network traffic, like in the training phase, but this time without knowing the traffic label. The principal setup of the classification pipeline resembles that described in the previous section, but now the decision engine is supposed to actually yield a result, i.e. its function d is queried to distinguish normal from anomalous traffic. The classification pipeline is shown in figure 4.3.

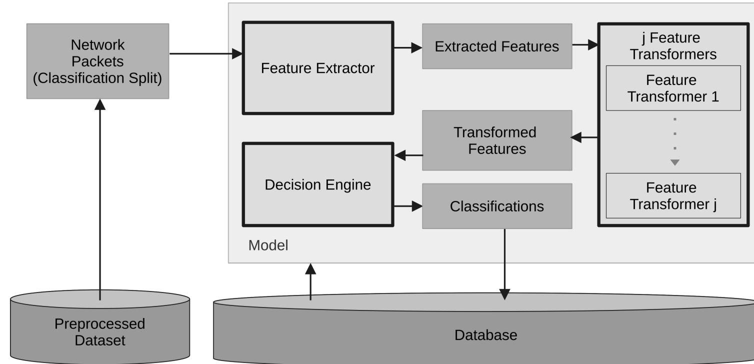

Figure 4.3.: Conceptualisation of the classification phase

The first step is to **load the state of a previously trained model** from the database, by restoring the state of all its components. As in the training phase, it takes a stream of network packets as its input. In a practical setup, it would be crucial to receive the result of the classification as soon as possible, so that measures against ongoing attacks can be undertaken immediately. However, as real-time responses are no substantial requirement for the evaluation's functioning, it is allowed for the classification step to feed in the packets as batches and retrieve the results in a bundled manner. The following describes the classification procedure for one such batch. As before, the number of its packets is represented by r.

The **feature extraction** works in a similar way as in the training phase and applies the function e on the raw network traffic, yielding n feature vectors. Additionally however, the feature extractor must now also keep track to which instance x_k each packet p_i is mapped. This can be expressed formally as a function

$$b\colon \{1, \ldots, r\} \to \{1, \ldots, n\}$$

which assigns a packet index $i \in \{1, \ldots, r\}$ to the feature vector index $b(i) = k \in \{1, \ldots, n\}$. The feature extractor must not provide this function b directly, but rather its inverse b^{-1}: It is later used to associate the decision engine's result (which is based on the extracted and/or transformed features) to the individual packets. The need for this association is also the reason why $r \geq n$ is required to hold, as with multiple extracted instances per packet the decision engine could classify one as an anomaly and the other as normal, resulting in a stalemate that would need special treatment.[18]

As described in the previous section, the **feature transformation** must use the function t_τ created in the training phase. $typetransform_\tau$ is not needed anymore in the classification phase, because the feature types that a transformer produces should be fixed by now (if a subsequent model component should rely on the types of its input instances, this information must be accessed in the training phase). When multiple feature transformers are used, they must be applied step by

[18]It should be noted that the function b is mainly needed for the packet-based evaluation of the NIDS, as it shows how many packets are actually influenced by a single classification of the decision engine. In practice it could be more convenient to omit it and instead create alerts directly based on the extracted features (such as the IP address of a flow which is reported to be an anomaly)

step. For describing this in a formal way, a new function $t_{\mathcal{T}^{all}}$ is introduced in the following, where $\circ$ denotes the composition operator of two functions:

$$t_{\mathcal{T}^{all}} = t_{\mathcal{T}^{j}} \circ \cdots \circ t_{\mathcal{T}^{1}}$$

Given the n instances provided by the feature extractor, the feature transformation must then be run consecutively on each instance x_k.

$$(x_1, \ldots, x_n) \mapsto (t_{\mathcal{T}^{all}}(x_1), \ldots, t_{\mathcal{T}^{all}}(x_n))$$

Finally, the **decision engine**'s function d is queried to determine whether an instance represents an attack or not. Using the inverse of function b, those classifications are then stored for each packet in the database. That is, for each instance x_k the following tuples are persisted, each one assigning the decision engine's output to the corresponding packet index:

$$\langle i, d(t_{\mathcal{T}^{all}}(x_k)) \rangle \mid \forall i\colon i \in b^{-1}(x_k)$$

Together with them, additional information about the classification context, like the name of the dataset, which part of the network traffic is used for the classification, and an ID that can distinguish the used model, can be persisted. It is also useful to time how long the classification, in detail the duration between the feature extraction and the result of the decision engine, took in total. This time can then be divided by the number of packets to get the average classification time per packet.

4.5. Evaluation

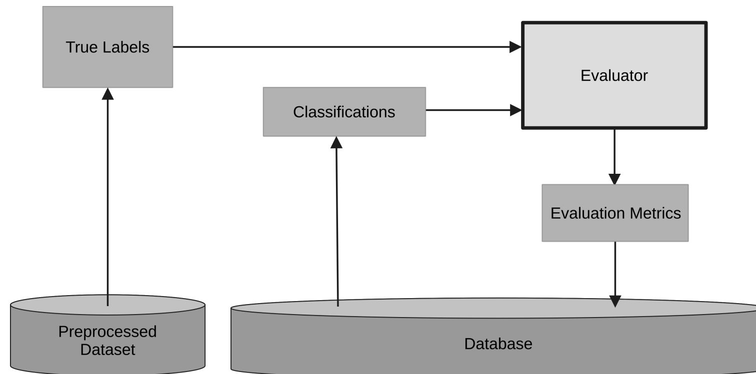

Figure 4.4.: Conceptualisation of the evaluation phase

The evaluation step assesses the performance of the classification, i.e. how well the model could detect attacks within the given network traffic stream. For this purpose, it loads both the classification results from the database and the actual packet-wise labels from the preprocessed dataset. By comparing them, a confusion matrix is created as described in section 2.4. Based on this, various measurements can be calculated, such as precision, recall and the f1-score.

After multiple approaches are trained, tested and evaluated, they can be compared to each other. Using the calculated metrics, the influence of one or multiple model parameters on the detection performance can be described.

To gain more insights into a model's ability for detecting a specific type of network attacks, it might be useful to perform the evaluation separately on each of the categories that are defined in the preprocessing step. For achieving this, various subsets of the classified packets are built. Per attack category, all of the corresponding attack packets as well as all benign traffic are evaluated together. This has the same effect as a retrospective modification of the classification, causing the anomaly detection model to only be faced with attacks of one category, but still all of the benign traffic. A downside of this method is that the categories might not be presented proportionally in the dataset, which leads to distortions. Furthermore, the classifications of benign packets, and hence the number of false positives and true negatives, are constant between all evaluations. As a consequence, precision, which depends on the number of both the true and false positives, is not an adequate measurement here, as it depends on how many packets belong to a category. Instead, for comparing different categories, the recall can be consulted. The following example illustrates this: Suppose a classifier which classifies 1000 benign packets incorrectly (causing a high number of false positives), but all 10 packets of a small attack category correctly. It would have a recall of 100% for this particular attack category, but a low precision of only 1%. For another category consisting of 2000 attack packets, of which it correctly classified 1000, it would have a precision and recall of 50%, despite intuitively it can be argued that the first attack category was detected better.

4.6. Hyperparameter Search

To facilitate multiple iterations of the training, classification and evaluation steps of the same model but with different parameters, a hyperparameter search can be utilized. A simple method for that is a grid search, which executes those steps with all possible combinations within a given set of allowed parameter values. In contrast to other techniques, it does not need feedback about how well a certain parameter combination performed in order to yield the next one. Providing this feedback would have to rely on a single numeric value, but determining which one of the evaluation metrics should be chosen for it might not be a simple choice. While consuming more resources than other approaches, a grid search allows easy comparison of a broad set of parameter values and thereby helps to obtain an overview of their influence on a model's performance. Figure 4.5 illustrates the entire hyperparameter search process. It is defined by the selected dataset, the utilized feature extractor and decision engine, and the set of hyperparameter values for them. The feature transformers are treated as such parameters as well, so that different arrangements for them can be compared to each other. In each iteration of the search, a model configuration is generated from the given set of possible parameter combinations. Using this configuration, the subsequent steps are then executed as described in the previous sections.

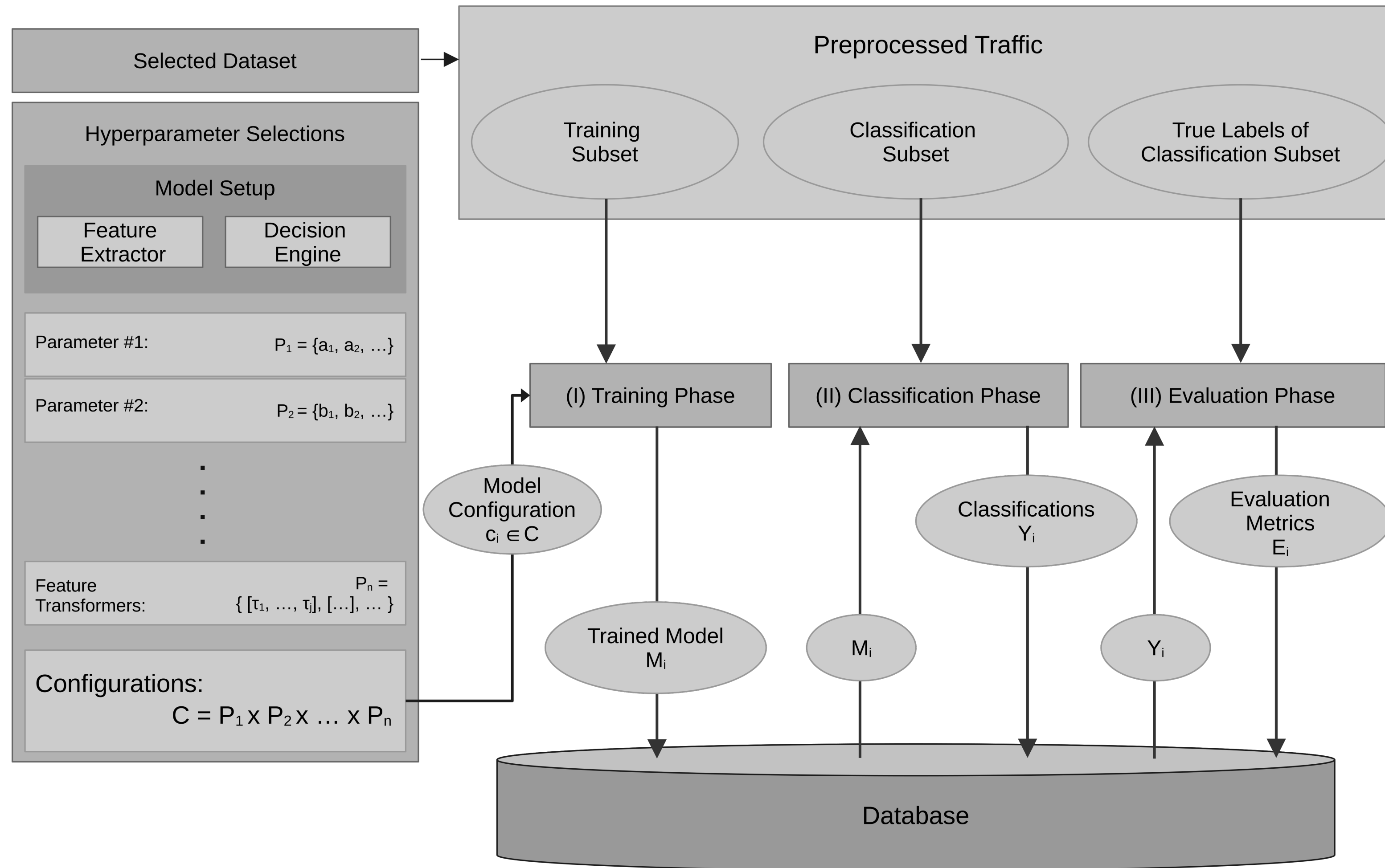

Figure 4.5.: Conceptualisation of the hyperparameter grid search.

Chapter 5 | Implementation

5.1. General Overview and Utilized Technologies

This chapter describes the realization of the previously proposed concept for a system dedicated to comparing anomaly-based network intrusion detection approaches. It starts with details about the **preprocessing** of the two chosen datasets and problems that must be solved for it. Then, the implemented **feature extractors** are described, namely a network flow generator, which mainly uses statistical data, and payload-based approaches that utilize byte-wise frequency distributions of the packet contents. Afterwards, the **feature transformers**, implementing standardization, min-max scaling, one-hot encoding, and principal component analysis, are described, followed by the **decision engine** approaches. Two types are utilized in the implementation: autoencoders and one-class support vector machines.

The proposed system is implemented in the programming language python and utilizes several open-source libraries, namely:

- **dpkt**[19] for reading and processing network packets from PCAP files,
- **pandas**[20] for processing tabular data, which is particularly used during the dataset preprocessing and the evaluation,
- **numpy**[21] for all numeric calculations,
- **scikit-learn**[22] for machine-learning functionalities, especially for the implemented one-class support vector machine and feature transformers,
- **keras**[23] and **tensorflow**[24] for building the autoencoders, and
- **plotly**[25] for visualizing the evaluation results.

For persisting data, python's internal sqlite[26] module is drawn on, which provides a self-contained, single-file database.

In the following sections, at some points pseudocode is utilized to describe the developed algorithms.

[19] https://dpkt.readthedocs.io/en/latest/
[20] https://pandas.pydata.org
[21] https://numpy.org
[22] https://scikit-learn.org
[23] https://keras.io
[24] https://www.tensorflow.org
[25] https://plotly.com
[26] https://sqlite.org

It resembles the actual python code of the implementation, but only focuses on the main ideas. The source code, which can be consulted as an in-depth reference of the implementation, is publicly available in a git repository hosted on GitHub[27].

5.2. Dataset Preprocessing

5.2.1. Assigning Packets to Flows

(A) Problem Description

Two datasets are chosen for this thesis, which both reflect reasonable recent network traffic: The CIC-IDS-2017 and the UNSW-NB15 dataset. A common trait that they both share is that they are providing their raw network traffic in the form of packet captures. However, the corresponding labels are not given directly. Instead, only a list of network flows, with the information of whether or not a flow belongs to an attack, is provided. Speaking with the terminology of the previous chapter, they provide the results of a feature extraction function e, but not an easy way to find out which packets belong to a specific flow (i.e. access to the function b). Since such a packet-wise labelling is a goal of the preprocessing, it must be created as a part of it. That is, for each IP packet in the dataset, the corresponding flow (which includes the packet) has to be found so that the flow label (*benign* or *attack*), and its attack category can be associated with the packet.

A trivial way to achieve this would be an exhaustive search for each network packet within all provided network flows, which checks whether the packet can belong to a flow based on certain properties, like its timestamp, protocol or IP addresses. The problem with this method is that it is inefficient and will take a long time.[28] For the given dataset sizes, it would already make the preprocessing step infeasible. Effectively, its time complexity lies in $\mathcal{O}(R \cdot F)$, R being the number of network packets and F the number of flows as supplied in the dataset.

To be more efficient, the flows could be sorted by their starting time. For a packet with a timestamp t_p, only flows that have a starting time t_f with $t_f \leq t_p$ must then be examined. This would reduce the number of comparisons, but the overall runtime would still be inordinately long, which would make reproducing and validating the preprocessing difficult for other researchers.

To further reduce the number of possible flows for a packet, flow identifiers can be used. They must be constructible from both the information of a flow given in the datasets and each packet in the raw traffic capture. In fact, the CIC-IDS-2017 dataset provides such a *flow ID* for each flow. It is a character string constructed in the format

```
[IP address A]-[IP address B]-[port A]-[port B]-[IP protocol number].
```

In the same manner flow IDs can be easily generated in the UNSW-NB15 dataset. The needed information can be extracted from each packet as well; however, there may be multiple possible flow IDs per flow: Firstly, it is not clear in which order the source and destination hosts of an IP packet are declared in the flow ID, and secondly, there might be multiple flows with the same id, but at various times. Using the flow ID as a mean to associate packets with flows therefore still requires some additional work, as described in the following.

[27] `https://github.com/dhelmr/bachelor-thesis`

[28] In fact, this approach was first tried out during the work on this thesis, but it quickly was clear that the preprocessing was practically not feasible that way.

(B) Proposed Algorithm

The main idea of the algorithm is to exploit the following points:

1. Detailed information must only be extracted for attack flows. For a packet that contains benign traffic, on the other hand, it suffices to only recognize that it is *not* an attack.

2. It can be assumed that $R \gg F$. Therefore, limiting the influence of R on the runtime complexity can have a desired effect.

3. The number of attack flows is significantly smaller than the number of benign ones. This implicates in turn, that the number of benign packets R_B is significantly bigger than the number of attack packets R_A. Furthermore, the number of packets that have ambiguous flow identifiers (from both benign and attack flows), in the following called R_U, are assumed to be relatively low, so that holds $R \gg R_U$.

In a first step all attack and ambiguous flows are found in the dataset and identifiers (*flow IDs*) are generated for them. The main part of the algorithm is then a loop, which successively reads in the network packets provided by the dataset. Per packet, an identifier is generated, together with a set of potential flow IDs under which the packet can be found in the dataset. If those are not part of the initially generated attack IDs, the packet must be of benign nature and does not need to be inspected any further. Otherwise, all potential flows that the packet could be a part of are loaded. Based on the packet's timestamp, the exact flow is then determined. Normally, a suitable flow for the packet should be found and information like its type and, if it's an attack, the respective category, can be extracted. The following code listing shows the algorithm described so far as python-like pseudocode.

```
attack_flows = get_attack_flow_ids()
for packet in pcap_reader():
  packet_id = get_packet_id(packet)
  flow_ids = make_flow_ids(packet)
  if attack_flows.contains_none(flow_ids):
    write_to_csv(packet_id, "benign traffic")
    continue
  potential_flows = attack_flows.find_all(flow_ids)
  flow = get_exact_flow(potential_flows, packet)
  if flow is None:
    write_to_csv(packet_id, "undecidable")
    continue
  write_to_csv(packet_id, flow.traffic_type, flow.attack_info)
```

The sub-steps of this algorithm, together with their runtime analysis, are explained in more detail as follows:

(I) Finding attack flows and corresponding ids: The purpose of this first step, `get_attack_flow_ids()`, is to reduce the number of potential flows that the algorithm later must search through. To achieve this, all flows with an attack label are filtered. Afterwards, for each of these attack flows, a search is run to find all remaining flows which have the same destination and source IPs (no matter in which order) and the same protocol. The motivation behind this is that a packet that is assigned to a certain attack flow ID can as well be part of an identically identified benign flow if only the flow identification is looked at. For determining the packet's true flow, its timestamp must be considered as well and be compared with each flow's starting time. This step has a worst-case runtime complexity in $\Omega(F \cdot F)$, as each flow has to be examined, and for each attack flow, corresponding flows have to be found.

(II) Creating the packet identifier: `get_packet_id()` generates a unique identification for each of the dataset's network packets. It is used to correctly assign the traffic type label to a packet later in the evaluation phase. A simple way to generate the ID is an incrementing

numerical counter, together with the name of the PCAP file where the packet is read from. This step's runtime does not depend on the number of packets or flows and is therefore assumed to be constant here.

(III) Creating the packet's flow identifiers: This step creates possible flow identifiers for each packet. In detail, an identifier is created for the flow in forward direction (with the packet's source and destination IP addresses in order) and backward direction (with those addresses in reversed order). As before, this step runs in constant time.

```
def make_flow_ids(packet):
    src_port = packet.transport_layer.source_port
    dest_port = packet.transport_layer.dest_port
    fwd_flow = (packet.source_ip, packet.dest_ip, src_port, dest_port, packet.
                                      protocol)
    bwd_flow = (packet.dest_ip, packet.source_ip, dest_port, src_port, packet.
                                      protocol)
    return [fwd_flow, bwd_flow]
```

(IV) Finding the matching flow: This step checks for all of the potential flows, whether the packet can be a part of it. This decision is grounded on the packet's timestamp and the flow's starting time. Additionally, if the flow is known to be unidirectional, it is checked whether or not the packet is sent in the same direction as indicated by the flow. If no flow matches the packet's timestamp, `None` is returned, which indicates a problem with the dataset, as explained later.

```
def get_exact_flow(potential_flows, packet):
    filtered_flows = filter_uni_directional_flows(potential_flows, packet)
    sorted_flows = sort(filtered_flows, key=lambda flow: flow.start_time)
    selected_flow = None
    for flow in sorted_flows:
      if flow.start_time > packet.timestamp:
        # abort criterion: the packet must lie in the previously selected flow
        break
      selected_flow = flow
    return selected_flow

def filter_uni_directional_flows(flows, packet):
    filtered_flows = []
    for flow in flows:
      if flow.is_uni_directional() and flow.src_ip != packet.src_ip:
        continue
      filtered_flows.append(flow)
    return filtered_flows
```

In the worst case, the runtime of this step is log-linear in the number of potential flows, due to the sort operation (assuming the application of merge sort, for instance). It can be assumed however that in the average case the sorted list will accelerate the following iteration through it, as the abort criterion presumably is reached faster. Moreover, the number of potential flows is no further estimated here. Instead, an upper bound is stated for the case that all flows of the dataset need to be iterated through. This yields a worst-case runtime in $\Omega(F \cdot log(F))$

For the overall runtime complexity analysis, the number of packets is divided into two parts: R_U, the number of packets whose traffic type is *unknown* at first and $R - R_U$, which is the number of packets that are immediately identified as *benign*. For the latter, the last sub-step, *(IV)*, must not be executed. This results in the following worst-case upper bound, where each of the above sub-steps is represented by a big Roman number:

$$\Omega(I + R \cdot (II + III + IV)) \tag{5.1}$$
$$= \Omega(F \cdot F + (R - R_U) \cdot (II + III) + R_U \cdot (II + III + IV)) \tag{5.2}$$
$$= \Omega(F \cdot F + (R - R_U) + R_U \cdot F \cdot log(F)) \tag{5.3}$$

This is based on the facts that $R = R_U + (R - R_U)$ (5.2) and that the sub-steps *(II)* and *(III)* run in constant time (5.3). Assuming that $R \gg F$ and $R \gg R_U$, this can be expected to run faster than the previously stated $\mathcal{O}(R \cdot F)$, as R only influences the runtime linearly.

5.2.2. Occurring Problems

Even if the described method for the packet-to-flow association is working in theory, there are problems when applying it to the chosen datasets:

Problems with CIC-IDS-2017: The accuracy of the provided timestamps is only exact to the seconds, and sometimes only to the minute, which is not suitable with the millisecond-exact timestamps of the network packets. It can be observed that the actual timestamps seem to be rounded down, i.e. their milliseconds (and sometimes seconds) are cut off. Having a timestamp t_s which denotes the flow's declared (and inaccurate) starting time, and a timestamp t_r, which is the real starting time, accurate on the milliseconds, with $t_r > t_s$, a packet will be misclassified if its timestamp lies within $[t_s, t_r)$. This case is illustrated in figure 5.1 and can't be detected, as t_r is not known. It could be argued, however, that a packet within this interval should in some cases be part of the flow, because it was sent only a short time before the flow is declared to begin, and is therefore likely to belong to the attack. This argumentation would assume that the flow generation method of the dataset is not accurate for some packets, and an implication would be that those packets are actually labelled correctly.

Furthermore, due to the inaccurate timestamps, some flows are not distinguishable from each other at all, but have distinct labels. The first selected flow is then chosen during the preprocessing.

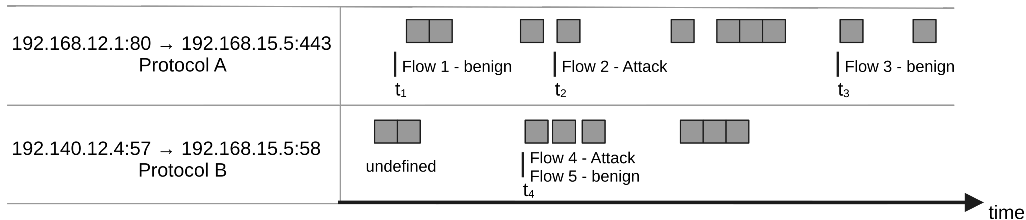

Figure 5.1.: Illustration of three problems that occur while preprocessing the CIC-IDS-2017 dataset The given timestamps t_1-t_4, which indicate the starting times of the flows, represent those that are stated in the dataset. Suppose that t_2 and t_4 are imprecisely specified: In the packet stream above, it would then be possible that the fourth packet should actually be assigned to the first flow. In the stream below, the first three packets cannot be assigned to any flow and are assumed to be benign. Furthermore, the fourth and fifth flow can't be distinguished, as they both share the same starting timestamp.

UNSW-NB15: Also this dataset has duplicate flow records, however mostly they only differ in the provided attack category. Assuming that such records are no error in the dataset, they would indicate that the same network packet belongs to multiple attack categories. This case is neglected in the preprocessing, and only the first attack category is used for labelling the packets.

Furthermore, the dataset provides some names for transport protocols which could not be correctly identified. Among them there are seven names which are also assigned to some attacks: `any` (411 flows), `ip`, `pri-enc`, `zero`, `isis`, `sccopmce` and `ib` (137 flows, respectively). As their protocol number in the IP header is not known, the corresponding packets cannot be identified and hence no matching flow can be found. As a consequence they will be labelled as *benign*.

	CIC-IDS-2017	UNSW-NB15
Packets with Indistinguishable Flows	112,296	27,693
Packets with no Matching Flow	32	640
Non-IP Packets	412,516	25,481

Table 5.1.: Preprocessing Measurements for UNSW-NB15 and CIC-IDS-2017

5.2.3. Preprocessing Result and Validation

For describing the degree of inaccuracy that emerges through the problems indicated in the previous section, the following validation metrics are collected:

Number of expected and actual packets per attack category: For each flow, the number of packets in forward and backward direction is provided in the datasets. As a packet should only belong to one flow, the sum of those numbers can be seen as the number of expected packets and might be broken down by attack categories. Thereby a comparison is possible with the number of packets that are associated with each attack during the preprocessing. Both the absolute and relative difference (in relation to the number of expected packets) is calculated.

Number of uncertain assignments due to indistinguishable flows: This number counts all packets for which at least two flows are found that cannot be distinguished based on the provided information. For the CIC-IDS-2017 dataset, this seems to be due to the inaccurate timestamps, whereas the UNSW-NB15 dataset provides flows that are completely identical except for the attack category.

Number of packets that cannot be assigned to any flow: Packets for which the step `get_exact_flow` returns 'None' are counted here. This happens if the packet's timestamp lies before the first starting time of the examined potential flows.

Number of packets for which no flow ID can be generated: Effectively, this contains all non-IP packets, and does not directly indicate an error, but instead provides additional information about the dataset.

Table 5.1 shows the last three measurements, whereas tables 5.2 and 5.3 display the distribution of attack flows and the ratio between expected and associated packets for each category. It can be observed that for both datasets, a significant portion of the expected packets cannot be found during the preprocessing. Despite the mentioned problems in the datasets it cannot be ruled out that the reason for those inaccuracies lies within the implementation of the preprocessing itself. However, an argument against this assumption is that in both datasets there is an attack category for which the number of associated packets exactly equals the expectations, indicating that the preprocessing at least yields correct results for those. Furthermore, it should be noted the number of wrongly associated packets in the UNSW-NB15 dataset is likely influenced by the fact that for each one of the duplicate flows, a packet is expected, but it only can be assigned to one of them. However, the impact on the evaluation of an NIDS is not as severe, because the packet in question is still classified as an attack. For the CIC-IDS-2017 dataset, on the other hand, the preprocessing seems to result in more misclassifications, i.e. packets which belong to an attack but are classified as benign.

Category	Flows in the Dataset	Expected Packets	Associated Packets	Absolute Difference	Relative Difference
Exploits	44140	2552199	2298198	254001	09.95%
Reconnaisance	13938	168003	138127	29876	17.78%
DoS	16045	741314	641025	100289	13.53%
Generic	215439	730630	314966	415664	56.89%
Shellcode	1511	13983	13983	0	00.00%
Fuzzers	24176	481398	409338	72060	14.97%
Worms	174	14120	13114	1006	07.12%
Backdoor	2280	19061	7792	11269	59.12%
Analysis	2621	19907	9346	10561	53.05%
All Attacks	320324	4740615	3845889	894726	18.87%

Table 5.2.: Preprocessing Result of the UNSW-NB15 dataset

Category	Flows in the Dataset	Expected Packets	Associated Packets	Absolute Difference	Relative Difference
FTP-Patator	7938	105618	86806	18812	17.81%
SSH-Patator	5897	163095	160169	2926	01.79%
DoS slowloris	5796	46327	46543	-216	00.47%
DoS slowhttptest	5499	36874	36317	557	01.51%
DoS hulk	231073	2191807	2160242	31565	01.44%
DoS golden-eye	10293	99003	103266	-4263	04.31%
Web Attack Brute Force	1507	27771	23039	4732	17.04%
Web Attack XSS	652	7483	5387	2096	28.01%
Web Attack Sql injection	21	120	94	26	21.67%
Infiltration	36	59754	59754	0	00.00%
Bot	1966	12853	12895	-42	00.33%
DDoS	128027	989426	780791	208635	21.09%
Portscan	158930	321442	318684	2758	00.96%
All Attacks	557635	4061573	3793987	267586	06.59%

Table 5.3.: Preprocessing Result of the CIC-IDS-2017 dataset

5.3. Feature Extraction

5.3.1. Network Flow Generation

Inspired by the feature extraction methods of the two chosen datasets, a feature extractor that creates network flows is presented in this section. The reason why a new one is implemented, instead of using already-existing solutions such *zeek*, *argus*, or *CICFlowMeter* (cf. section 2.1), is that, usually, such flow generators discard the packet payloads. For this thesis, however, the payload will later be taken into consideration as well, as described in the next section. Furthermore, an own implementation can be configured more flexible and ensures reproducibility, which might be beyond control when using third-party solutions.

The feature extractor groups IP packets based on their source and destination IP address, on the IP protocol number, and on their source and destination ports. The latter two can only be used if the packet's transport protocol is understood[29] and if the protocol itself is based on ports. If this is not the case, the particular packets are only grouped based on their IP addresses and protocol number. Thus, for each packet, a flow identification, consisting of these five values, is created for determining its respective flow. As all generated flows should be bi-directional, the IP addresses within this flow identification are not ordered according to the direction of the packet, but instead sorted based on the numerical value of the addresses. For example, a packet with IP protocol number 6 (TCP) from address `192.168.12.1` and port `80` to `172.182.4.4`, port `4111` is grouped into a flow with the ID `(172.182.4.4,192.168.12.1,6,4111,80)`.[30]

Additionally, packets are also grouped based on their timestamp. A configurable timeout value determines if a packet will be appended to an already-existing group or if a new one will be opened with the same identification. For TCP packets, the packets can also be grouped based on the observed connections, as shown later.

For each of the created groups, a set of statistical features is afterwards extracted. Each feature has one of the types `int` (for discrete numerical values), `float` (for continuous numerical values), `binary` (which only permits `0` or `1` as valid values) or `categorial`. The latter indicates that the feature values should not be compared based on the ordinal relationships of their numerical representations.

In the default configuration, the generated features are grounded on values from the IP headers, on the port of the transport protocol, and on statistics about the overall number of packets, packet sizes and arrival times (cf. table D.1). The packet size is defined as the size of its payload in bytes, i.e. only the transport layer data is taken into account here. That way, the length of the IP header does not have any influence on features that depend on the packet's length. Additionally, as also done in the CIC-IDS-2017 and UNSW-NB15 datasets, the packets of a flow are separated by their direction. For each flow, the *forward* direction is determined by the first packet; all packets in the reverse direction are assigned to the *backward* direction. Tables D.2 and D.3 show the features which are extracted using those separations.

In addition, it is possible to specify further operation modes for the feature extractor. As described in the following, they may add new features or alter how the network flows are grouped and how the above-mentioned features are created.

Include IP addresses: With the mode `with_ip_addr`, the source and destination IP addresses for each flow are appended to the extracted features. For this purpose, their binary representation

[29] More precisely, this means that the used library *dpkt* can parse the payload of the IP packet. It supports most of the protocols which are relevant in practice, e.g. TCP, UDP, ICMP, and IPX. `https://dpkt.readthedocs.io/en/latest/api/index.html` can be referred to for a complete listing.

[30] For illustration purposes, the two IP addresses are here compared by their dot-decimal representation. For a more efficient implementation, comparing the binary representations achieves the same purpose.

is converted to an integer. The question of whether or not IP addresses should be considered for anomaly detection is answered differently in literature. On the one hand it can be argued that their incorporation makes it harder to generalize the characteristics of attacks between different networks (cf. [40, p. 3], [36, p. 7] and [33, p. 29]). On the other hand, they can be helpful to detect unusual traffic patterns within the same network, as shown in various works. For example, messages from unauthorized IP addresses can be filtered to prevent analysis attacks [42, p. 695] and counting the number of similar flows from distinct IP addresses can help to reveal decentralized botnets [8, p. 248]. In order to measure the effect of using the IP address features with different network types, the mode can be turned on or off according to the requirements of a particular evaluation setup.

Consider the dot-decimal representation of an IP address: The described way of extracting features from the packet's IP addresses does not take into account any semantic meaning of its parts in the dot-decimal representation. For example, `192.168.231.2` and `192.168.231.210` could identify hosts from the same subnet, but this relationship is not adequately modelled with the default feature extraction which calculates the decimal value of the complete address. For that reason, the mode `ip_dotted` creates separate features for each part in the dot-decimal representation of an IPv4 address. More generally, it creates an additional feature for each 8-bit part of the address. For example, `192.168.231.2` then results in the four new feature values `192, 168, 231` and `2`. As both IPv4 and IPv6, which provides 128-bit addresses, must be supported by the feature extractor, a total number of 16 features are generated for each IP address (for IPv4, the first twelve features are padded with zeros).

Generate features based on the digits of the port number: Similarly, the basic implementation described above does not consider the semantics of port numbers that are used by the transport protocol. For example, some applications only use certain port ranges in TCP and many operating systems require administrator rights for opening ports of lower ranges (e.g. port `80` or 443). A naive way for reflecting such information in the extracted features could be the segmentation of the port number based on its decimal digits. This is implemented with the mode `port_decimal`: For instance, port `8080` results in the feature values `8, 0, 8` and `0`. Altogether eight additional features per port number, each representing a digit in it, are extracted, so that all common port number lengths can be handled. In the case that the port number has less than eight digits, the remaining digits are filled with zeros.

Creation of Subflows: Inspired by the functionality of CICFlowMeter (the feature extractor used for the CIC-IDS-2017 and CIC-IDS-2018 datasets), the mode `subflows` examines a flow for packets that are sent closely together[31]. Essentially, the same grouping technique as for the flow creation is used, but with a smaller timeout, which is passed as a hyperparameter with a default value of 0.5 seconds. By doing this, the behaviour of some transport protocols which segment the payload into multiple IP packets (like TCP) can be analysed, as such segments are likely to be sent in short succession. Table D.4 shows the features that are additionally extracted using this information. Besides statistics like the average subflow length, the entire flow can also be divided into *active* and *idle* phases, depending on whether or not any subflow is yet open, i.e. waiting for the next packet. Figure 5.2 illustrates the recognition of subflows in backward and forward direction and shows the flow's active and idle times.

TCP features While the feature extractor is supposed to be agnostic of any transport protocol in its basic mode (except taking into account the port information), the `tcp` mode exploits several characteristics of this protocol to extract additional features. First, the flow grouping mechanism is altered, so that flows are not only closed by a timeout, but also by a TCP packet with a `FIN` flag set, which signals the end of a connection. Optionally, with the mode `tcp_end_on_rst` the `RST` flag can also be considered. For each of those connection-based flows, additional TCP features are extracted, as shown in table D.5. If the flow's protocol is not TCP, those feature values are

[31] Confer the features `subfl_fw_pk`, `subfl_fw_byt`, `subfl_bw_pk` and `subfl_bw_byt` on the CIC-IDS-2018 description website [14]. They represent the average of the transmitted bytes and packets per subflow in forward and backward direction, respectively.

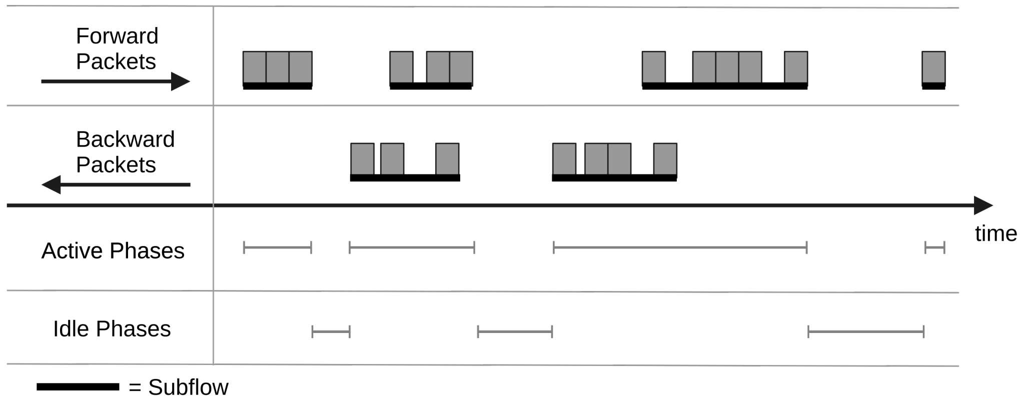

Figure 5.2.: Division of a flow into subflows as well as idle and active times

set to `-1`, so that they are distinct from any possibly-observable value.

Inspection of the last n flows Analogical to similarly extracted features in the UNSW-NB15 dataset, the mode `hindsight` compares certain attributes of a flow with the last n flows before it (ordered by the flows' starting timestamps). n is a configurable parameter, called the *hindsight window* and is set to 100 by default. Within this window, flows with the same IP addresses, ports or protocols are counted. This can help to detect decentralized attacks. Table D.6 shows the corresponding features.

Only extract basic features: The last of the described operation modes is called `basic` and only selects a very small subset of the features generated by default, namely `src_port`, `dest_port` and `protocol`. It can be expected to only be useful in conjunction with enhancements of the network flow generator that generate additional features, such as the payload analysers described in the following. The rationale for this mode is the need to determine the influence of such enhancements with as little interference as possible.

5.3.2. Flow-Based Payload Analysis

This section describes two feature extraction approaches that take into account the payload of an IP packet. They are based on the previously described flow generator, i.e. they inspect the aggregated payload of all packets in a flow and extract features from it. All of the operation modes can be utilized in conjunction with the following approaches.

(A) Append relative frequency of bytes as features

The first of the two described methods simply counts the frequency of each possible byte in the flow's concatenated payload. It then calculates the relative frequency for each byte (by dividing the byte count by the overall payload length) and thereafter appends these values as 256 new features.

It can be seen as a baseline for payload analysis methods as it is very easy to implement, running in linear time complexity.

(B) Build and compare 1-byte n-grams distributions (resembling PAYL)

A more sophisticated method is inspired by PAYL [60], which also counts 1-byte n-grams but rather builds a distribution over these counts during the training phase. For feature extraction, it then calculates the distance between the observed payload and the one seen during the training using a simplified version of the Mahalanobis Distance. These values are used for generating features that

indicate how anomalous the payload of a flow is. This is insofar different from the original PAYL as it takes into account the flow groupings.

The following describes the steps and functionalities of the PAYL feature extractor in detail.

Profile Building In the training phase, the feature extractor builds a distribution over the observed byte frequencies for each distinct `(protocol, port, packet-length)`-tuple (with regard to the transport protocol). This is done for each packet, without taking into account flow information yet. For each distinct and observed protocol-port-length combination, the following data is stored in the memory:

- n, the total number of observed payloads (resp. packets) for the corresponding `(protocol, port, payload-length)`-tuple
- $\overline{x_i}$: a distribution over the mean relative byte frequency for each possible byte value $i \in \{0, \ldots, 255\}$
- σ_i, the standard deviation for each of the observed byte distributions, which is later used for the simplified Mahalanobis distance
- $\overline{x_i^2}$: the mean of the squared relative byte frequencies, which is later used for updating the corresponding σ_i

Note that n can be stored as a simple integer, whereas the other data structures are arrays of length 256, as they contain a value for each possible byte value i. For each packet in the training set, the above data structures are initialized if they do not exist yet for the corresponding protocol-port-length combination. Otherwise, if already existing, they are updated with the payload's byte frequencies y_i as follows (cf. [60, p. 9]):

- $n \leftarrow n + 1$, as the number of observed payloads has increased by one
- $\overline{x_i} \leftarrow \overline{x_i} + \frac{\overline{x_i} - y_i}{n}$, which updates the mean byte frequencies with the newly observed ones
- $\overline{x_i^2} \leftarrow \overline{x_i^2} + \frac{\overline{x_i^2} + y_i^2}{n}$, analogically updates the mean squared byte frequencies
- $\sigma_i \leftarrow \sqrt{\overline{x_i^2} - (\overline{x_i})^2}$ updates the standard deviation for each byte. This equation is based on the fact that the standard deviation is the square root of the variance, and the variance can be calculated with the equation $Var(X) = E(X^2) - (EX)^2$. E denotes the expected value, which is approached here using the observed mean byte frequencies.

These variables are stored in the database after the training phase is ended.

Feature Extraction The feature extraction takes place both in the training and classification phase and uses the trained profile from above. In the training phase, it is run after the profile was built, so that the features are extracted in the same way as later during the classification.

For each packet in a flow, the relative byte frequencies are calculated. Then, a lookup is made in the profile for the flow's protocol-port-length combination. If the particular payload length was not observed during the profile training, the nearest available packet length is taken.

For the calculated byte frequencies y, the loaded mean byte frequencies from the profile $\overline{x}$ and the standard deviations σ, the simplified Mahalanobis distance is then calculated with the following

equation:

$$d(\overline{x}, \sigma, y) = \sum_{i=0}^{255} \frac{|y_i - \overline{x_i}|}{\sigma_i + \alpha} \tag{5.4}$$

α is a configurable *smoothing factor*, with $\alpha \in \mathbb{R}, \alpha > 0$. It prevents the denominator from becoming zero. It is also a measurement for the statistical confidence, i.e. how representative the training data is for the actual distributions. It is recommended to decrease α with increasing size of the training data. In the implemented version of PAYL, α does not change automatically, as the training set does not change after the training ended.

The Mahalanobis distance calculation is done for all packets of a flow. This results in a list of distance values, which is then used for the actual feature generation, as shown in table D.7. For each flow, the mean, minimum, maximum, and standard deviation of its packet's distance values are calculated, which give information about the degree of abnormality to the payloads observed during the training phase.

Clustering An additional step, which is executed after the profile's training, is clustering the distributions based on neighbouring packet lengths. This reduces the size of the profile and can also enhance the ability for anomaly detection (cf. [60, p. 9f.]). The clustering process merges two neighbouring distributions if their Manhattan distance is below a certain (configurable) threshold. Per clustering iteration, each distribution is only merged once. The entire process is repeated until no distributions are merged anymore.

For merging two distributions $\overline{p}$ and $\overline{q}$, the following calculations are done, resulting in a new distribution $\overline{x}$:

- $n \leftarrow n_p + n_q$
- for each $i \in \{0, \ldots, 255\}$: $\overline{x_i} \leftarrow \frac{p_i \cdot n_p + q_i \cdot n_q}{n}$

The new packet length is the arithmetic mean of the two old ones. The mean squared byte frequencies must not be updated because they are not needed after the training.

5.4. Feature Transformation

5.4.1. Min-Max-Scaling

The min-max scaler $\tau^{\texttt{min-max}}$ aims to transform a feature value into the $[0, 1]$ interval, by comparing it with minimum and maximum values of the feature. Those interval limits are determined during the training phase, therefore they do not change later, even if a bigger or smaller feature value is observed. Thus, during the classification, the feature values can also lie outside the targeted interval.

For the transformation of an instance $x = (v_1, \ldots, v_{\tau_{in}^{\texttt{min-max}}})$, a new feature value v_i is calculated as follows:

$$\texttt{min-max}(v_i) = \frac{v_i - min_i}{max_i - min_i} \tag{5.5}$$

where min_i and max_i are the minimum and maximum values of the feature which are observed during the training phase. This results in a function

$$t_{\tau^{\texttt{min-max}}}(x) = (\texttt{min-max}(v_1), \ldots, \texttt{min-max}(v_{\tau_{in}^{\texttt{min-max}}}))$$

As a consequence, the transformer does not change the length of the instances. The feature types are respectively transformed to `float`.

5.4.2. Standardization

Another common method for transforming features is standardization (or `z-score`). It works analogous to min-max scaling, but modifies the feature values so that they have unit variance and zero mean, as follows:

$$\texttt{standard}(v_i) = \frac{v_i - \overline{v_i}}{\sigma_i} \tag{5.6}$$

where $\overline{v_i}$ is the arithmetic mean of the feature's values during the training, and σ_i their standard deviation. As with the min-max scaler, those variables are not changed after training, so that the transformation will stay consistent between classification and training. $\tau^{\texttt{standard}}$ transforms the features with:

$$t_{\tau^{\texttt{standard}}}(x) = (\texttt{standard}(v_1), \ldots, \texttt{standard}(v_{\tau_{in}^{\texttt{standard}}}))$$

5.4.3. One-Hot Encoding

A one-hot encoding bijectively represents the possible values of a feature as binary numbers with exactly one `1`-bit. The remaining bits are set to `0`. Each bit is then mapped to a new feature. Each one of these newly generated features uniquely represents a certain value of the encoded feature, and attains 1 only for this value. As a consequence, its value is 1 if and only if the other one-hot features are 0. Fig. 5.3 shows an example for transforming the IP protocol numbers. One-hot encodings are a widely-used encoding mechanism in machine learning [47, p. 7].

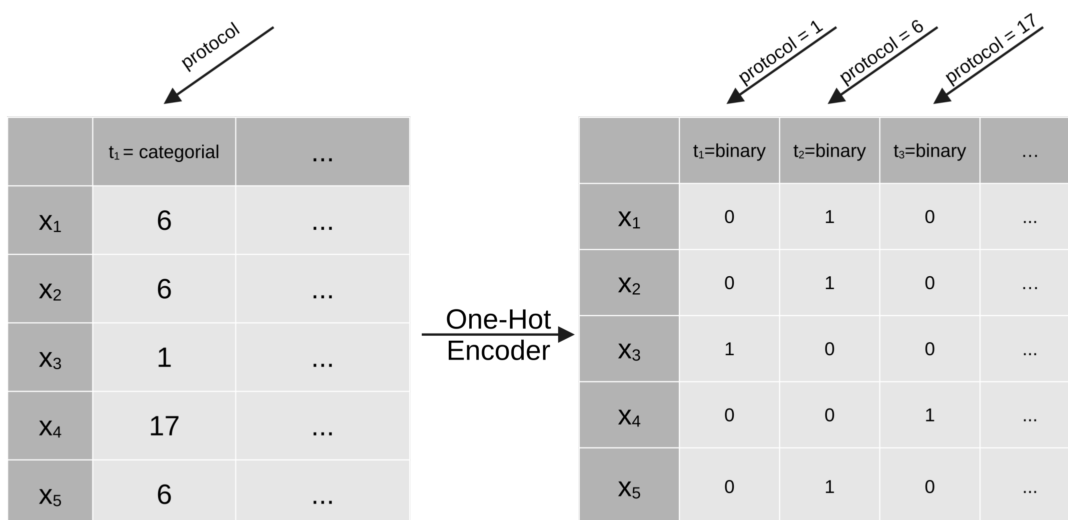

Figure 5.3.: Example of the feature transformation using a one-hot encoder
It creates a new feature (presented by a column) for each observed protocol value. Other (non-categorial) features are left untouched.

The one-hot encoder only expands *categorial* features in this manner, all other features are left untouched. Furthermore, it only creates new features for values that were observed during training, which has two implications. Firstly, the transformer doesn't need to have any knowledge about the feature and all its possible values, but instead learns them during the training phase. Secondly, if it encounters a feature value during classification that was not observed in the training it cannot create a one-hot encoding for it. In this case, all one-hot features are set to `0`. While this loses the information about the original feature value, it keeps the overall feature set consistent between training and classification, which is essential for the consecutive model pipeline steps (other feature transformers or the decision engine). Also, it is in accordance with the assumption that categorial features should not be compared based on their numerical value. Instead, the only necessary information is that the unknown feature values are of a *different* category[32].

5.4.4. Principal Component Analysis

The last of the implemented feature transformers execute a principal component analysis (PCA) on the feature matrix. They are available for 10, 20, 30, and 50 principal components. Thereby, a dimensionality reduction of the feature space can be achieved. This could have a positive effect on decision engines that cannot handle high dimensional data well, such as the one-class support vector machines.

5.5. Decision Engines

5.5.1. One-Class SVM

The implementation of the one-class SVM is based on the libraries `scikit-learn` and `libsvm` [16]. The following parameter can be specified:

Kernel: It can be chosen between a linear, polynomial, sigmoid and rbf (radial basis function) kernel (see section 2.3.4).

nu: ν must be within the interval $(0, 1]$ and is an "upper bound on the fraction of training errors and a lower bound of the fraction of support vectors" [52]. Thus, lower values will cause the model to fit the training data more accurately, while higher values will increasingly exclude present outliers in it.

gamma: γ is a parameter appearing as a coefficient in the rbf, sigmoid and polynomial kernel function. Like ν, it influences the trade-off between generalization and over-fitting. For the rbf kernel, $\gamma \to 0$ makes the kernel mappings converge at one point, which causes an over-generalized model without any classification ability, while $\gamma \to \infty$ results in over-fitting due to the kernel mappings being orthogonal. [63, p. 76]

degree: This parameter specifies the degree of the polynomial kernel and has no effect for other kernels.

coef0: Like `degree`, it only has an effect for the polynomial kernel where it appears as a constant coefficient.

tolerance: This parameter declares the tolerance for the stopping criterion of the one-class SVM.

[32] Suppose two additional feature rows in 5.3 with protocol numbers 42 and 23, which were not seen during the training phase. Both will be mapped to the one-hot feature vector `(0,0,0)`, which makes it impossible for the subsequent model components to distinguish them. However, this can be seen as a desired effect, as, given the training data, there should not be any special conclusion that can be made for one of them that cannot also be made for the other one.

shrinking: It can be set to `true` or `false` and specifies whether or not the so-called *shrinking heuristic* should be used. If turned on, `libsvm` tries to remove some instances in order to create a smaller optimization problem, which can be solved in shorter time. [15, p. 13]

cache size: The cache size has an impact on the runtime. With more available memory, it can be set higher and accelerate the training time.

max iter: If this value is set to a positive integer, it specifies the number of iterations after which the training will stop, even if the stopping criterion is not yet met. By default, there is no such fixed limit.

5.5.2. Autoencoder

The autoencoder implementation is realized with the libraries `keras` and `tensorflow`. During the training phase, it tries to fit the training instances with a low reconstruction error. Afterwards, a threshold is determined, which depends on the observed reconstruction errors using the training set. For determining whether a new instance is an anomaly or not, it is reconstructed by the autoencoder and the error is compared with the threshold: If it is greater, the instance is classified as an attack, otherwise as benign traffic.

The following parameter influence the behaviour of the autoencoder:

Loss Function: The loss function is used for training the autoencoder and for determining the reconstruction error. It can be chosen between the mean absolute error (`mae`) and mean squared error (`mse`).

Activation Function: The available activation functions are provided by `keras`. Among them are `relu`, `sigmoid`, `softmax`, `tanh`, and `exponential`.

Layer Sizes and Depth: This parameter is a textual representation of the autoencoder's architecture. It states the size of each layer, divided by commas, and thereby also specifies its depth (i.e. the number of layers). The layer sizes can be either given as *(i)* a fixed number, *(ii)* as a factor that is multiplied by the number of neurons in the previous layer or *(iii)* as a reference to a previous layer. For example, the value `"20,*0.5,5,#2,#1"` yields an autoencoder with five hidden layers, which respectively have 20, 10 ($= 20 \cdot 0.5$), 5, 10 (as the second layer), and 20 (as the first layer) neurons. As the size of the input layer (and therefore also those of the output layer) is influenced by the previous model component, it is not a modifiable hyperparameter and cannot be changed here.

Threshold Percentile: This value determines how many instances are used for determining the threshold during the training phase. It is expressed as the percentile of all training reconstruction errors and therefore must lie between zero and 100. If it is set to 100, the greatest observable reconstruction error is taken as the threshold.

Batch Size: It specifies the number of instances of a mini-batch. The dataset is split into multiple mini-batches for the training.

Epochs: This parameter determines the number of epochs for which the autoencoder is trained.

Early Stopping Patience: This parameter specifies the number of epochs after which the training stops early if the model's performance was not improved within them. If set to -1, the training is never stopped early.

Chapter 6 | Experiments

6.1. Overview and Experiment Setup

This chapter describes the experiments that were run with the implemented evaluation framework in the context of this thesis. They were executed on a computing cluster using the *Slurm Workload Manager*. The cluster provided an NVIDIA Tesla V100 GPU, which was utilized for the autoencoder tests, and altogether 512 GB RAM, of which between 20 GB and 320 GB were used for each run (depending on the expected memory consumption of the model components and the processed traffic). Furthermore, a sufficiently big file system was available, which was essential for persisting the datasets, their preprocessing results, the trained anomaly detection models, and evaluation measures.

Each experiment was designed for the evaluation of a single model component. Therefore, the other components were kept more or less fixed, while the parameters of the component that was the subject of the experiment were varied using a grid search approach.

The experiments were run on both the CIC-IDS-2017 and the UNSW-NB15 dataset. For both, the endorsed training and testing sets were ignored, and instead, the preprocessing results (cf. section 5.2) were used. For CIC-IDS-2017, the *Monday* set, which only contains benign traffic (ca. 11 GB), was used for the training, and the remaining workdays (*Tuesday* - *Friday*; ca. 39 GB; 44,660,731 packets) for the classification phase. For the UNSW-NB15 dataset, it was needed to define own splits, as those are not present in the dataset itself. It was observed, however, that certain PCAP files only contain benign traffic. Based on this fact, the splits displayed in table 6.1 are defined (and referred to hereinafter). As can be seen, not only one training set is provided, but instead two distinct ones, to make them of a more similar size as the CIC-IDS-2017 *Monday* subset. Most experiments in the following use the "A" training and "b" test split.

At first, different parameters for both of the implemented decision engines were examined, while using a fixed configuration for the network flow feature extractor. The latter is borrowed from reports of well-performing models in literature: the flow timeout is set to 12 seconds, the subflow timeout to 500 milliseconds and both the `tcp` and `subflows` mode are used (cf. section 5.3.1). The set of resulting features partly resembles those that would be generated by the CICFlowMeter or

Subset Name	Usage	PCAP files	Number of Packets	Size
A	Training	Day 1: `10.pcap` - `31.pcap`	(not determined)	ca. 21 GB
B	Training	Day 1: `32.pcap` - `53.pcap`	(not determined)	ca. 21 GB
a	Classification	Day 1: `1.pcap` - `9.pcap`	15,655,696	ca. 8.5 GB
b	Classification	Day 2: `1.pcap` - `14.pcap`	49,331,908	ca. 27 GB
c	Classification	Day 2: `15.pcap` - `27.pcap`	43,228,007	ca. 23 GB

Table 6.1.: Defined subsets of the UNSW-NB15 dataset

that are used in the UNSW-NB15 dataset. For the one-class SVMs, the rbf, sigmoid and polynomial kernels were investigated (whereas the examined parameters are mostly *nu* and *gamma*, as well as kernel-specific parameters). As a one-class SVM might yield better results with a reduced input dimensionality (cf. section 3.1), some tests incorporating PCA were run. The autoencoder experiments, on the other side, vary its depth, layer sizes and the threshold percentile, as well as its training batch size, the number of training epochs and the loss function.

Afterwards, well-performing approaches were used for comparing different feature extraction methods: the flow extractor's different modes were tested independently, as well as different values for the hindsight window and the flow and subflow timeouts. Moreover, experiments with the payload-analysing feature extractors from section 5.3.2 were run. Here, closer attention was given to finding out which attack types could be detected better by analysing the payload.

In all of the tests outlined above, the feature transformers applied one-hot encoding on categorial features, as well as min-max scaling and standardization (in this order), as described in section 5.4. This is based on the rationale that the implemented approaches are expected to benefit from those transformations. For completeness, different combinations of the feature transformers were tried out as well.

Lastly, the case of unclean training data (i.e. network traffic containing attacks), was simulated, due to its practical relevance (cf. section 2.3.3). In this way, a more profound conclusion regarding the influence of attacks in the training set can be drawn. In the end, a larger subset of the UNSW-NB15, almost encompassing the whole dataset, was used in order to evaluate the best-performing one-class SVM configuration.

For the evaluation of the made classifications, a set of measurements based on a generated confusion matrix was calculated, including precision, recall, MCC, balanced accuracy and the F1-score. As mentioned in section 2.4, the false detection rate can be inferred from the precision. Furthermore, the total count of false negatives and false positives are given in the following, so that the number of errors an approach is committing can be reasoned. Finally, considering the objective of comparing the approaches under practical aspects, the average classification time per packet is measured. Of course, this highly depends on the utilized implementations, which are not optimized, and further on the availability of system resources during the experiments, which could not be ensured to be stable. Nevertheless, this value can help to determine which approaches are faster than others, by considering its relative difference.

6.2. Results

The detailed results are shown in the respective tables in the appendix. It should be noted that the experiments sometimes were not executed in completeness due to technical complications on the computation cluster or because it became apparent that particular configurations do not improve the detection performance. Furthermore, models which did not classify any packet as an attack are not shown, as in this case no confusion matrix can be calculated due to the lack of any false or true positives. In any case, such models have no relevance in practical setups.

For a compact representation, the tables use the following abbreviations: *RC* for recall, *PR* for precision, *BA* for balanced accuracy, *F1* for the f1-score, *FN* for the false negatives, *FP* for the false positives and *ms/p* for the mean classification time per packet in milliseconds.

6.2.1. One-Class SVM

Tables D.8 and D.9 show the results for the experiments using the `rbf` kernel. The grid search here was executed over the parameters γ and ν. The parameters yielding the highest MCC of 87.26%

for the UNSW-NB15 dataset are $\gamma = 0.001$ and $\nu = 0.001$, with a tolerance of 0.01. Even if all other configurations yield a recall of 1, they result in lower precision values. Nevertheless, the best configuration found only has a precision of 77.78%, which might result in too many false alarms for a practical setup. The results for the CIC-IDS-2017 were much worse and overall showed precision values around 10%, which is unacceptable from a practical point of view. Accordingly, the best configuration only achieved an MCC of 19.01%. As later noted, it must be questioned whether the underlying packet labels used for the evaluation are accurate enough for permitting any meaningful inferences from the results for this dataset.

A little bit more promising for the CIC-IDS-2017 dataset is the polynomial kernel, which led to higher precision values, but at the same time much lower recalls. Two grid searches were executed for this kernel, one with varying values for the parameters `degree` and `coef0`, and the other for different γ and ν values (cf. tables D.10 and D.13). The best configuration here achieves a precision of 90.88% and an MCC of 30.76%. For the UNSW-NB15 dataset (cf. tables D.11 and D.12), on the other hand, the polynomial kernel here yielded worse results and achieved a maximum MCC of only 19.73%.

The grid search for the sigmoid kernel was not executed in detail, but only for some of the possible combinations. The results suggest that it is inferior to the rbf kernel for the UNSW-NB15 dataset and the polynomial kernel for the CIC-IDS-2017 dataset (cf. tables D.14 and D.15).

Table D.16 shows the combination of the PCA feature transformer with the one-class SVM (featuring the `rbf` and `polynomial` kernel). The assumption of implicating a better detection performance cannot be confirmed clearly, as no significant improvement can be observed. However, the classification times per packet are slightly shorter and for the UNSW-NB15 dataset, a configuration yielding a slightly better precision and MCC was found with 30 principal components.

The results for the hyperparameter search with different combinations for the feature transformers performing standardization, min-max scaling, and the one-hot encoding are shown in table D.17. It can be seen that, for a one-class SVM employing the network flow generator, standardization is the most important feature transformation step. Furthermore, if a min-max scaler is applied, its position should be before the standardization. Surprisingly, the one-hot encoding does not seem to have a big influence (the MCC values are even slightly higher without it); this could be due to the fact that the one-class SVM may handle smaller feature numbers better, or because there are not many features which are expanded by the one-hot encoder, so that it has not much effect.

6.2.2. Autoencoder

The experiments for the autoencoder first were executed by a grid search iterating the layer sizes, the threshold value, and the loss function, while using the `relu` activation function. The results are shown in the tables D.18 and D.19. For the CIC-IDS-2017 dataset, a similar trend as described above is observable: The best configuration only achieves an MCC of 17.93% and a precision of 12.66%. For the UNSW-NB15 dataset on the other hand, an MCC of 88.00% and a precision of 78.25% was achieved by an autoencoder employing the architecture `"*0.9,*0.8,*0.8,#2,#1"`, the mean squared error as the loss function, and a threshold percentile value of 99.99%. The best overall precision was 87.36%, but here, a recall of only 36.88% could be attained.

Table D.20 shows the results of the grid search that tried out different values for the training epochs and training batch size, as well as the early stopping patience. It shows that those parameters have no big influence on the detection ability, but smaller training batches seem to favour the resulting precision values, while greater ones yield higher MCCs.

6.2.3. Flow Feature Extractor

The experiments for the parameters of the network flow generator were only run with the UNSW-NB15 dataset, as the approaches examined so far achieved no convincing performance with the CIC-IDS-2017 dataset. A one-class SVM with the `rbf` kernel and the parameters $\gamma = 0.0005$ and $\nu = 0.001$ was utilized. Table D.23 shows a grid search iterating various values for the flow and subflow timeouts, while the `tcp` and `subflow` modes were applied. Due to the number of tested configurations, only those with a precision greater than 70% are shown in the table. It can be observed that the influence of both the flow and subflow timeout parameters is existent, but rather marginal, as the MCC values only range from 81.56% to 84.88%. The best found configuration consists of a flow timeout of 12 seconds and a subflow timeout of 0.55 seconds.

Table D.21 shows the best flow generator operation mode combinations for each of the attack categories in the UNSW-NB15 dataset. It should be noted that a direct comparison between the categories is not meaningful, as their number of attack packets varies (cf. section 4.5). However, the MCC can be used for comparing the performance of different configurations in the same category; and in this way, the most promising combinations were determined. Therefore, the table shows the configuration with the best MCC for each attack category. An insightful value, besides the balanced accuracy, is the recall, which indicates how many of the category's attack packets were found. In order to reduce the number of experiments, all of them utilized the `subflows` mode, except when the `basic` mode was turned on. The experiments were run with an autoencoder as the decision engine, featuring an architecture described by `"*0.9,*0.8,*0.8,#2,#1"`, the `relu` activation function and the mean squared error as the loss function. The results show that the best configurations include the `tcp` mode for all attack categories, and never use the `basic` mode. The `port_decimal` mode only improves the detection performance for the categories "worms" and "generic". For the former, this might be explainable with the fact that some worm implementations try to infect a certain range of ports, which is more easily recognizable by only considering parts of the port number. For the categories "fuzzers", "exploits", "reconnaissance", and "shellcode", the modes `with_ip_addr` and `ip_dotted` yield the best results in conjunction. For the "dos" category solely using `with_ip_addr` is better. Nevertheless, this shows that the inclusion of the IP addresses provides valuable information for the detection of attacks when using the UNSW-NB15 dataset. All of the categories described until now could be detected with recalls greater than 75%, and only the categories "backdoor" and "analysis" could not be recognized to a satisfactory extent.

The described experiments for the operation modes did not include the `hindsight` mode, since it was examined in the following. Here, the grid search varied the hindsight window, as well as the subflow timeout. Both the `tcp` and `subflows` modes were used. Moreover, the flow timeout was set to a fixed value of 6 seconds. The results in table D.22 indicate that the hindsight mode helped to detect both "backdoor" and "analysis" attacks, which can be detected with a recall of more than 93.19% and 55.60%, respectively. Further recall improvements can be made for the "worms", "fuzzers", and "dos" categories, while the recall values for "exploits" and "generic" decreased in comparison with table D.21. A hindsight window of 500 flows seems to be an acceptable value for practical circumstances, as with higher values, the classification time increases significantly. In comparison, a hindsight window of 2000 flows already raises it by a factor of approximately 1.5.

Finally, table D.24 shows the influence of the `tcp_end_on_rst` mode, which causes the flow generator to end flows when a `RST` TCP flag is observed. The experiments followed the setup of the autoencoder experiments utilizing the `"*0.9,*0.8,*0.8,#2,#1"` architecture. It can be seen that the results barely differ with this mode in comparison to table D.19. However, somewhat surprisingly, the experiments also revealed that the `subflows` mode, which was always utilized in the previous experiments, might not have a positive influence at all on the detection performance.

6.2.4. Payload Analysis

Table D.26 shows the results for the simple payload extractor which appends 256 features that represent the frequency of each byte in the payload. The experiments were run with an autoencoder similar to the previous section. It can be seen that the overall recall and precision decrease in comparison with the normal network flow generator (category “all”), but considerable recall values could can achieved for the categories “worms”, “shellcode”, “backdoor”, and “analysis”. It should be noted that the classification time per packet suffers and is almost twice as high as with the standard network flow generator.

The PAYL flow generator results with the autoencoder are shown in tables D.27 and D.28. Again, an `relu` autoencoder was used for determining the best values for the parameters `smoothing` and `clustering_threshold`. For the UNSW-NB15 dataset, the training set was reduced in order to cope with memory issues on the computation cluster, and only half of the `A` training set was used. It can be observed that the MCC slightly decreased for the “all” category in comparison with table D.19. In comparison with the results by attack categories from table D.26, the PAYL analyser could improve the MCC of the categories “analysis”, “exploits”, “fuzzers”, “reconnaisance”, “shellcode”, and “worms”. However, it had the highest of the observed classification times. On average, a packet needs 0.2679 milliseconds in order to be classified. For the CIC-IDS-2017 dataset, the PAYL analyser only was executed on the *wednesday* and *thursday* testing sets and only achieved a considerable performance for the `infiltration` category, for which an MCC of 71.63 % and a recall of almost 1 is achieved. However, the precision is at 51.63%, which means that almost every second packet that is classified as an attack is wrongly labelled as such. Furthermore, the other attack categories are barely detected better than by a random classifier.

Table D.29 shows the results of PAYL together with a one-class SVM featuring the rbf kernel ($\gamma = 0.0001, \nu = 0.0001$). They do not indicate any advantage of using PAYL in comparison with table D.9. This can be due to a too small training set, too different traffic profiles in the respective subsets, or because the traffic included in the dataset is not suited for payload analysis to a great extent, for example, because encryption methods on the application layer might have been used. Moreover, the implemented flow-based approach of PAYL could need further enhancement and it is not evident whether the selected features (cf. table D.7) are even suited for anomaly detection, even if this would be an intuitive rationale.

6.2.5. Unclean Training Data

These experiments, which were run with the autoencoder configuration from section 6.2.3, incorporated some attacks from the `a` subset to the `A` training split in the UNSW-NB15 dataset. As shown in table D.25, either one or two PCAP files with attacks are included, while the training phase operates as usual and assumes no attacks in the supplied data. As presumable, the performance detection goes down with a higher fraction of included attacks. It is notable however that mainly the recall is affected, while the precision roughly stays at the same level.

6.2.6. Bigger Subsets for the UNSW-NB15 Dataset

Table D.30 shows the results of a one-class SVM with a well-performing configuration from above (rbf kernel, $\gamma = 0.0001, \nu = 0.0001$) using almost the complete UNSW-NB15 dataset. Only the `a` testing split was spared. The results show a decrease of the MCC and recall; however, the precision for the complete traffic (category “all”) has improved in comparison to the former experiments, and is now at almost 89.30%. The other attack categories are shown for reference. It should be noted again that comparing the precision and MCC values between different categories does not provide meaningful insights. However, the results show that reasonable recall and balanced accuracy values can be achieved for the attack categories “worms”, “generic”, “dos”, and “exploits”.

Chapter 7 | Conclusion

7.1. Summary

With the present thesis, a systematic approach to evaluating and comparing anomaly-based network intrusion detection systems under practical aspects was proposed. What can be understood by *practical* was defined in the first chapter, where dataset independence, no reliance on the existence of class labels for the training, modern attack types, the system's runtime efficiency and low false detection rates were identified as crucial. Derived from those goals, the second chapter narrowed down the theoretical background and argued that one-class classification algorithms should be preferred for the anomaly-based detection of yet unknown attacks. The literature research shown in chapter 3 pointed out that most of the available related work disregards some of the mentioned points. Furthermore, it showed the overall lack of an objective, reproducible and generalized procedure for comparing different approaches to NIDS. In an attempt to address this very issue, chapter 4 proposed a concept for it, which was designed with the objective to minimize the gap between experimentation and real-world usage. It consists of the phases *preprocessing*, *training*, *classification*, and *evaluation*. A modular and flexible architecture for an anomaly-based detection model was shown, consisting of a feature extractor, feature transformers, and the decision engine. Notable here is the requirement of only taking network packets as the feature extractor's input, in order to achieve a practice-oriented and unbiased evaluation.

Chapter 5 then delved into the concrete implementation of this concept. Its first part showed the algorithm which was utilized in order to assign the corresponding traffic label (attack or benign) to each packet of the chosen datasets (UNSW-NB15 and CIC-IDS-2017). In addition, problems that arose from its application were discussed. For both datasets, a significant portion of packet labels could not be correctly identified, which caused problems for the later experiments. The root cause for those misclassifications could not be identified with absolute certainty, albeit both datasets show inaccuracies in their provided data. For the CIC-IDS-2017 dataset, those appeared to be more severe, as many packets belonging to attacks were not identified as such. The remainder of the chapter described prototypical implementations of the anomaly detection model components, such as a flow extractor, a packet payload analyser (PAYL), a one-hot encoder, a one-class support vector machine, and an autoencoder. Those components were then compared to each other in chapter 6, in order to demonstrate the capabilities of the implemented evaluation system.

The experiment results showed a big difference between both datasets. For the CIC-IDS-2017 dataset, only a few configurations had notable better measurements than a random classifier would. It must therefore be questioned whether the preprocessing for this dataset led to an unacceptable extent of inaccuracy for the packet labels, which would make an insightful evaluation impossible. For the UNSW-NB15 dataset, on the other hand, notable configurations for the autoencoder and the one-class SVM could be found. However, both showed relatively high false detection rates of around 20%, which is probably unacceptable for practical setups. No significant advantage of payload analysers could be found; in fact, the overall detection performance sometimes even decreases. Additionally, they had the costs of a significantly higher runtime, which must be considered as well in practice.

To conclude, the experiments could not find approaches that are particularly compelling for being used in practical setups, even if some of them are promising and could be further investigated. More importantly in this regard, the present thesis laid the foundation for an objective and reproducible evaluation procedure that permits to make statements about the practicality of implemented detection models, and which can be used in future for evaluating other anomaly-based approaches to NIDS. The next, and last, section gives suggestions for further work that could be built upon this foundation.

7.2. Future Work

A crucial point for improving the evaluation quality is the provision of accurate packet-wise labels. This can be done by either further investigating the utilized datasets in order to eliminate the described inaccuracies (albeit, due to the problems described in 5.2.2, this might not be achievable), by reviewing other datasets which provide such data in a more easily-accessible manner, or by creating a dedicated dataset with this objective. Before this is achieved, results based on the CIC-IDS-2017 dataset should be interpreted with caution.

As the described anomaly detection approaches should only be understood as prototypical demonstrations for the evaluation system, they leave much room for further enhancement. Besides using autoencoders for the compression of the extracted features (i.e. as feature transformers), this includes the usage of a more complex payload analysis, as shortly introduced in section 3.3. Both deep learning techniques, but also shallow approaches developed for natural language processing, such as word or document embeddings, are promising candidates for this.

The developed network flow feature generator can easily be extended in order to support a broader range of protocols. The incorporation of application layer protocols seems to be encouraging; however, here, the issue of not being able to handle encrypted traffic must be considered. The aggregation of network traffic based on sessions could be a potential enhancement.

Furthermore, some changes to the proposed evaluation concept can be imagined. Allowing updates of an anomaly detection model by giving it feedback about its made classification is an enhancement that could be relevant in practice. Moreover, a functionality for determining the influence of the training's data size on the model's performance could be of similar interest. Additionally, a more differentiated inspection of the false alarms on a flow-based, instead of a packet-based, level could help to define a notion of the concept *alarm* which might be more suitable for practical needs: With the previously proposed concept, each miss-classified packet is equalized with a *false alarm*, whereas in a practical system, an alarm could consist of multiple packets which belong together.

Lastly, the described hyperparameter search can be replaced by more sophisticated methods which use the results of previous runs as feedback in order to optimize the parameter configurations. An efficient and effective tuning process could enable researchers and developers to design performant anomaly detection models with more ease. Helpful could be the ability to evaluate and adjust models using incrementally increasing subsets of the datasets. In this way, unpromising approaches or configurations could be filtered out quickly. Conceivably, by this means an "evaluation-driven" development of anomaly-based NIDS can be made possible.

Appendix A | Bibliography

[1] M. Ahmed, A. Mahmood, and J. Hu, *A Survey of Network Anomaly Detection Techniques*, Journal of Network and Computer Applications, 60 (2015), pp. 19–31.

[2] J. An and S. Cho, *Variational autoencoder based anomaly detection using reconstruction probability*, Special Lecture on IE, 2 (2015), pp. 1–18.

[3] M. Barth, N. Hellemann, T. Kob, C. Krösmann, U. Morgenstern, T. Tschersich, T. Ritter, H. Shulman, D. Trapp, and R. Wintergerst, *Spionage, Sabotage und Datendiebstahl – Wirtschaftsschutz in der vernetzten Welt - Studienbericht 2020. (German) [Espionage, Sabotage and Data Theft - Economic Protection in the Networked World - Study Report 2020]*. Bitkom e. V., 2020.

[4] R. Basnet, R. Shash, C. Johnson, L. Walgren, and T. Doleck, *Towards Detecting and Classifying Network Intrusion Traffic Using Deep Learning Frameworks*, (2019).

[5] X. Bellekens, H. Hindy, R. Atkinson, C. Tachtatzis, D. Brosset, E. Bayne, and A. Seeam, *A Taxonomy and Survey of Intrusion Detection System Design Techniques, Network Threats and Datasets*, (2018).

[6] D. K. Bhattacharyya and J. Kalita, *Network Anomaly Detection: A Machine Learning Perspective*, 04 2013.

[7] M. Bhuyan, D. K. Bhattacharyya, and J. Kalita, *Network Traffic Anomaly Detection and Prevention: Concepts, Techniques, and Tools*, 01 2017.

[8] E. Biglar Beigi, H. Hadian Jazi, N. Stakhanova, and A. A. Ghorbani, *Towards effective feature selection in machine learning-based botnet detection approaches*, in 2014 IEEE Conference on Communications and Network Security, 2014, pp. 247–255.

[9] L. Bilge and T. Dumitras, *Before we knew it: An empirical study of zero-day attacks in the real world*, 10 2012, pp. 833–844.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2007.

[11] Bundesamt für Sicherheit in der Informationstechnik (BSI), *Die Lage der IT-Sicherheit in Deutschland 2020. (German) [The Situation of IT Security in Germany in 2020]*, September 2020.

[12] B. Çakir, *Zero-Day Attack Detection with Deep Learning*, 2019.

[13] Canadian Institute for Cybersecurity, *Applications, CICFlowMeter (formerly ISCXFlowMeter)*. `https://www.unb.ca/cic/research/applications.html#CICFlowMeter`, 2020. [Online; accessed on 13 December 2020].

[14] ——, *CSE-CIC-IDS2018 on AWS*. `https://www.unb.ca/cic/datasets/ids-2018.html`,

2020. [Online; accessed on 29 December 2020].

[15] C.-C. Chang and C.-J. Lin, *Training v-support vector classifiers: theory and algorithms*, Neural computation, 13 (2001), pp. 2119–2147.

[16] ——, *LIBSVM: A library for support vector machines*, ACM transactions on intelligent systems and technology (TIST), 2 (2011), pp. 1–27.

[17] D. Chicco and G. Jurman, *The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation*, BMC genomics, 21 (2020), pp. 1–13.

[18] C. Eckert, *IT-Sicherheit: Konzepte - Verfahren - Protokolle. (German) [IT Security: Concepts - Methods - Protocols]*, De Gruyter Studium, 2018.

[19] C. Estan, K. Keys, D. Moore, and G. Varghese, *Building a better NetFlow*, ACM SIGCOMM Computer Communication Review, 34 (2004), pp. 245–256.

[20] Europol Press Release, *World's most dangerous malware EMOTET disrupted through global action.* `https://www.europol.europa.eu/newsroom/news/world%E2%80%99s-most-dangerous-malware-emotet-disrupted-through-global-action`, January 2021. [Online; accessed on 02 February 2021].

[21] M. A. Ferrag, L. Maglaras, S. Moschoyiannis, and H. Janicke, *Deep Learning for Cyber Security Intrusion Detection: Approaches, Datasets, and Comparative Study*, Journal of Information Security and Applications, 50 (2019).

[22] S. Gatlan, *Emotet malware hits Lithuania's National Public Health Center.* BleepingComputer `https://www.bleepingcomputer.com/news/security/emotet-malware-hits-lithuanias-national-public-health-center/`, December 2020. [Online; accessed on 02 February 2021].

[23] K. Ghanem, F. J. Aparicio-Navarro, K. G. Kyriakopoulos, S. Lambotharan, and J. A. Chambers, *Support vector machine for network intrusion and cyber-attack detection*, in 2017 Sensor Signal Processing for Defence Conference (SSPD), IEEE, 2017, pp. 1–5.

[24] M. Gharib, B. Mohammadi, S. H. Dastgerdi, and M. Sabokrou, *AutoIDS: Auto-encoder Based Method for Intrusion Detection System*, arXiv preprint arXiv:1911.03306, (2019).

[25] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*, vol. 1, MIT press Cambridge, 2016.

[26] S. Hansman, *A taxonomy of network and computer attack methodologies*, (2003).

[27] S. Hansman and R. Hunt, *A taxonomy of network and computer attacks*, Computers and Security, 24 (2005), pp. 31–43.

[28] H. Hindy, R. Atkinson, C. Tachtatzis, J.-N. Colin, E. Bayne, and X. Bellekens, *Towards an Effective Zero-Day Attack Detection Using Outlier-Based Deep Learning Techniques*, 2020.

[29] K. Ingham and H. Inoue, *Comparing Anomaly Detection Techniques for HTTP*, 09 2007, pp. 42–62.

[30] A. Javaid, Q. Niyaz, W. Sun, and M. Alam, *A Deep Learning Approach for Network*

Intrusion Detection System, vol. 3, 12 2015.

[31] H. H. Jazi, H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, *Detecting HTTP-based application layer DoS attacks on web servers in the presence of sampling*, Computer Networks, 121 (2017), pp. 25 – 36.

[32] V. Kanimozhi and T. P. Jacob, *Calibration of Various Optimized Machine Learning Classifiers in Network Intrusion Detection System on the Realistic Cyber Dataset CSE-CIC-IDS2018 Using Cloud Computing*, International Journal of Engineering Applied Sciences and Technology, 4 (2019), pp. 2455–2143.

[33] K. Kostas, *Anomaly Detection in Networks Using Machine Learning*, 08 2018.

[34] F. Kuang, W. Xu, and S. Zhang, *A novel hybrid KPCA and SVM with GA model for intrusion detection*, Applied Soft Computing, 18 (2014), pp. 178–184.

[35] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach, 6th edition*, 03 2012.

[36] P. Lin, K. Ye, and C.-Z. Xu, *Dynamic Network Anomaly Detection System by Using Deep Learning Techniques*, 06 2019, pp. 161–176.

[37] H. Liu and B. Lang, *Machine Learning and Deep Learning Methods for Intrusion Detection Systems: A Survey*, Applied Sciences, 9 (2019), p. 4396.

[38] B. Lypa, O. Iver, and V. Kifer, *Application of machine learning methods for network intrusion detection system*, (2019).

[39] Melissa Eddy and Nicole Perlroth, *Cyber Attack Suspected in German Woman's Death.* The New York Times `https://www.nytimes.com/2020/09/18/world/europe/cyber-attack-germany-ransomeware-death.html`, September 2020. [Online; accessed on 02 February 2021].

[40] E. Min, J. Long, Q. Liu, J. Cui, and W. Chen, *TR-IDS: Anomaly-Based Intrusion Detection through Text-Convolutional Neural Network and Random Forest*, Security and Communication Networks, 2018 (2018), pp. 1–9.

[41] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, *Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection*, (2018).

[42] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, *A Detailed Investigation and Analysis of Using Machine Learning Techniques for Intrusion Detection*, IEEE Communications Surveys Tutorials, 21 (2019), pp. 686–728.

[43] N. Moustafa, J. Hu, and J. Slay, *A holistic review of Network Anomaly Detection Systems: A comprehensive survey*, Journal of Network and Computer Applications, 128 (2018).

[44] N. Moustafa and J. Slay, *UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)*, 11 2015.

[45] Z. Noumir, P. Honeine, and C. Richard, *On simple one-class classification methods*, in 2012 IEEE International Symposium on Information Theory Proceedings, IEEE, 2012, pp. 2022–2026.

[46] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee, *McPAD: A multiple classifier system for accurate payload-based anomaly detection*, Computer networks, 53 (2009),

pp. 864–881.

[47] K. Potdar, T. S. Pardawala, and C. D. Pai, *A comparative study of categorical variable encoding techniques for neural network classifiers*, International journal of computer applications, 175 (2017), pp. 7–9.

[48] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, *A Survey of Network-based Intrusion Detection Data Sets*, (2019).

[49] J. Schmidt, *Trojaner-Befall: Uni Gießen nutzt Desinfec't für Aufräumarbeiten. (German [Trojan Infection: University Gießen Uses Desinfec't for Clean-Up Operation].* heise online `https://www.heise.de/security/meldung/Trojaner-Befall-Uni-Giessen-nutzt-Desinfec-t-fuer-Aufraeumarbeiten-4617154.html`, December 2019. [Online; accessed on 02 February 2021].

[50] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, *Estimating the support of a high-dimensional distribution*, Neural computation, 13 (2001), pp. 1443–1471.

[51] scikit-learn developers, *scikit-learn 0.24.0 documentation - 1.4. Support Vector Machines / 1.4.6. Kernel functions.* `https://scikit-learn.org/stable/modules/svm.html#svm-kernels`, 2021. [Online; accessed on 31 January 2021].

[52] ——, *scikit-learn 0.24.0 documentation - sklearn.svm.OneClassSVM.* `https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html`, 2021. [Online; accessed on 18 January 2021].

[53] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani, *Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization*, 01 2018, pp. 108–116.

[54] C. So-In, *A Survey of Network Traffic Monitoring and Analysis Tools*, (2006).

[55] T. Tang, L. Mhamdi, S. Zaidi, F. El-moussa, D. McLernon, and M. Ghogho, *A deep learning approach combining auto-encoder with one-class SVM for DDoS attack detection in SDNs*, in Proceedings of the international conference on communications and networking, IEEE, 2019.

[56] D. Tax, *One-Class Classification; Concept-Learning In The Absence Of Counter-Examples*, (2001).

[57] E. Viegas, A. Santin, and L. Soares de Oliveira, *Toward a Reliable Anomaly-Based Intrusion Detection in Real-World Environments*, Computer Networks, 127 (2017).

[58] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman, *Deep learning approach for intelligent intrusion detection system*, IEEE Access, 7 (2019), pp. 41525–41550.

[59] K. Wang, G. Cretu, and S. J. Stolfo, *Anomalous payload-based worm detection and signature generation*, in International Workshop on Recent Advances in Intrusion Detection, Springer, 2005, pp. 227–246.

[60] K. Wang and S. Stolfo, *Anomalous Payload-Based Network Intrusion Detection*, vol. 3224, 09 2004, pp. 203–222.

[61] W. Wang, Y. Sheng, J. Wang, X. Zeng, X. Ye, Y. Huang, and M. Zhu, *HAST-IDS: Learning Hierarchical Spatial-Temporal Features Using Deep Neural Networks to Improve*

Intrusion Detection, IEEE Access, 6 (2018), pp. 1792–1806.

[62] P. Winter, E. Hermann, and M. Zeilinger, *Inductive intrusion detection in flow-based network data using one-class support vector machines*, in 2011 4th IFIP international conference on new technologies, mobility and security, IEEE, 2011, pp. 1–5.

[63] Y. Xiao, H. Wang, L. Zhang, and W. Xu, *Two methods of selecting Gaussian kernel parameters for one-class SVM and their application to fault detection*, Knowledge-Based Systems, 59 (2014), pp. 75–84.

[64] M. Zhang, B. Xu, and J. Gong, *An anomaly detection model based on one-class svm to detect network intrusions*, in 2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN), IEEE, 2015, pp. 102–107.

[65] J. Zhao, S. Shetty, J. Pan, C. Kamhoua, and K. Kwiat, *Transfer learning for detecting unknown network attacks*, EURASIP Journal on Information Security, 2019 (2019).

[66] X. Zhu, *Semi-Supervised Learning Literature Survey*, Comput Sci, University of Wisconsin-Madison, 2 (2008).

[67] A. Özgür and H. Erdem, *A review of KDD99 dataset usage in intrusion detection and machine learning between 2010 and 2015*, (2016).

Appendix B | List of Figures

2.1. The Internet Protocol Stack with examples for each layer. 4
2.2. Typical timeline of a vulnerability, adapted from [9, p. 3]. 8
2.3. Schematic representation of a training dataset for a supervised learning algorithm 10
2.4. Training dataset for an unsupervised learning algorithm 11
2.5. Training dataset for a semi-supervised learning algorithm 12
2.6. Training dataset and possible boundary for a one-class classification learning algorithm 13
2.7. Schematic illustration of an overcomplete (on the left side) and undercomplete autoencoder (on the right side) 16

4.1. Conceptualisation of the training phase 28
4.2. Depiction of a trained feature extraction function e. From an arbitrary number of packets of varying length it extracts instances with a fixed number of features, together with the corresponding feature type information. 29
4.3. Conceptualisation of the classification phase 31
4.4. Conceptualisation of the evaluation phase 32
4.5. Conceptualisation of the hyperparameter grid search. 34

5.1. Illustration of three problems that occur while preprocessing the CIC-IDS-2017 dataset 39
5.2. Division of a flow into subflows as well as idle and active times 44
5.3. Example of the feature transformation using a one-hot encoder 47

Appendix C | List of Tables

2.1. IP packet segments in IPv4 5
2.2. Relationships between learning methods and requirements for their training dataset 13
2.3. Confusion matrix of the underlying binary classification problem. 17
2.4. Relevant properties of datasets dedicated to network intrusion detection 20

5.1. Preprocessing Measurements for UNSW-NB15 and CIC-IDS-2017 40
5.2. Preprocessing Result of the UNSW-NB15 dataset 41
5.3. Preprocessing Result of the CIC-IDS-2017 dataset 41

6.1. Defined subsets of the UNSW-NB15 dataset 50

D.1. Default features extracted from the entire flow ix
D.2. Default features extracted from packets in forward direction x
D.3. Default features extracted from packets in backward direction x
D.4. Extracted features with the `subflows` mode xi
D.5. Extracted features with the `tcp` mode xii
D.6. Extracted features with the `hindsight` mode xiii
D.7. Extracted features using PAYL xiii
D.8. Results for the rbf kernel on the CIC-IDS-2017 dataset xiv
D.9. Results for the rbf kernel on the UNSW-NB15 dataset xiv
D.10. Results for the polynomial kernel on the CIC-IDS-2017 dataset, trying out different values for *degree* and $coef0$ xv
D.11. Results for the polynomial kernel on the UNSW-NB15 dataset, trying out different values for *degree* and $coef0$ xvi
D.12. Results for the polynomial kernel on the UNSW-NB15 dataset, trying out different values for γ and ν with fixed $degree = 3$ xvi
D.13. Results for the polynomial kernel on the CIC-IDS-2017 dataset, trying out different values for γ and ν with fixed $degree = 3$ xvi
D.14. Results for the sigmoid kernel on the CIC-IDS-2017 dataset. xvii
D.15. Results for the sigmoid kernel on the UNSW-NB15 dataset. xvii
D.16. Results for the combination of PCA and a one-class SVM on the UNSW-NB15 dataset (`A` training set and `b` test set). Only parameter configurations with $MCC > 0.5$ are shown. xviii
D.17. Results for the grid search which iterates the feature transformers, while a one-class SVM is used (rbf kernel, $\gamma = 0.0001, \nu = 0.0001$, `tolerance`$= 0.01$) xviii
D.18. Results for the autoencoder on the CIC-IDS-2017 dataset with the `relu` activation function and varying architectures xix
D.19. Results for the autoencoder on the UNSW-NB15 dataset with the `relu` activation function and varying architectures. Always, a training batch size of 1024 and 200 training epochs are used. xx
D.20. Results for the autoencoder on the UNSW-NB15 dataset with the `relu` activation function, MSE loss function, and varying values for `training_epochs`, `training_batch`, and `early_stopping_patience` xxi
D.21. Best network flow mode combinations by attack category in the UNSW-NB15 dataset (with the `A` training and `b` test subset) xxi

D.22. Best hindsight window values by attack category in the UNSW-NB15 dataset . . . xxii
D.23. Results for the grid search which iterates the flow and subflow timeouts xxii
D.24. Influence of the `tcp_end_on_rst` mode on the UNSW-NB15 dataset (with the `A` training and `b` test subset) xxiii
D.25. Results for unclean training sets (UNSW-NB15) xxiii
D.26. Best results for the simple payload analyser counting byte frequencies in combination with an autoencoder xxiv
D.27. Best results for the PAYL generator on the UNSW-NB15 dataset, in combination with an autoencoder xxiv
D.28. Best results for the PAYL generator on the CIC-IDS-2017 dataset (test sets: *wednesday* and *thursday*), in combination with an autoencoder xxiv
D.29. Best results for the PAYL generator on the UNSW-NB15 dataset, in combination with a one-class SVM xxv
D.30. Results using a larger UNSW-NB15 subset (training: `A+B`, test: `b+c`) for a model consisting of the network flow feature extractor (flow timeout: 12 seconds, subflow timeout: 0.5 seconds, modes: `tcp` and `subflows`), the one-hot encoder, a PCA transformer with 30 principal components, min-max scaling, standardization, and a one-class SVM ($\gamma = 0.0001$, $\nu = 0.0001$) xxv

Appendix D | Tables

Name	Description	Type
`src_port`	the source port of the transport protocol, or 0 if the protocol is either not known or does not specify any ports	int
`dest_port`	the destination port, analogous to `src_port`	int
`protocol`	the IP protocol number, as read from the packet and used for the flow groupings	categorial
`total_sum_pkt_length`	the sum of all packet sizes within the flow	int
`total_mean_pkt_length`	the arithmetic mean of all packet sizes within the flow	float
`total_min_pkt_length`	the minimum packet size within the flow	int
`total_max_pkt_length`	the maximum packet size within the flow	int
`total_std_pkt_length`	the standard deviation of all packet sizes within the flow	float
`total_n_packets`	the total number of packets within the flow	int
`total_packets_per_s`	the average packet throughput of the flow, measured in *packets per seconds*	float
`total_bytes_per_s`	the average payload throughput of the flow in *bytes per seconds*	float
`total_avg_ttl`	the arithmetic mean of the time-to-live values, as read from the IP headers	float
`total_iat_std`	the standard deviation of the timespans between consecutive packets in seconds	float
`total_iat_min`	the minimum timespan between consecutive packets in seconds	float
`total_iat_max`	the maximum timespan between consecutive packets in seconds	float
`total_iat_sum`	the sum of the timespans between consecutive packets; this equals the overall duration of the flow in seconds	float

Table D.1.: Default features extracted from the entire flow

Name	Description	Type
fwd_sum_pkt_length	the sum of all packet sizes in forward direction	int
fwd_mean_pkt_length	the arithmetic mean of all packet sizes in forward direction	float
fwd_min_pkt_length	the minimum packet size in forward direction	int
fwd_max_pkt_length	the maximum packet size in forward direction	int
fwd_std_pkt_length	the standard deviation of all packet sizes in forward direction	float
fwd_n_packets	the total number of packets in forward direction	int
fwd_packets_per_s	the average packet throughput in forward direction, measured in *packets per seconds*	float
fwd_bytes_per_s	the average payload throughput in forward direction, measured in *bytes per seconds*	float
fwd_avg_ttl	the arithmetic mean of the time-to-live values, as read from the IP headers of the packets in forward direction	float
fwd_iat_std	the standard deviation of the timespans between consecutive packets in forward direction, in seconds	float
fwd_iat_min	the minimum timespan between consecutive packets in forward direction, in seconds	float
fwd_iat_max	the maximum timespan between consecutive packets in forward direction, in seconds	float
fwd_iat_sum	the sum of the timespans between consecutive packets in forward direction, in seconds	float

Table D.2.: Default features extracted from packets in forward direction

Name	Description	Type
bwd_sum_pkt_length	the sum of all packet sizes in backward direction	int
bwd_mean_pkt_length	the arithmetic mean of all packet sizes in backward direction	float
bwd_min_pkt_length	the minimum packet size in backward direction	int
bwd_max_pkt_length	the maximum packet size in backward direction	int
bwd_std_pkt_length	the standard deviation of all packet sizes in backward direction	float
bwd_n_packets	the total number of packets in backward direction	int
bwd_packets_per_s	the average packet throughput in backward direction, measured in *packets per seconds*	float
bwd_bytes_per_s	the average payload throughput in backward direction, measured in *bytes per seconds*	float
bwd_avg_ttl	the arithmetic mean of the time-to-live values, as read from the IP headers of the packets in backward direction	float
bwd_iat_std	the standard deviation of the timespans between consecutive packets in backward direction, in seconds	float
bwd_iat_min	the minimum timespan between consecutive packets in backward direction, in seconds	float
bwd_iat_max	the maximum timespan between consecutive packets in backward direction, in seconds	float
bwd_iat_sum	the sum of the timespans between consecutive packets in backward direction, in seconds	float

Table D.3.: Default features extracted from packets in backward direction

Name	Description	Type
n_subflows	the number of subflows within the flow	int
n_active_times	the number of active times	int
min_active_time	the minimum duration of a subflow, in seconds	float
max_active_time	the maximum duration of a subflow, in seconds	float
total_active_times	the summed duration of all subflows, in seconds	float
std_active_time	the standard deviation of the durations of all subflows, in seconds	float
mean_active_time	the mean of the duration of all idle phases in the flow, in seconds	float
n_idle_times	the number of idle times	int
min_idle_time	the minimum timespan between two subflows, in seconds	float
max_idle_time	the maximum timespan between two subflows, in seconds	float
total_idle_times	the overall time (in seconds) in which no subflow was active, i.e. in which the flow idled	float
std_idle_time	the standard deviation of the duration of all idle phases within the flow, in seconds	float
mean_idle_time	the mean of the duration of all idle phases in the flow, in seconds	float
fwd_subflow_avg_pkts	the average number of packets per subflow in forward direction	float
fwd_subflow_avg_length	the average subflow length in forward direction	float
bwd_subflow_avg_pkts	the average number of packets per subflow in backward direction	float
bwd_subflow_avg_length	the average subflow length in backward direction	float

Table D.4.: Extracted features with the subflows mode

Name	Description	Type
tcp_fwd_win_mean	the arithmetic mean of the window value of all TCP packets in forward direction	float
tcp_fwd_total_urg	the number of TCP packets in forward direction with the URG flag set	int
tcp_fwd_total_syn	the number of TCP packets in forward direction with the SYN flag set and the ACK flag not set	int
tcp_fwd_total_syn_ack	the number of TCP packets in forward direction with the SYN and ACK flags set	int
tcp_fwd_total_ack	the number of TCP packets in forward direction with the ACK flag set and the SYN flag not set	int
tcp_fwd_total_fin	the number of TCP packets in forward direction with the FIN flag set	int
tcp_fwd_total_push	the number of TCP packets in forward direction with the PUSH flag set	int
tcp_fwd_total_rst	the number of TCP packets in forward direction with the RST flag set	int
tcp_bwd_win_mean	the arithmetic mean of the window value of all TCP packets in backward direction	float
tcp_bwd_total_urg	the number of TCP packets in backward direction with the URG flag set	int
tcp_bwd_total_syn	the number of TCP packets in backward direction with the SYN flag set and the ACK flag not set	int
tcp_bwd_total_syn_ack	the number of TCP packets in backward direction with the SYN and ACK flags set	int
tcp_bwd_total_ack	the number of TCP packets in backward direction with the ACK flag set and the SYN flag not set	int
tcp_bwd_total_fin	the number of TCP packets in backward direction with the FIN flag set	int
tcp_bwd_total_push	the number of TCP packets in backward direction with the PUSH flag set	int
tcp_bwd_total_rst	the number of TCP packets in backward direction with the RST flag set	int
tcp_syn_synack	the duration between the first SYN packet and the first SYN-ACK packet from the reverse direction	float
tcp_synack_ack	the duration between the SYN-ACK packet (as in tcp_syn_synack) and the first ACK packet from the reverse direction	float
tcp_rtt	the round-trip time, calculated as tcp_syn_synack + tcp_synack_ack	float

Table D.5.: Extracted features with the tcp mode

Name	Description	Type
`hindsight_dest_addr_src_port`	the number of flows with the same destination IP address and source port within the hindsight window	int
`hindsight_src_addr_dest_port`	the number of flows with the same source IP address and destination port within the hindsight window	int
`hindsight_src_addr`	the number of flows with the same source IP address within the hindsight window	int
`hindsight_dest_addr`	the number of flows with the same destination IP address within the hindsight window	int
`hindsight_dest_addr_prot`	the number of flows with the same destination IP address and IP protocol number within the hindsight window	int
`hindsight_src_addr_prot`	the number of flows with the same source IP address and IP protocol number within the hindsight window	int

Table D.6.: Extracted features with the `hindsight` mode

Name	Description	Type
`mean_payl_dist`	the arithmetic mean of the Mahalanobis distance values	float
`min_payl_dist`	the minimum of the Mahalanobis distance values	float
`max_payl_dist`	the maximum of the Mahalanobis distance values	float
`std_payl_dist`	the standard deviation of the Mahalanobis distance values	float

Table D.7.: Extracted features using PAYL
They base on the list of Mahalanobis distance values that are calculated for a flow's packets

TRAIN	TEST	gamma	nu	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
Monday	Tuesday, Wednesday, Thursday, Friday	0.0001	0.0001	0.0100	0.0449	0.0096	0.3080	-0.2189	0.0159	3623477	17531238	0.1971
Monday	Tuesday, Wednesday, Thursday, Friday	0.0001	0.0005	0.0100	0.0508	0.0107	0.3066	-0.2197	0.0176	3601418	17884229	0.1897
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	0.0100	0.4411	0.0802	0.4858	-0.0159	0.1357	2120550	19189538	0.1936
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0005	0.0100	0.4498	0.0815	0.4897	-0.0115	0.1380	2087609	19223205	0.1884
Monday	Tuesday, Wednesday, Thursday, Friday	0.0500	0.0001	0.0100	0.7912	0.1217	0.6305	0.1464	0.2110	792136	21664232	0.1969
Monday	Tuesday, Wednesday, Thursday, Friday	0.0500	0.0005	0.0100	0.7896	**0.1233**	**0.6341**	0.1501	**0.2132**	798194	21310601	0.2108
Monday	Tuesday, Wednesday, Thursday, Friday	0.6000	0.0001	0.0100	**1.0000**	0.0857	0.5047	0.0283	0.1578	0	40485952	0.2190
Monday	Tuesday, Wednesday, Thursday, Friday	0.6000	0.0005	0.0100	0.9735	0.1097	0.6200	**0.1554**	0.1972	100374	29975951	0.3775

Table D.8.: Results for the rbf kernel on the CIC-IDS-2017 dataset

TRAIN	TEST	gamma	nu	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	0.0001	0.0001	0.0100	0.9899	**0.7778**	0.9897	**0.8726**	**0.8711**	17808	498804	0.1908
A	b	0.0001	0.0005	0.0100	**1.0000**	0.7072	**0.9923**	0.8345	0.8285	0	730004	0.1883
A	b	0.0050	0.0001	0.0100	**1.0000**	0.5453	0.9845	0.7269	0.7057	0	1470724	0.1768
A	b	0.0050	0.0005	0.0100	**1.0000**	0.6493	0.9900	0.7977	0.7873	0	952649	0.1792
A	b	0.0500	0.0001	0.0100	**1.0000**	0.3901	0.9710	0.6062	0.5612	0	2757721	0.1875
A	b	0.0500	0.0005	0.0100	**1.0000**	0.4513	0.9775	0.6565	0.6219	0	2144134	0.1906
A	b	0.6000	0.0001	0.0100	**1.0000**	0.0398	0.5533	0.0651	0.0766	0	42502068	0.2151

Table D.9.: Results for the rbf kernel on the UNSW-NB15 dataset

TRAIN	TEST	gamma	nu	degree	coef0	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	3	0.0000	0.0010	0.1135	**0.9160**	**0.5562**	**0.3074**	**0.2019**	3363538	39456	0.2021
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	4	0.0000	0.0010	0.0422	0.9137	0.5209	0.1866	0.0807	3633761	15133	0.1966
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	5	0.0000	0.0010	0.0907	0.8704	0.5447	0.2663	0.1643	3449714	51256	0.1920
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	7	0.0000	0.0010	0.2018	0.0542	0.4373	-0.0751	0.0854	3028525	13366532	0.1868
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	10	0.0000	0.0010	0.2288	0.0423	0.3737	-0.1413	0.0714	2925773	19673963	0.2034
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	3	0.2000	0.0010	0.0043	0.6633	0.5021	0.0489	0.0086	3777561	8339	0.1897
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	4	0.2000	0.0010	0.0061	0.6682	0.5029	0.0585	0.0122	3770685	11573	0.1925
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	5	0.2000	0.0010	0.0083	0.7020	0.5040	0.0703	0.0165	3762359	13425	0.1855
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	7	0.2000	0.0010	0.0127	0.7435	0.5061	0.0899	0.0249	3745976	16563	0.1957
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	10	0.2000	0.0010	0.0383	0.6277	0.5181	0.1405	0.0722	3648752	86130	0.1925
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	20	0.2000	0.0010	**0.6394**	0.0995	0.5512	0.0573	0.1723	1368268	21943377	0.1933
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	3	1.0000	0.0010	0.0036	0.6765	0.5017	0.0449	0.0071	3780448	6475	0.1859
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	4	1.0000	0.0010	0.0029	0.6374	0.5014	0.0391	0.0058	3782902	6307	0.1868
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	5	1.0000	0.0010	0.0033	0.6521	0.5015	0.0419	0.0065	3781644	6586	0.1952
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	7	1.0000	0.0010	0.0039	0.6863	0.5019	0.0473	0.0077	3779277	6724	0.1898
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	10	1.0000	0.0010	0.0044	0.6449	0.5021	0.0484	0.0087	3777305	9184	0.1861
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	3	0.0000	0.0010	0.0864	0.7548	0.5419	0.2381	0.1551	3466036	106553	0.2092
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	4	0.0000	0.0010	0.0852	0.8167	0.5417	0.2482	0.1543	3470774	72557	0.2040
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	5	0.0000	0.0010	0.1062	0.5634	0.5493	0.2190	0.1788	3390883	312380	0.2102
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	7	0.0000	0.0010	0.1059	0.5702	0.5493	0.2204	0.1787	3392118	302885	0.2087
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	3	0.2000	0.0010	0.0569	0.7579	0.5276	0.1934	0.1059	3577958	68998	0.2055
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	4	0.2000	0.0010	0.0677	0.7811	0.5330	0.2150	0.1246	3537155	71969	0.2280
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	7	0.2000	0.0010	0.0830	0.8077	0.5406	0.2432	0.1505	3479250	74926	0.2177
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	5	0.2000	0.0010	0.0772	0.8035	0.5377	0.2338	0.1409	3500991	71661	0.2123
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	10	0.2000	0.0010	0.0854	0.4849	0.5385	0.1768	0.1453	3469864	344312	0.2094
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	4	1.0000	0.0010	0.0501	0.7456	0.5242	0.1795	0.0938	3604008	64818	0.2096
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	5	1.0000	0.0010	0.0512	0.7500	0.5248	0.1823	0.0959	3599564	64805	0.2051
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	7	1.0000	0.0010	0.0521	0.7433	0.5252	0.1828	0.0974	3596332	68257	0.2051

Table D.10.: Results for the polynomial kernel on the CIC-IDS-2017 dataset, trying out different values for *degree* and *coef0*

TRAIN	TEST	gamma	nu	degree	coef0	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	0.0050	0.0001	3	0.0000	0.0010	0.0117	0.2875	0.5053	0.0517	0.0224	1742938	51017	0.1762
A	b	0.0050	0.0001	4	0.0000	0.0010	0.0000	0.0000	0.4927	-0.0230	0.0000	1763521	695967	0.1765
A	b	0.0050	0.0001	5	0.0000	0.0010	0.0502	0.0076	0.4032	-0.0845	0.0132	1675064	11593747	0.1794
A	b	0.0050	0.0001	7	0.0000	0.0010	0.0608	0.0039	0.2456	-0.1899	0.0074	1656296	27094934	0.1787
A	b	0.0050	0.0001	10	0.0000	0.0010	0.0000	0.0000	0.0510	-0.4892	0.0000	1763521	42714484	0.1738
A	b	0.0050	0.0001	20	0.0000	0.0010	0.0000	0.0000	0.5000	-0.0002	0.0000	1763521	68	0.1764
A	b	0.0050	0.0001	3	0.2000	0.0010	0.0000	0.0000	0.5000	-0.0010	0.0000	1763521	1456	0.1750
A	b	0.0050	0.0001	4	0.2000	0.0010	0.0000	0.0000	0.4998	-0.0035	0.0000	1763521	16330	0.1733
A	b	0.0050	0.0001	5	0.2000	0.0010	0.0000	0.0000	0.4996	-0.0053	0.0000	1763521	37586	0.1749
A	b	0.0050	0.0001	7	0.2000	0.0010	0.0497	**0.8241**	0.5246	**0.1973**	0.0937	1675935	18692	0.1783
A	b	0.0050	0.0001	10	0.2000	0.0010	0.0000	0.0000	0.5000	-0.0005	0.0000	1763521	347	0.1757
A	b	0.0050	0.0001	20	0.2000	0.0010	0.0000	0.0000	0.2973	-0.1543	0.0000	1763503	19288596	0.1729
A	b	0.0050	0.0001	10	1.0000	0.0010	0.0000	0.0000	0.5000	-0.0007	0.0000	1763521	689	0.1783
A	b	0.0050	0.0001	7	1.0000	0.0010	0.0000	0.0000	0.5000	-0.0007	0.0000	1763521	697	0.1936
A	b	0.0050	0.0001	5	1.0000	0.0010	0.0000	0.0000	0.5000	-0.0010	0.0000	1763521	1267	0.1730
A	b	0.0050	0.0001	4	1.0000	0.0010	0.0000	0.0000	0.5000	-0.0009	0.0000	1763521	1073	0.1848
A	b	0.0050	0.0001	20	1.0000	0.0010	0.0000	0.0000	0.5000	-0.0005	0.0000	1763521	384	0.1717
A	b	0.0050	0.0050	4	0.0000	0.0010	0.0000	0.0000	0.4998	-0.0038	0.0000	1763521	18785	0.2103
A	b	0.0050	0.0050	7	0.0000	0.0010	**0.1981**	0.0585	**0.5400**	0.0455	0.0903	1414092	5623539	0.2090
A	b	0.0050	0.0050	5	0.0000	0.0010	0.0522	0.6740	0.5256	0.1811	**0.0969**	1671454	44536	0.2112
A	b	0.0050	0.0050	20	0.0000	0.0010	0.0000	0.0000	0.2693	-0.1724	0.0000	1763521	21950682	0.2273
A	b	0.0050	0.0050	4	0.2000	0.0010	0.0000	0.0000	0.4998	-0.0038	0.0000	1763521	19417	0.2203
A	b	0.0050	0.0050	10	0.2000	0.0010	0.0000	0.0000	0.4984	-0.0106	0.0000	1763521	149241	0.2166
A	b	0.0050	0.0050	4	1.0000	0.0010	0.0000	0.0000	0.4990	-0.0084	0.0000	1763521	93741	0.2238

Table D.11.: Results for the polynomial kernel on the UNSW-NB15 dataset, trying out different values for $degree$ and $coef0$

TRAIN	TEST	gamma	nu	degree	coef0	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	0.0050	0.0001	3	0.0000	0.0010	**0.0117**	0.2875	0.5053	0.0517	0.0224	1742938	51017	0.1729
A	b	0.0050	0.0005	3	0.0000	0.0010	0.0024	0.2421	0.5010	0.0208	0.0047	1759342	13085	0.1799
A	b	0.0500	0.0001	3	0.0000	0.0010	**0.0117**	**0.8516**	**0.5058**	**0.0973**	**0.0230**	1742938	3586	0.1774

Table D.12.: Results for the polynomial kernel on the UNSW-NB15 dataset, trying out different values for γ and ν with fixed $degree = 3$

TRAIN	TEST	gamma	nu	degree	coef0	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	3	0.0000	0.0010	0.1135	0.9160	0.5562	0.3074	0.2019	3363538	39456	0.1877
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0005	3	0.0000	0.0010	**0.1147**	0.9088	**0.5568**	**0.3076**	**0.2037**	3358717	43702	0.1890
Monday	Tuesday, Wednesday, Thursday, Friday	0.0500	0.0001	3	0.0000	0.0010	0.1107	**0.9339**	0.5550	0.3071	0.1980	3373933	29707	0.1885
Monday	Tuesday, Wednesday, Thursday, Friday	0.0500	0.0005	3	0.0000	0.0010	0.1137	0.9155	0.5564	**0.3076**	0.2023	3362583	39830	0.1945

Table D.13.: Results for the polynomial kernel on the CIC-IDS-2017 dataset, trying out different values for γ and ν with fixed $degree = 3$

TRAIN	TEST	gamma	nu	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
Monday	Tuesday, Wednesday, Thursday, Friday	0.0005	0.0001	0.0010	0.0305	0.0099	0.3741	-0.1599	0.0150	3678350	11535754	0.1807
Monday	Tuesday, Wednesday, Thursday, Friday	0.0005	0.0050	0.0010	0.1407	0.0288	0.3505	-0.1692	0.0479	3260313	17964855	0.2244
Monday	Tuesday, Wednesday, Thursday, Friday	0.0005	0.0100	0.0010	0.1386	0.0280	0.3456	-0.1743	0.0465	3268217	18279330	0.2493
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0001	0.0010	0.0347	0.0109	0.3715	-0.1614	0.0166	3662435	11920437	0.1866
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0050	0.0010	0.2954	0.0583	0.4261	-0.0832	0.0974	2673246	18108321	0.2148
Monday	Tuesday, Wednesday, Thursday, Friday	0.0050	0.0100	0.0010	**0.3833**	**0.0707**	**0.4580**	**-0.0470**	**0.1194**	2339851	19099319	0.2362

Table D.14.: Results for the sigmoid kernel on the CIC-IDS-2017 dataset.

TRAIN	TEST	gamma	nu	tolerance	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	0.0005	0.0001	0.0010	0.1626	0.8155	0.5806	**0.3558**	0.2711	1476858	64866	0.1764
A	b	0.0050	0.0001	0.0010	0.1226	**0.8202**	0.5608	0.3097	0.2133	1547321	47383	0.1731
A	b	0.0050	0.0050	0.0010	0.3108	0.2721	**0.6400**	0.2627	**0.2902**	1215472	1465840	0.2263
A	b	0.0050	0.0100	0.0010	**0.3403**	0.1426	0.6322	0.1757	0.2010	1163476	3608385	0.2853

Table D.15.: Results for the sigmoid kernel on the UNSW-NB15 dataset.

kernel	gamma	nu	tolerance	pca_reducer_10	pca_reducer_20	pca_reducer_30	pca_reducer_50	RC	PR	BA	MCC	F1	FN	FP	ms/p
rbf	0.0001	0.0001	0.0100	0	0	1	0	0.9899	0.7818	0.9898	**0.8749**	**0.8736**	17828	487263	0.1276
rbf	0.0001	0.0001	0.0010	0	1	0	0	0.3593	0.8703	0.6786	0.5501	0.5086	1129961	94399	0.1280
rbf	0.0001	0.0001	0.0100	1	0	0	0	0.3751	**0.8750**	0.6866	0.5639	0.5251	1101952	94525	0.1315
rbf	0.0001	0.0001	0.0100	0	1	0	0	0.3593	0.8703	0.6786	0.5501	0.5086	1129959	94399	0.1313
rbf	0.0001	0.0001	0.0010	1	0	0	0	0.3751	0.8455	0.6863	0.5537	0.5197	1101966	120907	0.1316
rbf	0.0001	0.0001	0.0100	0	0	0	1	0.9899	0.7699	0.9895	0.8680	0.8662	17775	521639	0.1322
rbf	0.0001	0.0001	0.0010	0	0	1	0	0.9899	0.7818	0.9898	**0.8749**	**0.8736**	17828	487275	0.1287
rbf	0.0001	0.0001	0.0010	0	0	0	1	0.9899	0.7699	0.9895	0.8680	0.8662	17775	521639	0.1324
rbf	0.0001	0.0005	0.0100	0	0	1	0	0.9900	0.6902	0.9868	0.8195	0.8134	17582	783628	0.1331
rbf	0.0001	0.0005	0.0100	0	0	0	1	0.9899	0.7012	0.9871	0.8263	0.8209	17775	744037	0.1340
rbf	0.0001	0.0005	0.0010	0	0	1	0	0.9900	0.6902	0.9868	0.8195	0.8134	17582	783628	0.1316
rbf	0.0001	0.0005	0.0010	0	0	0	1	0.9899	0.7012	0.9871	0.8263	0.8209	17775	744037	0.1373
rbf	0.0050	0.0001	0.0100	0	1	0	0	0.6615	0.7503	0.8266	0.6943	0.7031	597040	388130	0.1334
rbf	0.0050	0.0001	0.0100	0	0	1	0	**1.0000**	0.7332	0.9933	0.8505	0.8460	0	641849	0.1319
rbf	0.0050	0.0001	0.0100	0	0	0	1	**1.0000**	0.5226	0.9831	0.7106	0.6865	0	1610879	0.1400
rbf	0.0050	0.0001	0.0010	1	0	0	0	0.3897	0.8490	0.6936	0.5658	0.5342	1076273	122245	0.1309
rbf	0.0050	0.0001	0.0010	0	1	0	0	0.6639	0.7696	0.8282	0.7051	0.7128	592780	350451	0.1315
rbf	0.0050	0.0001	0.0010	0	0	1	0	**1.0000**	0.7420	**0.9936**	0.8558	0.8519	0	613160	0.1333
rbf	0.0050	0.0001	0.0010	0	0	0	1	**1.0000**	0.5335	0.9838	0.7184	0.6958	0	1542301	0.1399
rbf	0.0050	0.0005	0.0100	0	1	0	0	0.7387	0.6957	0.8634	0.7061	0.7166	460757	569785	0.1304
rbf	0.0050	0.0005	0.0100	0	0	1	0	**1.0000**	0.7131	0.9925	0.8381	0.8325	0	709475	0.1344
rbf	0.0050	0.0005	0.0100	0	0	0	1	**1.0000**	0.5341	0.9838	0.7189	0.6963	0	1538079	0.1418
rbf	0.0050	0.0005	0.0010	0	1	0	0	0.7387	0.6957	0.8634	0.7061	0.7166	460843	569738	0.1321
rbf	0.0050	0.0005	0.0010	0	0	1	0	**1.0000**	0.7123	0.9925	0.8377	0.8320	0	712120	0.1350
rbf	0.0050	0.0005	0.0010	0	0	0	1	**1.0000**	0.5336	0.9838	0.7186	0.6959	0	1541226	0.1367

Table D.16.: Results for the combination of PCA and a one-class SVM on the UNSW-NB15 dataset (`A` training set and `b` test set). Only parameter configurations with $MCC > 0.5$ are shown.

transformers	PR	RC	MCC	F1	BA	ms/p
minmax_scaler	0.7541	0.9325	0.8321	0.8339	0.9606	0.1262
minmax_scaler, standard_scaler	**0.7765**	**0.9950**	**0.8742**	**0.8723**	**0.9922**	0.1261
one_hot_encoder, minmax_scaler	0.7412	0.9281	0.8225	0.8242	0.9581	0.1302
one_hot_encoder, minmax_scaler, standard_scaler	0.7755	0.9899	0.8713	0.8697	0.9896	0.1297
one_hot_encoder, standard_scaler	0.7755	0.9899	0.8713	0.8697	0.9896	0.1284
one_hot_encoder, standard_scaler, minmax_scaler	0.7412	0.9281	0.8225	0.8242	0.9581	0.1271
standard_scaler	**0.7765**	**0.9950**	**0.8742**	**0.8723**	**0.9922**	0.1267

Table D.17.: Results for the grid search which iterates the feature transformers, while a one-class SVM is used (rbf kernel, $\gamma = 0.0001, \nu = 0.0001$, `tolerance`$= 0.01$)

TRAIN	TEST	layers	loss	threshold_percentile	activation	RC	PR	BA	MCC	F1	FN	FP	ms/p
Monday	Tuesday, Wednesday, Thursday, Friday	400,200,10,200,400	mse	99.9999	relu	0.0102	0.0052	0.4149	-0.1276	0.0069	3755105	7376423	0.2189
Monday	Tuesday, Wednesday, Thursday, Friday	400,200,10,200,400	mse	99.9000	relu	0.6376	0.1069	0.5714	0.0797	0.1831	1374985	20216774	0.1972
Monday	Tuesday, Wednesday, Thursday, Friday	400,200,10,200,400	mse	95.0000	relu	0.9308	0.1090	0.6123	0.1403	0.1952	262361	28861593	0.2090
Monday	Tuesday, Wednesday, Thursday, Friday	400,200,10,200,400	mse	90.0000	relu	0.9659	0.1020	0.5880	0.1238	0.1844	129550	32278138	0.2041
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	100.0000	relu	0.0023	0.0018	0.4410	-0.1051	0.0020	3785311	4917836	0.1990
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	100.0000	relu	0.0159	0.0121	0.4475	-0.0928	0.0138	3733520	4939104	0.2106
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	99.9990	relu	0.0317	0.0070	0.3066	-0.2215	0.0114	3673824	17100042	0.2042
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	99.9990	relu	0.0315	0.0074	0.3188	-0.2101	0.0119	3674484	16099925	0.2175
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	99.9900	relu	0.1683	0.0345	0.3656	-0.1522	0.0573	3155283	17866450	0.2002
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	99.9900	relu	0.0879	0.0187	0.3293	-0.1943	0.0308	3460408	17547438	0.2039
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	99.9000	relu	0.5916	0.1011	0.5516	0.0576	0.1727	1549601	19955244	0.1968
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	99.9000	relu	0.6675	0.1165	0.5987	0.1101	0.1983	1261456	19212875	0.2145
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	99.5000	relu	0.7538	0.1125	0.6009	0.1136	0.1958	934146	22557458	0.2091
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	99.5000	relu	0.7485	0.1157	0.6087	0.1218	0.2004	954226	21707091	0.2091
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	99.0000	relu	0.7974	0.1131	0.6084	0.1234	0.1981	768549	23725033	0.2206
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	99.0000	relu	0.7824	0.1136	0.6079	0.1221	0.1984	825602	23158582	0.2200
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mse	95.0000	relu	0.8381	0.0994	0.5664	0.0822	0.1777	614430	28819447	0.2197
Monday	Tuesday, Wednesday, Thursday, Friday	300,200,75,10,75,200,300	mae	95.0000	relu	0.8372	0.0992	0.5658	0.0814	0.1774	617632	28835049	0.2123
Monday	Tuesday, Wednesday, Thursday, Friday	200,20,200	mse	99.9999	relu	0.0024	0.0036	0.4711	-0.0705	0.0029	3785060	2458735	0.2053
Monday	Tuesday, Wednesday, Thursday, Friday	200,20,200	mse	99.9000	relu	0.7044	0.1165	0.6043	0.1163	0.2000	1121672	20263080	0.1959
Monday	Tuesday, Wednesday, Thursday, Friday	200,20,200	mse	95.0000	relu	0.9387	0.1116	0.6225	0.1513	0.1995	232669	28346724	0.2032
Monday	Tuesday, Wednesday, Thursday, Friday	200,20,200	mse	90.0000	relu	**0.9765**	0.1029	0.5932	0.1314	0.1862	89222	32289751	0.1964
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	100.0000	relu	0.0129	0.0098	0.4463	-0.0953	0.0112	3745062	4918538	0.2122
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	99.9990	relu	0.0315	0.0077	0.3265	-0.2030	0.0123	3674648	15469773	0.2052
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mse	99.9900	relu	0.0647	0.0139	0.3196	-0.2058	0.0229	3548605	17390424	0.1987
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	99.9900	relu	0.0474	0.0102	0.3104	-0.2164	0.0168	3614066	17437885	0.2055
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mse	99.9000	relu	0.6067	0.1027	0.5574	0.0640	0.1757	1492189	20104509	0.2101
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	99.9000	relu	0.3559	0.0673	0.4488	-0.0573	0.1131	2443697	18725360	0.2251
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mse	99.5000	relu	0.7794	0.1145	0.6098	0.1240	0.1996	836879	22874759	0.2148
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	99.5000	relu	0.6462	0.1113	0.5837	0.0933	0.1899	1342429	19569795	0.2173
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mse	99.0000	relu	0.8800	**0.1266**	**0.6581**	**0.1793**	**0.2213**	455217	23042637	0.2035
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	99.0000	relu	0.7352	0.1189	0.6148	0.1282	0.2048	1004685	20660651	0.1940
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mse	95.0000	relu	0.8905	0.1061	0.5970	0.1196	0.1896	415508	28460371	0.2022
Monday	Tuesday, Wednesday, Thursday, Friday	*0.9,*0.8,*0.8,#2,#1	mae	95.0000	relu	0.8693	0.1164	0.6285	0.1487	0.2054	495904	25025742	0.1973

Table D.18.: Results for the autoencoder on the CIC-IDS-2017 dataset with the `relu` activation function and varying architectures

TRAIN	TEST	layers	loss	threshold_percentile	activation	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	400,200,10,200,400	mse	99.9999	relu	0.3688	**0.8736**	0.6834	0.5586	0.5187	1113066	94111	0.1886
A	b	400,200,10,200,400	mse	99.9000	relu	**1.0000**	0.6291	0.9891	0.7844	0.7723	0	1039907	0.1515
A	b	400,200,10,200,400	mse	95.0000	relu	**1.0000**	0.0688	0.7490	0.1851	0.1287	0	23877380	0.1633
A	b	400,200,10,200,400	mse	90.0000	relu	**1.0000**	0.0600	0.7098	0.1587	0.1133	0	27609724	0.1948
A	b	300,200,75,10,75,200,300	mse	100.0000	relu	0.2703	0.8566	0.6343	0.4722	0.4109	1286914	79759	0.1809
A	b	300,200,75,10,75,200,300	mse	99.9990	relu	0.9248	0.7792	0.9575	0.8429	0.8458	132608	462223	0.1849
A	b	300,200,75,10,75,200,300	mse	99.9900	relu	0.9899	0.7833	0.9899	0.8758	0.8746	17778	482904	0.1880
A	b	300,200,75,10,75,200,300	mae	99.9900	relu	0.9899	0.7845	0.9899	0.8765	0.8753	17817	479662	0.1861
A	b	300,200,75,10,75,200,300	mse	99.5000	relu	**1.0000**	0.3213	0.9608	0.5442	0.4864	0	3724642	0.1833
A	b	300,200,75,10,75,200,300	mae	99.5000	relu	**1.0000**	0.3246	0.9614	0.5473	0.4901	0	3669642	0.1886
A	b	300,200,75,10,75,200,300	mse	99.0000	relu	**1.0000**	0.1257	0.8710	0.3054	0.2233	0	12268466	0.2042
A	b	300,200,75,10,75,200,300	mae	99.0000	relu	**1.0000**	0.1684	0.9084	0.3708	0.2882	0	8711655	0.2020
A	b	200,20,200	mse	99.9000	relu	**1.0000**	0.7116	0.9925	0.8372	0.8315	0	714617	0.1833
A	b	200,20,200	mse	95.0000	relu	**1.0000**	0.0616	0.7176	0.1637	0.1160	0	26866658	0.1790
A	b	200,20,200	mse	90.0000	relu	**1.0000**	0.0533	0.6706	0.1349	0.1012	0	31333362	0.1869
A	b	*0.9,*0.8,*0.8,#2,#1	mae	100.0000	relu	0.5587	0.8461	0.7775	0.6786	0.6730	778287	179152	0.1800
A	b	*0.9,*0.8,*0.8,#2,#1	mse	99.9990	relu	0.9899	0.7861	0.9899	0.8774	0.8763	17866	475050	0.1831
A	b	*0.9,*0.8,*0.8,#2,#1	mae	99.9990	relu	0.9710	0.7840	0.9805	0.8674	0.8675	51123	471886	0.1877
A	b	*0.9,*0.8,*0.8,#2,#1	mse	99.9900	relu	**1.0000**	0.7825	**0.9948**	**0.8800**	**0.8780**	0	490047	0.1867
A	b	*0.9,*0.8,*0.8,#2,#1	mae	99.9900	relu	0.9899	0.7850	0.9899	0.8768	0.8756	17784	478209	0.1774
A	b	*0.9,*0.8,*0.8,#2,#1	mse	99.9000	relu	**1.0000**	0.6750	0.9911	0.8142	0.8059	0	849239	0.1786
A	b	*0.9,*0.8,*0.8,#2,#1	mae	99.9000	relu	**1.0000**	0.6359	0.9894	0.7889	0.7774	0	1009641	0.1845
A	b	*0.9,*0.8,*0.8,#2,#1	mse	99.5000	relu	**1.0000**	0.2701	0.9499	0.4930	0.4253	0	4766038	0.1900
A	b	*0.9,*0.8,*0.8,#2,#1	mae	99.5000	relu	**1.0000**	0.2837	0.9532	0.5071	0.4420	0	4452230	0.1842
A	b	*0.9,*0.8,*0.8,#2,#1	mse	99.0000	relu	**1.0000**	0.1488	0.8939	0.3423	0.2590	0	10092017	0.1862
A	b	*0.9,*0.8,*0.8,#2,#1	mse	95.0000	relu	**1.0000**	0.0747	0.7703	0.2009	0.1390	0	21854529	0.1798
A	b	*0.9,*0.8,*0.8,#2,#1	mae	95.0000	relu	**1.0000**	0.0629	0.7240	0.1679	0.1184	0	26254549	0.1821

Table D.19.: Results for the autoencoder on the UNSW-NB15 dataset with the `relu` activation function and varying architectures. Always, a training batch size of 1024 and 200 training epochs are used.

TRAIN	TEST	layers	training_epochs	training_batch	early_stopping_patience	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	*0.9,*0.8,*0.8,#2,#1	800	2048	3	**1.0000**	0.7817	**0.9948**	0.8795	**0.8775**	0	492546	0.1439
A	b	*0.9,*0.8,*0.8,#2,#1	800	512	-1	**1.0000**	0.7818	**0.9948**	**0.8796**	**0.8775**	0	492218	0.1413
A	b	*0.9,*0.8,*0.8,#2,#1	800	256	-1	0.9899	0.7819	0.9898	0.8750	0.8737	17775	487071	0.1405
A	b	*0.9,*0.8,*0.8,#2,#1	400	1024	-1	0.9899	0.7826	0.9899	0.8754	0.8741	17775	485023	0.1428
A	b	*0.9,*0.8,*0.8,#2,#1	400	32	-1	0.9899	0.7837	0.9899	0.8760	0.8748	17817	481845	0.1372
A	b	*0.9,*0.8,*0.8,#2,#1	200	1024	-1	**1.0000**	0.7801	**0.9948**	0.8786	0.8765	0	497163	0.1405
A	b	*0.9,*0.8,*0.8,#2,#1	200	64	-1	0.9583	**0.7898**	0.9744	0.8648	0.8659	73611	449834	0.1367
A	b	*0.9,*0.8,*0.8,#2,#1	100	32	-1	0.9310	0.7765	0.9605	0.8443	0.8468	121763	472422	0.1437
A	b	*0.9,*0.8,*0.8,#2,#1	50	256	3	0.9899	0.7841	0.9899	0.8763	0.8751	17841	480604	0.1392
A	b	*0.9,*0.8,*0.8,#2,#1	50	64	-1	0.9899	0.7835	0.9899	0.8759	0.8747	17775	482466	0.1422
A	b	*0.9,*0.8,*0.8,#2,#1	10	2048	-1	0.9899	0.7843	0.9899	0.8764	0.8752	17798	480065	0.1391
A	b	*0.9,*0.8,*0.8,#2,#1	10	2048	3	0.9950	0.7838	0.9924	0.8785	0.8769	8824	484050	0.1387
A	b	*0.9,*0.8,*0.8,#2,#1	10	1024	-1	0.9899	0.7840	0.9899	0.8762	0.8750	17775	480942	0.1398
A	b	*0.9,*0.8,*0.8,#2,#1	10	512	3	0.9899	0.7844	0.9899	0.8764	0.8752	17791	479943	0.1407
A	b	*0.9,*0.8,*0.8,#2,#1	10	256	3	0.9899	0.7838	0.9899	0.8761	0.8749	17786	481561	0.1420
A	b	*0.9,*0.8,*0.8,#2,#1	10	32	-1	0.9898	0.7841	0.9899	0.8763	0.8750	17925	480660	0.1464

Table D.20.: Results for the autoencoder on the UNSW-NB15 dataset with the `relu` activation function, MSE loss function, and varying values for `training_epochs`, `training_batch`, and `early_stopping_patience`

category	tcp	ip_dotted	port_decimal	with_ip_addr	basic	MCC	BA	RC	F1	ms/p
worms	1	0	1	0	0	0.1606	0.8810	0.7650	0.0649	0.1415
shellcode	1	1	0	1	0	0.1285	0.9960	1.0000	0.0328	0.1418
reconnaissance	1	1	0	1	0	0.3819	0.9914	0.9909	0.2582	0.1418
generic	1	0	1	0	0	0.5759	0.8819	0.7668	0.5550	0.1415
fuzzers	1	1	0	1	0	0.5653	0.9950	0.9980	0.4878	0.1418
exploits	1	1	0	1	0	0.8175	0.9881	0.9861	0.8084	0.1375
dos	1	0	0	1	0	0.7551	0.9092	0.8207	0.7543	0.1353
backdoor	1	0	0	0	0	0.3288	0.5541	0.1081	0.1951	0.1400
analysis	1	0	0	0	0	0.1945	0.5189	0.0378	0.0729	0.1400

Table D.21.: Best network flow mode combinations by attack category in the UNSW-NB15 dataset (with the `A` training and `b` test subset)

category	flow_timeout	subflow_timeout	hindsight_window	BA	RC	F1	MCC	ms/p
worms	6.0000	0.2000	500.0000	0.9169	0.8367	0.0754	0.1814	0.1642
shellcode	6.0000	0.5000	500.0000	0.9956	1.0000	0.0297	0.1222	0.1549
reconnaissance	6.0000	0.5000	500.0000	0.9949	0.9988	0.2408	0.3681	0.1549
generic	6.0000	0.5000	100.0000	0.8681	0.7389	0.5608	0.5763	0.1217
fuzzers	6.0000	0.5000	500.0000	0.9950	0.9988	0.4627	0.5459	0.1549
exploits	6.0000	0.2000	100.0000	0.8088	0.6207	0.7034	0.7043	0.1375
dos	6.0000	1.0000	1000.0000	0.9181	0.8387	0.7520	0.7543	0.1708
backdoor	6.0000	0.5000	500.0000	0.9615	0.9319	0.0152	0.0841	0.1549
analysis	6.0000	0.5000	2000.0000	0.7769	0.5560	0.0721	0.1460	0.2260

Table D.22.: Best hindsight window values by attack category in the UNSW-NB15 dataset

TRAIN	TEST	flow_timeout	subflow_timeout	RC	PR	BA	MCC	F1	FN	FP	ms/p
A	b	2.0000	0.4000	**1.0000**	0.7023	0.9921	0.8314	0.8251	9	747417	0.1443
A	b	2.0000	0.4500	**1.0000**	0.7052	0.9923	0.8333	0.8271	9	737047	0.1411
A	b	2.0000	0.5000	**1.0000**	0.7063	0.9923	0.8339	0.8279	9	733401	0.1419
A	b	2.0000	0.5500	**1.0000**	0.7076	0.9923	0.8347	0.8288	9	728611	0.1417
A	b	2.0000	0.6000	**1.0000**	0.7073	0.9923	0.8345	0.8286	9	729703	0.1449
A	b	2.0000	0.8000	**1.0000**	0.7047	0.9922	0.8329	0.8268	9	738973	0.1466
A	b	2.0000	1.0000	**1.0000**	0.7058	0.9923	0.8336	0.8276	9	734917	0.1397
A	b	2.0000	3.0000	**1.0000**	0.7239	0.9929	0.8448	0.8399	0	672500	0.1401
A	b	4.0000	0.4000	**1.0000**	0.7003	0.9921	0.8302	0.8238	9	754541	0.1456
A	b	4.0000	0.4500	**1.0000**	0.7046	0.9922	0.8328	0.8267	9	739423	0.1444
A	b	4.0000	0.5000	**1.0000**	0.7054	0.9923	0.8334	0.8273	9	736343	0.1448
A	b	4.0000	0.5500	**1.0000**	0.7106	0.9924	0.8366	0.8308	9	718143	0.1465
A	b	4.0000	0.6000	**1.0000**	0.7095	0.9924	0.8359	0.8300	9	722197	0.1418
A	b	4.0000	0.8000	**1.0000**	0.7010	0.9921	0.8306	0.8242	9	752359	0.1446
A	b	4.0000	1.0000	**1.0000**	0.7018	0.9921	0.8311	0.8248	9	749309	0.1449
A	b	6.0000	0.4500	**1.0000**	0.7081	0.9924	0.8350	0.8291	9	726900	0.1432
A	b	6.0000	0.5000	**1.0000**	0.7021	0.9921	0.8313	0.8250	9	748096	0.1432
A	b	6.0000	0.5000	**1.0000**	0.7021	0.9921	0.8313	0.8250	9	748096	0.1388
A	b	6.0000	0.5500	**1.0000**	0.7058	0.9923	0.8336	0.8275	9	735131	0.1378
A	b	6.0000	0.6000	**1.0000**	0.7064	0.9923	0.8340	0.8280	0	732916	0.1386
A	b	6.0000	0.6000	**1.0000**	0.7064	0.9923	0.8340	0.8280	0	732916	0.1404
A	b	6.0000	0.8000	**1.0000**	0.7001	0.9921	0.8300	0.8236	0	755518	0.1468
A	b	10.0000	0.5500	**1.0000**	0.7024	0.9921	0.8315	0.8252	0	747059	0.1486
A	b	10.0000	0.6000	**1.0000**	0.7023	0.9921	0.8314	0.8251	0	747441	0.1436
A	b	10.0000	3.0000	**1.0000**	0.7002	0.9921	0.8301	0.8237	0	755112	0.1149
A	b	11.0000	0.4000	**1.0000**	0.7129	0.9925	0.8380	0.8324	0	710141	0.1409
A	b	11.0000	0.4500	**1.0000**	0.7166	0.9927	0.8403	0.8349	0	697483	0.1402
A	b	11.0000	0.5000	**1.0000**	0.7185	0.9927	0.8415	0.8362	0	691009	0.1508
A	b	11.0000	0.5500	**1.0000**	0.7232	0.9929	0.8443	0.8393	0	675117	0.1445
A	b	12.0000	0.4000	**1.0000**	0.7237	0.9929	0.8446	0.8397	0	673429	0.1449
A	b	12.0000	0.4500	**1.0000**	0.7253	0.9930	0.8456	0.8408	0	668065	0.1409
A	b	12.0000	0.5000	**1.0000**	0.7256	0.9930	0.8458	0.8410	0	667033	0.1397
A	b	12.0000	0.5500	**1.0000**	**0.7304**	**0.9932**	**0.8488**	**0.8442**	0	651023	0.1391
A	b	12.0000	0.6000	**1.0000**	0.7264	0.9930	0.8463	0.8415	0	664144	0.1427
A	b	12.0000	0.8000	**1.0000**	0.7194	0.9928	0.8420	0.8368	0	687712	0.1501
A	b	12.0000	1.0000	**1.0000**	0.7204	0.9928	0.8427	0.8375	0	684322	0.1477
A	b	12.0000	3.0000	**1.0000**	0.7295	0.9931	0.8482	0.8436	0	654012	0.1395
A	b	16.0000	0.5500	**1.0000**	0.7198	0.9928	0.8423	0.8371	0	686523	0.1461
A	b	16.0000	0.6000	**1.0000**	0.7142	0.9926	0.8388	0.8332	0	705846	0.1455
A	b	16.0000	0.8000	**1.0000**	0.7084	0.9924	0.8352	0.8293	0	725770	0.1391
A	b	16.0000	1.0000	**1.0000**	0.7104	0.9924	0.8365	0.8307	0	718792	0.1436
A	b	16.0000	3.0000	**1.0000**	0.7167	0.9927	0.8403	0.8349	0	697246	0.1454
A	b	20.0000	0.4000	**1.0000**	0.7055	0.9923	0.8334	0.8273	0	736085	0.1458
A	b	20.0000	0.4500	**1.0000**	0.7078	0.9923	0.8349	0.8289	0	727941	0.1442
A	b	20.0000	0.5000	**1.0000**	0.7130	0.9925	0.8381	0.8325	0	709851	0.1402
A	b	20.0000	0.5500	**1.0000**	0.7152	0.9926	0.8394	0.8339	0	702413	0.1471
A	b	20.0000	0.6000	**1.0000**	0.7113	0.9925	0.8370	0.8313	0	715880	0.1446
A	b	20.0000	0.8000	**1.0000**	0.7052	0.9923	0.8332	0.8271	0	737284	0.1418
A	b	20.0000	1.0000	**1.0000**	0.7063	0.9923	0.8339	0.8279	0	733366	0.1419
A	b	20.0000	3.0000	**1.0000**	0.7116	0.9925	0.8372	0.8315	0	714636	0.1441
A	b	60.0000	0.4500	**1.0000**	0.7071	0.9923	0.8344	0.8284	0	730628	0.1425
A	b	60.0000	0.5000	**1.0000**	0.7051	0.9922	0.8332	0.8271	0	737430	0.1388
A	b	60.0000	0.5500	**1.0000**	0.7108	0.9925	0.8367	0.8310	0	717464	0.1428
A	b	60.0000	0.6000	**1.0000**	0.7096	0.9924	0.8360	0.8301	0	721718	0.1381
A	b	60.0000	0.8000	**1.0000**	0.7021	0.9921	0.8313	0.8250	0	748268	0.1439

Table D.23.: Results for the grid search which iterates the flow and subflow timeouts

tcp_end_on_rst	subflows	flow_timeout	subflow_timeout	RC	PR	BA	MCC	F1	FN	FP	ms/p
1	0	6.0000	0.2000	0.9900	0.7858	0.9900	0.8774	0.8762	17584	475857	0.1274
1	0	6.0000	0.5000	**1.0000**	0.7826	0.9948	0.8801	0.8780	0	489969	0.1331
1	0	6.0000	1.0000	0.9899	**0.7859**	0.9900	0.8774	0.8762	17775	475490	0.1316
1	0	12.0000	0.2000	0.9999	0.7841	0.9948	0.8809	0.8789	222	485546	0.1311
1	0	12.0000	1.0000	**1.0000**	0.7827	**0.9949**	0.8801	0.8781	0	489552	0.1275
1	0	60.0000	0.5000	**1.0000**	0.7841	**0.9949**	**0.8810**	**0.8790**	0	485477	0.1256
1	0	60.0000	1.0000	**1.0000**	0.7832	**0.9949**	0.8805	0.8784	0	488041	0.1317
1	1	12.0000	0.2000	0.9899	0.7843	0.9899	0.8764	0.8752	17780	480026	0.1426
1	1	12.0000	0.5000	0.9899	0.7840	0.9899	0.8763	0.8750	17778	480832	0.1433
1	1	12.0000	1.0000	0.9899	0.7822	0.9899	0.8752	0.8739	17775	486134	0.1431
1	1	60.0000	0.2000	0.9899	0.7826	0.9899	0.8754	0.8742	17778	484817	0.1387
1	1	60.0000	0.5000	0.9899	0.7840	0.9899	0.8762	0.8750	17775	481052	0.1417
1	1	60.0000	1.0000	0.9899	0.7837	0.9899	0.8761	0.8748	17778	481701	0.1417

Table D.24.: Influence of the `tcp_end_on_rst` mode on the UNSW-NB15 dataset (with the `A` training and `b` test subset)

TRAIN	TEST	basic	layers	PR	RC	MCC	F1	BA	ms/p
9-31	b	0	*0.9,*0.8,*0.8,#2,#1	0.8673	0.4522	0.6173	0.5945	0.7248	0.1415
8-31	b	0	*0.9,*0.8,*0.8,#2,#1	0.8519	0.3083	0.5033	0.4528	0.6532	0.1386

Table D.25.: Results for unclean training sets (UNSW-NB15)

category	basic	subflows	tcp	flow_timeout	subflow_timeout	RC	MCC	F1	BA	ms/p
worms	0	1	1	15.0000	0.5000	1.0000	0.1344	0.0358	0.9963	0.2387
shellcode	0	1	1	15.0000	0.5000	1.0000	0.1339	0.0355	0.9963	0.2387
reconnaissance	0	1	1	15.0000	0.5000	0.9084	0.3649	0.2544	0.9505	0.2387
generic	0	1	1	15.0000	0.5000	0.6924	0.3869	0.3333	0.8425	0.2387
fuzzers	0	1	1	15.0000	0.5000	0.9982	0.5814	0.5085	0.9954	0.2387
exploits	0	1	1	15.0000	0.5000	0.3582	0.4200	0.4232	0.6754	0.2387
dos	0	1	1	15.0000	0.5000	0.6159	0.6583	0.6587	0.8071	0.2417
backdoor	0	1	1	15.0000	0.5000	0.9228	0.0913	0.0180	0.9577	0.2387
analysis	0	1	1	15.0000	0.5000	0.4653	0.0759	0.0245	0.7299	0.2387
all	0	1	1	15.0000	0.5000	0.4853	0.5744	0.5763	0.7390	0.2387

Table D.26.: Best results for the simple payload analyser counting byte frequencies in combination with an autoencoder

TRAIN	TEST	category	basic	smoothing	clustering_threshold	flow_timeout	PR	RC	MCC	F1	BA	ms/p
12-20	b	worms	0	0.0000	20.0000	15.0000	0.0232	0.9516	0.1483	0.0454	0.9730	0.2669
12-20	b	shellcode	0	0.0001	7.0000	15.0000	0.0195	1.0000	0.1390	0.0382	0.9966	0.2650
12-20	b	reconnaissance	0	0.0001	4.0000	15.0000	0.1565	0.9895	0.3920	0.2702	0.9910	0.2687
12-20	b	generic	0	0.0010	7.0000	15.0000	0.2493	0.9872	0.4938	0.3980	0.9892	0.2687
12-20	b	fuzzers	0	0.0001	7.0000	15.0000	0.3583	0.9971	0.5956	0.5271	0.9951	0.2650
12-20	b	exploits	0	0.0010	20.0000	15.0000	0.6849	0.9861	0.8174	0.8083	0.9880	0.2679
12-20	b	dos	0	0.0010	20.0000	15.0000	0.3887	0.9949	0.6188	0.5591	0.9925	0.2679
12-20	b	backdoor	0	0.0001	7.0000	15.0000	0.0093	0.8753	0.0900	0.0185	0.9342	0.2650
12-20	b	analysis	0	0.1000	20.0000	15.0000	0.8657	0.0278	0.1550	0.0538	0.5139	0.2653
12-20	b	all	0	0.0010	20.0000	15.0000	0.7860	0.9899	0.8774	0.8762	0.9899	0.2679

Table D.27.: Best results for the PAYL generator on the UNSW-NB15 dataset, in combination with an autoencoder

TEST	category	basic	smoothing	clustering_threshold	flow_timeout	PR	RC	MCC	F1	BA	ms/p	F1.1	BA.1	classification_ms_per_packet.1
thursday	web attack – xss	1	0.0001	2.0000	15.0000	0.0000	0.0000	-0.0000	0.0000	0.5000	0.2325	0.0000	0.5000	0.2325
thursday	web attack – sql injection	1	0.1000	20.0000	15.0000	0.0000	0.0106	0.0002	0.0000	0.5023	0.2331	0.0000	0.5023	0.2331
thursday	web attack – brute force	1	0.0001	2.0000	15.0000	0.0000	0.0000	-0.0001	0.0000	0.5000	0.2325	0.0000	0.5000	0.2325
thursday	infiltration	1	0.1000	20.0000	15.0000	0.5163	0.9996	0.7163	0.6810	0.9968	0.2331	0.6810	0.9968	0.2331
wednesday	dos slowloris	0	0.0001	20.0000	15.0000	0.9226	0.0879	0.2841	0.1605	0.5439	0.3215	0.1605	0.5439	0.3215
wednesday	dos slowhttptest	1	0.1000	2.0000	15.0000	0.0068	0.0059	0.0034	0.0063	0.5016	0.2424	0.0063	0.5016	0.2424
wednesday	dos hulk	1	0.1000	2.0000	15.0000	0.7761	0.0501	0.1718	0.0941	0.5237	0.2424	0.0941	0.5237	0.2424
wednesday	dos goldeneye	1	0.1000	2.0000	15.0000	0.2336	0.0922	0.1420	0.1322	0.5447	0.2424	0.1322	0.5447	0.2424

Table D.28.: Best results for the PAYL generator on the CIC-IDS-2017 dataset (test sets: *wednesday* and *thursday*), in combination with an autoencoder

TRAIN	TEST	clustering_threshold	smoothing	pca_reducer_30	PR	RC	MCC	F1	BA	ms/p
A	b	8.0000	0.1000	1	0.7813	0.9899	**0.8747**	**0.8733**	0.9898	0.2637
A	b	5.0000	0.0100	1	**0.7814**	0.9899	**0.8747**	**0.8733**	0.9898	0.2714
A	b	5.0000	0.1000	1	0.7813	0.9899	**0.8747**	**0.8733**	0.9898	0.2729
A	b	4.0000	0.0100	1	**0.7814**	0.9899	**0.8747**	**0.8733**	0.9898	0.2666
A	b	8.0000	0.0100	1	**0.7814**	0.9899	**0.8747**	**0.8733**	0.9898	0.2704
A	b	2.0000	0.1000	1	0.7813	0.9899	**0.8747**	**0.8733**	0.9898	0.2683
A	b	2.0000	0.0100	1	**0.7814**	0.9899	**0.8747**	**0.8733**	0.9898	0.2730
A	b	4.0000	0.1000	1	0.7813	0.9899	**0.8747**	**0.8733**	0.9898	0.2845
A	b	2.0000	0.1000	0	0.7279	**0.9909**	0.8432	0.8393	0.9886	0.2867
A	b	4.0000	0.1000	0	0.7279	**0.9909**	0.8432	0.8393	0.9886	0.2763
A	b	5.0000	0.1000	0	0.7279	**0.9909**	0.8432	0.8393	0.9886	0.2822
A	b	8.0000	0.1000	0	0.7279	**0.9909**	0.8432	0.8393	0.9886	0.2724
A	b	2.0000	0.0100	0	0.7756	**0.9909**	0.8717	0.8701	**0.9901**	0.2799
A	b	4.0000	0.0100	0	0.7756	**0.9909**	0.8717	0.8701	**0.9901**	0.2715
A	b	5.0000	0.0100	0	0.7756	**0.9909**	0.8717	0.8701	**0.9901**	0.2694
A	b	8.0000	0.0100	0	0.7756	**0.9909**	0.8717	0.8701	**0.9901**	0.2708

Table D.29.: Best results for the PAYL generator on the UNSW-NB15 dataset, in combination with a one-class SVM

TRAIN	TEST	category	PR	RC	MCC	F1	BA	ms/p
10-53	54-81	worms	0.0456	0.8237	0.1936	0.0865	0.9107	0.1378
10-53	54-81	shellcode	0.0002	0.0029	0.0002	0.0003	0.5003	0.1378
10-53	54-81	reconnaissance	0.0336	0.0556	0.0416	0.0419	0.5267	0.1378
10-53	54-81	generic	0.4511	0.6393	0.5354	0.5289	0.8185	0.1378
10-53	54-81	fuzzers	0.1556	0.1165	0.1319	0.1333	0.5571	0.1378
10-53	54-81	exploits	0.8368	0.5485	0.6721	0.6627	0.7731	0.1378
10-53	54-81	dos	0.6814	0.8269	0.7489	0.7471	0.9123	0.1378
10-53	54-81	backdoor	0.0000	0.0006	-0.0003	0.0000	0.4992	0.1378
10-53	54-81	analysis	0.0002	0.0022	0.0004	0.0004	0.5006	0.1378
10-53	54-81	all	0.8930	0.5368	0.6844	0.6706	0.7672	0.1378

Table D.30.: Results using a larger UNSW-NB15 subset (training: `A+B`, test: `b+c`) for a model consisting of the network flow feature extractor (flow timeout: 12 seconds, subflow timeout: 0.5 seconds, modes: `tcp` and `subflows`), the one-hot encoder, a PCA transformer with 30 principal components, min-max scaling, standardization, and a one-class SVM ($\gamma = 0.0001$, $\nu = 0.0001$)

Erklärung / Statement of Originality

„Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.“

Ort: Leipzig

Datum: 09. April 2021

Unterschrift: