A Deep Learning-based Approach to Identifying and Mitigating Network Attacks Within SDN Environments Using Non-standard Data Sources

Matthew David Banton

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of PhD.

November 2020

Acknowledgements

This thesis was completed following extensive research into Deep Learning, and I would like to express my gratitude to several people who have supported me throughout.

First thanks must go to my principal supervisor, Dr Nathan Shone, whose expert knowledge and insight aided and guided the direction of the research project, and without whom this thesis may never have been finished. Thanks must also go to Dr William Hurst and Prof. Qi Shi, whose insight and support has been invaluable.

Secondly, thanks must be provided to my Sister, Carys, for living in the same house as me for the last few years, and not going mad. Thanks also to my parents, Bob and Val, whose love and support has been instrumental in my upbringing and encouraged me to take up the offer of the scholarship.

I also really appreciate the support from Liverpool John Moores University, and the department of computer science, in particular the research administrators, for being able to answer any questions quickly and efficiently.

Contents

List of Tables

List of Figures

List of Confusion Matrixes

List of Equations

List of Pseudocode

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
AWS	Amazon Web Services
BGP	Border Gateway Protocol
CIA	Confidentiality, Integrity and Availability
CNN	Convolutional Neural Network
DoS	Denial of Service
DDoS	Distributed Denial of Service
DNN	Deep Neural Network
DPI	Deep Packet Inspection
FNT	Flexible Neural Tree
FTP	File Transfer Protocol
GPGPU	General Purpose Graphics Processing Unit
HIDS	Host Intrusion Detection System
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IG	Information Gain
IMS	Intrusion Mitigation System
IPS	Intrusion Prevention System
LSTM	Long Short-Term Memory
MLP	Multi-Layer Perceptron
NAT	Network Address Translation
NHF	Network Health Flow
NIDS	Network Intrusion Detection System
NIMS	Network Intrusion Mitigation System
NIPS	Network Intrusion Prevention System
PCA	Principal Component Analysis
PSO	Particle Swarm Optimisation
RBF	Radial Bias Function
RDP	Remote Desktop Protocol
SDN	Software Defined Network
SMOTE	Synthetic Minority Oversampling Technique
SSH	Secure Shell
SVM	Support Vector Machine

Publications

M. Banton, N. Shone, W. Hurst and Q. Shi, "Intrusion Detection Using Extremely Limited Data Based on SDN," *2020 IEEE 10th International Conference on Intelligent Systems (IS),* Varna, Bulgaria, 2020, pp. 304-309, doi: 10.1109/IS48319.2020.9199950.

M. Banton, N. Shone, W. Hurst and Q. Shi, "A Fresh Look at Combining Logs and Network Data to Detect Anomalous Activity," *2019 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, Paris, France, 2019, pp. 1-6, doi: 10.1109/ICT-DM47966.2019.9032959.

M. Banton, N. Shone, W. Hurst, and Q. Shi, “Visualising Network Anomalies in an Unsupervised Manner Using Deep Network Autoencoders,” *The Fourth International Conference on Applications and Systems of Visual Paradigms VISUAL 2019*, Rome, Italy, 2019 pp.25-31

M. Banton, N. Shone, W. Hurst and Q. Shi, “Deep Learning Based Methods and Applications: A Survey,” *ACM Transactions on Privacy and Security*, In Preparation.

Abstract

Modern society is increasingly dependent on computer networks, which are essential to delivering an increasing number of key services. With this increasing dependence, comes a corresponding increase in global traffic and users. One of the tools administrators are using to deal with this growth is Software Defined Networking (SDN). SDN changes the traditional distributed networking design to a more programmable centralised solution, based around the SDN controller. This allows administrators to respond more quickly to changing network conditions. However, this change in paradigm, along with the growing use of encryption can cause other issues.

For many years, security administrators have used techniques such as deep packet inspection and signature analysis to detect malicious activity. These methods are becoming less common as artificial intelligence (AI) and deep learning technologies mature. AI and deep learning have advantages in being able to cope with 0-day attacks and being able to detect malicious activity despite the use of encryption and obfuscation techniques. However, SDN reduces the volume of data that is available for analysis with these machine learning techniques. Rather than packet information, SDN relies on flows, which are abstract representations of network activity.

Security researchers have been slow to move to this new method of networking, in part because of this reduction in data, however doing so could have advantages in responding quickly to malicious activity. This research project seeks to provide a way to reconcile the contradiction apparent, by building a deep learning model that can achieve comparable results to other state-of-the-art models, while using 70% fewer features. This is achieved through the creation of new data from logs, as well as creation of a new risk-based sampling method to prioritise suspect flows for analysis, which can successfully prioritise over 90% of malicious flows from leading datasets. Additionally, provided is a mitigation method that can work with a SDN solution to automatically mitigate attacks after they are found, showcasing the advantages of closer integration with SDN.

1 Introduction

Modern society is heavily reliant on computer networks, which are key to delivering an ever-expanding array of critical services. Therefore, it is unsurprising that the volume of annual global IP traffic is forecast to grow to 4.8ZB by 2022, which is a 220% increase from 1.5ZB in 2017 [1]. Much of this growth will come from regions that currently have poor internet access, however, even in Western Europe, the number of users and devices is set to increase, along with internet speeds (353 million users in 2022, up from 331 million in 2017, and 4 billion networked devices in 2022, up from 2.3 billion in 2017) [1], [2].

As the volume of data continues to increase, so too does the need for new networking solutions to handle it. Software Defined Networks (SDNs) are rapidly becoming one of the main tools used to aid this growth. Traditional network infrastructure relies upon hardware routers, each maintaining a routing table for the potential paths for packets between devices, and the most efficient paths that can be taken. On a small scale, this is relatively easy to manage as changes can be made by reprogramming the routers as needed, and even small outages can be planned for and automatically corrected. However, at scale, the reprogramming of numerous individual devices becomes a complex and time-consuming task. This problem is negated in SDN environments as instead of each router maintaining its own route table, a central controller manages the route table. If a router has not previously encountered a route that a new packet requires, then it will ask the SDN controller for instructions on how to handle the packet and any future similar packets. In effect, instead of being distributed between all routers on the network, the routing table is now centralised on one device, which is separate from the devices that route the packets. This separation is what defines SDN, and it is described as “The physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices” [3], [4].

However, security systems have been slow to respond to this new way of networking. Most existing IDSs still analyse data from packet headers directly, requiring additional hardware in all relevant locations, which runs counter to the SDN paradigm of having a centralised solution providing a holistic view of the network. Successfully integrating an IDS into the northbound interface of a SDN controller would bring all the same benefits to network security that SDN has brought to networking more generally. Upon finding a malicious flow, a SDN controller would be able to automatically send instructions to attempt to mitigate it. SDNs would provide an IDS with a more holistic view of a network and negate the need for individual devices in selected parts of the network. However, moving to such a model for IDSs does come with additional problems.

One problem is a problem that current IDSs suffer from, data rates. Moving to a centralised solution would mean that the IDS must not only monitor all traffic in its local network segment, but all traffic from across the network. While SDN does aid with this somewhat, by providing an abstracted view of network activity in the form of network flows, a network will still produce many network flows creating a great deal of work. Additionally, these flows suffer from another problem. Most current IDS technologies are designed to work with traditional network packets (e.g. Flags), which provide a great deal of data useful for determining malicious intent. SDNs do not typically keep track of this kind of data since it is not useful for determining overall network activity levels. In addition, some IDSs monitor data from inside packet bodies (known as deep packet inspection (DPI)), again a rich source of information that is lost when moving to an SDN-based solution. This means that when attempting to create an IDS that detects malicious activity based on SDN flows, there is both too much data (in terms of sheer volume of flows) and not enough (in terms of the features collected.)

Another potential issue that could exacerbate this is the rise in popularity of machine learning. Recently, machine learning and in particular deep learning, has been seen as a potential boon to security-minded administrators. It has been shown that machine learning can detect attacks using various forms of data, the goal being to train an algorithm on various examples of data, which will then be able to detect new variants through generalisation [5]. This generalisation can aid in detecting novel and 0-day threats [6], threats for which a signature has not been produced or even seen before. However, machine learning models and in particular deep learning models, require large volumes of training data, with many different data points or features. As has been explained, SDNs goal is to abstract the features machine learning models need in order to get high accuracy rates. Additionally, deep learning models are particularly computationally expensive, which increases issues given the volumes of data that must be analysed.

This work proposes a solution to these dilemmas. Proposed is a deep learning solution that uses only data that is readily available, without the use of techniques such as DPI, and can be integrated into the northbound interface of a SDN controller, to make use of the holistic view a SDN provides.

1.1 Motivation

One of the advents to aid with the increasing amounts of data has been SDN [4], which allows networks to respond more dynamically to changing network conditions. However, IDSs have been slow to embrace this new way of networking. As such, the motivational factors for this research are:

1. **To provide a solution to the issue of data rates overloading IDSs**
   One of the issues facing traditional network infrastructure is that data rates are increasingly forcing Network Intrusion Detection System (NIDS) to either act in parallel to the network data (off-line), and potentially only detect attacks after the attack has succeeded, or use sampling which could increase the risk of attacks being missed due to packets not being sampled. SDN provides a high-level representation of the network activity, meaning less data must be analysed and malicious activity should still be detectable.
2. **To provide a solution to the amount of time it takes to mitigate an attack**
   It has been found that the time from first action in an event chain to initial compromise of an asset is most often measured in seconds or minutes [7]. While reducing the load on IDSs can aid with this (ensuring breaches are found in a timely manner) other solutions may be required to limit the damage of a successful breach. SDN provides one way to do this, through automated changing of the network conditions to hinder an attacker's progress.
3. **To provide a solution to the contradiction between AI and SDN in an IDS setting**

Machine learning provides a method to detect 0-day threats, being able to generalise well enough to determine if a flow is malicious even if it is “novel”, simply because many “novel” attacks are just slightly altered versions of existing attacks. However, machine learning requires large volumes of data, which SDN attempts to abstract

4. **To provide a solution to the issue of training time for AI based IDSs.**
   While machine learning, and deep learning has potential to aid with intrusion detection, the amount of time it takes to train these systems is prohibitive, especially the large and deep models proposed in research. Machine learning models must be updated regularly to keep abreast of changing network conditions. This is a time-consuming process that must be repeated regularly, and reducing this time is a major issue.

1.2 Aims and Objectives

To address the research challenges outlined, the aim of this research is to develop an IDS solution that can be tightly integrated within a SDN-based network, capable of accurately identifying 0-day attacks and mitigating those threats that are detected. This broad aim can be separated into two primary goals:

1 **Investigation of the role of data sources and machine learning in mitigating 0-day attacks**
One of the issues identified is that SDN flows only provide a high-level representation of network data. This high-level representation typically comes in the form of a network “flow” that contains very few data points. This means there is less to analyse when attempting to make changes to a network and allows faster changes. However, machine learning typically requires large volumes of data points to be accurate. The effect of this will need to be investigated, and alternative data sources will need to be identified if (as is believed) accuracy will be affected.

2 **Development of a system that can utilise the advantages that SDN has to offer**
As has been stated, SDN has advantages when it comes to identifying and mitigating malicious activity. The smaller flows, while offering less raw data than packet data would, do allow for faster processing, meaning processing could potentially happen at line rate. Additionally, SDN allows for incredibly granular control of flows. This allows for the potential of mitigation that has been customised for the attack that is detected. For example, flows identified as botnet activity could lead to the infected machines being quarantined, while DoS attack flows could simply be dropped.

In order to meet these aims, the following objectives have been established:

1. **Discover the effect of limited SDN flows on machine learning IDSs (Aim 1)**
   Current literature will need to be examined on to what effect the limited data within SDN flows has on IDS detection rates, as well as what methods have been proposed to overcome the issues posed.
2. **Develop a method to mitigate the effects of the limited data. (Aim 1)**
   A machine learning model will need to be built that can operate at similar levels to that in other state of the art research, but with significantly less data. This will additionally need investigation of overfitting causes and mitigations**.**
3. **Identify other potential data sources (Aim 1)**

One potential method of mitigating the effects of the limited data on detection accuracy is to incorporate other accessible data sources into the detection system. These alternative sources will need to be identified, and methods of aggregating them into the SDN flows will need to be examined and tested.

4. **Determine how to run the model at near line speed (Aim 2)**
   The aggregation of any alternative data as well as the analysis as to the maliciousness of any flows will need to happen at line speed. This is a consideration that will need to be incorporated into the earlier objectives, i.e. the speed of aggregating the data and how computationally expensive the detection system is, but a new sampling method for occasions where network load exceeds capacity to process flows will also need to be developed, especially since machine learning tends to be computationally expensive.
5. **Determine a method to mitigate even 0-day threats (Aim 2)**
   One of the primary advantages of SDN is its ability to react to changing network conditions. A method should therefore be developed to make use of this ability to mitigate malicious activity once it has been detected in real time. Additionally, a primary advantage of machine learning in intrusion detection is the ability to handle 0-day attacks. These should also be mitigated, and so a method to identify a mitigation for previously unknown malicious activity needs to be devised.

1.3 Novelties

As a multi-disciplinary project, this project provides the following novel contributions:

1 **Use an alternative to network packet data**

SDN flows have less data available than reading packet data directly. This can result in reductions in accuracy for anomaly detection. As such, the flow data will be supplemented with network log data. Although Hybrid IDSs are not a novel idea, existing methods classify logs and network packet data in two separate processes. The outputs of both are used to determine if an attack has taken place. The contribution offered by this work is a novel technique that aggregates log and network flow data into a single data stream. Using logs like this should additionally allow for parallelisation of the flow creation process (network flows and log flows), which will aid in distributing computational load.

2 **Initial automated risk assessment mechanism**

A HDBScan-based mechanism has been devised that enables autonomous and near-wire speed grouping of similar network flows based on behavioural similarity and the assignation of a priority rating based on the percentage of flows that have been found to be malicious. HDBScan works significantly faster than other machine learning methods, enabling rapid grouping of each flow to be grouped at or near wire speed. This means that any flows that are potentially malicious (or high-risk) will be analysed by the IDS, rather than potentially being missed by sampling methods. It is believed that this is the first-time unsupervised clustering will have been used to assign a risk score in this way. Being able to run in real time is a significant issue, as even traditional IDS methods struggle to deal with the volume of data, and the use of machine learning adds more complexity.

3 Custom classification model for combination log and network flows

The initial classification of benign or malicious flows will be performed using a specifically developed classification model. The model is based upon a single convolutional neural network with customised layers and uses various techniques to reduce overfitting such as dropout and batch normalisation. In addition, techniques such as PCA, SMOTE or ADASYN will be employed, to attempt to ensure the training data is as useful as possible. As has been stated, ML, and convolutional, models in particular require many features to learn from, and this work shall use only 27% of the features (11 out of 41) other works require. This is a significant issue, as has been stated the more data that is available the better the classification accuracy can be. Deep models are seen as requiring large volumes of data that is not available here.

4 Adaptive mitigation framework

The proposed mitigation framework is capable of identifying a diverse range of threats and implementing differing mitigations depending on the scenario. For example, if a botnet is detected, then quarantining infected systems may be an appropriate response, whereas if a probe is detected than an appropriate response may be to drop the flow. The framework leverages four sequential machine learning models in order to compare the flow to widely accepted attack taxonomies. It is believed this will be the first-time machine learning has been used to determine which leaf of a taxonomy an attack belongs to, and then use this information to devise an appropriate response to an attack.

1.4 Research Scope

This work is targeted at enterprise network infrastructure i.e. networks that are large enough to enjoy the benefits of SDN but not as large or complex as data centre networks or backbone internet structure. However, it is believed that this work could be expanded to other areas where SDN is proving useful, such as IoT and Fog. Specifying enterprise networks is still a broad scope. We specifically target networks that make use of their own (for example) webservers, email servers or file servers. Some (or even all) of these services may have been moved to the cloud in order to allow easier remote access, however it is still the case that many enterprises use local servers that are synced to the cloud to ensure on-site access, legal compliance, or security concerns.

It is envisaged that the kind of company that could make use of this would likely have multiple sites (either nationally or locally), with custom connections linking those sites. Such a business would likely have custom Internet connections, and multiple connections for redundancy. Each site could potentially have hundreds of employees. For businesses of this scale, SDN is a very appealing technology as it simplifies what could otherwise be very complex network management. Similarly, companies of this size may want to use custom IDS services. It is worth noting that potentially office space providers may also find this technology useful, since they are typically responsible for Internet access for their customers (making SDN solutions attractive), however they rarely need to manage servers, or offsite connections.

2 Background

In this chapter, will seek to provide additional context for some of the key areas of this work and discuss in more detail some of issues this work seeks to resolve through discussing previous works on the subjects. Data rates and malicious activity, SDN and how it operates, Deep Learning and intrusion detection methods shall be discussed. Finally, the state of current deep learning IDSs within SDN environments shall be explained, and briefly discuss current relevant datasets.

2.1 Recent Data Rates and Malicious Activity

From Chapter 1 is can be seen that data rates are increasing, and this is of concern for network administrators. However, the statistics need additional context to understand the extent of the issues that are arising. While fixed line internet speeds are expected to almost double between 2017 and 2022 (increasing from an average of 37.9Mbps to 76Mbps for Western Europe) [1] most of that extra bandwidth is not being used on fixed devices. In 2017, roughly 48% of all IP traffic came from fixed devices, whereas in 2022 that is predicted to drop to 29% of all IP traffic. It will instead be replaced by wireless devices, mostly smartphones. In 2017, smartphones accounted for 18% of global IP traffic. This is a significant amount but dwarfed by the 41% of traffic accounted for by PCs alone. In 2022, PC traffic is expected to only account for 19% of all IP traffic, while smartphone traffic is expected to increase to 44%.

This change in usage is important, as it shows some of the shifting demands that administrators are going to have to deal with. Generally, Wi-Fi access points are going to be in more demand as people grow more accustomed to working from their mobile devices. This is important as mobile traffic is more potentially more prone to temporal behavioural variations (e.g. an unexpected number connections, or traffic volume more generally), with high peaks and low troughs. In addition, those peaks may change location depending upon time, even within the same LAN. With fixed terminals, it was understood where most traffic within a business originated from, and when (where the largest number of terminals are located, and when they are scheduled to be in use). This predictability makes it comparatively easy to plan for. The administrator can determine a minimal acceptable service level, and plan to have enough bandwidth to accommodate it. With the move to wireless, this is no longer the case. While the number of people within LAN can still be estimated with some reliability, where those people wish to access local resources is constantly changing. For instance, an impromptu company meeting could see excessive surge in one room, which previously may have only housed a couple of fixed terminals. Offices with many terminals may see less use as workers move more freely to other locations to discuss matters with other staff not located in the same room. This kind of activity puts more pressure on wireless access points, some of which may only see those kind of data rates very infrequently.

Along with this change in how networks are being used, there has been a change in what malicious activity is undertaken. Ransomware attacks have generally decreased since reaching a peak in 2017 [8], however the number of ransomware attacks that specifically target businesses increased 12% between 2017 and 2018 [8][7]. Ransomware relies on being able to spread through a local network, in order to encrypt both active files and any backups (or the victim could just wipe the affected computer and restore it). In addition, there have been instances of attacks that initially look like ransomware; however, the files are encrypted with a random key that cannot be recovered, meaning that the files are permanently lost. While using the same malicious software and tools, this means that the attacks are purely being used for disruption and destruction, with little intention of making large amounts of money directly. The reasons for these kinds of attacks vary, but again they are primarily targeted at businesses, not individuals.

There has also been an increase in “Living off the land” attacks, which target commonly used applications and attempt to hide malicious activity as benign. Applications used like this include Remote Desktop Connection (RDP). Similarly, worms have seen a decrease in spreading via remote exploit vulnerabilities [7], [9]. Instead malware has moved to using simple techniques such as brute forcing passwords to laterally move across a network. Tied in with this has been an increase in “supply chain” attacks. These involve exploiting third party software or services in order to compromise a final target. A prominent example is the Petya/NotPetya [8][10] ransomware attack that was propagated through the misuse of a software update of MEDoc, a Ukrainian tax preparation program. Administrators generally have little control over the content of software updates and are frequently advised to install them without being able to gauge the security risks of the update. This kind of attack will allow an attacker to gain a foothold in many companies that use the software exploited, and can provide a foothold to launch further attacks using different methods if the attacker chooses (for example, NotPetya uses a version of Mimikatz to gain administrator passwords to spread across local networks).

Finally, there has also been an increase in IoT-specific attacks [8] (an average of 5,200 per month against Symantec’s IoT honeypot), as these devices are often relatively poorly protected, but provide an excellent foothold for an attacker to launch more attacks. Many routers for instance provide all the tools needed to initiate a password attack against a server (using SSH for instance, or telnet), and are often themselves left with default or unsecure passwords.

Moving away from attacks that specifically target businesses, there has been an increase in malware that targets mobile and other IoT devices within the last three years. This may be unexpected, considering the increasing prevalence these devices have in general society. This does pose an issue for business administrators however, as they need to be aware of the risks of the mobile malware and how to best prevent its spread. It is estimated that 1 in 36 mobile devices used within organisations could be classed as high risk [7]. This includes devices that had been routed or jailbroken, along with devices that had a high degree of certainty that malware had been installed on the device. Taken in conjunction with the fact that most data is going to come from mobile devices, this means that a large number of devices that are not under the administrator’s control, with a high risk of being compromised, are being taken into the heart of an organisations infrastructure and placed onto the same LAN as potentially sensitive documents or other servers. These devices seem to offer ideal opportunities as jumping off points, either to infect computers they are connected to or as ways to infect wireless routers they connect to.

In summary, just as the use of computer networks has changed, so has the malicious use of computer networks. IoT devices were once seen as secure and are now a potential major weakness. Updates from suppliers once only needed to be vetted enough to ensure they did not break existing processes, but now can be another point of weakness. Mobile device malware infections have grown steadily, and these devices are invited into the heart of company networks. Malicious activity can be started from any IP connected device and can target any IP connected device. To meet this threat, IDS's need to be able to detect threats from all over a network, not just isolate particularly sensitive areas.

2.2 SDN

The volume of network traffic is increasing, however the volume itself is less of a concern than the complexities that come with managing it are. As data rates grow, the infrastructure required to deal with the growth becomes more complex, often requiring hundreds (if not thousands) of switches and paths.

2.2.1 Traditional Networking

Traditional networking works through devices having two planes, a control plane, and a data plane, which are vertically integrated into a single device (the network switch) [11]. The data plane is responsible for the movement of packets from one port to another, based upon a routing table. When a packet enters a port, packet header information is recorded and a match in the table is searched for in order to route the packet correctly. If no information for the packet exists within the routing table, the data plane forwards the packet meta-data to the control plane. The control plane is responsible for building a map of the network (or the network topology), and creating the routing tables for the data plane, dependant on that map. So, when it receives packet information from the data plane that doesn't match any existing rules, it refers to the topology it has built (through the use of protocols such as the Border Gateway Protocol (BGP) [12] or the Spanning Tree Protocol (STP) [13]) to determine how to forward that packet, and updates the routing table accordingly.

This is an effective system, which limits the impact of failures through not being centralised. If an individual switch fails, then neighbouring switches can learn of this failure and attempt to reroute traffic around the failed node. This was an important consideration of early IP networks that were supposed to work even in the event of catastrophic damage to infrastructure (given the military background). However, the same system leads to issues of scalability. Each individual switch needs to build its own topology map, and no switch will have a complete map of a sufficiently sized network. This will be due to limits in:

- **Memory:** Even if you add more memory to a switch (which can be costly), that memory needs to be fast enough to allow the data plane to search the entire routing table to forward the packet at line speed.
- **Processing power:** Building the network topologies is not a simple task computationally and each node added to a network increases that complexity. Larger networks require increasingly powerful processors to build and maintain the network map**.**
- **Available bandwidth**: Network switches attempting to gather information about the network will need to use some network capacity.

None of these issues are insurmountable but the solutions are costly on an individual switch basis, and when you multiply the costs hundreds of times for multiple switches in a large network, they can easily spiral out of control.

2.2.2 How SDN is Different

SDN changes the traditional view of networks by separating the control and data planes. Switches continue to contain the data plane and are responsible for the forwarding of packets depending upon a routing table. However, that is all they do. The control plane is moved to a centralised device, the controller, which many switches can contact and ask for instructions. This does implement a single point of failure in networks (if the controller is compromised then the network could be), however it also centralises the costs into a single device.

The model also provides other advantages. Traditionally, managed switches needed to be programmed individually before being placed into a network, as well as needing to be individually reprogrammed if the network structure changed. While tools to aid this process did exist, they were typically locked to a single provider through the use of proprietary protocols (causing issues if the vendor were to stop supporting that protocol) and were propagated through the network on a switch to switch basis (causing issues with a potential lack of clarity on when/if switches had received updated instructions). In contrast, SDN switches need minimal configuration before being placed in a network, and when a network structure changes, switches can receive the new instructions from the controller directly, simplifying and quickening the process. In effect, it allows what would normally be managed switches to become unmanaged. This advantage combined with the centralisation of cost allows for another advantage – dynamically programmable networks.

Since the controller is a singular device, it gives the opportunity to add another plane above the control plane, the application plane. This application plane can receive information about the network state from the control plane (named the northbound interface), and issue instructions depending upon several factors and different programs (see Figure 2.2-1). For example, you could program an application to redirect traffic through certain paths at certain times of day for maintenance reasons or redirect traffic in response to malicious traffic being detected. These rules would be difficult to implement in the traditional networks, for two main reasons:

1. The decentralised nature of networks means that getting multiple switches to act in a coordinated manner, even switches that are not directly connected or aware of each other, is a difficult task.
2. The custom programs would require additional computational resources, multiplied by every switch in the network that needs to be able to run these programs.

What this effectively means, is that while under the old, decentralised paradigm, traffic was routed depending upon IP or MAC address only, under SDN, traffic can be routed depending upon many other conditions, all of which are configurable. In summary, a SDN architecture is:

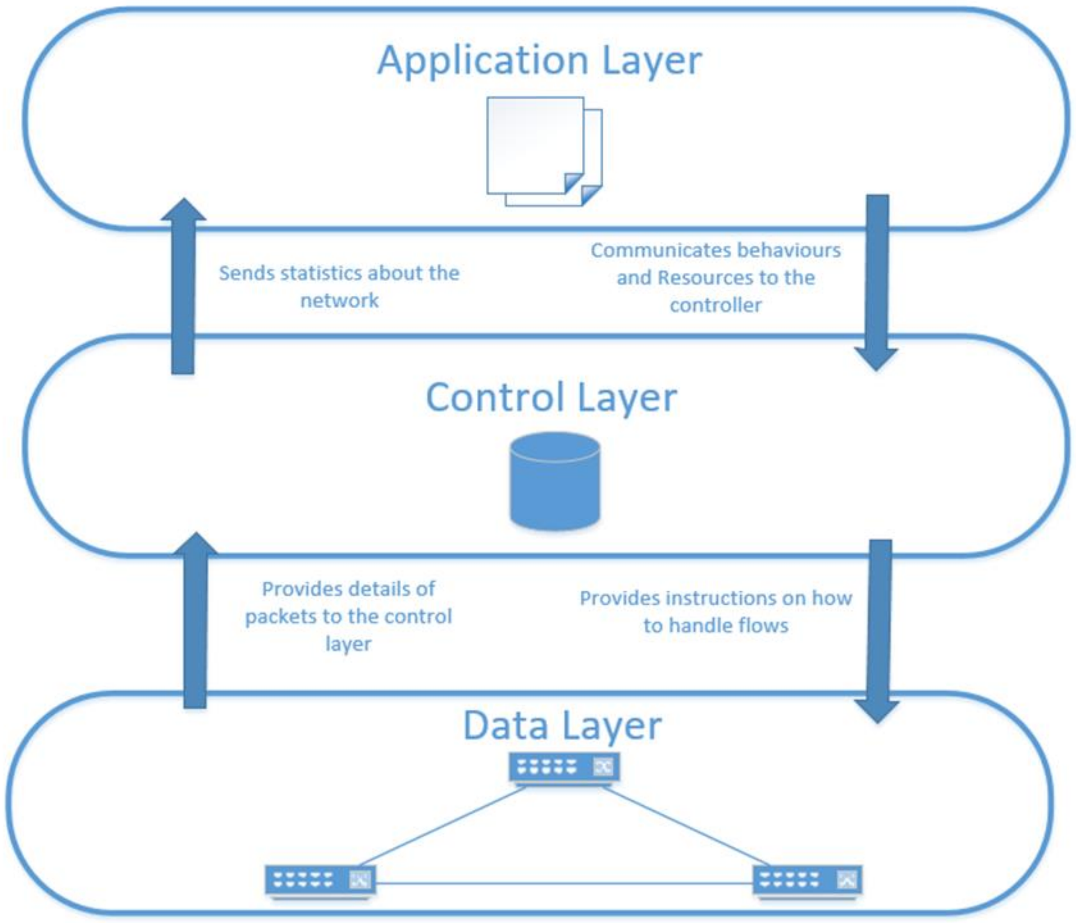

Application Layer
Sends statistics about the network
Communicates behaviours and Resources to the controller
Control Layer
Provides details of packets to the control layer
Provides instructions on how to handle flows
Data Layer

Figure 2.2-1 – How the layers of a SDN interact.

1. **Directly Programmable:** As the network control is decoupled from the forwarding functions, network devices can be programmed directly.
2. **Agile**: Abstracting the control plane from the data plane allows administrators to dynamically adjust network-wide traffic flow to meet the changing needs of dynamic network conditions.
3. **Centrally managed:** The network intelligence is centralised in the controller, which maintains a near real-time global view of the network, which appears to applications as a single logical switch.

2.2.3 The Effects of SDN on Intrusion Detection

As established in Subsection 2.2.2, SDN is a technology that has been created in response to growing data rates and the resulting increase in network complexity. However, the benefits or drawbacks to network security as a result are less clear cut. Other academic works (discussed in Chapter 3) have indicated that SDN can have some security benefits, making some attacks more difficult, but also that the controller being a single point of failure, makes other attacks easier. For example, it may be possible to cause a DoS by flooding the controller with many new flows. While attacks like this may have been possible previously, they would target single switches, and the distributed nature of networks would significantly reduce the potential impact. Broadly, SDNs should be more resilient against DoS attacks. The purpose of SDNs is to increase the utilisation of network hardware, and to decrease the reaction time in response to changing network conditions. A SDN should lessen the impact of DoS attacks since they should be able to automatically detect the increasing volumes of traffic or failure of network devices, and redirect traffic accordingly. Additionally, once the DoS attack has been detected, the SDN should make it easier to react to the attack, blocking or redirecting malicious flows.

Generally, it is agreed that while SDNs do have associated risks that need to be accounted for when designing an SDN, they also bring benefits that should not be ignored. Another example is that SDNs allow for the redirection of traffic to middle boxes anywhere on the network for traffic to be analysed. Currently, middle boxes (IDSs, firewalls, load balancers etc.) need to be in specific parts of the network and only affect traffic within those parts. SDNs allow these devices to be separated, with traffic redirected to the devices if the SDN controller decides it is necessary. This offers some advantages, including separation of the security middle boxes from the main network, more selective analysis of traffic (only analysing traffic that SDN controller deems important) and centralisation of the middle boxes themselves (only a single device is able to analyse multiple areas of the network). Of course, these adaptions also have implications that need to be considered. If a NIDS is going to analyse all traffic across a network (rather than just a small section) then that NIDS needs to be computationally powerful enough to cope with the extra traffic.

2.3 Deep Learning

Representational learning is a technique that enables a system to automatically determine the representations for a given classification based upon the raw data. Machine learning is an example of representational learning, in that a machine learning algorithm determines the features that differentiate the classes automatically. Deep learning is a subset of machine learning that uses multiple levels of representation, with each level transforming the representation to a slightly more abstract model [5]. For example, rather than an input layer containing the pixels of an image and a shallow classifier determining the picture contains a face (something most shallow classifiers would not be able to do with sufficiently complex pictures), a deeper model may have the same pixel input, and then the first hidden layer detect edges. The second layer might detect the edges from an eye or mouth. The final layer may detect the image contains eyes, a mouth, a nose, and ears while the final classification determines the picture contains a face.

In the same way, a deep model attempting to classify malicious traffic may have a first layer that detects "edges" of packets (e.g. a small packet size, a short Time To Live (TTL) or the use of a particular protocol). The second layer may add some context to these features, while abstracting the actually features (e.g. the packet is small or the protocol is unusual) and the final layer may be used as a final classifier (e.g. a DDoS is being attempted). This usage of multiple layers allows greater generalisation and can lead to better classification accuracy than shallow machine learning models are capable of, as each layer is building on the representations of previous layers [5].

Deep learning algorithms generally come with greater computational expense, which is something that initially held back their adoption since their conception in 1965 [14]. However, the advances in processing capabilities, such as the introduction of General-Purpose Graphics Processing Units (GPGPUs), has reduced these barriers.

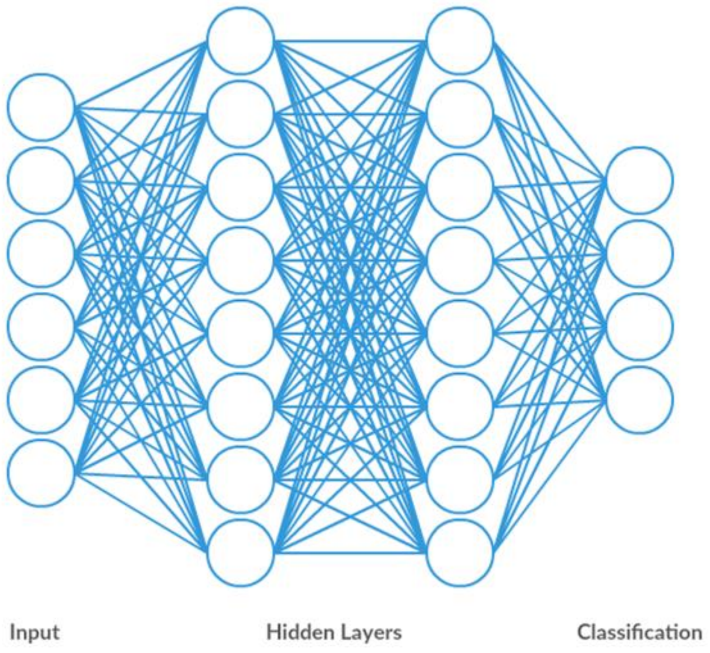


Figure 2.3-1 – An example of a DNN structure

Figure 2.3-1 shows an example of a Deep Neural Network (DNN) structure. The input layer contains 6 features, which is expanded into the first hidden layer with 8 nodes, which maps to another hidden layer with 8 nodes, before resulting in classification layer with 4 nodes. In this case, the network would be detecting 4 discrete things. If applied to detecting network attacks there could be classifications for benign traffic, DoS traffic, Probe traffic and other malicious traffic.

The first layer, termed the Input layer, is the raw data of the thing being classified. For example, in a grayscale image this would consist of individual pixels, typically ranging in values from 0-255 depicting the colour depth. For intrusion detection, this consists of packet data, including things like protocol type, TTL, size and source/destination addresses. The middle two layers are termed “hidden” layers as they are not directly observable from the system input and outputs. Each hidden layer transforms its own input data (i.e. the data from the layer before it) into a slightly more abstract version. With enough of these transformations, complex features can be learned [5]. This process is termed representational learning and higher levels of representation (i.e. further along the model) aspects of the input that are important for the classification task are amplified, and less important aspects are supressed. The key aspect is that these representational layers are not coded by an engineer but are learned from data.

Finally, is the output layer (also known as the classification layer), which is commonly a more traditional shallow learning classifier, which receives its input from the final hidden layer. As opposed to being another layer of the same Multi-Layer Perceptron’s (MLP) node that make up the hidden layers. These are typically linear layers that compute a weighted sum for each of the potential classes. If this weighted sum is above a threshold then it determines that the input belongs to a particular category. The issue with these shallow classifiers has been that they are sensitive to irrelevant changes in the input, which leads to misclassification when used alone. However, when paired with a suitable feature extractor (the hidden or deep model for example), they can perform well, being able to detect small changes in input. Previously, the feature extractors were designed by hand, specific to the kind of input that was being classified. This took great amounts of engineering skill and domain expertise, however the automatic feature extraction enabled by deep learning simplifies the process significantly.

While all the nodes are connected to every other node on the next layer within Figure 2.3-1 (known as a fully connected network), this is not required, or even necessarily recommended. It has been found that limiting connections can aid in reducing overfitting. Overfitting is a phenomenon that can occur when a model is trained too much. A goal of machine learning is to identify not only samples it has been shown, but examples that are similar to what it has been shown but it has never seen before. The ability to adapt to new data is referred to as generalisation. When overfitting occurs, generalisation loses accuracy (typically on testing or evaluation datasets) as the models start to only associate the training data with the classifications. Overfitting can be avoided in three main ways:

1. **Using more data:** Both increasing the number of samples you have, as well as increasing the number of features within the dataset can aid in overfitting. Increasing the amount of data forces the model to attempt to learn all new data and prevents it from fitting too closely to the original training data.
2. **Using less data**: While this may seem to contradict the first point, often a lot of data within datasets can be “noisy” – i.e. data that does not help with the end classification. This can be features that heavily resemble other features within the dataset, or samples that are similar (or even identical) to other samples within the dataset. Both examples force the model to learn the same thing multiple times, increasing bias towards these values.
3. **Using regularisation**: Regularisation techniques (e.g. dropout) ensure that all nodes are not always connected to every node in the next layer by randomly “dropping” a specified amount of these connections. This prevents the network from relying too heavily on a few strong features and forces it to use other features to learn as well. Other types of regularisation exist (to be discussed in later Chapters) but the principle is the same for them all.

Broadly, machine learning can be separated into three different classes: Supervised, Unsupervised and Semi-supervised.

Supervised learning is used when all the data used in the training process is labelled. It explicitly matches inputs to a list of outputs depending upon training examples that have been provided previously. This is achieved through generation of a function to match the training inputs to the labelled output and uses this function to infer the output for new samples [15]. Examples of this in practice include Artificial Neural Networks (ANNs), decision trees, linear and logistic regression and Support Vector Machines (SVM) [16]. Generally, these methods have advantages in terms of accuracy and false positives over other forms of machine learning, but also tend to be slower to train. Additionally, as these require fully labelled data, they also have more issues with the difficulties of getting high quality pre-labelled training data.

Semi-supervised [17] models use a mixture of labelled and unlabelled data. The goal is typically to lessen the burden of obtaining large, labelled datasets. The assumption is that if the dataset contains a few labelled examples of every class, then the unlabelled data can aid in forming decision boundaries. Similar to supervised learning, semi-supervised requires examples of every class within the training data, however, the use of unlabelled data allows larger training datasets, that can be constructed comparatively inexpensively (in terms of expertise and time required for labelling). Generally, semi-supervised models achieve higher accuracy rates than unsupervised methods, and it can achieve higher accuracy rates than supervised learning on the same datasets if the extra unlabelled data does help form the decision boundary. However, in cases where all the training data is labelled, or the unlabelled training data is low quality or does not help form a decision boundary then semi-supervised learning will not hold any advantage over fully supervised methods.

Unsupervised models use completely unlabelled data. They seek to detect shared attributes across the data and group records together, based upon the presence or absence of these shared attributes [18]. Clustering is one example of this, where records are plotted on an n-dimensional graph (where n is the number of features) and clustered depending upon their location. Autoencoders are another example, where an attempt is made to reduce the number of features of a record to a set amount while retaining as much information as possible (i.e. to learn a lower dimensional representation of the original data) [15], [19]. Unsupervised methods have advantages in being able to use unlabelled data and are frequently less computationally expensive than supervised methods but can have more issues relating to accuracy. Unsupervised methods are commonly used to reduce the dimensionality of a problem before a supervised algorithm is used to classify it. Most unsupervised algorithms will attempt to reduce the dimensionality of a problem in some way [15], making them effective feature selectors to decrease the computational complexity of the supervised classifier.

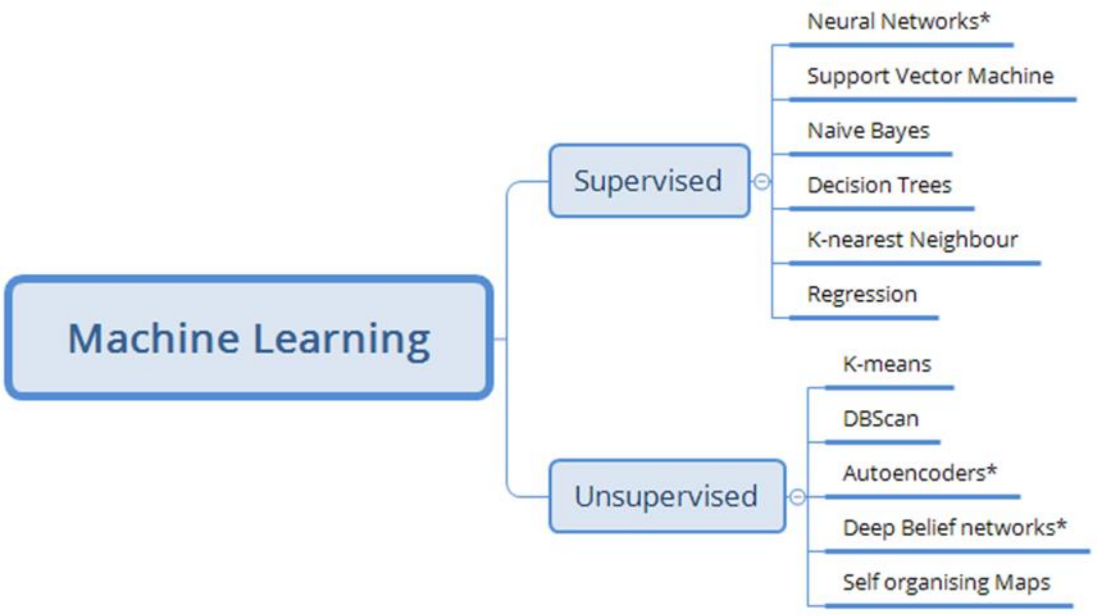


Figure 2.3-2 – A Figure to some example techniques used in the primary categories of Machine Learning – Asterisk is used to denote an algorithm that is frequently used in a deep manner

Figure 2.3-2 shows some example techniques used in machine learning and whether they are primarily supervised or unsupervised. The * denotes an algorithm that is frequently used in a deep manner (i.e. multiple levels with hidden layers between the input and output). As can be seen, there are examples in both supervised and unsupervised models, meaning that deep models can be supervised or unsupervised depending upon the exact type of model chosen e.g. you can have a deep unsupervised autoencoder network, or a deep supervised neural network.

Also of note in recent years is reinforcement learning, which differs from supervised and unsupervised learning in that it is active (i.e. it changes the input space it exists in), whereas supervised and unsupervised are passive (they classify the input space, and do not make changes to it). This makes it useful for applications where changing the input space is desirable, for example, game theory (e.g. AlphaGo [20]) or control theory (e.g. load balancing [21]). However, it is not a classifier in the same way as supervised or unsupervised learning can be.

2.3.1 Data Preparation Techniques for Machine Learning

The volume and quality of training data for machine learning models is particularly important and overfitting is a common problem for deep models. This subsection will discuss a few common methods for improving the quality of training data.

Increasing the variety of data to ensure that all scenarios are represented is important. However, this can be challenging if your dataset is imbalanced or you only have access to limited volumes. One workaround for this problem is to use techniques to generate synthetic data points. Synthetic Minority Oversampling Technique (SMOTE) [22] is one such technique. Rather than undersample or oversample, SMOTE creates new data for the minority classes. This is performed using k-nearest neighbours. A real point is taken, and a vector drawn from that point to one of its *k* neighbours. The vector is then multiplied by a random value between 0 and 1. Adding this to the original data point creates an entirely new data point. There is an assumption that k-Nearest Neighbour can accurately classify data which is not necessarily the case, however, the technique has worked well for intrusion detection [23].

ADASYN [24] works on a similar premise to SMOTE, in that it creates new minority samples using existing values. However, unlike SMOTE, it attempts to discover which minority samples are more difficult to learn through analysis of the data distribution. The goal is not only to reduce the bias caused by uneven data distribution but also to shift the decision boundary by creating more examples of the difficult to learn samples.

2.4 Existing Intrusion Detection Systems

It is commonplace for attacks that attempt to spread through an internal network after gaining a foothold in a poorly secured device. With the growth of supply chain attacks (which an administrator may have little initial control over) [8][10], the ability to monitor the internal network becomes more important than ever. Malicious activity needs to be detected and mitigated as soon as possible. Previously, detection of malicious activity was accomplished using a combination of signatures and statistical analysis.

2.4.1 Signatures

Signatures for network IDSs work on the same principal as common anti-virus or anti-malware software. As new forms of attacks are identified, a signature of the type of data the traffic creates is made, typically by companies specialising in IDS technologies. For example, the cmd.exe attack that was used by Nimda and Code Red uses a specific sequence of bytes that it can be identified with (bytes that correlate with copying the cmd.exe file to an accessible location). It is extremely unlikely there would be a legitimate reason for a remote user to perform that action, and so by monitoring all traffic for that sequence of bytes the attack can be identified. However, some potential issues do arise from this method of detection:

- **It is slow by nature**
  For attacks to be classified in this manner, successful attacks must first have taken place and been identified through other methods. After being identified, then the process of creating a useful signature is undertaken (i.e. a signature that will reliably detect the attack, without being too broad to detect benign activity).
- **Encryption can interfere with the detection**
  Going back to the example of the cmd.exe attack, if performed over an encrypted connection (HTTPS in this example, as it targets IIS servers), then applying the signature on a NIDS will not be of any use, as the sequence of bytes will have been encrypted. This means the signature instead needs to be run on the end point, as this will be the first instance where the unencrypted data will be analysable.
- **Small changes to the malicious activity can bypass the signature**

Since signatures rely on finding common byte sequences in malicious activity, that signature can be bypassed by simply not including the specific byte sequences. It becomes obvious that malicious users are exploiting this when examining the volume of variants of some pre-existing malware. For example, Zeus has had hundreds of variants, some of which are even still in use today [25].

2.4.2 Statistical Analysis

This method of malicious activity detection acts as a complement to signatures. While signatures can find malware and worms (exploit type attacks) propagating across a network, statistical analysis is usually employed to identify DoS type attacks. For example, if general traffic towards a system exceeds a set amount, then the traffic may be classified as DDoS attack. This is often more useful in cases where traffic from a single source appears unusual. For example, a common statistical rule might run along the lines of 'Classify as malicious any connection where the percentage of TCP SYN packets exceeds 50% of all packets, and there has been more than 50 packets sent.' This kind of rule would attempt to classify TCP SYN flood attacks, by providing a firm barrier to the volume of TCP SYN packets the system would accept as benign. This detection method doesn't suffer from the same kind of issues as signatures. However, attackers are aware that systems using such rules are in place and as a result, attempt to ensure attacks look as much like background traffic as possible. This is evident in the rise in the number of ransomware attacks that are used specifically as destruction and disruption attacks. Rather than pay for a DDoS attack to bring down systems for a short period before malicious traffic is identified and controlled, malicious users have started to employ ransomware attacks as pseudo DoS attacks [8]. Data on infected machines is encrypted and the encryption key is deleted, meaning there is little hope of recovering any data even if a ransom is paid [8]. These attacks have been targeted, and there appears to have been little effort in monetary reward, instead the goal appears to have been to be as disruptive as possible for as long as possible. This goal is what DoS attacks are typically used for.

2.4.3 Reputation Based Analysis

Reputation based IDSs have some similarities to statistical analysis in that the goal is to detect activity that is unusual, however, it goes further in that the potential malicious activity is marked and the reputation of the system is considered while deciding if the activity is malicious [26]. If the system producing the anomalous activity is highly trusted, then the anomalous activity can then be trusted as well. This can aid further detection attempts by also being a point of reference later. If a less trusted system starts to display the same anomalous activity, than the system can compare it to the early activity, see that it was listed benign and allow it to continue. The issue with reputation-based systems becomes one main question. How do you decide on the reputation of systems? Some solutions have involved using reputation scores that change based upon the proportion of anomalous activity that has been detected in the past or how accurate a node has been at detecting other malicious activity [26]. The issue with this method is that it takes time to determine how trustworthy a node should be, after all, how can you decide based on past actions if there have been no past actions? Other more traditional methods revolve around setting a score based upon how secure the node should be, with unknown sources being low trusted, and known, well secured nodes being highly trusted.

2.4.4 Types of IDS

There are generally considered to be two main types of IDS: Host-based (HIDS) and Network-based (NIDS). HIDS reside on individual network devices and protect only those devices, whereas NIDS reside in strategic locations on the network and protect all devices within the corresponding network segment. Despite only monitoring singular devices, HIDSs do have some advantages over NIDSs, in that they have access to more detailed data. A NIDS might only have access to packet header data as well as data derived from that data (e.g. packets per second). A HIDS that can monitor the network interface can compare what is being broadcast, to what is shown as running, and check if those programs have the authority to broadcast packets in that way. They also potentially can check packet body data, even if that data is encrypted on the network level. If a program is broadcasting data, then a HIDS might be given permissions to read any data before encryption on the network level to ensure it is not malicious. Alternatively, it could be given permission to read any received data after decryption. NIDSs do not have this luxury typically. The fact that they must analyse data coming from and to several sources, means that packets must be analysed even more quickly to keep at wire speed.

NIDSs have advantages in terms of scope and protection as a single NIDS can protect several network devices and protect them earlier than a HIDS. For example, a NIDS could detect a DoS attack (many of which do not require complex processing to detect) and block that malicious activity from ever reaching the source. A HIDS may well be able to block the same activity, however by the time a HIDS is analysing the attack, the goal may have already been achieved (e.g. saturating the target's bandwidth). NIDS are also able to detect distributed malicious activity. If a malicious user targets several different devices, it is possible that individual HIDSs could miss the attacks. It is only when combining the disparate flows that the pattern becomes noticeable. An example includes botnets that attempt to spread through password attacks. The goal of the malware is to remain undetected while spreading, however targeting a single device with many password attempts is noticeable. So rather than perform ten password attempts on a single device inside a minute, they perform the same amount of password attempts on different devices. From the attacker's point of view, this may have the disadvantage of different devices having different passwords, and ultimately take longer to infect those different devices. However, in practice, passwords are frequently shared between devices to ease operational maintenance. Once the password for one device is found, it can be attempted on all the other devices. A NIDS may not be able to detect that all of these flows are password attempts (due to encryption or obfuscation techniques), however, it would likely recognise that the system has initiated an unusual number of flows (TCP SYN packets) to different devices in a short amount of time.

In practice, HIDS and NIDS are used symbiotically, with NIDS designed to detect as many network attacks as possible as early as possible, and HIDS being used to detect attacks that either did not spread through the network or were missed. For instance, a NIDS is not going to be able to detect a malicious actor typing in different passwords to a physical console, and may not be able to detect multiple password attempts using something like SSH due to the encryption inherent in such protocols (without including other obfuscation techniques a malicious actor may employ). A HIDS however, should always be able to detect this activity through monitoring of system logs.

Both HIDS and NIDS often use a mixture of statistical and signature-based techniques. For instance, SNORT can be configured with rules to detect port scans dependent upon connection count per IP, ports connected per IP and connections per port (a statistical method), or can use signatures to detect malware through identifying the presence of specific byte code sequences in an executable (a signature).

In order to detect a password attack like the one described above, a NIDS would generally need to analyse the packet body. The connections set up for these kinds of attacks can be completely legitimate, with nothing to indicate they are malicious within the packet header. This kind of analysis (termed Deep Packet Inspection) is more computationally expensive than examining packet header data. The extra expense limits the locations where it can be performed, and how many different signatures can be used before network traffic is slowed.

While Figure 2.4-1 shows how an inline NIDS would be positioned, Figure 2.4-2 shows how an offline NIDS might be positioned. This kind of NIDS requires the switch copy and forward data to it (e.g. using a span or mirror port), while allowing the flow to continue. This kind of system is not designed to monitor data in real time, instead sacrificing speed of response for completeness. If a system is not waiting for benign data to be analysed, more time can be taken in that analysis to ensure any attacks are detected. Due to the delay, this is an area where DPI can be used, as the computational complexity does not matter as much.

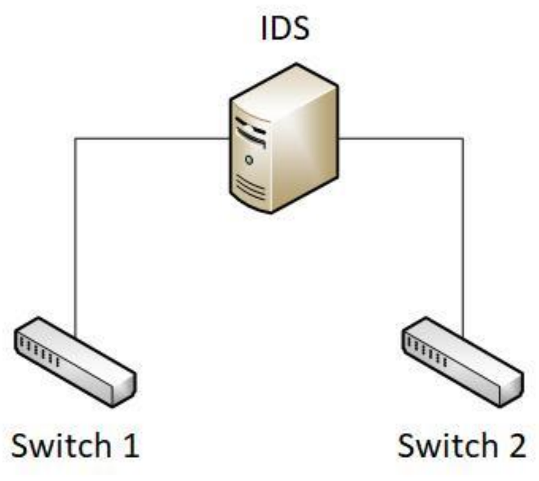


Figure 2.4-1 – An Inline NIDS

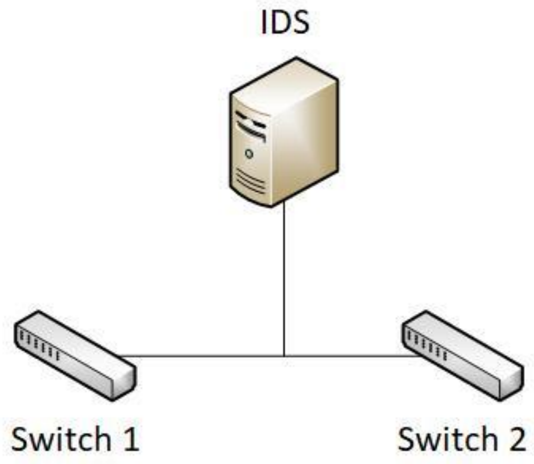


Figure 2.4-2 – An offline NIDS

However, the continued use of DPI is coming into question for other reasons beyond its computational complexity. Increasing use of encryption is still rendering DPI less useful. While issues with network level encryption can be bypassed by placing the NIDS after the data has been decrypted (in the case of a Secure Socket Tunnelling Protocol (SSTP) for instance) it will not bypass application layer encryption. Often, packet body data is encrypted before it is separated into packets and transmitted, and the receiving application holds the decryption key (which is decided on a per-connection basis) [27].

Given the difficulties in increasing use of encryption as well as privacy concerns, ideally only packet header information should be used for intrusion detection at the network level. Packet headers contain a great wealth of information and many features are considered important by academics and industry. For example, if attempting to detect an ACK port scan, it could be considered good practice to set a rule that detects any packets that have a sequence number of 0 and have the ACK flag set. Both features sit within the packet header and would not be encrypted at an application level. However, detecting higher level threats does become more difficult without access to packet body data, meaning that if a NIDS is required to detect higher level threats than other measures start to become necessary. For example, if a user is connecting to several different servers (SYN or SYN/ACK messages) with those servers sending reset packets shortly after (RST messages) then this could be an indication that password attacks are taking place. The issue with this (statistical) analysis is a greater degree of uncertainty. While the activity could be potentially malicious, there are other benign reasons it could be happening. In conclusion, it is not feasible to continue to rely on statistical and signature-based approaches if all that is accessible, is the packet header.

2.4.5 Deep Learning IDSs

While it has more in common with statistical analysis (both ultimately being anomaly detection methods) deep learning as a tool for anomaly detection has several advantages. As it is an anomaly detection method, deep learning is more resilient to obfuscation techniques than signatures. This is because instead of matching a sequence of binary data or using set rules, anomaly detection methods simply look for data that is unusual. It is also more resilient to 0-day threats as all that is needed for an attack to be detected is for the activity to be unusual.

At first glance, this restriction for malicious activity to look different may seem to be the same as the kind of restriction on statistical methods. However, statistical models typically rely on broad rules that cannot capture nuanced activity, and which can be bypassed with little effort. For example, statistical methods may be used for detecting password attacks (e.g. if there are more than 5 password attempts in 1 minute, flag it as malicious). Frequently, malware and malicious users attempting password attacks will simply limit the number of attempts per minute that they make. Deep learning IDSs can bypass issues like this as the automated feature extraction allows it to detect patterns that broad rules cannot encompass. For example, a deep learning IDS could determine that all sent packets being fairly short, all responses being the same or a similar size (with a standard password not accepted response), and the number of interactions before a RST ACK are important features when detecting password attacks, and that flows that have more of these features are more likely to be password attacks.

Most deep learning based IDSs use packet header data, and information derived from that data (number of packets per second, or number of SYN packets, etc.). Most research datasets also follow the example of using packet header data and data derived from it [28]–[30]. This makes sense considering that mostly this is the data that has been available to network administrators. Additionally, this aids with one of deep learning's disadvantages. Deep Learning models typically require large amounts of both features and training data. Even without packet body data, packet headers can create the large volume of features required. For example, the UNSW-NB15 [31] dataset contains 47 features, of which 43 could be obtained from the packet header alone. When compared to other situations where hundreds or even thousands of features are used, this may not sound like a lot, however several of the features are categorical. This means that they could feasibly be split into features themselves through a process such as binarisation. Binarisation is the process of taking categorical data and turning it into binary data.

So, if you have a categorical entries such as the protocol type e.g. TCP, UDP and ICMP, binarising this list would result in three columns where 1 is used for the presence of that protocol, and 0 used for its absence, as shown in Table 2.4-1

Categorical Label	**Converted Binarised Labels**		
Label	TCP	UDP	ICMP
TCP	1	0	0
UDP	0	1	0
ICMP	0	0	1
TCP	1	0	0
TCP	1	0	0

Table 2.4-1 – Example Binarising of Protocol Types

The volume of network data being generated could be considered a boon to these models. Deep learning models require large volumes of data to learn from, as such, one would imagine training data is plentiful for NIDS. Unfortunately, this is not necessarily true, as the labelling of network data is particularly difficult due to a combination of legal and technical issues. In terms of academic studies, the sharing of network datasets becomes difficult when considering the potential privacy issues involved. Data will need to be anonymised (potentially damaging useful features) at the very least, and permission may need to be sought from anyone using the network at the time of the recording in order to share the data as a dataset with other academics. Additionally, deciding which groups of network data are malicious is a difficult and time-consuming task. With potentially millions of rows of network data, it would be normal to use another program to detect any malicious activity that has been recorded. However, if that is done then any model that is generated as a result of the dataset is likely only going to be as good as the tool used to label it. This is not to say these issues cannot be overcome, but that getting high quality network datasets is difficult, and deep learning models are potentially more reliant on their training datasets than tools using signatures or statistical based approaches would be.

Deep learning also has issues when it comes to accuracy and false positives. Simply because it is an anomaly-based approach, deep learning can label benign flows as malicious as they appear like other malicious activities. A common example from statistical analysis would be the account lockout function. Usually, if an account is locked due to too many password attempts, it is because the user has forgotten their password rather than a malicious actor attempting to gain access to the account. However, the statistical analysis rule (lock the account if there are x passwords attempts within y seconds) cannot tell the difference between a benign user forgetting their password and a malicious access attempt. It only detects that the access attempt is an anomaly and reacts accordingly. Deep learning has the same problem, except while an administrator could tweak the statistical analysis rule to attempt to accommodate forgetful users while still blocking malicious activity, the deep learning model is dependent on their being data that can distinguish the two types of activity.

However, machine learning models are susceptible to a form of attack both signatures and statistical methods do not suffer from. Termed adversarial attacks, these are samples that are perceptually indistinguishable from one class, but are classified incorrectly [32]. Assuming *C* is a correctly classified sample by a machine learning system *M*, then *M(C) = ytrue*, however it is possible to create a sample such that *M(C') ≠ ytrue*. Adversarial samples are classified according to the knowledge of the system that is being targeted by the malicious actor. These classes are white-box, grey-box and black-box. White-box adversarial attacks are when the attacker has detailed knowledge of the model they are attacking, including parameters and structure. In grey-box adversarial attacks, the knowledge of the attacker is limited to the structure of the model being attacked, and in black-box attacks the attacker has only the knowledge gained from query access [33]. However, it has been found in many cases that attacks created for one model will transfer over to another model successfully [34], which means it is possible to create adversarial examples and perform a misclassification attack on a machine learning system with no knowledge of the model. It has been shown that adversarial attacks can still take place within machine learning IDSs [35], [36], even with the more constrained environment networks operate in where there is less scope to change values within packets without causing errors (for instance, changing the packet type from TCP to UDP would cause errors, since the packet structure is different).

Finally, deep learning has issues in terms of computational complexity. Signatures and statistical rules are very computationally inexpensive to operate. They typically amount to simple “if statements”, whereas deep learning requires much greater computational power to run. This is particularly important in NIDSs, since the goal is typically to analyse network data at line speed, so that malicious activity can be stopped before it has had chance to cause any damage, or even to reach the intended target.

2.5 Deep Learning IDSs within SDN environments

As noted in Section 2.2, the goal of a SDN is to separate the control plane from the data plane, which allows the control plane to become programmable. This is done through switches (on the data plane) sending information about the packet header to the controller, which then decides how to handle the packet. All packets are not forwarded to the controller however, just enough information to create an entry in a flow table. The flow table is what is used in order to decide how flows (collections of related packets) should be treated. This information is also what is provided to the application layer in order to do any other more complex calculations. For example, if a SDN controller was going to be used to load balance between two parts of a network, it would get the information on number of packets, the size of the flows and number of separate flow traversing each network segment. Using this information, it can determine which segment is least active and can assign new flows it. For load balancing, this is all the information that is needed. However, in Section 2.4 it was seen that IDSs typically use far more information than that, going as far as to read individual packet body data. In practice, flows can be thought of as abstract representations of data traversing the network. For instance, OpenFlow only requires enough information to identify what flow a packet belongs to, exclusively from the packet header (IP or MAC source and destination addresses, protocol type (TCP, UDP etc.) and port or service numbers). In addition, the controller will keep track of statistics about the flows, including priority, number of packets and number of bytes transmitted.

When comparing the 10 features listed to the number of features available when training machine a learning based NIDS, the issue quickly becomes apparent. For instance, the NSL-KDD dataset (a research dataset commonly used in the networking community for testing machine learning based NIDS) contains 41 features. This is significantly more than the 9 required fields [37] provided by an OpenFlow SDN controller. Much of Section 2.2.3 showed that IDSs are more reliable when they have more data available. Encryption is reducing the amount of information (preventing reading of packet bodies) and the volume of data itself is reducing the amount of data that can be read (if you have more data to analyse, you have less time to analyse it). In Section 2.3 it was established that deep learning models require large volumes of high-quality data in order to train effectively. If the data quality is low, or if there is not enough of it, then overfitting can occur. In other words, the amount of data available for NIDSs is decreasing, and machine learning based NIDSs, and deep learning based NIDs, need even more data than even traditional signature or statistical analysis NIDSs.

2.6 Datasets

This brings us to datasets more generally. Datasets for intrusion detection have traditionally been focused on providing data for what can be obtained from network packet data, and not what SDN flows provide. This started with the KDD’99 [38], which was produced before SDN was created. Prior to the KDD’99 the IDS research community lacked a highly detailed realistic dataset on which to conduct research. This led to researchers creating their own datasets which causes issues on reproducibility. Additionally, the dataset still contained several issues, such as redundant and repeated records, too many records and records that were too easy to classify. The NSL-KDD [28], [39] dataset sought to correct some of these issues, particularly the number of redundant records and the ease of classification. More recently the dataset has come under criticism for not being realistic, which is a by-product of being more than a decade old, in an environment which changes quickly. Still, the NSL-KDD is one of the de facto datasets for intrusion detection of this kind. As has been stated, the NSL-KDD consists of 41 features, found in Table 2.6-1:

Number	Name	Number	Name
1	Duration	22	Is_guest_login
2	Protocol_type	23	Count
3	Service	24	Srv_count
4	Flag	25	Serror_rate
5	Src_bytes	26	Srv_rerror_rate
6	Dst_bytes	27	Rerror_rate
7	Land	28	Srv_rerror_rate
8	Wrong_fragment	29	Same_srv_rate
9	Urgent	30	Diff_srv_rate
10	Hot	31	Srv_diff_host_rate
11	Num_failed_logins	32	Dst_host_count
12	Logged_in	33	Dst_host_srv_count
13	Num_compromised	34	Dst_home_same_srv_rate
14	Root_shell	35	Dst_host_diff_srv_rate
15	Su_attempted	36	dst_host_same_src_port_rate
16	Num_root	37	dst_host_srv_diff_host_rate
17	Num_file_creations	38	dst_host_serror_rate
18	Num_shells	39	dst_host_srv_serror_rate
19	Num_access_files	40	dst_host_rerror_rate
20	Num_outbound_cmds	41	dst_host_srv_rerror_rate
21	Is_host_login		

Table 2.6-1 – The NSL-KDD features

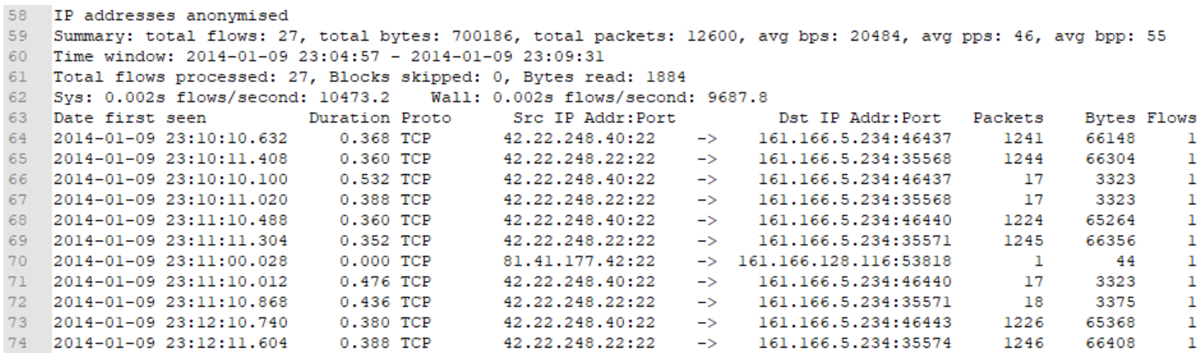
More recently, the UNSW-NB15 dataset has started to become popular. The dataset is designed to be a more modern replacement for the NSL-KDD, featuring more modern attacks with low footprints. This was built using the IXIA PerfectStorm and consists of 49 features that have been extracted from a combination of network packets, Argus and Bro-IDS tools. The features are found in Table 2.6-2

Feature Number	**Feature Name**	**Feature Number**	**Feature Name**	**Feature Number**	**Feature Name**
1	id	**16**	dloss	**31**	response_body_len
2	dur	**17**	sinpkt	**32**	ct_srv_src
3	proto	**18**	dinpkt	**33**	ct_state_ttl
4	service	**19**	sjit	**34**	ct_dst_ltm
5	state	**20**	djit	**35**	ct_src_dport_ltm
6	spkts	**21**	swin	**36**	ct_dst_sport_ltm
7	dpkts	**22**	stcpb	**37**	ct_dst_src_ltm
8	sbytes	**23**	dtcpb	**38**	is_ftp_login
9	dbytes	**24**	dwin	**39**	ct_ftp_cmd
10	rate	**25**	tcprtt	**40**	ct_flw_http_mthd
11	sttl	**26**	synack	**41**	ct_src_ltm
12	dttl	**27**	ackdat	**42**	ct_srv_dst
13	sload	**28**	smean	**43**	is_sm_ips_ports
14	dload	**29**	dmean		
15	sloss	**30**	trans_depth		

Table 2.6-2 – The UNSW-NB15 features

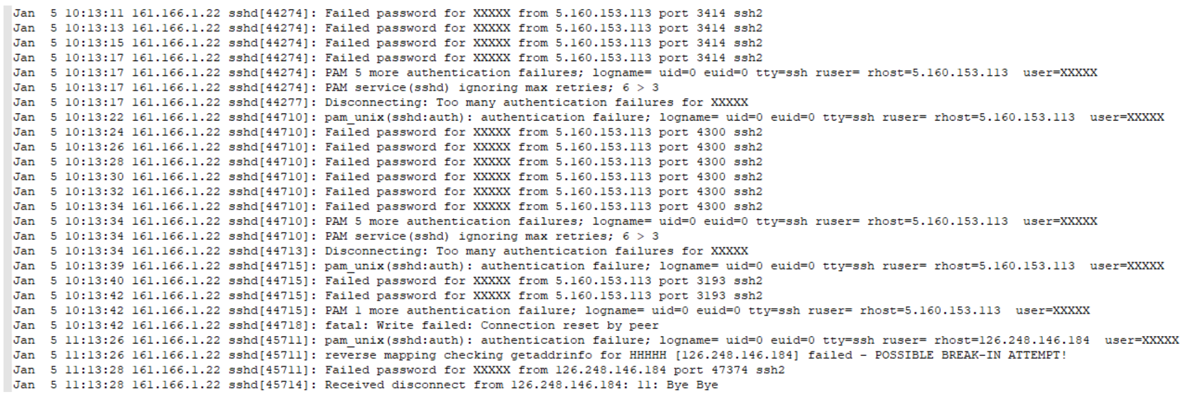
Many of these features are similar to those found in the NSL-KDD, including successful logins, and commands used, which are features that are only available from logs if the network packets are encrypted end-to-end.

Specific to this work is the NetFlow/IPFIX SSH compromise detection dataset from the University of Twenté [40]. This dataset was originally designed to test a method of SSH compromise detection using the NetFlow/IPFIX SDN flows. Of interest to us is that it consists of both flows and logs from the servers, and that it consists of real-world data. The flows were collected from edge routers and contains all SSH traffic entering and leaving the campus network, while logs were collected from workstations and servers that have a publicly accessible daemon. The data comprise a period of one month, collected in January and February of 2014. The fact that the dataset specifically has log files that have been created as a direct result of flows is important for this work, since this allows us to monitor the impact of additional log data being added to the network flows in a real world environment. The data is not pre-processed, which would make comparisons to other works more difficult since the data will inevitably be split into different training and test sets. Example of the dataset are shown in Figure 2.6-1 and Figure 2.6-2:



```
IP addresses anonymised
Summary: total flows: 27, total bytes: 700186, total packets: 12600, avg bps: 20484, avg pps: 46, avg bpp: 55
Time window: 2014-01-09 23:04:57 - 2014-01-09 23:09:31
Total flows processed: 27, Blocks skipped: 0, Bytes read: 1884
Sys: 0.002s flows/second: 10473.2    Wall: 0.002s flows/second: 9687.8
Date first seen          Duration Proto      Src IP Addr:Port          Dst IP Addr:Port   Packets    Bytes Flows
2014-01-09 23:10:10.632     0.368 TCP       42.22.248.40:22    ->    161.166.5.234:46437     1241    66148     1
2014-01-09 23:10:11.408     0.360 TCP       42.22.248.22:22    ->    161.166.5.234:35568     1244    66304     1
2014-01-09 23:10:10.100     0.532 TCP       42.22.248.40:22    ->    161.166.5.234:46437       17     3323     1
2014-01-09 23:10:11.020     0.388 TCP       42.22.248.22:22    ->    161.166.5.234:35568       17     3323     1
2014-01-09 23:11:10.488     0.360 TCP       42.22.248.40:22    ->    161.166.5.234:46440     1224    65264     1
2014-01-09 23:11:11.304     0.352 TCP       42.22.248.22:22    ->    161.166.5.234:35571     1245    66356     1
2014-01-09 23:11:00.028     0.000 TCP       81.41.177.42:22    ->  161.166.128.116:53818        1       44     1
2014-01-09 23:11:10.012     0.476 TCP       42.22.248.40:22    ->    161.166.5.234:46440       17     3323     1
2014-01-09 23:11:10.868     0.436 TCP       42.22.248.22:22    ->    161.166.5.234:35571       18     3375     1
2014-01-09 23:12:10.740     0.380 TCP       42.22.248.40:22    ->    161.166.5.234:46443     1226    65368     1
2014-01-09 23:12:11.604     0.388 TCP       42.22.248.22:22    ->    161.166.5.234:35574     1246    66408     1
```

Figure 2.6-1 – An extract of the raw flows from the NetFlow/IPFIX dataset



```
Jan  5 10:13:11 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:13 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:15 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:17 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:17 161.166.1.22 sshd[44274]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:17 161.166.1.22 sshd[44274]: PAM service(sshd) ignoring max retries; 6 > 3
Jan  5 10:13:17 161.166.1.22 sshd[44277]: Disconnecting: Too many authentication failures for XXXXX
Jan  5 10:13:22 161.166.1.22 sshd[44710]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:24 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:26 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:28 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:30 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:32 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:34 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:34 161.166.1.22 sshd[44710]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:34 161.166.1.22 sshd[44710]: PAM service(sshd) ignoring max retries; 6 > 3
Jan  5 10:13:34 161.166.1.22 sshd[44713]: Disconnecting: Too many authentication failures for XXXXX
Jan  5 10:13:39 161.166.1.22 sshd[44715]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:40 161.166.1.22 sshd[44715]: Failed password for XXXXX from 5.160.153.113 port 3193 ssh2
Jan  5 10:13:42 161.166.1.22 sshd[44715]: Failed password for XXXXX from 5.160.153.113 port 3193 ssh2
Jan  5 10:13:42 161.166.1.22 sshd[44715]: PAM 1 more authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:42 161.166.1.22 sshd[44718]: fatal: Write failed: Connection reset by peer
Jan  5 11:13:26 161.166.1.22 sshd[45711]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=126.248.146.184  user=XXXXX
Jan  5 11:13:26 161.166.1.22 sshd[45711]: reverse mapping checking getaddrinfo for HHHHH [126.248.146.184] failed - POSSIBLE BREAK-IN ATTEMPT!
Jan  5 11:13:28 161.166.1.22 sshd[45711]: Failed password for XXXXX from 126.248.146.184 port 47374 ssh2
Jan  5 11:13:28 161.166.1.22 sshd[45714]: Received disconnect from 126.248.146.184: 11: Bye Bye
```

Figure 2.6-2 – An extract of the logs from the NetFlow/IPFIX dataset

3 Related Work

In this chapter shall provide an overview of the cutting-edge and relevant existing pertinent to this project.

3.1 Deep learning within Intrusion Detection

This subsection will provide an overview of cutting-edge research related to deep learning within intrusion detection. This will come in three sub-sections, Datasets, the models being used, and data preparation.

3.1.1 Datasets

As outlined in Section 2.3, deep learning has many advantages when it comes to addressing the challenges within modern network security. Numerous works and datasets have been proposed over the years, starting with the KDD'99 dataset released in 1999. This dataset came into criticism for several issues, including [39]:

- Redundant records, which can lead algorithms to bias more common data
- An unrealistic attack variance and background noise, which does not match or emulate real network conditions
- Too many records, which has led to researchers using different subsets of the data, meaning that works using the same data set cannot necessarily be directly compared
- An attacker focused taxonomy of attacks, which is not how an IDS sitting on the edge of an attacked network would see data

In response to these criticisms, an alternative dataset was created, the NSL-KDD+ dataset which was significantly more challenging. However, the realism of the dataset is still an issue, especially as networks have changed significantly since it was devised [28], [31].

The UNSW-NB15 dataset [30], [31] was developed to be a more modern alternative, with attacks that mimic more closely the low footprint attacks found today. This dataset contains 47 features, which have been extracted using a combination of Argus, Bro-IDS and twelve algorithms which cover characteristics of network packets. The dataset comes in two forms, a collection of four CSV files which contain all 2,540,044 records, and a partition of that dataset, separated into training and testing. The dataset contains 9 attack families, with Normal being a 10th classification. These are as follows:

• **Fuzzers** – Attempting to cause a program or network suspend by feeding randomly generated data

• **Analysis** – Different attack of port scan, spam and html file penetrations

• **Backdoors** – System security mechanisms are bypassed to stealthily access a computer or its data

• **DoS** – An attempt to make a server or resource unavailable to users by temporarily interrupting or suspending the services of a host

• **Exploits** – Targeting a known security problem within an operating system or piece of software

- **Generic** – A technique that works against block-ciphers without consideration about the structure of the block-cipher
- **Reconnaissance** – Contains strikes that can simulate attacks that can be used to gather information
- **Shellcode** – A small piece of code used as a payload in exploiting a vulnerability
- **Worms** – A self-replicating program that uses a network to spread, relying on security failures to do so.

The training/test split can be more difficult for several reasons. To start there is simply less data. As stated in Section 2.3, machine learning models gain better results when exposed to more data. This is for a few reasons. More data can aid in reducing overfitting and it allows the model to learn more examples of anomalous data. Importantly there are examples of attacks in the smaller testing data that do not appear in the training data. This can result in a difference of accuracies for different methods. For example, works such as [41], [42], [43], [44] achieve an average accuracy of 89.56%, precision of 87.13, recall of 90.42 and F-Score of 88.58 while using the pre-prepared dataset. Compare this to works such as [45] (which used the full CSV files, and split the dataset into train/test sets separately), where accuracy is over 90% and it becomes clear that these factors do play a part.

3.1.2 The Models Used

Recently convolutional models have started to become popular within the research space for supervised deep learning. Originally designed as a method to extract complex features from an image [46], this has since been expanded to several different areas including speech recognition, sentiment analysis and medical diagnostics. Krizhevsky *et al.* [46] note that the main advantages of CNNs over standard feed forward networks (with similar sized models) are that they have fewer connections and parameters. This makes CNNs easier and quicker to train, while theoretically best classification performance will likely only be slightly worse. They create a model that achieves a winning top-5 test error rate of 15.3% in the ILSVRC-2012 competition, compared to 26.2% achieved by the second-best entry.

Azizjon *et al.* show how the transformation to network datasets can work in [41]. Using a 1-dimensional CNN, they achieve accuracy of 91.2%, precision of 87.53%, recall of 95.15% and an F1 of 91.59% on the pre-prepared UNSW-NB15 test set after balancing the data using a random oversampling method. Their model consists of three convolutional layers in total, all with the same filters (32), stride (1) and kernel size (5). The data is passed through the first two convolutional layers before moving to a max pooling layer, and then the final convolutional layer. Batch normalization is performed, as is drop out and a final two fully connected layers. Min/max normalization is used on some features such as duration, sbytes and dbytes, where the values can proceed outside the range 0-1. They also test models with only a single convolutional layer, and two convolutional layers, as well as unbalanced data. They found that balancing the data increases accuracy and precision by more than 5% in all cases, whereas recall actually falls slightly (less than 3%). This results in an F-score topping 90% for all instances for the balanced data, while falling short of that with the unbalanced data. This shows the importance of balancing the dataset; however, the method of the balancing is also important. Through oversampling the data, they effectively add more data to train on, which as discussed in Section 2.3 aids in reducing overfitting. As such, just from this work alone it cannot be ascertained whether it is the dataset balancing or additional data that helps increase accuracy and precision, or a combination of both. Increasing the number of layers has a similar effect, improving accuracy and precision (both for balanced and unbalanced data) whilst slightly decreasing recall. This results in a slight net increase in F1-score. This is not surprising, as other models have found similar advantages when increasing the number of layers or decreasing the kernel size [47]. As such, it is becoming increasing common to have models with kernel sizes of only 5, 3 or even 2.

When comparing Azizjon's work to Al-Zewairi et al. [45], then the issues explained in Section 3.1.1 surrounding accuracy rates for different versions of the same datasets become more apparent. Using a DNN with five fully connected layers, each with ten neurons they achieve an accuracy of 98.99%, and a FAR of only 0.56%. However, they create their own training, test, and evaluate sets from the original four CSV files from the UNSW-NB15 dataset. As there is extra data to learn from, the model employed by Al-Zewairi can afford to be deeper than that of Azizjon, as overfitting is not as likely. Additionally, as the training, test and evaluation sets were randomly generated, it is likely that there were some representations of every attack within both the training and test sets. This is not true of the prepared datasets, which contain examples of attacks in the testing set that do not appear in the training set. The lack of some examples within the training set of the pre-prepared datasets is an attempt to replicate 0-day threats, the expectation being that the models should be able to detect the attacks even though they have not previously been trained on them. This, along with the difference in dataset size, means care should be taken when comparing the pre-prepared dataset with the raw CSV files.

Further questioning whether the dataset needs to be balanced (or whether simply adding more data is sufficient) is Kim et al [48]. Using the KDD'99 dataset, they show that increasing the proportion of attack packet data in the training set does not have a significant effect on accuracy, ranging from 98.95% accuracy with 30% attack packet data, to 99.08% accuracy at 50%, 70% and 80% attack packet data. However, the false alarm rate does change, peaking at 0.47% with 90% attack packet training data (from 0.01% at 10% attack packet data). The detection rate also mimicked this pattern of the FAR. The minimum recall achieved was 99.19% (with 30% attack data, not the 10% as might be assumed), and the maximum was 99.81% (with both 80% and 90% of attack data). This shows that balancing the dataset may not help accuracy, but it will affect precision and recall.

An Autoencoder network's desired output is the same as its input. So, for input *i*, and output *o*, the desired output would be *i*=*o*. However, intermediate layers of the network are smaller than the input and output, meaning that the network is forced to encode a representation of the original input with reduced dimensionality that compliments the layer size [19]. The goal is to reduce noise or unneeded features in an automatic manner. Traditionally, this is performed by creating a symmetrical model with output containing the same number of nodes as the input, and hidden layers decreasing and increasing in a symmetrical manner. So, for a model with 3 hidden layers with the size being decreased by 2 in total, and an input size of x, the layer sizes would be x, x-1, x-2, x-1 and output of x. An example of this model is given in Figure 3.1-1.

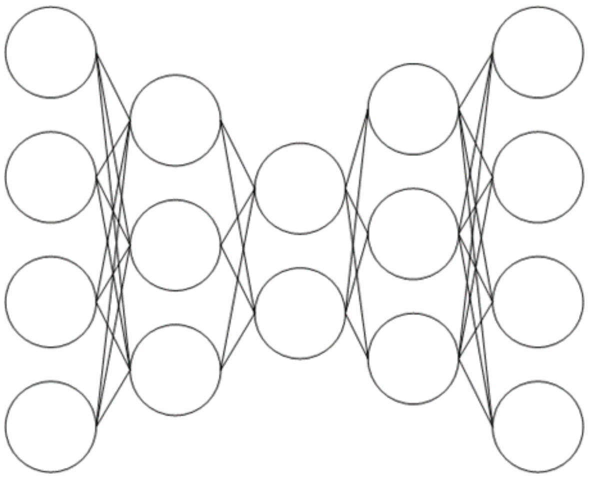


Figure 3.1-1 – An example of a small autoencoder structure

This means that an Autoencoder network is effectively two separate networks, an encoder and a decoder network. The encoder network maps the inputs into a smaller representation. This can be represented with:

$$N = S_1(Wx + b_1) \qquad \textit{Equation 3.1-1}$$

Here, *W* is the weight, *b* is the bias vector, *S* is the activation function and *N* is the latent representation.

The decoder network can likewise be represented as:

$$O = S_{O-1}(Wx + b_{O-1}) \qquad \textit{Equation 3.1-2}$$

Here, *O* is the output.

Typically, a network like this is combined with a shallow classifier to get the final classification. For example, Al-Qatf *et al.* [49] utilise a sparse Autoencoder model with a SVM classifier in order to classify both the NSL-KDD and KDD'99 datasets. They use a sigmoid activation for the Autoencoder network. They achieve 80.48% accuracy on the NSL-KDD test set, as well as 93.96% on the full KDD '99 dataset. This again shows the effect of having more data available. As mentioned, the NSL-KDD dataset is a version that corrects some of the issues in the original KDD'99 dataset, and the difference in accuracy between the two datasets on the same model highlights how much of an effect this had.

Alternatively, Andresini *et al.* [50] utilise a combination of 2 Autoencoder networks (one trained on normal data and one trained on attack data only), and then perform a final classification based upon a CNN and fully connected ANN. The Autoencoders have 3 layers of sizes 40, 10, 40 and use mean squared error as the loss function. ReLu is used as the activation function for the hidden layer and linear activation used for the output. The convolutional layer uses a 1x1 kernel size with 64 filters. CNNs with a 1x1 kernel size were first used like this by Szegedy *et al.* [34] in submission for the ImageNet Large-Scale Visual Recognition Challenge 2014. From here, the data is flattened (to size 1280) and moves through 3 fully connected layers of sizes 320, 160 and 2 for the final classification. ReLu is again used for all layer activation except for the final classification layer which is softmax. They obtain accuracy of 92.49% on the KDD'99 dataset, and 93.40% on the UNSW-NB15 Test dataset. While the performance on the KDD'99 dataset could be improved, the accuracy on the UNSW-NB15 compares favourably to the other works listed, the next highest managing only 91.2% accuracy.

3.1.3 Data Preparation

Data preparation is clearly important as a model can only be as good as the data used to train it on. If poor quality training data is used that does not represent the end use of the model, then the model will not accurately predict the outputs. However, even beyond gaining the initial high-quality training data, there are additional tools and techniques that can be used to prepare the data so that accuracy can be increased.

Balancing the data is often seen as being important. In Section 3.1.2, Kim *et al.* [48] was referred to, who used various proportions of attack and benign data and showed that recall and precision can be affected. The same kind of thinking was behind Andresini *et al.*'s paper when they used two Autoencoder networks, one trained solely on attack data. Kim *et al.* use oversampling to achieve the desired attack data proportions, however other methods of data balancing are available. For example, Phetlasy et al. [51] show SMOTE can increase accuracy, precision and recall when using a number of classifiers, including J48, Multilayer Perceptron (MLP) and K-Nearest Neighbour. When using J48 alone, recall increased from 68.16% to 84.6%, precision dropped slightly from 97.27% to 97.02%, while accuracy overall increased from 80.69% to 89.95% on the NSL-KDD testing set. Using a combination of both the MLP and J48 saw results improve further, with recall moving from 77.13% to 91.48% and precision from 92.13% to 96.57%. Finally, the combination of all 3 classifiers saw recall increase from 81.36% to 92.01% and precision increase from 91.95% to 95.61%. Of note, is that in all instances apart from J48 being used by itself, both precision and recall increased. This is in contrast to Kim *et al.* who showed that just the proportion of attack data will negatively affect recall while positively affect precision, or vice versa.

Alternatively, as part of their work Labonne *et al.* [52] compare a number of different balancing techniques, including SMOTE, ADASYN, SMOTEENN, SMOTE Tomek links, random undersampler and random oversampler. Results when comparing the Area Under Curve – Receiver Operating Characteristics curve (AUC-ROC) are in Table 3.1-1.

Sampling Technique	AUC-ROC
No Sampling	94.53%
SMOTE	95.33%
SVM SMOTE	96.93%
ADASYN	94.93%
SMOTEENN	95.61%
Tomek Links	96.02%
Random Undersampler	96.24%

Random Oversampler	95.95%

Table 3.1-1 – Results found in [52] *when comparing different sampling methods*

As can be seen in Table 3.1-1 all methods of balancing increase the AUC-ROC score, with SMOTE based methods performing the best. It should be noted that the AUC-ROC score may not be the best metric to use for this kind of study. While it provides a method to determine how well a model can predict a class, it does not help us to see how the false positives or false negatives are being affected.

An alternative to Kim et al. comes from Dong *et al.* [53]. Their work compares deep learning methods of classifying network data, to traditional methods. They do this by showing the precision results (rather than accuracy) for SVM-RBM, SVM, Naïve Bayes and C4.5. The deep SVM-RBM model outperforms all the other models on the KDD'99 data set and the paper highlights the difficulty of detecting U2R and R2L attacks within the dataset due to the small volume of attacks within the dataset. Precision for all results peaks at around 0.85, whereas for U2R attacks precision peaks at around 0.4, with the SVM-RBM still achieving the highest precision results. The authors also used SMOTE to solve some of the issues with imbalance in the data set. Showing results for precision both with and without using SMOTE on U2R attacks, accuracy peaked at 0.56 with SMOTE and 0.45 without.

Feature selection itself is also important, as explained by Akashdeep, *et al.* [54]. They observe that removing features that do not contribute (or have a very low contribution) to detecting attacks can aid in increasing accuracy and decreasing computation time. The authors use the KDD`99 data set and rank the features into two tables, one table based upon information gain (IG) and the other correlation (CR). After this ranking, the tables are merged (a Union Join) and the top 25 features are selected. There is a direct (and deliberate) contradiction between IG and CR, as the higher the IG, the lower the CR. The method does show an improvement in accuracy and FPR over other methods of feature selection mentioned in the paper.

Praneeth *et al.* [55] specifically look at Principal Component Analysis (PCA) as a method of feature reduction for the KDD'99 dataset. They compare accuracy against the number of components using a SVM classifier (including RBF, Linear and Polynomial kernels). They find that accuracy is matched with and without PCA at 25 components (or features) for the linear kernel, 20 for the RBF kernel, and that accuracy is consistently higher using PCA for the polynomial kernel. The main advantage of PCA comes with the amount of time to process, which is smaller for all kernels with PCA, and follows the pattern you would expect (more components taking longer to process). At worst they find use of PCA allows for accuracy matching that without using it, but allows for quicker processing, something that becomes important in a networked environment.

Bahrololum *et al.* [56] compare decision tree, particle swarm optimisation (PSO) and Flexible Neural Tree (FNT) as methods to select features. Through analysis of the KDD'99 dataset, they determine that there are several features that do not participate in the final decision, and therefore can be removed with no loss of accuracy. However, different classifiers do prioritise different features. For example, decision tree finds that src_bytes are important for all classification tasks, whereas it doesn't feature as important for any class with FNT. It can be concluded that there is no one "best" feature reduction technique, and which method is used will need to be evaluated on a case by case basis. However, feature reduction should be considered in any work where processing time is important.

More recently, Liu and Chung [57] compare a number of feature reduction techniques, including stacked Autoencoders, decision trees, neural networks and SVM. The process is to train on the dataset and obtain a baseline accuracy. Then to remove features and check the extent of the accuracy change. This process is automated using grid search and 10-fold validation. They come to a number of conclusions, including highly correlated features may lead to overfitting, the feature extraction step is important since it re-represents the original feature set in a way that is more sensitive to later classification steps and that weaker separate features can become stronger when combined with other weaker features

3.2 SDN as a security solution

Shin et al. [58] identify four characteristics that could aid SDN in improving network security. These areas include:

- Dynamic flow control
- Network-Wide Visibility with Centralised Control
- Network Programmability
- Simplified Data Plane

They give firewalls as an example of the advantage of dynamic flow control. Upon receiving a new packet, an SDN-enabled switch forwards the information about the packet to the controller. The controller can forward the data to a firewall application, which can determine if the packet passes the security policies. Upon receiving the pass or fail response, the controller can forward instructions on how to manage the packet to the switch, which will add the instructions to its flow table. The advantages of this beyond a regular firewall are that any SDN-enabled switch can become a firewall, reducing the need for specialist hardware, and increasing the resilience of the network. An example given for network-wide visibility advantages, is the detection of particular DDoS attacks initiated by bots, where the ability to monitor traffic from multiple unrelated devices, allows for the detection of malicious activity that may otherwise have remained undetected. For example, DDoS and flood attacks frequently transmit small volumes of data towards the target, none of which individually would be dangerous, and so not detected. The benefit of the network programmability revolves around middle boxes. It is commonly not easy to change or modify a middle box's security functions, but it is also difficult to predict exactly which security functions are required. The programmable nature of SDNs allows an administrator to create their own security functions as a supplement to the middle boxes, using scripting languages such as FRESCO [59]. They offer the now comparative simplicity of the data plane as an advantage to security since more security functions can now be implemented in an easier manner.

Yoon *et al.* [60] show the feasibility of implementing some of these security functions to the control layer of a SDN. They implement a firewall, NIDS, NIPS, an anomaly detector, stateful firewalls and a reflector network. Among the advantages of the NIDS and NIPS they implement, they state the lack of need for middle boxes and the lack of need to carefully place the devices. These advantages pass over to other security functions that were implemented as well. The example given being a distributed firewall, capable of managing traffic network wide. They also state the primary disadvantage as being performance. In-line security functions such as NIDS and NIPS need to analyse traffic in real time, so as to not slow all other network traffic, and this becomes more difficult as you add more data. Specifically for anomaly detection, the authors state the advantages being the comparative ease of collecting information, however the disadvantage being one that has been mentioned in Section 2.5, the loss of other information that is frequently used (e.g. TCP session information).

Despite these advantages, SDN does come with additional security issues. Scott-Hayward *et al.* [61] discuss some of these issues, which they split into six areas:

- **Unauthorised access** to either the controller or unauthenticated applications.
- **Data leakage** in terms of flow rule discovery or forwarding policy discovery
- **Data modification** through flow rule modification to modify packets
- **Malicious applications** creating fraudulent rules or controller hijacking
- **DoS** – through switch-controller flooding or switch flow table flooding
- **Configuration issues** such as lack of authentication techniques or policy enforcement.

However, they also find potential advantages, such as a “moving target defence” where the actual IP of machines is hidden and replaced with a virtual IP. While conceptually similar to a NAT, the idea is a SDN controller would hold a list of randomly changing virtual IP addresses that are mapped to a specific physical IP address. Named hosts are available via the virtual IP, but the real IP is only accessible by authorised entities [62]. It is found this solution can help invalidate the results of information scanners. More recently, Scott-Hayward *et al.* have expanded on the initial paper, with research examples given to address some of the security issues that have been presented. However, they note that although SDN has matured since the initial paper, there is still much work to be done. Strong themes they identify include projection of potential security issues and automated response for quick reaction to network threats.

One example of the potential security issues is given by Sin and Gu [63] who propose a SDN scanner and DoS method. They successfully fingerprint a SDN using specially crafted IP packets and measure the response times. The theory is that the first packet to traverse a SDN will take longer than the second. This is because the first packet will need to be referred to the controller, and rule generated and transmitted back to the switch. Alternatively, the second packet will still have the new rule in place. Upon identifying an SDN, the attacker could then proceed to disrupt the network through sending multiple new crafted packets through the target network, using up network resources. This would essentially be a DoS that targets a SDN controller and is unique to SDN environments.

An example of how a SDN can improve network security comes from Shin *et al.* [64][58], who through the use of actuating triggers determine a method of responding more quickly to standard DoS attacks within SDN environments. They also propose an extension of the OpenFlow data plane called connection migration, which reduces the amount of data needing to be passed to the controller, thus reducing the risk of the Southbound interface becoming saturated. They claim these two measures would likely be enough to mitigate or prevent the malicious activity proposed by [63].

3.3 SDN and Deep Learning within Intrusion Detection

Other authors can be found highlighting the contradiction between SDN and deep learning made in Section 2.5

For example, Tang *et al.* [65] propose a fully-connected DNN using an six-dimensional input layer comprised of common features found within both the NSL-KDD dataset and OpenFlow. The authors attempt binary classification and achieve an accuracy of 75.75%, precision of 83%, recall of 76% and an overall F-score of 75% on the NSL-KDD test set. This should be contrasted with the works noted in Section 3.1.2, for example, Al-Qatf *et al.* [49] utilise a sparse Autoencoder model (unsupervised) to achieve 80.48% accuracy on the NSL-KDD test set, or Azizjon et al. [2] using a 1-dimensional CNN achieve an accuracy of 91.2%, precision of 87.53%, recall of 95.15% and an F1 of 91.59%. Considering only six features are used, and that these are not even the most useful features (according to Akashdeep, *et al,* Praneeth *et al* or Bahrololum *et al,* in Section 3.1.3), and that the DNN does achieve higher accuracy then other tested shallow methods (Naïve Bayes at 45%, SVM at 70.9% and decision tree at 74%), the result could be considered good. However, the authors acknowledge that it would still not be good enough to be adopted into a commercial solution, or to be used as an alternative to signature based IDSs. It should be noted that the default OpenFlow specification does call for a larger flow table than the six common features included in this work. This means while the work is useful in highlighting the issue, further work with more realistic datasets is needed.

More evidence of the contradiction between SDN and deep learning is provided by Janarthanan and Zargari [30]. Using several attribute selectors, including the CfsSubsetEval, InfoGainAttibuteEval and Ranker methods in Weka, they use several machine learning algorithms to evaluate binary classification performance. The goal is to determine which features within the UNSW-NB15 dataset are irrelevant or redundant, so that these can be omitted, and the curse of dimensionality reduced, resulting in less training and testing time. They find that the most important features included service (e.g. http, ftp etc), sbytes (number of bytes sent from source), sttl (source to destination time to live), smean (mean of packet size transmitted by the source) and ct_dst_sport_ltm (No. of rows of the same destination IP and source port in 100 rows). Of these, none is a required feature within the OpenFlow specification [37]. Service could be inferred from a combination of protocol type (TCP, UDP etc.) and port number, though port number itself is not a required feature. Sbytes is an optional field within the specification, which would likely be used by many administrators, and smean could likely be inferred as it can be determined through a combination of the optional fields received bytes and packet count (both of which again, an administrator is likely to use). However, sttl would not be configurable at all since time to live only really has a purpose on the data layer. ct_dst_sport_ltm exists within the derived category of features within the UNSW-NB15, which means it is a feature that has been added after the dataset has been generated. This kind of feature could exist, but would require more processing in general, since it will change multiple records with the addition of a single new flow. In practice, it may be better to use a feature that compares the number of connections to the same destination address and port within the last *n* minutes. This would be simpler to compute, as it would only be updated once (when the flow is generated), however, this is not a feature that exists within the UNSW-NB15 dataset.

Niyaz et al. [66] take a different approach, instead of limiting themselves to a few common features like Tang *et al.* [65], they create their own dataset. This is achieved through creating their own network and performing various DDoS attacks from ten different attackers towards 5 different hosts. To collect benign data, they use a home wireless network with up to 12 devices connected, though not concurrently. After collecting data from both networks, they create their SDN and replay the data collected over it using TCPReplay. Using a deep sparse autoencoder model, they achieve 95.65% accuracy for binary classification. It should be noted that whilst the authors do use a SDN implementation based on OpenFlow, it is still a custom design, with data being recorded that would not typically be recorded by an SDN. For instance, the authors use the flags that are set in TCP flows as features, which is not an optional or required table entry within the OpenFlow framework [37]. This could lead to scalability issues on larger networks (more data being collected and analysed), though the authors suggest this could be overcome using a hybrid approach with tools such as sFlow or individual packet capture. The details of such a system are not provided. It should also be noted that the focus was on DDoS attacks, and not the broad range of attacks that can be found within a dataset such as the NSL-KDD or UNSW-NB15.

This is an issue Yoon *et al.* [60] also mentioned in their work integrating IDS and IPS into SDN. As mentioned in Chapter 3.2, they do find advantages for IDS and IPS within SDN, namely that being that the middle boxes no longer need to be placed in-line. Upon receiving a flow table miss, the controller can order the packet to be redirected to IDS component for analysis, and then future benign packets can be forwarded onto their final location. This has the advantage of reducing the processing overhead of the network. Packets will only be analysed when it is unknown if they are benign or malicious, and the IDS will not be a bottleneck for the network. However, one of the disadvantages they mention is the limited access to packet data that comes from a network controller. Ideally, the IDS/IPS would just be using controller data to perform this analysis, as redirecting packets to the IDS introduces additional overheads. However, the comparative lack of data within the controller, means that the authors' (non-machine learning-based) solution would be hampered.

3.4 Taxonomies

After identifying a flow is malicious, a suitable mitigation measure needs to be determined. It has been stated that SDN is suited to carrying out automated mitigations of attacks, however what kind of mitigation needs to be discussed. Typically, a system will identify an attack by name, and list possible counter measures, allowing the administrator to choose something suitable, or it will just block the flow. This is not appropriate for the proposed system since part of the goal is to mitigate even 0-day attacks which do not have prepared counter measures. This subsection therefore examines taxonomies which are designed to identify a mitigation (unlike most which are designed to identify an attack).

Souissi [67] shows that the system being proposed could theoretically work with “A novel response-oriented attack classification”. Souissi proposes a taxonomy consisting of the classes source, attack vector, target, and impact. Additionally, a matching defence mechanism is proposed that varies depending upon the parameters found in the taxonomy. While the system seems logical, it is tested on only two test cases, and so more study should be performed to ensure the system operates well in practice.

A similar paper from Fu *et al.* [68], this time using the dimensions of Attack plane, Vector, Target and Effect. Effect is functionally equivalent to Impact from Souissi’s [67] paper, however only uses system damaging or resource occupying (verses the use of DoS, Access privilege, Harm implementation, information disclosure or no result from Souissi). Fu *et al.* do not include an access privilege option, as it is believed this can fall within the remit of system damage. It should be noted that the focus is on routing hardware, not a full network. We do not share this belief, as routers and basic switches can still be susceptible to access privilege attacks that do not perform harm to the switch or router themselves. For instance, the Mirai botnet was formed using primarily access privilege attacks, and most users of an infected router did not realise there was an issue until the botnet was leveraged. The attack plane, vector and target also have differences; however, these are explained by differences in the goals of the taxonomy. For instance, while Souissi has local or distant for attack source, Fu et al. have data plane, control plane or management plane, mimicking the three layers of a SDN network.

Wu *et al.* [69] feel that only three dimensions are needed, source, technique and response. Like Souissi, Wu *et al.* propose a taxonomy based on responding to a network event, however, they do not address vector at all. For a response-orientated taxonomy, vector should be crucial, as without it, the countermeasures that involve restricting access to the attacker become more difficult. Apart from this, the solution is flexible and able to identify attacks and formulate responses.

Simmons *et al.* [70] show a different approach in AVOIDIT: A Cyber Attack Taxonomy. In it, they state that to be a complete and useful taxonomy then several aspects must be met. These include that it must be built on previous works, that an attack must only be classified into one class, it must be clear and concise, it must be exhaustive, it must be unambiguous, it must be repeatable (i.e. the same attacks should be classified the same way each time), the terms should be well defined and that it should be useful. Some of these are related, for example, unambiguous and repeatable. If a term is ambiguous then there is a higher probability of some attacks not being classified the same way multiple times (as the class is open to interpretation). Applying these requirements, they devised AVOIDIT, which contains 5 classes consisting of Attack Vector, Operational Impact, Defence, Informational Impact and Target. Most of the classes are again broadly like others, but the inclusion of defence is an interesting choice. This consists of both Mitigation and Remediation and is designed to give a defender an appropriate starting point to defend against the attack. However, the authors state that this area could potentially be the weakest part of the taxonomy, with it needing more research to provide an exhaustive list of defence strategies.

3.5 Summary and Discussion

It has been determined that there are a number of challenges facing this project, including:

- A lack of suitable datasets
- Uncertainty about the need or best way to balance data
- Potential security issues around SDN itself
- A significant reduction in accuracy when models are being trained only on the features available within SDN flows

While the UNSW-NB15 is a more modern dataset and includes more modern attacks, it still includes a lot of data that is not readily available to SDN networks. Another dataset must be found in order to confirm any results. Additionally, the prepared dataset should be used, as using the raw dataset could lead to issues like those found in the earlier versions of the KDD'99 dataset. Those being that manually separating training, testing and evaluation sets can change the distribution of attacks making it easier or harder. The prepared dataset avoids these issues and allows us to compare any results directly (though with the knowledge other authors were using all features available). Some experimentation will also need to be done into balancing the data. It becomes obvious there is no "one size fits all" solution to data balancing, and that perhaps data does not need to be completely balanced at all. Instead, there may only need to be enough examples of all attacks for generalisations about other attacks to be made. The major issue surrounding the project is the lack of data available to teach deep models. Section 3.3 outlined multiple papers which did not achieve results comparable to those of state-of-the-art papers using the whole dataset, or only did so by adding excess data back in and causing potential issues with scalability.

Section 2.5 highlighted the contradiction that is apparent between SDN and deep learning for the purposes of intrusion detection. SDNs operate using extremely limited datasets, deep learning models, by contrast require large volumes of data with many features. Past works have achieved high accuracy rates using creating a custom SDN solution and generating more data points (Section 3.3, notably [66]). This solution could however lead to issues of scalability and is not a solution that would be compatible with pre-existing SDN installations. However, they do show that there is a benefit to more data being added to standard SDN flows.

Additional data could come from several sources. Past papers have shown the benefit of using features that are effectively just basic features applied over time, or some other measure, such as the count in the NSL-KDD dataset (Number of connections to the same host as the current connection in the past two seconds). Indeed, some of these features are the most useful [54], and often also have the advantage of being easy to calculate using SDN flow characteristics. For the count, no packet body data is required, or even any header data beyond the destination of the flow. Every two seconds the flow table can be examined, and the number of connections can be counted. Additionally, there are sources of data outside of network data alone. Log data is a valuable source of information, and often a resource companies already manage using automated log servers. Again, some of the most valuable features are ones that are available through this resource [29], [71]. Going back to the NSL-KDD dataset, the root_shell (whether root was obtained), su_attempted (count of number of attempts), num_shells (number of shells opened) and num_access_files (number of files accessed) are all found to be valuable, and are all features that would be readily available within logs. This is not surprising, the combination of root_shell and su_attempted gives a clear indication of whether a service may be under a password attack. If root is being requested many times, with few (or no) successes, it is a clear sign that the service may be under attack, as a forgetful administrator would likely move to regain access to the service through a more legitimate route then guessing many times.

There have been attempts to merge logs with network data for the purposes of intrusion detection. Typically referred to as Hybrid NIDS [72][73], these systems generally attempted to use different methods to detect anomalous activity in logs or the network data separately, and registered it as an intrusion attempt depending upon a ruleset (e.g. If either system detects anomalous activity, register an intrusion, or If the network system registers an intrusion but the logging system doesn't, register it as benign, which may help with false positives). Few researchers have attempted to incorporate logs into a NIDS to improve network anomaly detection specifically. This is likely because traditionally this kind of merging has been difficult to achieve. For more traditional systems, it would require rewriting all of the signatures that have been generated over the years (for both network-based as well as host-based intrusions), whereas statistical measures would gain little from such a merger and the added information it provides. For self-learning systems there was also the difficulty of how you incorporate both log and network data. Often, network data is only kept for a short amount of time (the time needed to manage it), and so you would need to either copy all the network data to another location in order to have it analysed (a system for which there wasn't a simple or standard solution) or attempt to do the analysis on the routers handling the data (which would lead to issues with computational resources). SDN can solve this problem for us.

A common feature of all SDN designs is the use of a northbound interface, which can be used to receive information from the controller and issue instructions back. Without a northbound interface, a system cannot be considered an SDN, as there is no way to issue the commands that define the idea. Typically, the northbound interface can also be accessed through software, allowing automation of network resources dependent upon network load. As such, it is possible that an intrusion system could sit on this northbound interface, getting information from the SDN controller about the currently active network flows. In addition, if this were to sit on separate hardware from the main controller, it is entirely possible it could also receive data from log servers in a similar manner. A large database of flows, both past and present, could be built, which includes log data as well as flow data and is regularly analysed for signs of anomalous activity.

This configuration process comes with an additional advantage. Since the IDS is sitting on the northbound interface, it would be possible to alter flows based upon network conditions or anomalous activity. This is the purpose of the northbound interface after all. Once the IDS has found malicious activity, then a system could be put in place to mitigate the risk posed through manipulation of network traffic. This could be similar to a Network Intrusion Prevention System (NIPS), however there would be some important differences.

Network Intrusion Prevention Systems (NIPSs) are typically hardware-based devices that seek to prevent attacks by blocking traffic (possibly through the use of reconfiguring firewalls), resetting connections or simply dropping packets. They typically work inline and are placed just before important infrastructure that needs to be protected. The process of manipulating SDN flows to attempt to stop malicious activity does not fit directly into this paradigm, particularly as the device will not be sitting inline (Section 3.3, notably [60]). As such, proposed is that an intrusion detection and mitigation system that sits on the northbound interface of a SDN be called a Network Intrusion Mitigation System (NIMS).

A NIPS will prevent attacks through blocking or dropping traffic, the NIMS does not have this restriction. Through manipulating flows, a NIMS could redirect an attack to a honeypot, for example, with the attacker still believing they are targeting the original target. This could allow for gaining potentially important intelligence on the attacker (what the motivation or end goal is for example). If the attack is of a low threat, then it could be highlighted to the administrator, but no further action be taken, or if mitigating the attack could potentially produce undesirable results, then the same action could be taken. While dropping or blocking certain flows is still something that a NIMS could perform, the goal would be focussed more on mitigating the effects rather than preventing the attack entirely.

The advantages for this approach become clearer when you examine some of the most common threats facing businesses (Section 1 [8][7]). Living off the land attacks, malware propagation using exploits, password attacks, and supply chain attacks all have one thing in common. They use software and tools that are already commonplace within business networked environments to propagate and are therefore more difficult to detect and stop. Deep learning can provide the means to spot the attacks, but the means to stop them requires both more finesse, and a more holistic view of the network, than is possible with traditional NIPS. At best, a NIPS would only ever be able to protect the service it has been placed immediately in front of. Even then, without SDN to aid in the identification of exactly which flows are malicious, the options would be to either perform a DoS attack against any compromised machines (through blocking all traffic from them) or allow the attack to continue. With SDN, flows can be blocked or redirected, instead of the agents creating the flows. Additionally, the NIMS would be able to monitor the entire network and block or redirect malicious flows, not just those directed at the equipment being protected by a NIPS. This is noteworthy as while access to these protected machines is desirable by attackers, this is not the goal. With automated botnets and worms, the goal is simply to spread as far as possible and turn those infected machines into resources to spread further. Later, instructions can be sent to utilise this network (as was seen with Mirai and Petya/NotPetya [74]).

A NIDS may help to detect the malware and alert an administrator to its existence, and a NIPS may help stop the spread to essential network components, however, without some sort of automated response system the attack may have compromised the entirety of the rest of the network. Removing such an infection could feasibly take months, as frequently the best course of action is to replace the infected system with one that is patched to not be infected in the same way.

4 Overview of Assumptions

Previous chapters, have discussed the apparent conflict between the SDN paradigm (a centralised and programmable network, based upon a high level representation of the state of the network) and deep learning based IDSs or IPSs (requiring direct packet analysis and in the case of IPSs changing the network state without utilising the SDN). Also discussed (at a high level) have been solutions to this issue, and how other researchers have approached it. This chapter, shall provide an overview of the assumptions that have been made whilst developing the solution, and provide the environmental context the solution is designed to work in.

4.1 Overview of Hardware Assumptions

Figure 4.1-1 illustrates an example of how a network may look with the proposed solution deployed. This can be separated into three sections, mimicking a SDN layout. In this section will discuss what these areas are and what components fit into them.

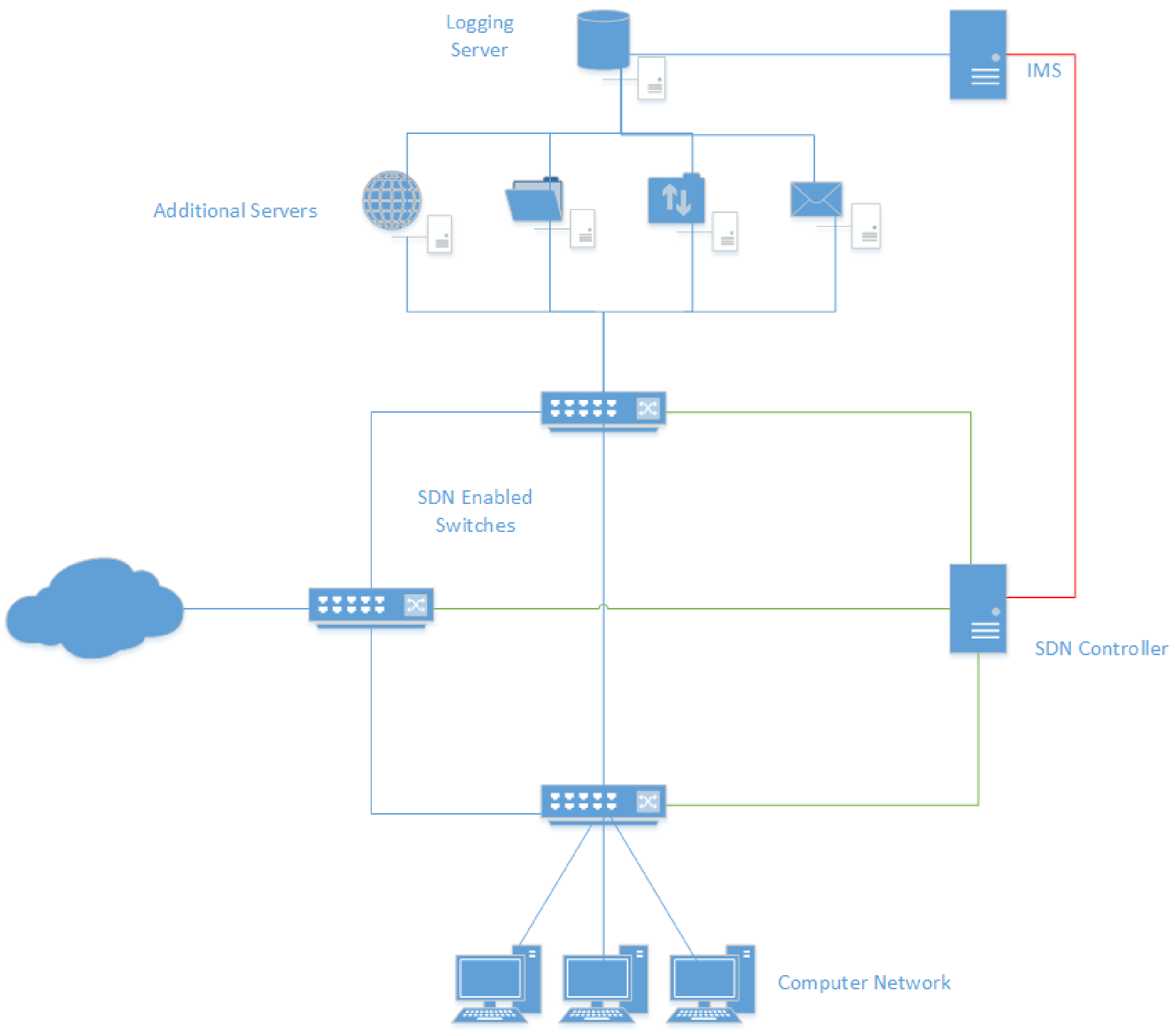


Figure 4.1-1 – Architectural view of an example network it is believed the solution could be placed into

The three core areas are highlighted with the use of the different coloured lines indicating network connections. These consist of the Data Plane (blue lines), the Control Plane (green lines) and the Mitigation Plane (red lines).

It should be noted that with this setup, both the IMS and the SDN controllers are single point of failures. This holds true for SDN controllers in general (one of the disadvantages of the SDN paradigm is that the controller becomes a point of weakness and could be a target of attack). However, the same benefits that have led to SDN becoming more widespread can aid security. There is reduced operational cost (one device protecting the network vs several protecting different locations), and easier management as a result. Additionally, it offers advantages in terms of reaction to threats. If an internal attack has been detected in one area of the network, then the whole network can be configured to block or redirect that threat, should the same symptoms be detected elsewhere. More to the point, this can be done automatically, leveraging the power of SDN.

Within the diagram, the logging controller and IMS have been shown as two separate devices, however it may be beneficial to have them run on the same device, or at least have access to the same storage (in the case of Storage Area Networks or Fibre Channel Storage). The IMS will want to check logs frequently to see if there are any updates for matching flows and having them connected would decrease the overhead. However, it should also be considered that running the IMS is likely to be computationally expensive (in terms of CPU and memory specifically) and having it reside on the same device as the log server could lead to bottlenecks. This is slightly outside the scope of this work and is something that would need to be explored further for a future system. In the same vein, the mitigation component and detection component are shown as being on the same device (the IMS). In practice, they could be on separate devices, and the benefits of separating them would need to be weighed against the costs.

4.1.1 Data Plane

In this plane, the data that will eventually be analysed is created. As such, this consists of the SDN-enabled switches, additional servers, and general PCs.

- **Servers**
  Shown in Figure 4.1-1 are additional servers that represent the kind of servers that may be seen in a business (such as a web server, FTP server, SSH server and a mail server). Logs from these servers are transferred to the logging server, a common component of business networks which aggregates and sort logs for easier processing later. In the proposed system, this is called the log controller, as this title better fits the intended role. Log levels for these services are set to notice (or the equivalent, i.e. usually the level before warning) and be transferred with existing log management software (such as syslog). It should be noted that the analysis of logs is done later on the log controller.
- **SDN Enabled Switches**
  These provide the network flows, which are the base of the detection system. Upon registering a flow, they were not previously aware of, a message is sent to the SDN controller with details

of the new flow, including source and destination addresses, protocols involved (UDP, TCP etc.) and start time. It is also updated regularly with number of packets, and size of flow.

- **General PCs**
  While general use PCs rarely have extensive logging turned on by default (as the scope for damage is typically less), DNS lookups can be logged, along with other general activity information. This is often used to diagnose malware. Activity at unusual times (for example when the office is closed) can be a sign of malicious use, as can requests for unusual addresses. Tracking of activity usually requires an agent/server relationship, in which an agent is set to start-up automatically on the user PC and sends an aggregated view of activity for a defined time period, to the server. The server can then provide an overall view of general activity, which can then be focused on specific computers if needed. Unlike the specific servers above, the user PCs typically do the initial data aggregation, rather than a log server, which naturally means the data is less granular.

4.1.2 Control Plane

The control plane consists of two main devices, the SDN controller (as with SDNs) and the log controller. The purpose of the control plane is to aggregate and format the data so that it can be used later by the mitigation plane.

- **SDN Controller**
  The SDN controller is configured as it would normally be, keeping a list of current and past network flows. While the switches provide the controller with source, destination, size of instances of flows and number of packets, the flow controller aggregates this data to create a larger view. For example, a switch may provide the controller with an update saying "A flow that has been matched to one in my database with this source and destination, lasted x seconds, with y number of packets and a size of z bytes." The controller would take this information and add the number of packets, bytes and duration to the flow data already compiled. In addition, controllers typically keep note of the length of time the flow has been idle (which will be reset).
- **Log Controller**
  The log controller receives logs as it would if it were a normal log server, from servers and devices on the data plane. These are then integrated into a database from which additional features are obtained. Required features would include things like date and time of creation (to match against flows that may have created it), as well as source of the log (which server or host the log was generated on) and the log text itself. Ideally, the remote IP address would also be accessible. Derivable features include examples such as login attempted (was the log created because a user attempted a login), login successful (was the log generated because a user successfully logged in), a sentiment score of the log text (how positive or negative the log text sounds), and log number (how many logs have been generated matching certain criteria (such as from the same IP within n minutes)).

4.1.3 Mitigation Plane

The mitigation plane consists of two main components, the detection component and the mitigation component, both of which could run on the same device (The IMS in Figure 4.1-1) due to the close integration these two components would need. As stated in 4.1 they could be separated onto two devices, however much of the work they will do is very similar, and they will need access to the same data sources, so it is assumed that they would be on the same device.

- **Detection Component**
  This component is responsible for combining the separate log and network flow data, detecting useful features, and making an initial classification. Logs are combined with flows through matching the source and destination addresses, as well as ensuring the log has been created within *n* minutes of the flow. A delay is allowed as flows will be created before the logs they generate, and so cannot be matched perfectly. In addition, the time period allows us to spot instances where a user has started a short flow, stopped it, and then restarted it again a few minutes later. Depending upon load and configuration this activity may appear to be new flows to the SDN controller, especially if a FIN message is sent in the case of a TCP flow. For instance, this can happen in the case of a password attack. The attacker will attempt two or three passwords, and then pause. This can aid in bypassing a lockout limit (for number of attempts within a set time period).
- **Mitigation Component**
  The component is responsible for determining what kind of attack the network is under and determining a mitigation method for it. This is done not through the traditional method of comparing a flow to previous examples of attacks, but by comparing the flow to several different attack characteristics within an attack taxonomy (examples of which have been discussed in Section 3.4). Through comparing the flow to different leaves of the taxonomy it is possible to determine a mitigation that is custom to the attack that is being experienced. This is done as some attacks that may be considered quite different may have similar mitigations. For example, a SQL injection attack designed to leak data, and a brute force password attack designed to gain access to a system, where the mitigation for both could consist of blocking the offending IP. The reverse is also true, where similar attacks may have different mitigations. For example, a SYN flood mitigation may be to block the flow, whereas a DDoS attack mitigation might be to redirect traffic. However, in both cases the attack being made is a DoS attack.

4.2 Overview of Process Assumptions

This section shall provide an overview of the assumptions that are made about how both server logs and network flows are created and interact. These are based on the hardware assumptions made in Section 4.1, and seek to clarify exactly where and how the data being used in the rest of this thesis would come from in a real-world application. To do this, process diagrams that outline the individual steps taken will be shown. All these steps are taken within the Data Plane explained in Section 4.1.1, and these processes are termed initialisation processes later in the thesis. An initialisation is considered an event that results in the creation of a network flow. These events are outside the ability of a network administrator to control, however there are several ways this could be achieved, and administrators need to be mindful of these. These are discussed below, as is their impact on this work.

It should be noted that these processes could happen in sequence, or in parallel, or in any other combination. The goal is to show the types of communication, and what outputs might be available to analyse, as well as where in the system this analysis might take place. To aid with this, sections of the flows have been highlighted red, green, or blue, and this will correspond to the colour segments of the network diagram Figure 4.1-1.Figure 4.1-1 – Architectural view of an example network it is believed the solution could be placed into

External Client to Server Communication

This is the bulk of what would be considered typical communication and is what most datasets attempt to represent. Communication is initiated by an external actor, typically using the internet. The first notice of this communication received is from the firewall, and flow is created at the first switch it meets after passing through the firewall. The outline of the flows' path is shown in Figure 4.2-1. This shows the process of the flow lifecycle and is generally like that found in most SDNs. Three components can be identified clearly, the SDN-enabled switch (green), the SDN controller (blue) and the server, which could be a SSH server, or an FTP server etc. (red). Note that a flow is always created or updated within this process, while a log is not. It is possible that flows could be created that do not require the creation of a log. For example, it is unlikely a web server would log every page request it receives, as this would quickly lead to unmanageable log sizes.

It should also be noted that the firewall will be dropping packets before they meet the SDN-enabled switch, meaning that no flows are ever created, and the data is not analysed by the system. While SDN-based firewalls have been researched, it is likely that exposing a controller to the level of traffic experienced by an external firewall would lead the controller to be become overloaded, and for little benefit.

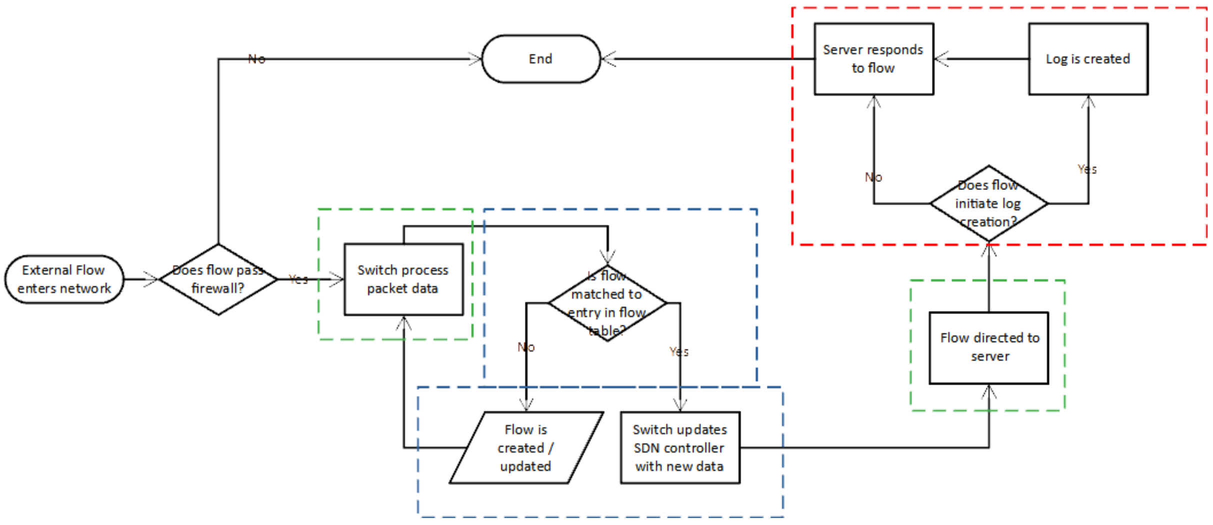

External Flow enters network
Does flow pass firewall?
No
Yes
End
Switch process packet data
Is flow matched to entry in flow table?
No
Yes
Flow is created / updated
Switch updates SDN controller with new data
Flow directed to server
Does flow initiate log creation?
No
Yes
Server responds to flow
Log is created

Figure 4.2-1 – External Client to Server Flow

Internal Client to Server Communication

Figure 4.1-1 shows a secondary firewall that monitors all data directed to the servers regardless of the location. This leads to a process flow similar to the external client to server, with the exception that all communication produced by the internal computers will produce a flow to analyse. Additionally, the second firewall leads to a situation in which flows may be created, but logs are not. For instance, if somebody mistakenly attempted to start an FTP connection to the webserver (through typing the wrong URL), it is likely the firewall would drop that connection. This is shown in Figure 4.2-2.

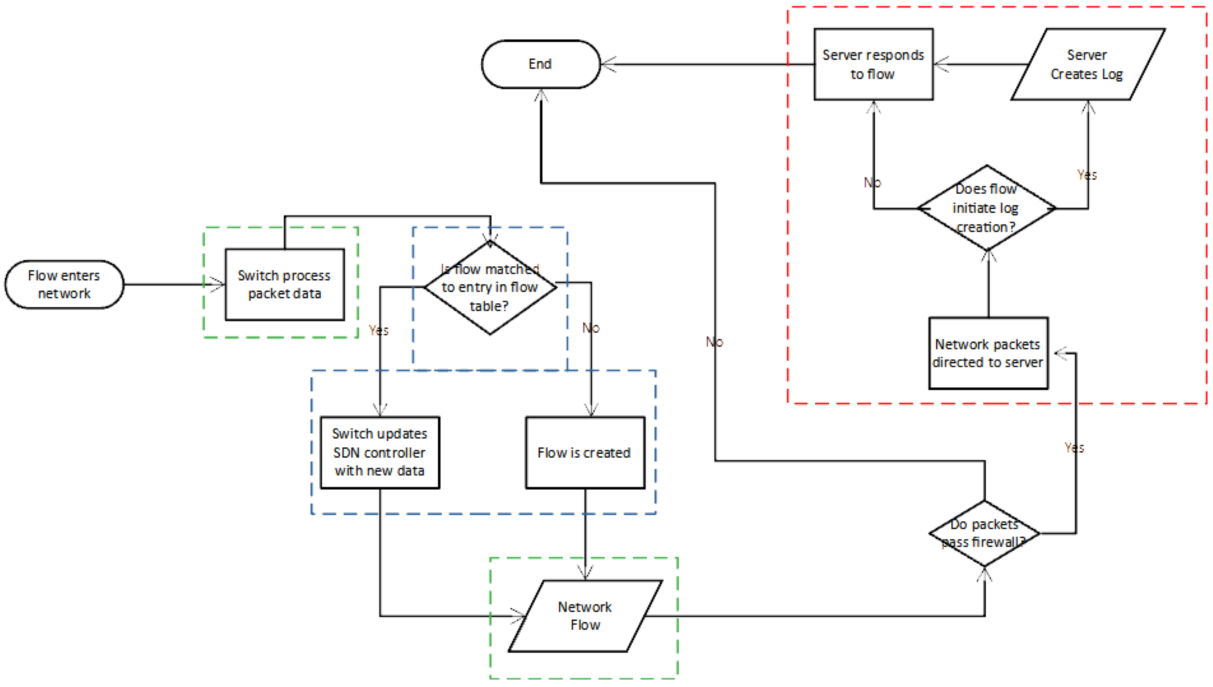
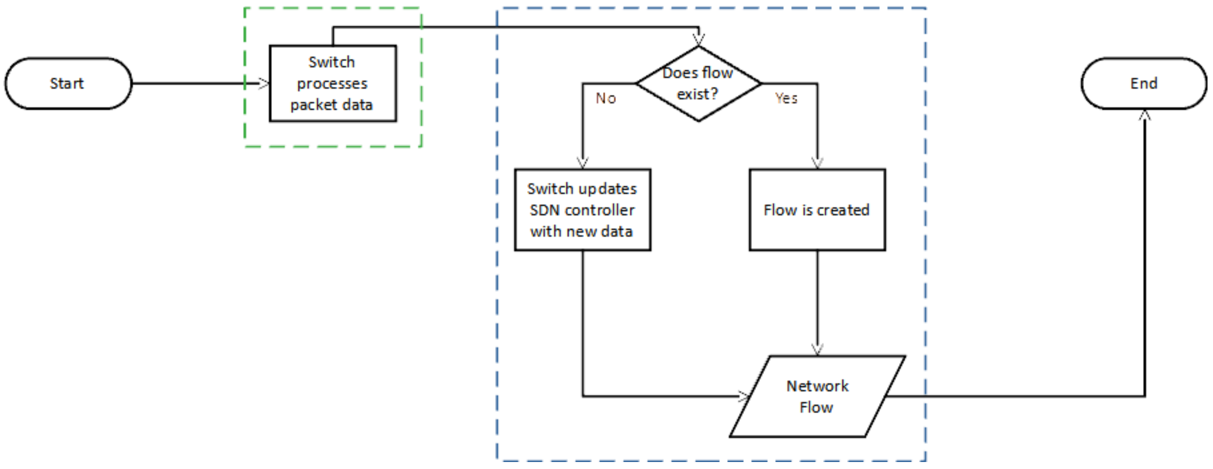

End
Server responds to flow
Server Creates Log
No
Yes
Does flow initiate log creation?
Flow enters network
Switch process packet data
Is flow matched to entry in flow table?
Yes
No
No
Network packets directed to server
Switch updates SDN controller with new data
Flow is created
Yes
Do packets pass firewall?
Network Flow

Figure 4.2-2 – Internal Client to Server Communication flow

The primary difference in the flows between Figure 4.2-1 and Figure 4.2-2 is the location of the firewall. With external communication, the firewall is going to be the first thing the flow comes against, and not passing the firewall means that the flow never enters the network. With internal communication the firewall is only accessed after the flow has entered the network, and so will be analysed regardless of whether it can pass the firewall or not.

Internal Client to Client Communication

While comparatively rare, this kind of communication is becoming more common as applications have moved away from client-server relationships to peer-to-peer relationships. For example, the file sharing tool Dropbox enables peer-to-peer file sharing for shared folders on the same network, in order to speed up synchronisation speed versus that of syncing over the internet. It is also this kind of communication that would result from a botnet infection, as the bot attempts to infect other computers on the same network. This is important as ransomware infections spread by botnets are increasing within business environments [8], and this is seen as an inexpensive way of disrupting a network. The flow for this kind of activity is shown in Figure 4.2-3.



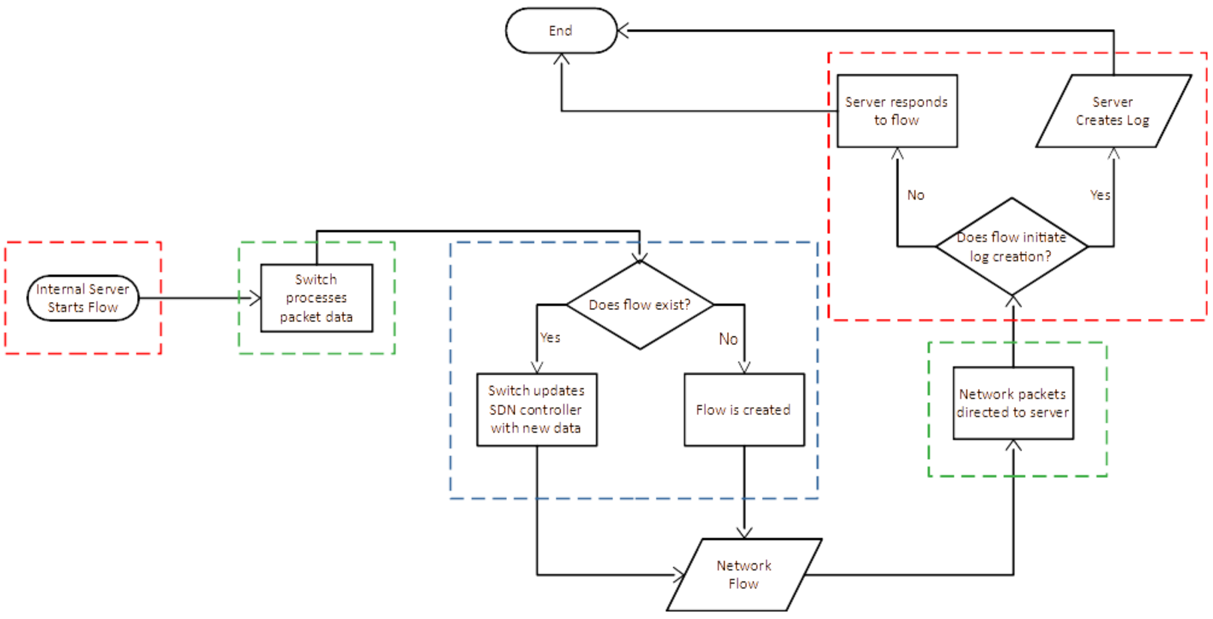
Start
Switch processes packet data
Does flow exist?
No
Yes
Switch updates SDN controller with new data
Flow is created
Network Flow
End

Figure 4.2-3 – Client to client communication flow

There is no firewall involved in this system, as typically client computers are not specifically protected before they access a switch. This makes for a simpler flow diagram but is still communication that needs to be analysed. It should also be noted this kind of communication typically will not produce server logs, as the flow is not directed at a server.

Internal Server to Server Communication

While similar to the internal client to server model, this kind of communication would rarely feature any significant defences. The reasoning is simple, this kind of communication typically comes as a result of servers requiring data from each other in pursuit of a request from another client or process. Time to reply is often critical in this kind of situation, and many networks cannot afford the computational complexity of having another comparatively slow firewall protecting servers from each other, especially when this should be the most secure area of the network, with no malicious activity.



End
Server responds to flow
Server Creates Log
No
Yes
Does flow initiate log creation?
Internal Server Starts Flow
Switch processes packet data
Does flow exist?
Yes
No
Switch updates SDN controller with new data
Flow is created
Network packets directed to server
Network Flow

Figure 4.2-4 – Server to server communication

Figure 4.2-4 shows this, with no firewall included in the diagram, but still containing the log creation.

External Client to Internal Client Communication

While traditionally rare, this communication type is becoming more common as remote working becomes common place, and applications such as TeamViewer make the experience easier. Although cloud computing and remote software can aid in remote working, often users want to use the same computer they would be working on inside the office, or need specialist tools or hardware that are only available through logging into their office computer remotely. This type of communication has the same issue as remote client to server communication, in that if the connection is stopped at the firewall, no flows or logs of the event will enter the system. However, it is unlikely that a traditional firewall would stop this, as typically these services (while peer to peer) utilise outbound connections to bypass stateful firewall checks. Figure 4.2-5 shows the process for this, again showing that if the incoming flow does not meet the requirements to pass the firewall, then the process will never begin. Again, as the flow never interacts with a server, logs are not recorded.

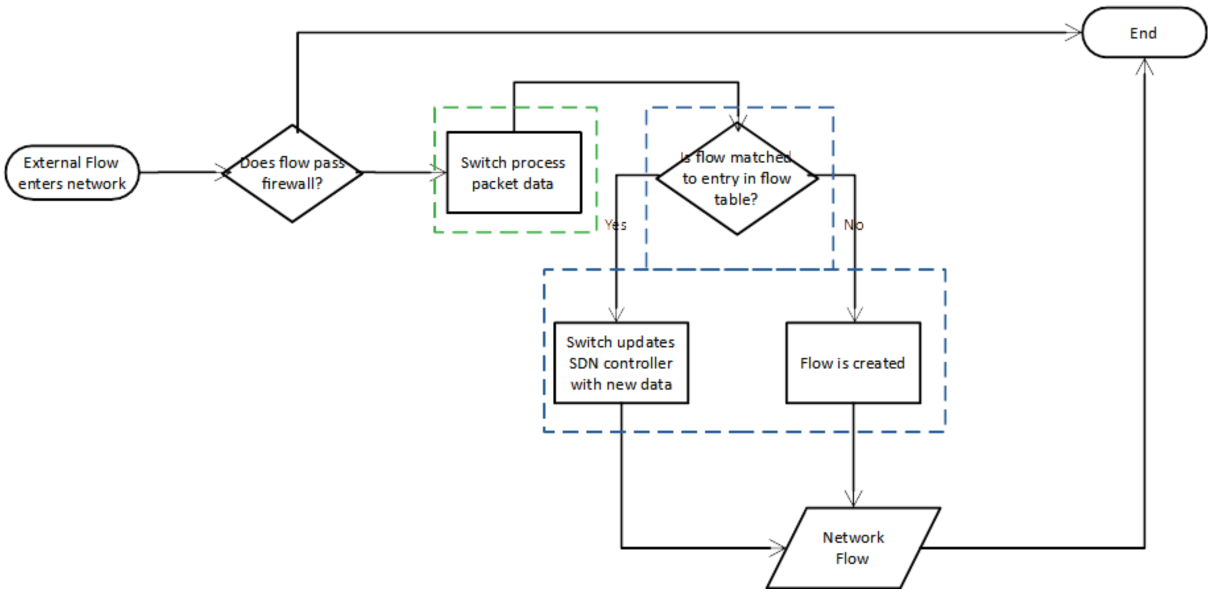

End
External Flow enters network
Does flow pass firewall?
Switch process packet data
Is flow matched to entry in flow table?
Yes
No
Switch updates SDN controller with new data
Flow is created
Network Flow

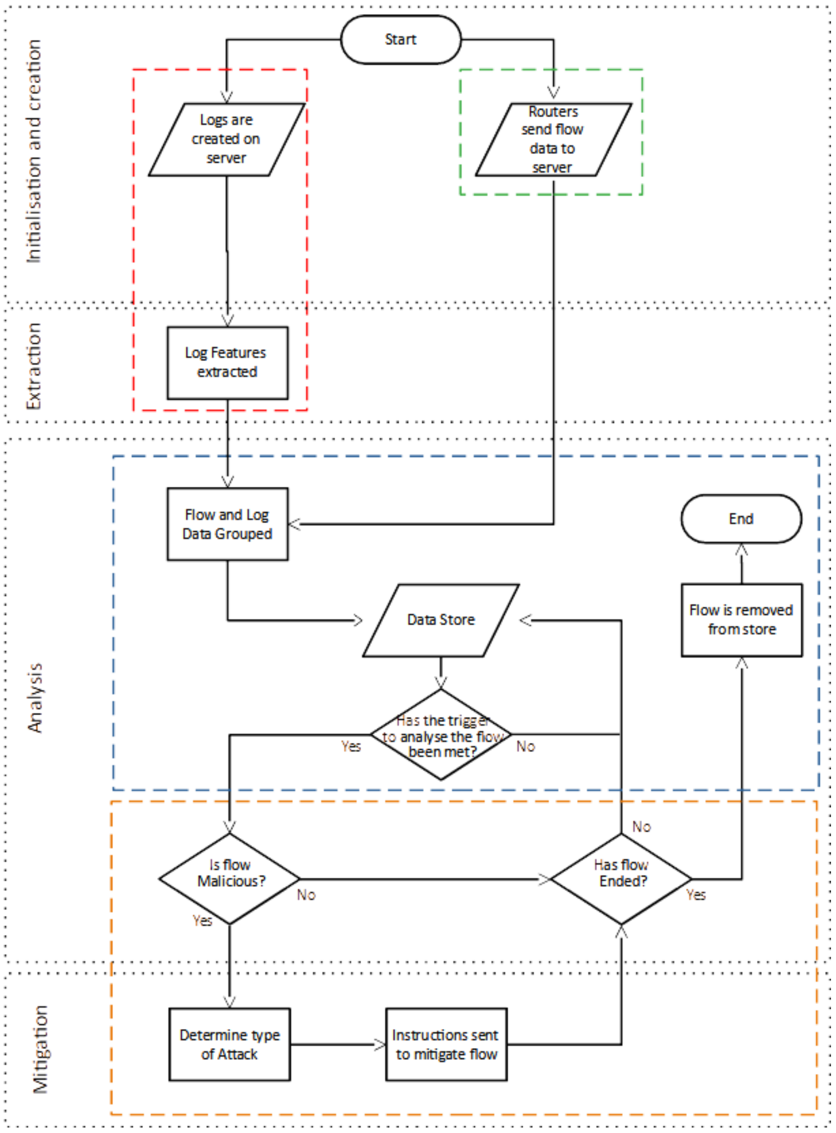
Figure 4.2-5 – External Client to Internal Client

5 Methodology

This chapter shall discuss the methodology and reasoning behind the system being created, linking decisions to the previous assumptions that have been made, and showing the reasoning behind them. To achieve this, process diagrams detailing the individual steps being taken will be presented, including code extracts where applicable. A process flow diagram will be followed (to be shown in the next subsection), while mentioning where on the hardware diagram the process is taking place.

5.1 The process being followed

In Figure 5.1-1 the overall process laid out. As presented, it consists of four main stages. These stages are organised according to the state of the data and the kind of analysis or transformation being done to it. This is in contrast to the hardware overview, which is organised according to what the goal of the hardware is within the system (creation of data, control and organisation of the data, and manipulation of the data). The process stages are Initialisation and Creation, Extraction, Analysis and Mitigation.



Initialisation and creation
Extraction
Analysis
Mitigation
Start
Logs are created on server
Routers send flow data to server
Log Features extracted
Flow and Log Data Grouped
End
Data Store
Flow is removed from store
Has the trigger to analyse the flow been met?
Yes
No
Is flow Malicious?
No
Yes
Has flow Ended?
No
Yes
Determine type of Attack
Instructions sent to mitigate flow

Figure 5.1-1 – Flow diagram to show the flow of data within the system

Of note is the new orange box. This is the activity that is envisioned would happen on the IMS. As mentioned in Section 4.1.3, this could be the same physical device as the SDN controller, or a separate physical box that uses the SDN controllers' northbound interface to collect relevant data.

It should be noted that the logs are processed in parallel to the network flows until they are finally grouped. This is done to attempt to ensure the system will scale, as processing both logs and flows in series could affect scalability. Additionally, as has been mentioned in Section 4.2 there is an assumption that while flows will always be created, logs may not be, and this design aids that assumption by allowing logs to be matched to flows when they are created but doesn't wait for them or require them. This is also tested with the SSH Compromise Detection Using Netflow/IPFIX Dataset, which did not have corresponding logs for every flow. The log data is supplementary data to aid classification, and not essential.

5.2 Initialisation and Creation

The initialisation process includes all actions required to create a network flow. This is significant as it is possible to create a flow without creating any logs. For example, with the network hardware given above, an internal HTTP request for an external website would likely not generate any logs, but would generate multiple network flows (DNS query, HTTP flow itself, etc.). A request like this would still need to be analysed, especially as these requests can be used to track botnet activity. It should also be noted that logs that are created without flows are not being considered, as this would fall within the scope of HIDSs. As stated in Section 4.1.1, creation of the data is handled by devices in the Data plane. This includes logs from servers, as well as network flows from switches. More specifically, switches provide flow data to the SDN controller, and the servers create logs that are provided or accessed by the log server. As such in the rest of this section shall discuss the creation of the logs and flows.

5.2.1 Flows

Flows are the first kind of data to be created and are the primary source of data for the IMS. A network flow is created whenever data arrives at an SDN-enabled switch and does not match any previously encountered data. When comparing a packet to previously encountered data to determine which flow entry a packet belong to, OpenFlow specifies 12 fields a packet can be compared against:

1. Switch Input Port
2. VLAN ID
3. VLAN Priority
4. Ethernet Source Address
5. Ethernet Destination Address
6. Ethernet Frame Type
7. IP Source Address
8. IP Destination Address
9. IP Protocol
10. IP Type of Service bits
11. Source Port (For TCP or UDP)

12. Destination Port (For TCP or UDP)

Not all the fields are required, and there are three types of compliance:

- Full compliance: where all fields are supported.
- Layer 2 conformance: where the layer 2 headers are supported (fields 2-6)
- Layer 3 conformance: where the layer 3 headers are supported (fields 7-12)

It is noteworthy that packets can be matched to multiple flow entries (effectively belonging to multiple flow entries), and in order to manage this the OpenFlow specification states that the packet will be assigned to the flow entry with the highest priority, and no more searches for other matches will be made.

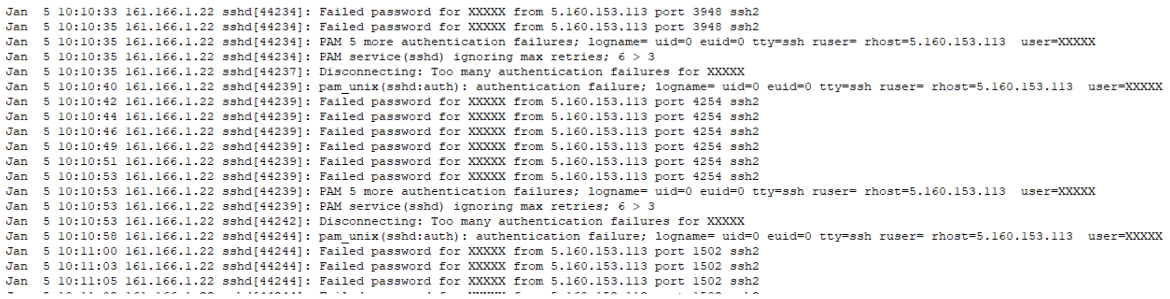
Typically, NIDS datasets (such as the UNSW-NB15, and University of Twenté dataset outlined in Section 2.6) do not contain fields 1-6, but only IP addresses and ports. This work assumes an organisation will be using a layer 3 compliant switch in order to gather the data, thus providing the following 6 fields for us to use:

1. IP Source Address
2. IP Destination Address
3. IP Protocol (TCP/UDP etc)
4. IP Type of Service Bits
5. Source Port
6. Destination Port

However, an OpenFlow table also contains counters and timeouts. The specification [37] requires a counter to be used for active entries, duration, received packets and received bytes. This increases the features used to 9. There are other counters available, but these are not required by the OpenFlow specification and so are not assumed to be available for the purposes of this work. Other SDN solutions may provide more data, however this is on a case by case basis, and the project seeks to create a solution that should be applicable to as many SDN based solutions as possible. OpenFlow has been chosen as it forms the basis of other commercial SDN solutions [4], and so even if a company is not using an OpenFlow product specifically, it is likely the same assumptions about available data can be made.

5.2.2 Server Logs

Logging is enabled on most server-based systems by default, and is generally required for legal and compliance reasons, as well as security and monitoring. Figure 5.2-1 shows a log from the NetFlow/IPFIX dataset from the University of Twenté [40].



```
Jan  5 10:10:33 161.166.1.22 sshd[44234]: Failed password for XXXXX from 5.160.153.113 port 3948 ssh2
Jan  5 10:10:35 161.166.1.22 sshd[44234]: Failed password for XXXXX from 5.160.153.113 port 3948 ssh2
Jan  5 10:10:35 161.166.1.22 sshd[44234]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:10:35 161.166.1.22 sshd[44234]: PAM service(sshd) ignoring max retries; 6 > 3
Jan  5 10:10:35 161.166.1.22 sshd[44237]: Disconnecting: Too many authentication failures for XXXXX
Jan  5 10:10:40 161.166.1.22 sshd[44239]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:10:42 161.166.1.22 sshd[44239]: Failed password for XXXXX from 5.160.153.113 port 4254 ssh2
Jan  5 10:10:44 161.166.1.22 sshd[44239]: Failed password for XXXXX from 5.160.153.113 port 4254 ssh2
Jan  5 10:10:46 161.166.1.22 sshd[44239]: Failed password for XXXXX from 5.160.153.113 port 4254 ssh2
Jan  5 10:10:49 161.166.1.22 sshd[44239]: Failed password for XXXXX from 5.160.153.113 port 4254 ssh2
Jan  5 10:10:51 161.166.1.22 sshd[44239]: Failed password for XXXXX from 5.160.153.113 port 4254 ssh2
Jan  5 10:10:53 161.166.1.22 sshd[44239]: Failed password for XXXXX from 5.160.153.113 port 4254 ssh2
Jan  5 10:10:53 161.166.1.22 sshd[44239]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:10:53 161.166.1.22 sshd[44239]: PAM service(sshd) ignoring max retries; 6 > 3
Jan  5 10:10:53 161.166.1.22 sshd[44242]: Disconnecting: Too many authentication failures for XXXXX
Jan  5 10:10:58 161.166.1.22 sshd[44244]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:11:00 161.166.1.22 sshd[44244]: Failed password for XXXXX from 5.160.153.113 port 1502 ssh2
Jan  5 10:11:03 161.166.1.22 sshd[44244]: Failed password for XXXXX from 5.160.153.113 port 1502 ssh2
Jan  5 10:11:05 161.166.1.22 sshd[44244]: Failed password for XXXXX from 5.160.153.113 port 1502 ssh2
```

Figure 5.2-1 – Extract from the logs of the Netflow/IPFix Dataset

The logs in this dataset come from Kippo and OpenSSH servers, and contain date, time, service, source IP address or hostname as well as log text. This is broadly in line with syslog (RFC 5424), which specifies a log should contain Hostname or IP address, Timestamp, Facility, Severity, and the message itself. Windows system logs contain Date, Time, Computer, Source, Event ID, Level and Category, as well as log text. While logs are not standardised, there are some similarities between different log formats, and data that can be gleamed almost universally, that being the date and time of creation of the log, hostname or IP address of the system, log text and service that resulted in the log creation. It is logical that these attributes would be included in most log systems, simply because for a log to be useful these are the minimum attributes required. If a log does not have a time or date, an administrator cannot tell when the log was created or attempt to discern what events may have been happening at the same time to cause its creation. Similarly, while IP address or hostname may not be required when dealing with a single machine, if transferred to a central log server (a common practice in business environments) an administrator will need to know where a log originated, and the service it relates to. As such, it can be assumed that logs within a business environment contain:

1 Date
2 Time
3 Hostname or IP Address
4 Service
5 Log text

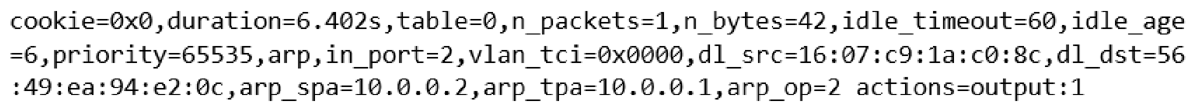
However, the only feature this easily adds to the proposed system is Service, since port number is not necessarily a reliable representation of the service being used on a network level. Date, time, and IP are already provided by the network flow, and log text can vary enough within a single service that without further processing it simply cannot be used. It may also be possible to add severity to the list, however, different log formats use different ratings for severity. This comes in the form both rating number and scale itself. For example, syslog uses a rating of 0-7 to represent Emergency, Alert, Critical, Error, Warning, Notice, Informational and Debug, whereas windows uses a rating of 1-5, representing Critical, Error, Warning, Information and verbose. If using this measure to determine the severity of the event, the question becomes how should these separate ratings be compared? For example, should a critical error in syslog (level 2) be comparable to a critical error on windows (level 1), or an Error on windows (level 2). Given the range of different log services, and the variety of potential different levels, an alternative method to determine the severity is required.

5.3 Extraction

The base level features that can be extracted have been identified in Section 5.2. However, this base level data is not suitable for intrusion detection alone. Additionally, it was stated that log text needs more processing in order to be useful. The SDN flows do not require the same kind of processing as the logs do, and features derived from network flows are features the SDN controllers keep track of anyway. For example, one such feature includes total size of flow (both in terms of packet number and bytes). Switches send the size of individual flow instances to the controller, and the controller updates the flow table with the cumulative sum of the flow instances to get the size of the flow in total. While more data can be derived (such as an average flow instance size) the goal is to create a system that will work with the data provided by a SDN controller.

5.3.1 Flow Features are Extracted

The first step is to load flows into the database since the network flows are always generated and are used as a comparison to the logs. Flows are received in a textual form, an example of which can be seen in Figure 5.3-1:



```
cookie=0x0,duration=6.402s,table=0,n_packets=1,n_bytes=42,idle_timeout=60,idle_age
=6,priority=65535,arp,in_port=2,vlan_tci=0x0000,dl_src=16:07:c9:1a:c0:8c,dl_dst=56
:49:ea:94:e2:0c,arp_spa=10.0.0.2,arp_tpa=10.0.0.1,arp_op=2 actions=output:1
```

Figure 5.3-1 – An example of an OpenFlow flow

This can effectively be read as a CSV, with cookie=0x0 being the first column, duration=6.402s being the second, etc. This can be performed with pseudocode like:

```
import csv
string = getFlow()
x = csv.reader(string)
for row in x:
    #Process the row
```

Pseudocode 5.3-1 – Pseudocode for reading a flow

In Pseudocode 5.3-1, the getFlow() function runs an appropriate command on the SDN controller to view flows (for OpenFlow this would be the ovs-ofctl dump-flow command), and returns the contents. From here, the contents of various fields can be accessed through row[x] where x is the number of the field you wish to access. For features that consist of just a number, a regular expression filter such as re.sub("[^0123456789\.\:]","",row[x]) can be applied, which will remove any characters except numbers, decimal points or colons (for IPv6 addresses). So, for the string in Figure 5.3-1, Table 5.3-1 shows the features obtained, and where they were obtained within the string:

Feature	Was Filter Used?	Row Number	Result
IP Source Address	Yes	Row[13]	10.0.0.2
IP Destination Address	Yes	Row[14]	10.0.0.1
IP Protocol	No	Row[8]	arp
IP Type of Service Bits	Yes	Row[7]	65535

Source Port	Yes	Row[15]	2
Destination Port	Yes	Not applicable for ARP, same as Source Port	2
Duration	Yes	Row[1]	6.402
Received Packets	Yes	Row[3]	1
Received Bytes	Yes	Row[4]	42

Table 5.3-1 – Table to show features gained from an Example ARP flow, including location they were taken from

For the TCP flow shown in Figure 5.3-2, the corresponding table is shown in Table 5.3-2.

```
cookie=0x0, duration=16.012s, table=0, n_packets=12, n_bytes=2945, idle_timeout=60,
idle_age=16,priority=65535,tcp,in_port=1,vlan_tci=0x0000,dl_src=6e:04:ff:d0:7c:fc,
dl_dst=82:48:b7:2b:56:0b,nw_src=10.0.0.1,nw_dst=10.0.0.2,nw_tos=0,tp_src=80,tp_dst
=52930 actions=output:2
```

Figure 5.3-2 – An example of a TCP OpenFlow flow

Feature	Was Filter Used?	Row Number	Result
IP Source Address	Yes	Row[13]	10.0.0.1
IP Destination Address	Yes	Row[14]	10.0.0.2
IP Protocol	No	Row[8]	tcp
IP Type of Service Bits	Yes	Row[7]	65535
Source Port	Yes	Row[16]	80
Destination Port	Yes	Row[17]	52930
Duration	Yes	Row[1]	16.012
Received Packets	Yes	Row[3]	12
Received Bytes	Yes	Row[4]	2945

Table 5.3-2 – Table for an example TCP flow

This data can now be used to compare the flow received to flows within the main IMS database. This database, similar to a SDN flow table, keeps a list of flows that are traversing the network, and its corresponding data aggregated data. The term Network Health Flow (NHF) is proposed to describe the aggregated data being kept on the IMS. An SDN flow can be matched to a NHF using IP Source and Destination, IP Protocol, and Source and Destination ports collectively. If a match is found, then Duration, Received Packets and Received Bytes can be updated with the new information, otherwise a new NHF can be entered into the IMS database. This is shown in Pseudocode 5.3-2.

```
import mysql.connector

mydb = mysql.connector.connect(
host=host,
user=user,
passwd=password,
database="flowDB"
)
mycursor = mydb.cursor(buffered=True)

```

```
query=”SELECT count(*) FROM flowTable WHERE ip_src = ip_src AND ip_dst = ip_dst AND port_src = port_src AND port_dst = port_dst AND protocol = protocol;”

mycursor.execute(query)
result = mycursor.fetchone()

if result[0] > 0:
```

Pseudocode 5.3-2 – Pseudocode for creating a new flow entry to updating an existing entry

After the flows have been updated and new flows added, any logs that have been generated can be matched to the flow data. The first stage in this is adding data to a temporary log database, to make the processing quicker. This is performed using a query that is very similar to that used to match the network flows, but the match is made on the host IP address (after being converted if necessary), as well as the service and the time created. Pseudocode for this is provided in Pseudocode 5.3-3. This simultaneously gives us the first feature derived directly from the logs, the number of instances of duplicate logs being generated.

```
import mysql.connector

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB"
)

mycursor = mydb.cursor(buffered=True)
query=”SELECT count(*) FROM logTable WHERE hostIP = hostIP AND service = service AND time>= time AND logText = logText

mycursor.execute(query)
result = mycursor.fetchone()

if result[0] > 0:
        query = “UPDATE logTable SET instances = instances + 1, time = time, date = date
        WHERE hostIP = hostIP AND service = service AND time>= time;”
Else:
        query = “INSERT INTO logTable hostIP, service, date, time, logText, instances
        VALUES hostIP, service, date, time, logText, 1;”

mycursor.execute(query)
```

Pseudocode 5.3-3 – Code to match log entries to flow entries

5.3.2 Log Features are Extracted

More features can be extracted from the logs; the first aspect considered is whether there is a successful login. The only way to get this data without the use of decryption techniques and deep packet inspection is to analyse the log text. Because logs are generally highly structured, and the text does not deviate significantly between different instances of the same log it is possible to use source code similar to Pseudocode 5.3-4:

```
log = getLogFromDataServer()
x = log.split()
for word in x:
  if (word == "Accepted"):
    accepted = 1
  elif (word == "password"):
    password = 1

if (password == 1) && (accepted == 1):
    loginAttempt = 1
  else:
    loginAttempt = 0
```

Pseudocode 5.3-4 – Entry to show whether a login attempt has been made

The code simply looks for the words “Accepted” and “Password”. If a single log contains both of these words, it is considered to have been a successful login. It should be noted that these words were chosen specifically for Kippo and OpenSSH logs, which log a successful login with a log like that shown in Figure 5.3-3:

```
Accepted password for XXXXX from xxx.xxx.xxx.xxx port xxxxx ssh2
```

Figure 5.3-3 – Accepted Password Log

And log failed logins with Figure 5.3-4:

```
Failed password for XXXXX from xxx.xxx.xxx.xxx port xxxxx ssh2
```

Figure 5.3-4 – Failed Password Example

If the log is recorded with a different message along the lines of Figure 5.3-5:

```
Password not accepted for XXXXX from xxx.xxx.xxx.xxx port xxxxx ssh2
```

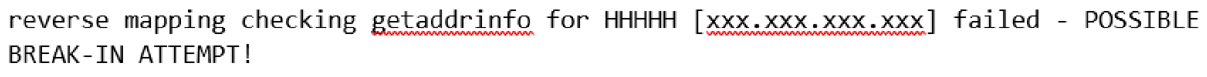
Figure 5.3-5 – Alternative Failed Password Log

The relevant code would have to change to accommodate an additional qualifier of the word “not”. If the word “not” is found, then the result of the code is multiplied by -1 in order to generate a minus number if the log displays a failed login. In the code, this is replicated through use of another if statement to detect the word “Failed” (which only appears in failed password logs). Upon finding this, the loginAttempt variable is set to -1. This means on the log database now contains:

- Date
- Time
- Host IP
- Service
- Log Text
- Instances
- Login Success

The next step is to determine a sentiment score for the log. The sentiment score needs to be an indicator of how positive or negative the log text is overall. The reasoning for this is that logs depicting negative events are typically going to contain negative words, such as “error”, “failure” and “closed”. If a flow is producing mainly negative logs, then that could be taken as an indicator that the flow is malicious.

The Python textblob library has been selected, which contains the sentiment.polarity function. This function scores words individually and is based upon Vader used in the Natural Language Tool Kit (NLTK). This creates a normalised, weighted composite score based upon the lexicon that was generated within VADER [75]. VADER is a sentiment lexicon that is sensitive to both the intensity and polarity of the statement being read and is fast enough to be used with online streaming data. The score it generates varies between -1 and 1, where scores close to -1 are almost entirely negative, whereas scores close to 1 are almost entirely positive, and scores close to 0 are neutral. While normally used for sentiment analysis for online reviews, Vader has some advantages for this work as well. Vader attempts to qualify sentiment not only using its own lexicon, but also qualifies the sentiment being expressed by including punctuation and use of capital letters. This allows it to shift sentiment appropriately when confronted with a phrase like: “VADER is VERY SMART, handsome, and FUNNY!!” compared to: “Vader is very smart, handsome, and funny.” The first phrasing clearly has more emphasis on the positive attributes, given by the capitalisation and exclamation marks, and so will gain a more positive score. While logs are more structured than online comments, the use of capitals and punctuation is not uncommon the bring attention of the administrator to major errors or warnings. From the NetFlow/IPFIX dataset, there is evidence of this in some logs such as in Figure 5.3-6:



```
reverse mapping checking getaddrinfo for HHHHH [xxx.xxx.xxx.xxx] failed - POSSIBLE
BREAK-IN ATTEMPT!
```

Figure 5.3-6 – Capitalisation of a Log being used for emphasis

Another advantage of using textblob and Vader is that is computationally efficient [75], testing having shown it producing sentiment scores faster than more complex machine learning methods of producing sentiment, without losing accuracy. This speed is noteworthy, since any delays in producing the NHF will inevitably delay the final classification and slow any possible response. Pseudocode for the textblob found in Pseudocode 5.3-5.

```
import mysql.connector
from textblob import TextBlob

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB"
)

mycursor = mydb.cursor(buffered=True)

log = getLogFromDataServer()

blob = TextBlob(log)
sentiment = blob.sentiment.polarity
```

```

query = “UPDATE logTable SET sentiment = sentiment
        WHERE hostIP = hostIP AND service = service;”

mycursor.execute(query)
```

Pseudocode 5.3-5 – Example code for the textblob

It is also possible to obtain the remote IP address from the Log text. This is typically not recorded separately and so does need log analysis to determine the address. Pseudocode for this is found in Pseudocode 5.3-6.

```
import mysql.connector
import re

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB"
)

mycursor = mydb.cursor(buffered=True)

log = getLogFromDataServer()
re.finadall(r'(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})', log)
```

Pseudocode 5.3-6 – Code to determine IP Address from a Log

Where IP is the IP address found within the text of the log. The regex in the final line captures four 1-3 digit inclusive numbers, which are separated by a full stop. The log database now contains the following fields:

- Date
- Time
- Host IP
- Service
- Log Text
- Instances
- Login Success
- Sentiment
- Remote IP Address

The final structure of an NHF can be found in Table 5.3-3:

From both flows and logs – Mandatory for matching				
Date	Time	Host IP	Remote IP	Service
Logs	**Flows**			**Logs**
Instances	Received Packets	Received Bytes	Duration	Sentiment

Table 5.3-3 - Structure of an NHF

5.4 Analysis

The analysis consists of a number of stages, laid out as follows:

- **Aggregation**

  The logs and network flows need to be aggregated into an NHF. This is achieved through matching IP Addresses, services and ports that have been extracted in Section 5.3.1 and Section 5.3.2. Also utilised are two rules that state that the log must have been generated within a 5-minute period of a flow, and that the log must have been generated after a flow. This is because logically, logs must be created after a network flow, however, logs should not be matched to flows that have been idle too long. Because the amount of time between a flow being registered and a log being created can vary depending upon network conditions, server usage, server location within the network and a number of other conditions, a 5-minute grace period was selected. This period was based upon works such as [40] where similar periods have been chosen. Another advantage of this is that it aids in detection and consolidation of micro flows. Micro flows are flows that can last less than a second, and typically contain little data. Due to their small size and duration, micro flows are often not kept on the flow table by SDN controllers very long, but these can still be a sign of malicious activity. By using a process to drop flows that depends solely on the amount of time a flow has been idle it is possible to detect flows that are small and do not last long but can be frequent.

- **Initial Analysis**

  It has been mentioned that there will need to be a way to organize the NHFs into a priority queue. Deep learning is resource intensive, and while the NHFs contain fewer features than many other datasets, some form of sampling is still likely. Here a risk score to evaluate which flows are high risk shall be calculated. This is done as a way to ensure that malicious flows can be analysed at line speed.

- **Initial Classification**

  An initial binary classification of the NHF is made. Separate to the risk analysis made earlier, this uses a deep learning model that produces a classification as to whether the flow is malicious. The machine learning model for this is a deep one and consists of multiple convolutional and dense layers to arrive at the classification. While this is a complex model, its slowness can be compensated for in the risk analysis mentioned earlier, which should ensure that most benign flows are not required to be analysed.

5.4.1 Aggregation

Once features have been extracted from both the logs and the flows, they can be aggregated to create the NHF. From examining the log table and the flow table, it can be seen that there are several features that are shared (IP addresses, times and service). These will be used to match the various features of both logs and flows. Pseudocode for this is shown in Pseudocode 5.4-1.

```
import mysql.connector

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB"
)

mycursor = mydb.cursor(buffered=True)

destIP, sourceIP, time, port = getFlowData()
query = select instances, loginSuccesses, sentiment, FROM
logFlows
         WHERE (destIP == hostIP || destIP == remoteIP) AND
(sourceIP == hostIP || sourceIP == remoteIP) AND
port == service AND
time <= logTime AND
time + datetime.timedelta(minutes=5) >= logtime
```

Pseudocode 5.4-1 – Creation of the NHF

As can be seen, the first two WHERE statements match the IP addresses. The OR statements are used, as a reply from a server will have the source as the server IP, whereas the original message will have the server IP as destination. Marking these separately would result in two separate flows, when they are clearly supposed to be connected. Additionally, on the final line, timedelta is utilised. Logs will be generated after flows inherently, and so included is a requirement that logs must have been generated after the initial flow time. However, logs that are generated hours or even days after the initial flow time should not be grouped together. So, the final line gives a time limit of 5 minutes. If a log is generated more that 5minutes after the initial flow time, then it will be assigned to a new flow. This matches the flow entries, where a flow is dropped from the database if it has not been updated within 5 minutes. A time of 5 minutes has been chosen as this broadly matches the time of many password lockout policies. If a malicious user initiates a brute force or dictionary attack, and the account gets locked, the malicious user would need to wait longer than five minutes for the system to not group the malicious attacks together. More accurately, if the malicious user started the attack at time *s*, and paused after receiving an account locked message at time *t* the amount of time the malicious user would have to wait (*w*) is shown in Equation 5.4-1:

$$w = 5 + (t - s)$$ *Equation 5.4-1*

5.4.2 Initial Analysis

Now that the NHF has been formed, the initial analysis and risk assessment can be implemented. Every flow will need to go through this process, sometimes more than once as flows get updated with new data. As such, the process for this initial analysis needs to be computationally efficient.

Deep learning models are computationally expensive, and this process will be adding more expense on top of that. As such, the first goal for the initial analysis will be to provide a method to decrease processing.

Often, IDS/IPS solutions will require sampling for the solutions to work at line speed. Sampling strategies are a focus of research, however, these have often come across what is ultimately the same issue, that being that less you sample, the more likely that some attacks will not be noticed simply because they are not sampled [76][77]. The NHFs being analysed should be quicker to analyse then individual packets, since the NHFs can be considered an abstract representation of the packets they represent. However, there still needs to be some consideration given to the analysis happening at line speed.

A slight variant to sampling involves creating a “priority list”, whereby rather than packets being sampled randomly, or based upon position in a flow, packets are prioritised for analysis based on some measure. The difference being that if the computational resources are available, then every packet will be analysed, however those that are deemed to be high risk are analysed first. As such, a priority list is implemented. This leads to the second requirement, to provide an indication of the amount of risk the flow contains.

This should not be a definitive sign that a flow is malicious, but that it contains some features that have been shown to be malicious. This stage can be completed with clustering of the flows. Clustering algorithms are unsupervised algorithms that place records into groups depending upon how alike they are. Several clustering algorithms exist; however, HDBScan is chosen. This is because HDBScan has several advantages. These include:

- **Not requiring a set number of groups**

  Several clustering algorithms (such as K-Means) require that the number of clusters be known in advance. This is not known as the data has not been analysed. While datasets such as the NSL-KDD or UNSW-NB15 could have the number of clusters configured to be the number of different classes within the dataset, it should not be assumed that this is known in advance for most intrusion detection systems.

- **Being resilient to noise**

  Another issue with K-means (and other centroid-based clustering algorithms) is that it assumes every record is part of some group, when this may not be the case. Network datasets are inherently noisy, and it is entirely possible to get benign flows that do not seem to be related to any other flow (a benign anomaly). As HDBScan is a density-based approach to clustering, it does not have this issue. Instead, there is are hyper-parameters to be set to control at what point a group of records can be considered a cluster. Any records that appear outside one of these clusters is placed into the -1 category (or simply unclassified).

- **Not assume a Gaussian cluster**

  K-means assumes that clusters are Gaussian in nature and can misclassify based upon this. HDBScan holds no such assumption and can classify clusters of any shape or density distribution.

- **Being efficient**

  The process overall is supposed to ensure that any malicious flows are analysed quickly. If this process is computationally expensive, then this negates the benefits of the system as malicious flows may have stopped before they are detected. While HDBScan is not quite as efficient as K-Means (not achieving O(n log(n)) complexity), it is more efficient than many other clustering algorithms, and has less than O(n^2) complexity [78]. K-means also has an advantage in that increasing the number of features has little effect on the efficiency of the computation. However, this work is inherently limited in the number of available features and as such the advantage of k-means in this regard is of limited use.

Clustering will ideally group malicious flows together, however the goal is not classification but to determine a risk for the various clusters. This allows us to analyse flows in high risk groups first, leaving low risk groups for later should the system become too busy to analyse all the flows. The risk factor can be determined by calculating the proportion of malicious flows within each cluster. If a cluster has an unusually high proportion of malicious flows, then any later flow assigned to it can be assigned a higher risk.

Assuming *M* is a malicious flow, and that a *C* is a cluster with size *S*, then the risk factor *R* can be determined using Equation 5.4-2:

$$R = \frac{\sum_{i=0}^{M \in C} i}{S}$$ *Equation 5.4-2*

This will result in a value between 0 and 1, where 0 is a very low risk, and 1 is very high risk. The process logically requires supervised learning, since flows will have needed to be declared malicious before the risk analysis can be conducted. This feeds into the choice of a supervised model later for the analysis.

Pseudocode for the process can be found in Pseudocode 5.4-2.

```
import hdbscan

groupList = []
var = []
count = 0

def UpdateGroupRisk(group):
  query = SELECT COUNT(*) from flowTable WHERE group = group
  malicious = SELECT COUNT(*) from flowTable WHERE group = group
AND malicious = 1
         mycursor.execute(query)
  number = mycursor.fetchone()
         mycursor.execute(malicious)
  maliciousInGroup = mycursor.fetchone()
         newRisk = maliciousInGroup / number

         query = UPDATE riskTable SET risk = newRisk
                 WHERE group = group

         return

```

```
training, y = getTrainingData()
data = getFlowData()

clusterer = hdbscan.HDBSCAN(min_cluster_size=x, gen_min_span_tree=True,
                            metric='manhattan', min_samples=y, prediction_data=True)

clusterer.fit(training)

for flow in data:
    mark = 0
test_cluster, tested_strength = hdbscan.approximate_predict(clusterer, data)
    for label, y, in zip(clusterer.labels_, y):
        mark = 0
            for cluster in clusterList:
if cluster[0] == y:
cluster[1] = cluster[1] + 1
mark = 1
                    if y == 1
                        cluster[2] = cluster[2] + 1
                if mark == 0:
                    if y == 0:
                        appendList = [cluster, 1, 0]
clusterList.append(appendList)
                    else:
                        appendList = [cluster, 1, 1]
clusterList.append(appendList)


query = "UPDATE flowTable SET group = group[0]
WHERE hostIP = hostIP AND service = service AND time = time;"
    execute.query()

    UpdateGroupRisk(group[0])
```

Pseudocode 5.4-2 – Code to show how risk is created and updated

Within this, min_cluster_size and min_samples are hyper-parameters that need to be configured according to the dataset. min_cluster_size is the amount of records required for a cluster to be considered a cluster. At a min_cluster_size of 1, all records would be clustered, though some would only be clustered with themselves (which logically does not make sense). At values larger than 1, these points would be put into group -1, and considered noise. As min_cluster_size increases, the number of clusters decreases and the volume of records in -1 increases, as smaller clusters no longer reach the requirements to be considered a cluster at all. min_samples is similar, however refers to the number of records within a neighbourhood for a point to be considered a core point.

5.4.3 Initial Classification

After the initial analysis and risk detection, a classification can be made. For this precision is prioritised over recall, as this will reduce the number of false positives that are passed onto the mitigation system.

Ideally both recall and precision will be achieved, but when it is possible to have benign usages blocked or redirected in an automatic fashion, it is believed that minimising the number of these would be preferable. Other false negatives could be caught by a later security process (for example, malware could still be caught through local anti-malware scans, even if it evades detection of its transmission over the local network).

Data preparation will also need to be undertaken. Studies [79][80] have found that good data preparation can impact results significantly. As such, the flows that have been formed will undertake the following steps:

1. **Convert Categorical Data**

   The datasets contain several pieces of categorical data, for example protocol type. This comes in the form of TCP, UDP, ICMP etc. These need to be converted to numerical data, as the model does not recognise string input. This can be done simply by assigning a number to each different element (0 for TCP, 1 for UDP etc.). However, this is not enough by itself. Machine learning can be prone to giving bias to larger numbers [81]. As such, with the procedure set out above, it would give UDP greater weight than TCP, simply because 1 is larger than 0. In order to counter this, any categorical fields that have been converted to an integer then need to be binarised. This takes the values and converts them into a binary vector, e.g. 001 for TCP, 010 for UDP. This has the by-product of increasing the feature space, as instead of 1 column for protocol type, there is now have one column for each protocol.

2. **Scaling**

   Not all the numerical data conforms to the same scales. For example, duration is measured in seconds and rarely lasts longer than a few minutes (or hundreds of seconds). Similarly, the duration of flows can also frequently last less than 1 second, leading to decimal values of less than 1. Port number on the other hand, can extend up to 65535. Additionally, the range from 49152-65535 are officially unassigned, and free for use, which means than when determining which port to contact a client on with a client/server relationship, it will likely be one of those ports. The different scale of these numbers can lead to issues with bias being given to the larger numbers [81][82]. As such Min-Max scaling is employed. This will bring all elements of the sets to within the range 0-1.

3. **Reduce the Feature Space**

   This may at first seem at odds with the rest of the thesis. Much of this paper has so far consisted of gathering additional features to those generated by a SDN controller. However, in step 1 on this process the feature space was increased by binarising categorical features. This leads to the situation where some features are redundant. To take the protocol example again, assuming the dataset only consists of TCP, UDP and ICMP transmissions, then the sum of TCP, UDP and ICMP features will always be 1. Therefore, the ICMP feature could be removed completely, as if the sum of the remaining TCP and UDP columns does not equal 1, then the transmission must have been ICMP. Removing features like this has been shown to not have any significant effect on the model's overall accuracy and can increase the processing speed of the model dramatically [54], [55].

4. **Balance the Training Set**

   Raw network datasets are incredibly imbalanced. The NetFlow/IPFIX dataset contains a 96% to 4% imbalance of benign to anomalous flows for instance. This imbalance is a significant issue and can result in overfitting to the majority class (i.e. the model classifies every record as benign, as this gives greatest accuracy and precision). Balancing the dataset can have a marked improvement on accuracy, through forcing it to learn the minority classes.

More details of each of these steps will be given later in the chapter. The classification itself can then proceed. As discussed, efficiency is important, while steps have been taken earlier in the process to decrease computational complexity, it is still important that the model at this point is efficient. This is the primary reasoning for choosing a CNN, as they have been shown to have an advantage in accuracy over ANNs [16], but for network datasets this advantage is typically small [83]. However, they are also significantly more efficient, requiring fewer parameters to get those results. For example, a convolutional layer with a kernel size of 3 and 16 filters has 160 parameters to compute. The equivalent fully connected layer of 144 has 17,568 parameters [46]. This is also the reason a CNN is used over other RNNs such as LSTM. LSTMs have been shown to be effective in this kind of classification [84], but are more computationally expensive, and CNNs can still get comparable (or better) accuracy [41]. The major disadvantage CNNs face is that it is a supervised learning model. Unfortunately, within the intrusion detection space, large and well-labelled datasets are scarce and require a great deal of expertise to gain. When moving to local networks that will likely require their own custom dataset for a system to work, (since different networks have different architectures and designs) this means that the expertise to create and label a dataset needs to be within the company responsible for maintaining the network. Additionally, as networks develop over time, it is likely that the data used to originally train the network will become outdated, meaning the training process needs to begin again. For this reason, unsupervised methods are generally seen as being a more practical approach, as data does not need to be labelled. However, this lack of labelling does come with drawbacks. Unsupervised models are often less accurate overall than their supervised counterparts. In addition, the proposed solution requires labelled data for the risk analysis stage (even though HDBScan is unsupervised). This is generally going to be true of any system that attempts to sample data based upon risk, as logically you cannot determine a risk level without knowing what kind of activity is risky. As such it is determined that a CNN best suits the requirements for the classification.

The structure of the model is also important. Multiple papers have found it is generally better to have multiple small convolutional layers [85][86][87], rather than a single larger convolutional layer. Two convolutional layers with a kernel of size 3 are equivalent to a single layer of size 5 (each decreasing the feature space by 4), however the two separate layers would generally lead to superior results. It has also been found that pooling, and in particular max pooling can be beneficial in reducing overfitting [88], [89]. However, pooling dramatically reduces the feature space, halving it each time it is used. In most cases this is beneficial, as it reduces the parameters that need to be calculated. However, we have a small feature space to begin with. This limited feature space dramatically decreases the variety of layers available to us within the CNN, and while decreasing overfitting would be beneficial, it needs to be considered within the context of its effect on the model in general. This leads us to conclude that batch normalisation would be the best choice for reducing overfitting within the CNN itself. Batch normalisation was originally thought to decrease overfitting by reducing internal covariate shift, which changes the distribution of inputs of each layer. By reducing the variance of the hidden values, the values of the next layer can be more tightly controlled. There is some minor benefit of regularisation as well. Similar to dropout multiplying some nodes by 0 to drop them from the network, and prevent a network from overusing that node, batch normalisation can have a similar effect, reducing the chance of overuse of a single node and forcing the rest of the network to be used. The theory of reducing internal covariate shift is disputed [90][91], but batch normalisation does aid in reducing overfitting [90][92], as well as offer some minor regularisation effects [89], [93].

Pseudocode for the model can be found in Pseudocode 5.4-3:

```
#Training and testing data are located, for testing data
# this would be new flows or flows that need to be reanalysed
# with the model being called as an object

x_train, y_unsampled = getTrainingData()
x_test = getFlow()
#Scaling is performed on the training data
scaler = MinMaxScaler()
scaler.fit(x_train)
x_train_unsampled = scaler.transform(x_train)

#Encoding is performed on the training data
encoding = OneHotEncoder()

#This allows for additional columns to be added later
cols = ['protocol'']

for n in cols:
    encoding.fit(x_train[:, n])
    x_train[:, n] = encoding.transform(x_train[:, n])

#PCA is performed to removed redundant data and quicken learning
pca = PCA(H)
pca.fit(x_train)

x_train = pca.transform(x_train_unsampled)

#The training data is balanced, this will not be applied to testing
#data at any stage
ada = ADASYN()
X, y = ada.fit_resample(x_train, y_unsampled)

#The model is created using Keras
model = Sequential()

#A single dense layer allows us to know the shape of the data
#after PCA changed it to an unknown shape
model.add(Dense(121, activation='softmax', input_dim=int(inputShape)))

model.add(Reshape((11, 11, 1)))

#Convolutional Layers followed by batch normalaization
model.add(Conv2D(H, (H, H), activation='H'))
model.add(BatchNormalization())

# More layers are added after this initial one
model.add(H Conv2D layers)

model.add(Flatten())
# Output
model.add(Dense(H, activation='H'))

optimizer = H(lr=H)
model.compile(optimizer=optimizer,
              loss='binary_crossentropy',
              metrics=['categorical_accuracy', 'accuracy']
             )
model.fit(X, y, validation=x_unsampled, y_unsampled)

#Testing can be performed, this allows for individual flows to be tested
#as they come into the system
for x in x_test:
  x = scaler.transform(x)
  x = encoding.transform(x)
  x = pca.transform(x)
          prediction = model.predict(x)
```

Pseudocode 5.4-3 – Code for the model, H represents Hyper-parameters to be tuned

As can be seen, the code had space for multiple hyper-parameters (H) that will need to be individually tested and configured. This testing will be explained in the next Chapter (implementation). The following will need to be configured:

1 **PCA components**

As has been explained, a system is required to reduce the data after it has been expanded to remove features that do not aid with classification. PCA has been chosen for this task, as it is efficient [94], [95] and has been shown to be helpful in reducing features and increasing accuracy on IDS datasets in the past [55][96]. This is done so the model is not hindered in attempting to use features that have little effect on the end classification, either because they have high correlation with other features, or contain a low entropy. Previously, in binarising the dataset features that have a high correlation have been explicitly added, and so this step is important in reducing these features and decreasing computational complexity and time.

2 **Convolutional Kernel Size**

The kernel size relates to the size of the convolutional filter that traverses the feature space. These have seen a steady reduction in size over recent years, from 7x7 kernels being common to 2x2 or 3x3 sizes being more common throughout the models, with occasionally 5x5 or 7x7 being used to start the model.

3 **Convolutional Filters**

This indicates the dimensionality of the output space. As more layers are added the feature space decreases, but the output space should increase. This allows prominent features of the input to be recognised in early layers, and later layers to recognise less prominent features.

4 **The Activation Function**

The activation function determines whether a neuron fires (or activates) or not. If the input values to a neuron are above a threshold, then that neuron will activate, if they are not then it will not. The threshold can be thought of as the centre of a curve that varies from function to function. As such, activation functions are important in the make-up of deep learning models. There are a few types of activation functions available, with various advantages and disadvantages. It would also be possible to design a custom activation function, however that is outside the scope of this work. One of the most prominent functions would be sigmoid. Sigmoid tends to be used for binary classification at the end of a model (i.e. end classification), since it produces a steep curve near the centre that will naturally mean most outputs will be pushed to one side or the other, producing clearer results. However, it does suffer from the issue of vanishing gradients at either side of the curve, meaning that learning will slow or stop if used in the middle of a deep model. Tanh has similar issues, being a scaled version for sigmoid.

Rectified Linear Unit (ReLu) has a different approach, it uses a linear activation for values above 0, else 0. It has greater efficiency than both sigmoid or tanh since any value less than 0 is simply made 0. However, this can cause the dying ReLu problem. This is caused when areas of the network have a negative or 0 value, the gradient of the network in that area will become 0, preventing learning. This issue can be aided with a leaky ReLu. Instead of 0, this multiplies the figure by 0.01, ensuring the value is small and close to 0, but not actually 0 itself. This can aid if overfitting is occurring due to the dying ReLu problem. Finally, softmax is another activation function that performs well as an end classifier. Another potential issue of sigmoid is that its outputs are not related to each other. For a single class problem, this is not an issue, however for multi-class problems this can lead to a sum of probabilities becoming greater than 1. With softmax, the outputs sum to 1, which makes sense for multi-class problems where the output must be one of the classes.

5 Learning Rate

The learning rate determines the extent of change in the model, in response to the loss. There is a trade off in learning rate, as higher rates will not see the model converge to the optimum solution, whereas smaller rates will take more epochs to converge. Generally accepted rates are in the range 0.01-0.0001, but this is a parameter that needs to be tuned according to the network, as well as the choice of optimiser.

6 Optimiser

The role of the optimiser is to update the weights and biases within the model in order to reduce the loss. Constant rate optimisers always change the weights and biases by the same amount (the learning rate), stochastic gradient decent falls into this category. Adaptive optimisers change the learning rates on a per-parameter basis, reducing the risk of the issue outlined in point 5, where the model can fail to converge. An example of this is Adagrad, which uses larger updates for infrequent parameters, and smaller ones for more frequent updates. However, Adagrad runs into issues of diminishing learning on more frequently updated parameters, which can cause models to stop learning prematurely. Optimisers such as Adadelta, RMSProp and Adam attempt to fix this. Adam is frequently used in practice, as it is again computationally efficient [97].

With the model completed, a probability for the final classification will have been made. This can be attached to the flow and if it is over a threshold send it to the mitigation system. The probability is included as it gives an indication of the level of certainty that exists about the flow and the requirement to mitigate it. With a high degree of certainty that a flow is malicious, more dramatic mitigation measures can be employed.

A table showing the model dimensions is shown in Table 5.4-1

Layer (Type)	**Output Shape**	**Number of parameters**
dense_1 (Dense)	(121)	8349
reshape_1 (Reshape)	(11, 11, 1)	0
conv2d_1 (Conv2D)	(9, 9, 8)	80
batch_normalization_1 (Batch Normalization)	(9, 9, 8)	32
conv2d_2 (Conv2D)	(7, 7, 16)	1168
batch_normalization_2 (BatchNormalization)	(7, 7, 16)	64
conv2d_3 (Conv2D)	(5, 5, 32)	4640
batch_normalization_3 (Batch Normalization)	(5, 5, 32)	128
conv2d_4 (Conv2D)	(3, 3, 64)	18496
batch_normalization_4 (Batch Normalization)	(3, 3, 64)	256
conv2d_5 (Conv2D)	(1, 1, 128)	73856
batch_normalization_5 (Batch Normalization)	(1, 1, 128)	512
flatten_1 (Flatten)	(128)	0
dense_2 (Dense)	(32)	4128
dense_3 (Dense)	(2)	33

Table 5.4-1 - Table to show Deep Model Layers

5.5 Mitigation

The final mitigation is achieved through classifying malicious flows according to an attack taxonomy. This is achieved using an ensemble of classifiers that classify according to each dimension of the taxonomy. Whilst this may sound especially computationally complex, in reality it is envisioned that this stage will only be used on a small proportion of traffic, as any benign flows should be declared as such, and no mitigation needs to take place. The goal of the taxonomy is to determine the threat the malicious flow poses and recommend a course of action. For flows where the threat is low, no action beyond notification to the administrator is a possible choice. For medium threats, flows can be redirected to protect essential services, whereas high threats can result in the flow being dropped.

The intended taxonomy differs from other taxonomies in that the goal is to formulate an automated response to the attack that is being identified. This means that the branches should be designed in order to provide a solution that can be undertaken. Many taxonomies are designed to separate different attacks, ensuring that each attack type is classified in only one way (the uniqueness of the classification). This is beneficial for administrators seeking to classify an attack in the forensic analysis stage of an attack (i.e. after an attack has happened and the goal is to assess the impact and determine any additional threat), or testing phase (to ensure the system is secure). However, it is not as beneficial for the initial response. For the initial response, it rarely matters if two different attacks are classified a similar way if the response should be the same to both [67]. The target for this taxonomy is also important. Many taxonomies seek to address localhost (e.g. someone performing a password attack by typing in passwords on the computer they seek to gain access to) and physical attacks (e.g. someone physically turning off a computer), whereas this work is strictly concerned with attacks that can be recognised and mitigated through the network. As such, this work is built upon the work of Wu *et al.* [69], Souissi [67] and Fu *et al.* [68]. Wu *et al.* propose a response orientated taxonomy that consists of Source, Technique and Results, proposing that defence can be built from these. Souissi builds upon this and noting that Wu *et al.* acknowledge that blended attacks would pose an issue for their taxonomy propose a solution that includes Source, Vector, Target and Impact. Both papers, however, are for generic attacks that can be carried out on computer networks, including local attacks. As such, inspiration is also taken from Fu *et al.* who designed a taxonomy for routing systems, and use Attack Plane, Vector, Target and Impact. The use of attack plane rather than source makes more sense within the confines of a strictly network attack taxonomy, as while source can refer to internal or external attacks, attack plane allows the system to see exactly what level of the SDN is being used to initiate the attack.

A diagram of the Taxonomy used is shown in Figure 5.5-1.

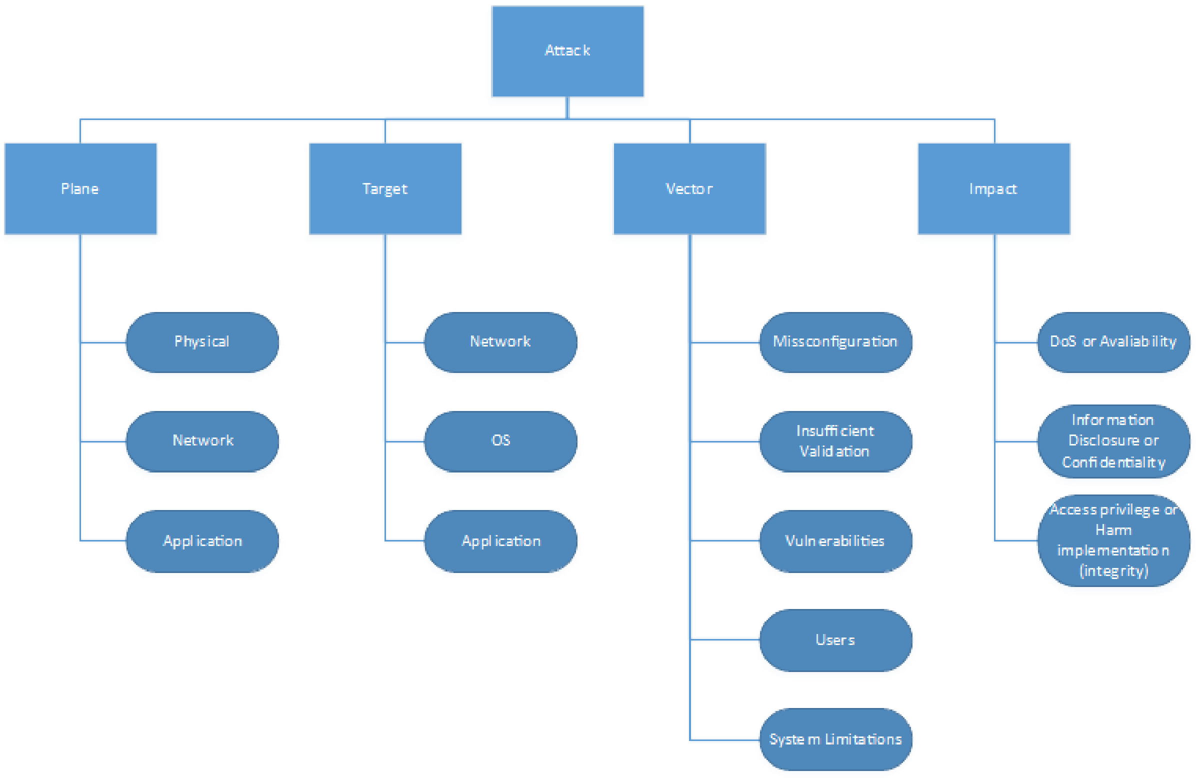


Figure 5.5-1 – The taxonomy used

The four major aspects (Plane, Target, Vector and Impact) of the taxonomy is discussed below:

Plane

Related to the OSI [98] model this refers to the level the attack is taking place on. Fu *et al.* utilize physical, network or application as the attack planes. The physical layer refers to individual devices. Unplugging a cable, turning off a server or using a radio frequency transmitter to interfere with wireless signal would belong on the physical layer. The network layer includes attacks that target network infrastructure, but do not depend on a specific application or OS. An example would be a port scan, or many types of DDoS or flood attacks, where the goal is to overload processing resources, rather than a SYN flood, which requires open ports. The final layer is the application layer, which includes anything that requires an application or OS. The SYN flood mentioned earlier is one example, while another would be a XSS (Cross Site Scripting) attack.

Target

Different from the Attack Plane, this is specifically for what the attack is targeting. For example, a SYN flood specifically targets an OS, while the Plane is the Network (a network vulnerability being used to target a singular device). This contains three levels, as described:

- Network

  This contains attacks such as floods or port scans. These attacks operate on the network level and work irrespective of the OS or Applications on the device or devices being targeted. If something attacks the network, rather than a particular service or OS it should be placed here.

- OS

  This contains attacks such as fingerprinting, or attacks that target specific operating systems. Many (though not all) buffer overflow attacks operate on the OS level for example, as different operating systems may treat malformed network packets in slightly different ways.

- Application

  This contains attacks which specifically target the services being provided (HTTP, FTP etc.). This can involve XSS attacks, or slightly further down the stack buffer overflows that target an SQL server (for instance).

Vector

The Vector consists of what kind of attack is being performed, or what exploits attackers are hoping to take advantage of. Souissi shows this can be separated into six categories:

- Misconfiguration

  This consists of administrators themselves making a mistake in the configuration of a service. The root password being left at default may be an example of this, or configuration files for a web server being left in a publicly accessible location would be another.

- Insufficient Validation

  Insufficient validation is caused by a system failing to validate user input appropriately. Buffer overflows are typically validation errors, as are boundary condition errors or malformed input (such as SQL injection).

- Vulnerabilities

  Vulnerabilities are defined as potential exploits in the software being used. Administrators typically have little control over vulnerabilities, instead requiring the first party supplier to provide patches, fixes, or workarounds.

- Users

  Users is a large source of attack vectors and consists of user error providing the exploit. A user not following password policy may be one example. Another might be a user opening a suspect email attachment, and then agreeing to install the attached software.

- System Limitations

  System Limitation attacks involve taking advantage of the fact that systems do not have unlimited resources. Floods can fit here, taking advantage of the fact that networks do not contain unlimited bandwidth.

Impact

The impact involves assessing what the end effect of the attack would be if it were not mitigated or stopped. Related to the CIA (Confidentiality, Integrity and Availability) model, this contains three categories:

- Confidentiality

  This is used when the confidentiality of data is at risk. The obvious example is leaked usernames and passwords, or card details in an online commerce application with something like a SQL injection attack. However, most probes or fingerprinting attacks also fit into this category, as information about the network structure is leaked.

- Integrity

  These attacks edit or change the data or a service. SQL injection attacks can be placed here as well if the attack is used to insert false data into a database or change records within it, for example, adding an illegitimate username or password to a user credentials database.

- Availability

  These attacks seek to disrupt the availability of data or a service. Most DoS attacks would be placed in here for example, as would the ransomware attacks described in Section 2.1 where the encryption key was deleted after data was encrypted, making recovery almost impossible without access to unencrypted backups. Data destruction, rather than data leakage or modification falls within availability, as do any attacks that disrupt a service.

Using these four aspects, it is possible to determine a SDN mitigation for the malicious activity. Knowing the attack plane means it may be possible to block or redirect the attack at levels 2 or 3 of the OSI model, ensuring that devices are kept safe. Knowing the attack target potentially allows us to refine this further. For example, rather than blocking level 2 or 3 data, if it is determined that an attack is at level 6 or 7, then only that data could be blocked. The attack vector again allows us to refine this further again. For example, in the case of a brute force password attack, the attack would be at the OS or application level, but if it is known that all passwords are secure then the attack might be safely ignored. Finally, the attack impact provides the context required for other actions. If the attack is attempting to target the availability of a service, then quarantining the system (while protecting it from further damage) would be seen as a success from the attacker's point of view. Alternatively, if the target is the integrity of the system, then quarantining it for a short period may be preferable.

This culminates in a requirement to develop a system that can determine these aspects. Again, machine learning can be used in this situation, as it is effectively now a classification problem. Proposed for this is the usage of multiple deep learning models, which feed their answer from one model to the next. The way the four aspects have been arranged, information from each aspect is useful to the classification of the next aspect. For example, if it is determined that the first aspect involves an attack happening in the application plane, this is useful to the second step which can typically eliminate the network from consideration as it would be unusual (though not unheard of) to have an attack target the network through the application plane.

The mitigation actions the system can use in response to a malicious flow also need to be defined. These actions need to be suitable for a range of malicious activity and be implementable through SDN. These potential actions have been identified:

- **Action 1 – Block or Drop Flows**

  Result: Suspicious flows are permanently dropped from all devices.

  Impact: High **–** Used in situations where the integrity or confidentiality of data may be compromised and requires a high degree of certainty.

  Potential Side Effects: This could result in benign flows being permanently dropped or blocked, resulting in an unintended DoS. This is more serious when the flows are integral to the systems being used. For example, if a HTTP server had all its SQL requests to a SQL server blocked, this could result in a website going down.

This is an extreme action, with potentially serious consequences for mislabelled flows. However, this will also ensure that the malicious activity is stopped. Similar to the methods proposed by Yoon *et al.* [60], this would effectively turn every SDN-enabled switch within the network into an inline security appliance, programmed to drop matching flows. This means that the moment the flow is detected it is dropped, rather than having to wait until the flow meets an IPS. This could be used on flows that have a high degree of certainty of being malicious, and are performing malicious activity low down the network stack, which isn't targeting particular machines or services (for example, port scans, or flood DoS attacks.)

- **Action 2 – Block or Drop an IP address or machine**

  Result: A single IP address is no longer able to communicate with the network

  Impact: High - Used in situations where the integrity or confidentiality of data may be compromised and requires a high degree of certainty.

  Potential Side Effects: This could result in devices no longer being able to access network resources, resulting in an inadvertent DoS. For example, a user may not be able to access an FTP site or be able to login to a shared community hub.

While like Action 1, the difference is in the scale. Action 1 will only stop the malicious activity that is being undertaken in relation to a specific flow, while this action prevents an IP address from performing any network activity (even legitimate activity). Rather than an IPS, this action could be thought of more like making the switches a firewall, with a blacklist of banned IP addresses. Again, this is an extreme example, but could be useful in instances where the integrity of the network is at stake.

- **Action 3 – Redirect the flow**

  Result: A flow is redirected to a honeypot or other illegitimate destination

  Impact: High - Used in situations where the integrity or confidentiality of data may be compromised and requires a high degree of certainty.

Potential Side Effects: This could result in a legitimate user receiving illegitimate data as they are redirected to a site that seems like the data being requested. If not redirected to a honeypot, then it could result in an inadvertent DoS.

The flow could be redirected to a honeypot or other secure area of the network. This is potentially useful in cases where the flow is deemed to be attempting to exploit vulnerabilities, this could allow an administrator to monitor the activity and learn of any potential exploits without exposing actual systems to damage. However, this would also likely be a temporary measure while the activity is being analysed, before being blocked completely.

- **Action 4 - Quarantine a system or IP**

  Result: A system or IP is temporarily unable to communicate with the network

  Impact: Moderate – Used in situations where certainty is not as high and includes DoS attacks or potentially high-risk devices.

  Potential Side Effects: Could result in temporary inadvertent DoS.

While superficially like action 2, the difference is in the length of time a system will be blocked, as well as the extent of that blocking. Broadly, this action should be time limited, whereas actions one and two are indefinite and could be permeant (assuming there are no negative consequences). This could be useful in instances where a password attack is deemed to be taking place, slowing the speed of the attack.

- **Action 5 – Quarantine the flow**

  Result: A suspicious flow is temporarily blocked or dropped.

  Impact: Moderate - Used in situations where certainty is not as high and includes DoS attacks or potentially high-risk devices.

  Potential Side Effects: Could result in a temporary DoS of a service if used in a high-risk environment

Rather than blocking an IP address, a flow could be temporarily dropped. Again, while similar to Action 4, the difference is in the scope. If all activity from a service is quarantined then that service would suffer an unintended DoS, however, users would still be able to interact with most network resources.

- **Action 6 – Create a “sinkhole”**

  Result: Responses to a potentially malicious IP are never received by that IP

  Impact: Moderate – Used in situations where certainty is not as high and can include DoS attacks or potentially high-risk devices.

  Potential Side effects: May not guarantee stopping attacks where responses are not required and can result in inadvertent DoS as legitimate users could potentially have replies blocked or dropped.

While the term sinkhole typically refers to DNS applications, SDN can create a similar effect. Any traffic that is being directed towards a particular IP (or set of IPs) could be directed towards a non-existent address (such as 0.0.0.0) and dropped. This technique has proven successful in disrupting the command and control infrastructure (C&C) of botnets in the past. This is different from actions 1 and 2, since the focus is on traffic directed towards a remote IP. That remote IP would still be able to send data into the network but would never receive a response. Due to the fact this would only affect remote IP addresses, it can be considered to have a moderate impact.

- **Action 7 – Redirect through legitimate source**

  Result: Stress on individual devices is lessoned

  Impact: Low – Used in situations where certainty is low, and typically is used for DoS attacks to protect high risk devices.

  Potential Side Effects: Ineffective against access privilege or information disclosure attacks. May cause slightly slower services for legitimate users depending upon network composition.

Rather than redirecting to a honeypot, traffic can be redirected to a different but still legitimate destination. Like load balancing, a SDN could direct suspect traffic through an alternative route, or to secondary servers. These servers and routes would still contain the same data and be subject to the same risks as the primary servers and routes, however in the case of DoS attacks the impact of the secondary servers going down would be lessened. The network impact of this would be low.

- **Do Nothing**

  Result: No action is taken beyond logging and/or alerting an administrator

  Impact: Low - Used in situations where uncertainty about the attack is high, or the impact would be minimal

  Potential Side Effects: Ineffective against legitimate damaging attacks

If the potential impact of the attack is low, and the uncertainty of the attack is high, then potentially the wisest course of action would be to not employ any additional measures beyond informing the administrator of potential malicious activity. This could be used in password-based attacks where it is not certain that an attack is happening (rather than a user being forgetful). The potential impact to the network is low, assuming that password policies are upheld, and the services being targeted have methods to slow this kind of attack (timeout policies, account lockout policies etc.).

The response that will cause the least amount of disruption to the network if a classification is incorrect should also be sought. Now possible responses have been found, the next step is to determine the optimum response. This involves using the certainty of the classification; the lower the classification certainty, the more likely a low impact response will be chosen. These two statements are proportional, so it is possible to just take the certainty provided by the first model and use that as an indication as to which response is correct. The certainty will have come in the range 0.5-1.0 (as below 0.5 the model classifies it as benign) and so Min/Max normalisation is performed to scale these values to the range 0-1.

This can be calculated using Equation 5.5-1:

$$\bar{x} = \frac{x - 0.5}{1 - 0.5}$$ *Equation 5.5-1*

Where *x* is the probability from the first model. x̄ can not become less than 0, as any probability entering this stage will have been at least 0.5. The flow can then be assigned a mitigation according to Table 5.5-1.

	Low certainty (0-0.25)	**Moderate Certainty (0.26-0.75)**	**High Certainty (0.76-1)**
Block Flow			Access Privilege, Information Disclosure
Block IP			Access Privilege, Information Disclosure
Redirect Flow to honeypot			Access Privilege, Information Disclosure
Sinkhole		DoS	DoS
Quarantine IP		Access Privilege, Information Disclosure	
Quarantine Flow	Access Privilege, Information Disclosure	Access Privilege, Information Disclosure	
Redirect to legitimate source	DoS	DoS	
Do nothing	Access Privilege, Information Disclosure or DoS		

Table 5.5-1 – Mitigation according to probability

The certainty ranges were chosen as in the initial analysis we aim to have as few flows classified near 0.5 as possible. All flows reaching this point should have been classified in the initial assessment as being between 0.5-1, and the min-max normalisation will extend that range to 0-1, meaning differences in risk should be greater. This means that malicious flows that fall between 0-0.25 we were initially very unsure were malicious, while flows between 0.76-1 we were very sure were malicious. Risk management may normally use ranges 0-0.1 for low, 0.1-0.5 for medium and 0.5-1 for high, however these ranges are not suitable for us, as we expect many values around 0.5 (which can later be seen in 6.3.1). Ideally, we want the system to either be very sure of the result, or very unsure, so that correct responses can be organised. If the system is only moderately certain, then that could lead to situations where a quarantine is issued instead of a block, or a DoS is redirected to a legitimate source instead of a sinkhole being arranged. As such we use a H-range measurement to help determine whether the system is achieving its goals. Even so, these number could be changed depending upon the level of risk the administrator is willing to take, and different interpretations of these risk probabilities are given in the results.

As can be seen, there are numerous instances where there are multiple possible actions. For example, in the low-risk category there is always the option to do nothing, as well as quarantining or redirection depending upon the kind of attack. Which option is taken will depend upon the vector of the attack. Some vectors will also be higher risk that others. Misconfiguration and Insufficient Validation should be fairly low risk (if the developers and administrators have followed good practice) and the odds of such an attack working should be low. Vulnerabilities and Users indicate a higher risk. If the attacker is attempting to use vulnerabilities it indicates that some preliminary reconnaissance has been carried out and the attacker is now attempting to exploit a known vulnerability. Users have a lower level of expertise than administrators or developers and are more prone to making mistakes. Finally, system limitations are also considered low since the SDN itself should generally be capable of managing itself to aid with managing its resources.

As an example, if an attack is detected to be high certainty, and the impact is determined to be information disclosure while the vector is determined to be vulnerability then the response would be to block the flow (the most extreme action available for the high risk and high certainty attack.) Alternatively, if we were only moderately certain that there was an attack, then the flow would be quarantined instead.

5.6 Models

The deep learning models for the determination of the branches of the taxonomy are similar to those used earlier for the initial classification as it is a similar problem. However, there are of some differences. PCA does not need to be used a second time and using it multiple times on the same data would reduce the useful data being kept. While data is added after each model, the detrimental effect of using PCA outweighs the potential processing gains.

The problem has also shifted from a binary classification problem (i.e. is the flow malicious or not?) to several multiclass problems (i.e. Does the attack target the Network, OS, or Application levels?). This places potentially more emphasis on using softmax as the end classifier, over something like sigmoid. This, combined with the additional data, means that the structure of the models may be different. The training process for the models should also be considered.

Since the models should only experience malicious data, one potential option is to take all the malicious flows from the original training set, configure their outputs and use that data. However, this is not representative of the data that the model will receive. While false positives will be reduced to a minimum, the possibility of false positives still exists. If the process laid out above is used for training data, then these false positives will never be seen by the models. The alternative of this is also true, the model will be trained on true positives that are never seen. This leads to the best solution being that the model is trained on the training data that the initial model produces, even if that data is incorrect. The next step is to create pseudocode for the models that will determine which subcategory of Plane, Target, Vector and Impact the malicious activity falls into. As there are four models, there are also four sets of pseudocode (one for each model) as follows:

```
def plane():
  K.clear_session()
  model = load_model(path)
  tf.Session().as_default()
  with K.get_session().as_default()
         prediction = model.predict(flow)
         return prediction

def vector():
         K.clear_session()
         model = load_model(path)
         tf.Session().as_default()
         with K.get_session().as_default()
                 prediction = model.predict(flow)
                 return prediction

  def target():
         K.clear_session()
         model = load_model(path)
         tf.Session().as_default()
         with K.get_session().as_default()
                 prediction = model.predict(flow)
                 return prediction

  def result():
         K.clear_session()
         model = load_model(path)
         tf.Session().as_default()
         with K.get_session().as_default()
                 prediction = model.predict(flow)
                 return prediction


  x_train, y_unsampled = getTrainingData()
  x_trainResult = getTrainingResult()

  x_test = getFlow()
  x_testResult = getResult()
  X = np.concatenate(x_train, x_trainResult)
  #First model
  model.add(Conv2D(H, (H, H), activation='H'))
  model.add(BatchNormalization())

  # More layers are added after this initial one
  model.add(H Conv2D layers)

  model.add(Flatten())
  # Output
  model.add(Dense(H, activation='H'))

  optimizer = H(lr=H)
  model.compile(optimizer=optimizer,
                loss='categorical_crossentropy',
                metrics=['categorical_accuracy']
               )
  model.fit(X, y, validation=x_unsampled, y_unsampled)

model.save(plane.h5)
```

Pseudocode 5.6-1 – Showing how the first model is set up and how the models are called

Pseudocode 5.6-1 shows how the models are set up and they are called. Note that there are still hyper-parameters that will need to be configured, as with the initial model, and these hyper-parameters fall into the same categories with learning rate, the exact structure being used, and activation. There are also several classes to move from one model to another, pulling the saved weights and structures from training to run the models on demand.

From here the pseudocode is broadly similar, as the changes are in the hyperparameters and the name of the model being called. The basic shape is shown in Pseudocode 5.6-2.

```
1. #Basic model
2.
3. X = np.concatenate(X, previousResult)
4.
5. k.clear_session()
6.
7. model.add(Conv2D(H, (H, H), activation='H'))
8. model.add(BatchNormalization())
9.
10.  # More layers are added after this initial one
11.  model.add(H Conv2D layers)
12.
13.  model.add(Flatten())
14.  # Output
15.  model.add(Dense(H, activation='H'))
16.
17.  optimizer = H(lr=H)
18.  model.compile(optimizer=optimizer,
19.                loss='categorical_crossentropy',
20.                metrics=['categorical_accuracy']
21.               )
22.  model.fit(X, y, validation=x_unsampled, y_unsampled)
23.
24.  model.save(categoryName.h5)
25.
26.
27.  targetResult = model.predict(X)
```

Pseudocode 5.6-2 – Pseudocode to show the basic code structure for the models

In line 3, training results from previous models are concatenated into the training data of the next model. In line 24 the model is saved under the category name, and it is this model that will be restored within the classes of Pseudocode 5.6-1. The same hyperparameters as earlier still need to be configured, though of course this is on a model by model basis and may not be the same for all the models.

The final process is to select a mitigation that matches the results of the models, pseudocode for which can be found in Pseudocode 5.6-3.

```
X_test = np.concatenate(x_test, x_testResult)

y_test = plane(X_test)
X_test = np.concatenate(x_test, y_test)
```

```
Y_test = vector(X_test)

X_test = np.concatenate(x_test, y_test)
Y_test = Target(X_test)
target = Y_Test

X_test = np.concatenate(x_test, y_test)
Y_test =Result(X_test)

certainty = (y_test-0.5)/0.5
action = [1,2,3,4,5,6,7,8]

if y_test == DoS:
        action.drop(1,2,3)
else:
        action.drop(4,7)

certainty = x_testResult
if certainty < 0.25
        action.drop(1,2,3,4)
elif certainty > 0.75
        action.drop(5,6,7,8)
else:
        action.drop(1,2,3,8)

if action.count == 1
        EmployAction(action)
elif (target == vulnerability) || (target == user)
        action = max(action)
        EmployAction(action)
Else
        action = min(action)
        EmployAction(action)
```

Pseudocode 5.6-3 – Showing how the mitigation is chosen

This ultimately ends with one action being selected and undertaken by the SDN, using the code that comes with the SDN controller. This code will vary from controller to controller, but for an OpenFlow controller it would be similar to that shown in Table 5.6-1:

Action	Command
Block Flow	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx, nw_dst=xxx.xxx.xxx.xxx, ,actions=drop
Block IP	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx,actions=drop
Redirect Flow to honeypot	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx, nw_dst=xxx.xxx.xxx.xxx, ,actions=mod_nw_src:xxx.xxx.xxx.xxx
Sinkhole	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx, nw_dst=xxx.xxx.xxx.xxx, ,actions=mod_nw_src:0.0.0.0
Quarantine IP	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx, idle_time-out=Y,actions=drop

Quarantine Flow	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx, nw_dst=xxx.xxx.xxx.xxx, idle_time-out=Y ,actions=drop
Redirect to legitimate source	ovs-ofctl add-flow brX ip nw_src=xxx.xxx.xxx.xxx, nw_dst=xxx.xxx.xxx.xxx, ,actions=mod_nw_src:xxx.xxx.xxx.xxx

Table 5.6-1 – Showing commands that may be issued in response to threats

6 Implementation

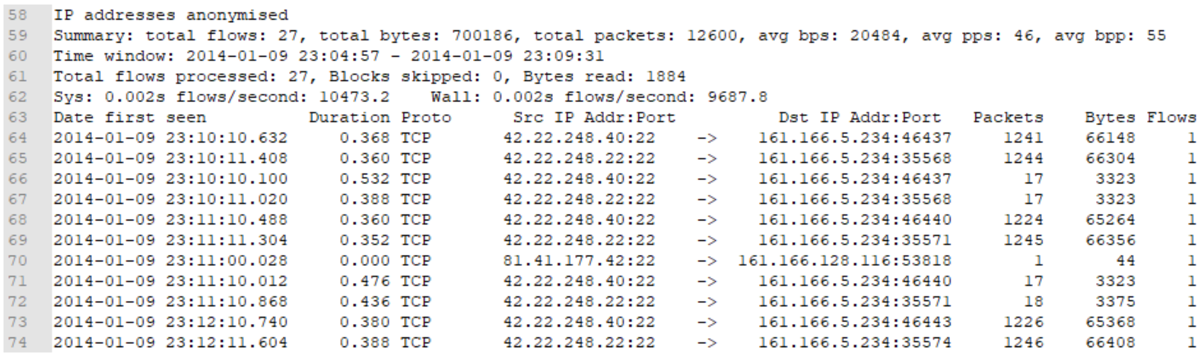
Discussed within this chapter is how the methodology is implemented for testing purposes, including actual code extracts linked to the pseudocode code discussed in previous chapters, as well as discussing any changes that have been made.

6.1 Initialisation and Creation

While Section 5.2 discusses how to potentially obtain the features available, it does not describe how these features will be obtained within this work. While creation of a new dataset is possible, through setting up a SDN (using a tool such as Mininet), this means that it becomes difficult to compare to other works (a problem that has been highlighted in Chapter 2). Instead a combination of the NetFlow/IPFIX dataset and the UNSW-NB15 dataset is used.

6.1.1 The SSH Compromise Detection using Netflow/IPFIX dataset

The NetFlow/IPFIX dataset consists of connection monitoring for multiple SSH servers, which is organised into flows matching the IPFIX standard. The dataset consists of both the flows, and the logs from the SSH servers. The purpose of this dataset is to show process the data goes through, and realistic results that could be obtained. An example of the flows is shown in Figure 6.1-1.



```
58  IP addresses anonymised
59  Summary: total flows: 27, total bytes: 700186, total packets: 12600, avg bps: 20484, avg pps: 46, avg bpp: 55
60  Time window: 2014-01-09 23:04:57 - 2014-01-09 23:09:31
61  Total flows processed: 27, Blocks skipped: 0, Bytes read: 1884
62  Sys: 0.002s flows/second: 10473.2    Wall: 0.002s flows/second: 9687.8
63  Date first seen          Duration Proto      Src IP Addr:Port          Dst IP Addr:Port   Packets    Bytes Flows
64  2014-01-09 23:10:10.632     0.368 TCP       42.22.248.40:22    ->    161.166.5.234:46437     1241    66148     1
65  2014-01-09 23:10:11.408     0.360 TCP       42.22.248.22:22    ->    161.166.5.234:35568     1244    66304     1
66  2014-01-09 23:10:10.100     0.532 TCP       42.22.248.40:22    ->    161.166.5.234:46437       17     3323     1
67  2014-01-09 23:10:11.020     0.388 TCP       42.22.248.22:22    ->    161.166.5.234:35568       17     3323     1
68  2014-01-09 23:11:10.488     0.360 TCP       42.22.248.40:22    ->    161.166.5.234:46440     1224    65264     1
69  2014-01-09 23:11:11.304     0.352 TCP       42.22.248.22:22    ->    161.166.5.234:35571     1245    66356     1
70  2014-01-09 23:11:00.028     0.000 TCP       81.41.177.42:22    ->  161.166.128.116:53818        1       44     1
71  2014-01-09 23:11:10.012     0.476 TCP       42.22.248.40:22    ->    161.166.5.234:46440       17     3323     1
72  2014-01-09 23:11:10.868     0.436 TCP       42.22.248.22:22    ->    161.166.5.234:35571       18     3375     1
73  2014-01-09 23:12:10.740     0.380 TCP       42.22.248.40:22    ->    161.166.5.234:46443     1226    65368     1
74  2014-01-09 23:12:11.604     0.388 TCP       42.22.248.22:22    ->    161.166.5.234:35574     1246    66408     1
```

Figure 6.1-1 – An example of the flows from the NetFlow/IPFIX Dataset

The first step is to clean up the flows and logs, keeping only the useful data. Lines such as those from 58-63 would need to be removed completely since these lines are just metadata about the later flows. This was done with use of a script, shown in Code 6.1-1.

```
#!/bin/bash
FILES=/media/dataset.log
sed -i -e '/^[0-9]/ !d' $FILES
```

Code 6.1-1 – Code to remove extra lines

This simply checks if the line starts with an integer (0-9). If it does, then the line is not deleted, otherwise it is. This way all flows are kept, while the additional lines are deleted. From here, the flow lines themselves need to be organized into a csv with categories previously identified for the flows.

There are no commas included within the flow lines, and so these can be added first.

```
sed -e 's/  */,/g' dataset.log > dataset2.log
```

This converts any amount of blank space into a comma. It also reads the new data into a new file, called dataset2.log. This allows comparing to the original file if needed. Finally, the -> between the two IP addresses can be removed with the use of:

```
sed -e 's/->/ /g' dataset2.log > dataset3.log
```

This results in lines that look like:

2014-01-09,23:11:10.898,0.478,TCP,42.22.248.22:22, ,161.166.5.234:35568,17,66356,1

Additionally separation the port numbers from the IP addresses is necessary, however this cannot be done in the exact same way as previously, as the symbol separating them is a colon. This is also what separates the hour, minute, and seconds within the time field. However, given that every line has the same format, the following code will achieve the desired effect:

```
sed -e 's/://2g' dataset3.log > dataset4.log
```

This will replace every colon after the first two within a line. The following features have now been obtained:

- Date
- Time
- Duration
- Protocol
- Source IP
- Source port
- Destination IP
- Destination Port
- Packets
- Bytes
- Number of Flows

The only differences between Table 5.3-1 and this is that the flows in the NetFlow/IPFIX dataset do not contain a priority field, and instead do contain a “number of flows” entry. This is acceptable since Number of flows was one of the optional fields discussed in Chapter 2. The loss of priority is unfortunate, however this is not considered important, since priority does not relate to the packets themselves, but to how the SDN has determined the priority of the rule for the flow as a whole, and therefore is another derived field.

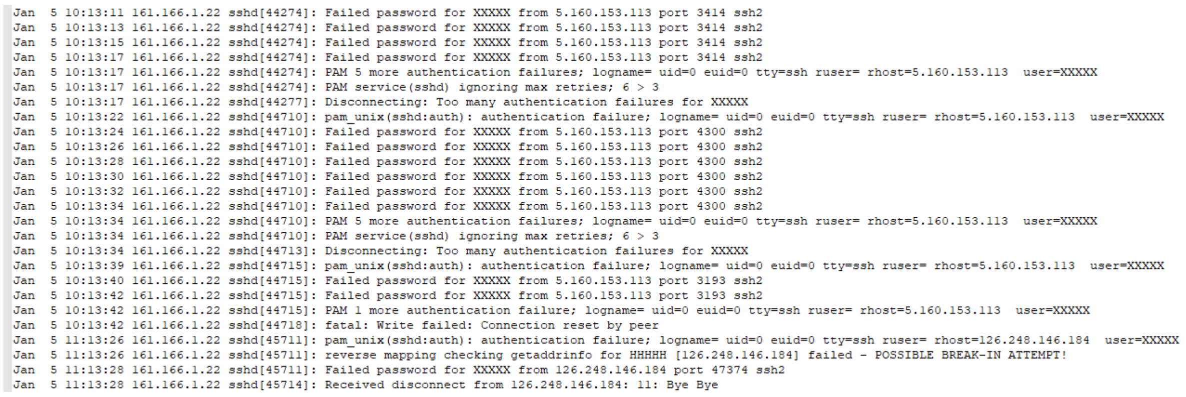
The data is then imported into the database for comparison with log data, as is explained in SectionFlow Features are Extracted 5.3.1. This is accomplished using the script presented in Code 6.1-2.

```
import csv
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="user",
  passwd="password",
  database="flows"
)

mycursor = mydb.cursor(buffered=True)
query="INSERT /*+ append */ into flows VALUES
  ('%s','%s','%s','%s','%s','%s','%s','%s','%s','%s','%s');"

with open(r'dataset4.log') as csv_file:
        reader = csv.reader(csv_file, delimiter=',')
        for row in reader:
                date = row[0]
time = row[1]
duration = row[2]
protocol = row[3]
sourceIP = row[4]
sourcePort = row[5]
destinationIP = row[7]
destinationPort = row[8]
packets = row[9]
bytes = row[10]
flows  = row[11]
                mycursor.execute(query%(date, time, duration,
  protocol, sourceIP, sourcePort,
destinationIP, destinationPort, packets, bytes, flows))
                mydb.commit()
print("Done")
```

Code 6.1-2 – Code to show import of flows to SQL DB

It should be noted that row[6] is not imported. This is because that row is empty. In the final example flow given above, it can be seen that there is a gap between the source and destination IPs, where the -> symbol was. This is still surrounded by two commas, and so is imported as a blank cell by the CSV reader.

The next process is the aggregation of the log files; an example of the log files provided is shown in Figure 6.1-2.



```
Jan  5 10:13:11 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:13 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:15 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:17 161.166.1.22 sshd[44274]: Failed password for XXXXX from 5.160.153.113 port 3414 ssh2
Jan  5 10:13:17 161.166.1.22 sshd[44274]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:17 161.166.1.22 sshd[44274]: PAM service(sshd) ignoring max retries; 6 > 3
Jan  5 10:13:17 161.166.1.22 sshd[44277]: Disconnecting: Too many authentication failures for XXXXX
Jan  5 10:13:22 161.166.1.22 sshd[44710]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:24 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:26 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:28 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:30 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:32 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:34 161.166.1.22 sshd[44710]: Failed password for XXXXX from 5.160.153.113 port 4300 ssh2
Jan  5 10:13:34 161.166.1.22 sshd[44710]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:34 161.166.1.22 sshd[44710]: PAM service(sshd) ignoring max retries; 6 > 3
Jan  5 10:13:34 161.166.1.22 sshd[44713]: Disconnecting: Too many authentication failures for XXXXX
Jan  5 10:13:39 161.166.1.22 sshd[44715]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:40 161.166.1.22 sshd[44715]: Failed password for XXXXX from 5.160.153.113 port 3193 ssh2
Jan  5 10:13:42 161.166.1.22 sshd[44715]: Failed password for XXXXX from 5.160.153.113 port 3193 ssh2
Jan  5 10:13:42 161.166.1.22 sshd[44715]: PAM 1 more authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=5.160.153.113  user=XXXXX
Jan  5 10:13:42 161.166.1.22 sshd[44718]: fatal: Write failed: Connection reset by peer
Jan  5 11:13:26 161.166.1.22 sshd[45711]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=126.248.146.184  user=XXXXX
Jan  5 11:13:26 161.166.1.22 sshd[45711]: reverse mapping checking getaddrinfo for HHHHH [126.248.146.184] failed - POSSIBLE BREAK-IN ATTEMPT!
Jan  5 11:13:28 161.166.1.22 sshd[45711]: Failed password for XXXXX from 126.248.146.184 port 47374 ssh2
Jan  5 11:13:28 161.166.1.22 sshd[45714]: Received disconnect from 126.248.146.184: 11: Bye Bye
```

Figure 6.1-2 – An example of the log file from the NetFlow/IPFIX Dataset

Again, some clean-up is needed before the lines can be incorporated into the database. Similar sed commands as earlier are used, however this time there is no need to delete any lines, as each line is an addition to the log. The first step is to remove some hidden characters (in particular a ^A character which normally results from a Control +A key combination in a console) with:

sed -i -e 's/ //g' logs.log > logs2.log

Then double spaces are removed from the file with:

sed -i -e 's/ */ /g' log2.log > log3.log

There are no headers or footers, and so the data can now be converted into a CSV format. This is done by replacing the 2nd, 3rd, 4th and 5th spaces with a comma, as shown below:

sed -e 's/ */,/2' log3.log > log4.log

sed -e 's/ */,/3' log4.log > log5.log

sed -e 's/ */,/4' log5.log > log6.log

sed -e 's/ */,/5' log6.log > log7.log

The following fields have been obtained:

- Date
- Time
- Server IP
- Service
- Log text

This again closely matches the listed available features in Section 5.3.2, only missing the derived fields of Instances, Login Successes, Sentiment and Remote IP. However, these steps are accomplished within the database. As such, the logs can be uploaded to the server using code like that of Code 6.1-2. The missing features can be obtained using the pseudocode throughout Section 5.3.2, and are explained as follows:

Remote IP Address

The remote IP address can be obtained using the *Pseudocode 5.3-6*, and in particular the string:

SELECT * FROM logs WHERE logText REGEXP '[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}+$';

However it will only obtain all the records that include a remote IP, in effect becoming the query required for getLogFromDataServer() in Pseudocode 5.3-6. The full code is found in Code 6.1-3:

```
import mysql.connector
import re

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB"
)

mycursor = mydb.cursor(buffered=True)

query = SELECT * FROM logs WHERE logText REGEXP '[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}+$';

result = mycursor.execute(query)
mydb.commit()

for text in result:
ip = re.findall(r'(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})', text)
query2 = INSERT INTO logs (rip) VALUES(ip) WHERE id = id
mycursor.execute(query2)
mydb.commit()
```

Code 6.1-3 – Getting remote IP Address

Instances

Instances becomes easier to check within the SQL database, and can be obtained using the code:

`SELECT count(*) FROM logs WHERE hostIP = hostIP AND service = service AND time>= time AND date = date AND remoteIP = remoteIP;`

Full code can be found in Code 6.1-4:

```
import mysql.connector
import re

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB"
)

mycursor = mydb.cursor(buffered=True)

query = SELECT count(*) FROM logs WHERE hostIP = hostIP AND service = service AND time>= time AND date = date AND remoteIP = remoteIP;

result = mycursor.fetchall(mycursor.execute(query))
mydb.commit()

for row in result:
        id = row[0]
        number = row[1]
query2 = INSERT INTO logs (instances) VALUES(number) WHERE id = id
mycursor.execute(query2)
mydb.commit()
```

Code 6.1-4 – Check Instances within SQL Database

Login Success

Again, the pseudocode from Section 5.3.1 is used, however the full SQL command can be substituted in, and the full code is shown in Code 6.1-5.

```
import mysql.connector
import re

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB")

mycursor = mydb.cursor(buffered=True)

query = SELECT id, text FROM logs;

result = mycursor.fetchall(mycursor.execute(query))
mydb.commit()

for row in result:
        id = row[0]
        text = row[1]
        password = 0
        Not = 1
x = text.split()
for word in x:
                if (word == "not") or (word == "Failed"):
                        Not = 1
                elif (word == "password"):
                        password = 1

if (password == 1):
                loginAttempt = 1
if (Not == 1):
                loginAttempt = loginAttempt * -1

query2 = INSERT INTO logs (loginAttempt) VALUES(loginAttempt) WHERE id = id
mycursor.execute(query2)
mydb.commit()
```

Code 6.1-5 – Get Login Successes

Seen is that the changes for the "not" have been included, so if a failed login occurs, the result should be -1 in the loginAttempt column. Additionally, if a password attempt has not been made then the result will be 0, and if one succeeded the result will be 1. The word "not" could be expanded to additional words that could indicate a failed password attempt (for instance, failed), however this is sufficient for the logs used.

Sentiment

The sentiment analysis can be completed in the same manner, with the final code being provided in Code 6.1-6:

```
import mysql.connector
import re

```

```
mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB")

mycursor = mydb.cursor(buffered=True)

query = SELECT id, text FROM logs;

result = mycursor.fetchall(mycursor.execute(query))
mydb.commit()

for row in result:
        id = row[0]
        text = row[1]
        blob = TextBlob(text)
sentiment = blob.sentiment.polarity

query2 = INSERT INTO logs (sentiment) VALUES(sentiment) WHERE id = id
mycursor.execute(query2)
mydb.commit()
```

Code 6.1-6 – Get Login Successes

Each of the steps described above was performed individually, and so the code is as written.

The dataset is not labelled, so the next step is to label the dataset appropriately. While the dataset provided is not labelled, the authors do provide details of what they consider to be an attack. They consider any flow that has six unsuccessful login attempts and has no idle period of more than one hour to be an attack. This requires comparing the flows to the logs, since the logs are the only way to confirm a login. The method used for this is described below.

Compared is the logs and flows using Remote IP address, time, and date. The flow time and log time are by necessity going to be slightly different. This is because the flow will always be generated before the log. Additionally, it is possible that a single flow can generate multiple logs. Indeed, Hofstede *et al.* [99] explicitly state that a single flow can result in three failed password logs, before the flow is reset and a new connection needs to be made. The SQL command used for this is:

```
SELECT * FROM logs WHERE logRemoteIP = flowRemoteIP AND logTime > flowTime – 5mins
AND logTime < flowtime
```

A timer of five minutes was chosen as this matches the account lockout duration of many security policies, and if connection attempts are still being registered after an account has been locked out this can be a sign of automated malicious activity (a brute force or password attack). After the process has finished, the completed NHFs contain:

- Date
- Time
- Duration
- Protocol

- Source IP
- Source Port
- Destination IP
- Destination Port
- Packets
- Bytes
- Flows
- Instances
- Login Success
- Sentiment

This comes from the code found in Code 6.1-7:

```
import mysql.connector
import re

mydb = mysql.connector.connect(
  host=host,
  user=user,
  passwd=password,
  database="logDB")

mycursor = mydb.cursor(buffered=True)

query = select * FROM flows;
query2 = SELECT instances, loginsuccess, sentiment FROM logs WHERE remoteIP = flowRemoteIP AND time > flowTime - 5mins AND time < flowTime AND date = flowDate;

result = mycursor.fetchall(mycursor.execute(query))
mydb.commit()

for row in result:
        flows = row[10]
        bytes = row[9]
        packets = row[8]
        dport = row[7]
        destip = row[6]
        sport = row[5]
        flowRemoteIP = row[4]
        protocol = row[3]
        duration = row[2]
         flowTime = row[1]
        flowDate = row[0]

results2 = mycursor.fetchall(mycursor.execute(query2))
for rows in results2:
        count = count + 1
        instances = rows[0] + instances
        sentimentsum = rows[2] + sentimentsum
        sentiment = sentimentsum / count
        if count > 6:
                attack = 1
```

Code 6.1-7 – Getting the Network Health Flows

The flows that can be used for the analysis are complete. The attack column is removed for most of this process and is only used as the Y value in any supervised training, and for analysing results at the end of the process.

6.1.2 The UNSW-NB15 Dataset

As has been explained in Section 2.6, the prepared UNSW-NB15 dataset contains 43 features and 2 labels. These are shown in Table 6.1-1.

Feature Number	**Feature Name**	**Feature Number**	**Feature Name**	**Feature Number**	**Feature Name**
1	id	**16**	dloss	**31**	response_body_len
2	dur	**17**	sinpkt	**32**	ct_srv_src
3	proto	**18**	dinpkt	**33**	ct_state_ttl
4	service	**19**	sjit	**34**	ct_dst_ltm
5	state	**20**	djit	**35**	ct_src_dport_ltm
6	spkts	**21**	swin	**36**	ct_dst_sport_ltm
7	dpkts	**22**	stcpb	**37**	ct_dst_src_ltm
8	sbytes	**23**	dtcpb	**38**	is_ftp_login
9	dbytes	**24**	dwin	**39**	ct_ftp_cmd
10	rate	**25**	tcprtt	**40**	ct_flw_http_mthd
11	sttl	**26**	synack	**41**	ct_src_ltm
12	dttl	**27**	ackdat	**42**	ct_srv_dst
13	sload	**28**	smean	**43**	is_sm_ips_ports
14	dload	**29**	dmean	**44**	attack_cat
15	sloss	**30**	trans_depth	**45**	label

Table 6.1-1 – List of features available in the UNSW-NB15 Dataset

However, most of these features are not available. There are many similarities to the features obtained from the Netflow/IPFIX dataset, however. The matched features are shown below:

1. Duration (Entry 2 – Dur)
2. Protocol (Entry 3 – Proto)
3. Packets (Entry 6 spkts and entry 7 dpkts)
4. Bytes (Entry 8 sbytes and entry 9 dbytes)
5. Login Success (Entry 38 Is_ftp_login)

Several other features can be used, using the same methodology as is described in Section 5.3, these include:

6. Source packets retransmitted or lost (Entry 15 sloss)
7. Destination packets retransmitted or lost (Entry 16 dloss)
8. Source bits per second (Entry 13 sload)
9. Destination bits per second (Entry 14 dload)
10. Number of flows that has a command in the ftp session (Entry 39 ct_ftp_cmd)

11. Number of flows that has methods such as Get and Post in http service (Entry 40 ct_flw_http_mthd)
12. If source equals destination IP addresses and port numbers are equal, this variable takes value 1 else 0 (Entry 43 is_sm_ips_ports)

Entries 6 and 7 are typically derivable through looking for flow entries with the same ACK numbers within a single flow. This is something many SDN solutions will keep track of regardless, since retransmitted packets can be a sign that a network is reaching capacity and starting to drop flows. The optional counters Receive Drops and Transmit Drops within the SDN specification track the same sort of activity, except they count a switch dropping packets (which would inevitably lead to retransmitted ACKS).

Entries 8 and 9 are extremely easily derivable from the data already collected, simply being a division of bytes by the duration of the flow, both of which are required by the OpenFlow specification. The same is true for entry 12, which compares the source and destination IP addresses and ports, and if they are equal assigns a 1.

Entries 10 and 11 can be determined in a similar manner to that of the Login Success, which is explained in Section 5.3.2. The change is that instead of searching for the word "Success" or "failure" you instead search for logs containing the HTTP or FTP commands you are interested in tracking. This adds load to the log processing sections of the system, but this is the area least concerned about load.

6.1.3 Comparison of Both Datasets

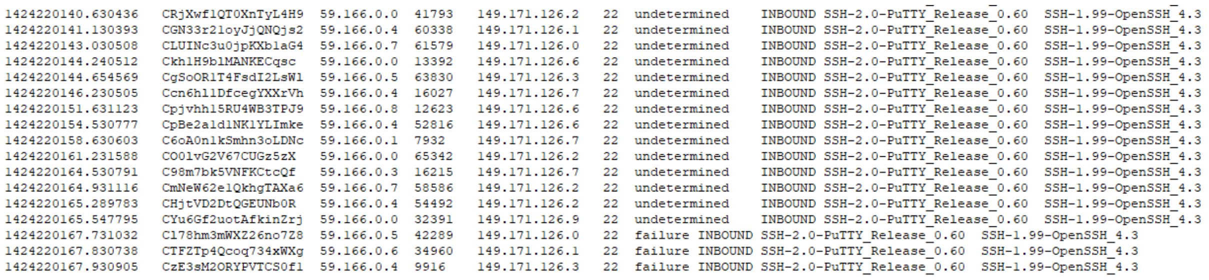
The features that have been determined as available from both datasets are shown in Table 6.1-2.

NetFlow/IPFIX from the University of Twenté		**UNSW NB-15**	
1	Date	**1**	
2	Time	**2**	
3	Duration	**3**	Duration
4	Protocol	**4**	Protocol
5	Source IP	**5**	Source retransmitted bits per second
6	Source Port	**6**	Destination retransmitted bits per second
7	Destination IP	**7**	Source bits per second
8	Destination Port	**8**	Destination bits per second

9	Packets	**9**	Packets
10	Bytes	**10**	Bytes
11	Flows	**11**	Number of flows that has a command in the ftp session
12	Instances	**12**	Number of flows that has a command in the HTTP session
13	Login Success	**13**	Login Success
14	Sentiment	**14**	Are source and destination IP address and ports equal

Table 6.1-2 – Table comparing features of the Datasets Used

It can be seen the datasets contain different features. However, this is typical of real-world situations. Networks are diverse, and very few have the same SDN configuration, let alone server configurations. However, all the features can be found using the methodology shown in Chapter 5. The most important difference is the lack of a sentiment score within the UNSW-NB15 dataset. This is because the pre-prepared version of the dataset does not come with logs to analyse in the same way as the NetFlow/IPFIX dataset. The full version does have Bro IDS logs, however, these logs do not contain log text to analyse in the same way sentiment analysis requires. Instead, Bro-IDS logs consist of listing requests and the server result of that request. For instance, in Figure 6.1-3 the result for inbound requests on the SSH server can be found. The "log text" for this would simply be "Failure", "Success" or "Undetermined" which allows for simpler methods to be used to be understood.



```
1424220140.630436   CRjXwf1QT0XnTyL4H9  59.166.0.0  41793   149.171.126.2   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220141.130393   CGN33r2loyJjQNQjs2  59.166.0.4  60338   149.171.126.1   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220143.030508   CLUINc3u0jpKXblaG4  59.166.0.7  61579   149.171.126.0   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220144.240512   CkhlH9blMANKECqsc   59.166.0.0  13392   149.171.126.6   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220144.654569   CgSoOR1T4FsdI2LsW1  59.166.0.5  63830   149.171.126.3   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220146.230505   Ccn6hl1DfcegYXXrVh  59.166.0.4  16027   149.171.126.7   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220151.631123   Cpjvhh15RU4WB3TPJ9  59.166.0.8  12623   149.171.126.6   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220154.530777   CpBe2ald1NK1YLImke  59.166.0.4  52816   149.171.126.6   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220158.630603   C6oA0n1kSmhn3oLDNc  59.166.0.1  7932    149.171.126.7   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220161.231588   CO01vG2V67CUGz5zX   59.166.0.0  65342   149.171.126.2   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220164.530791   C98m7bk5VNFKCtcQf   59.166.0.3  16215   149.171.126.7   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220164.931116   CmNeW62e1QkhgTAXa6  59.166.0.7  58586   149.171.126.2   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220165.289783   CHjtVD2DtQGEUNb0R   59.166.0.4  54492   149.171.126.2   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220165.547795   CYu6Gf2uotAfkinZrj  59.166.0.0  32391   149.171.126.9   22  undetermined    INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220167.731032   C178hm3mWXZ26no7Z8  59.166.0.5  42289   149.171.126.0   22  failure INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220167.830738   CTFZTp4Qcoq734xWXg  59.166.0.6  34960   149.171.126.1   22  failure INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
1424220167.930905   CzE3sM2ORYPVTCS0f1  59.166.0.4  9916    149.171.126.3   22  failure INBOUND SSH-2.0-PuTTY_Release_0.60  SSH-1.99-OpenSSH_4.3
```

Figure 6.1-3 – Bro IDS logs from the UNSW-NB15 Dataset for SSH connections

The FTP and HTTP logs are much the same, except they also list the command being used (for example, "RETR README.txt - 226 Transfer complete" for FTP logs)

However, it does highlight the ease of determining the counts that have been used within the dataset. After comparing time and IP addresses from the flows and logs, and on a match add one to the corresponding count in the log table.

Ultimately, depending on the logs being generated, sentiment analysis may not be required or suitable and exactly what is suitable will vary from server to server. It is believed that the sentiment analysis makes sense for the Kippo/Open SSH logs for the NetFlow/IPFIX dataset, but not for the UNSW-NB15 dataset. With the different features it becomes obvious that the datasets are not directly comparable. However, the reasoning behind using both datasets is different. The NetFlow/IPFIX dataset advantage is that it does have both logs and flows and so the full process can be followed from start to finish. However, it is not as well known or used, and so comparisons to other works are difficult. The UNSW-NB15 dataset is better known and understood. Additionally, direct comparisons to other works can be made as no additional unusual features are being used. Finally, the NetFlow/IPFIX dataset is for SSH connections only, while the UNSW-NB15 dataset contains a mix of services. This allows for examination of how the system works when moved over to services outside SSH.

6.2 Extraction

From here the data preparation needs to be performed. The first step is to convert any text data into numerical data. This is performed through the use of the LabelEncoder() function within sklearn. This encodes strings into an integer between 0 and n_classes-1. So, for a dataset that contains TCP, UDP, ICMP and ARP, the values would be converted into 0, 1, 2 and 3. Within the dataset only protocol needs to be converted this way. The code for this is shown in Code 6.2-1:

```
from sklearn.preprocessing import LabelEncoder
encoding = LabelEncoder()
i = ['proto']

for n in i:
    print("Processing column", n)
    encoding.fit(X_train[:, n])
    X_train[:, n] = encoding.transform(X_train[:, n])
    X_test[:, n] = encoding.transform(X_test[:, n])
```

Code 6.2-1 – Code to shown LabelEncoder Function

Where X_train is the training set, and X_test is the testing set. However, this method potentially introduces bias within the process. The implication is that protocol is ordinal, and so that larger values are “more significant” than smaller ones. In practice, this is not true. The data is therefore transformed once more using OneHotEncoding(). This converts the integer into a binary vector. Using the example of the four protocols above, this means there would be 0,0,0,1 – 0,0,1,0 – 0,1,0,0 and 1,0,0,0. Each new column refers to one of the protocols, and these in practice become new features. The code for this is shown in Code 6.2-2:

```
from sklearn.preprocessing import LabelEncoder
encoding = OneHotEncoder()
i = ['proto']

for n in i:
    print("Processing column", n)
    encoding.fit(X_train[:, n])
    X_train[:, n] = encoding.transform(X_train[:, n])
    X_test[:, n] = encoding.transform(X_test[:, n])
```

Code 6.2-2 – Code to show OneHotEncoder

Recently, sklearn has allowed OneHotEncoder to take strings as inputs, however that was not the case when this code was originally developed, and the data need to be converted into numerical form before One Hot Encoding could be applied. As such the only code needed now is the code shown in Code 6.2-2 for the OneHotEncoder.

The data will then need to be scaled. Later processes can show bias towards larger numbers, deciding that they are more significant. Again, this is not necessarily true. As such, the data will be transformed using Min-Max scaling. The process for this is carried out by the sklearn MinMaxScaler() function, with the code very similar to the above encoding functions and is shown in Code 6.2-3:

```
scaler = MinMaxScaler()
scaler.fit(X_train)
training_data = scaler.transform(X_train)
testing_data = scaler.transform(X_test)
```

Code 6.2-3 – Code for the MinMaxScaler

This will fit the data according to the training data. If values in the testing data are larger than those supplied within the training data appear, then this will mean that those values will appear as over 1 with the testing data. Similarly, smaller values in the testing data then those that appear in the training data will become negative values. However, this is not considered a significant disadvantage, since these values are statistically abnormal, and so appearing outside the normal 0-1 range would emphasise this discrepancy.

The code for the functions in this section does not change for the different datasets.

6.3 Analysis

While this Section does consist of aggregation, initial analysis, and initial classification, in practice the aggregation for the datasets has been completed in the previous step. This means this subsection will consist of initial analysis and initial classification, for which the primary steps to be discussed are the way the values for the hyper-parameters for the models discussed in Section 5.4.2 and Section 5.4.3 have been determined.

6.3.1 Initial Analysis

The goal of this assessment is to gain an initial idea as to the category of risk the flows contain. In Section 5.4.2 it was discussed how this would take the form of a clustering algorithm that creates a risk score based upon the ratio of benign to malicious flows within each cluster. As the goal is ultimately to create a risk score for the clusters, metrics such as accuracy are not the best indication of success. It is not significant if occasional benign samples are mixed with malicious samples, so long as the mixing of benign and malicious is not equal (i.e. clusters of benign and malicious samples evenly mixed). The goal is to achieve two sets of groups of clusters (or two sets of super clusters, i.e. clusters of clusters), one with as high to a value of 1 as possible, and one with a value as close to 0 as possible as this gives the best indication as to whether flows are malicious or benign. If placed on a graph with risk score as the X axis, and number of samples in the Y axis, this should result in a graph that resembles an y^2 + 0.5 line, so that when x reaches 0.5 reaches y reaches 0, where y is the number of samples.

Additionally, minimising the number of groups would be beneficial, as too many groups defeat the purpose of being efficient. As discussed, there are 2 hyper-parameters to be configured, min_cluster_size and min_samples. Both values can be increased to increase the number of records that are clustered into a specific group.

HDBScan works through plotting the records onto an n-dimensional graph (where n is the number of features). It then groups together records depending upon their distance from other clusters. As stated in chapter 3, for a cluster to be defined, there must be at least n records, where n is the min_cluster_size. Single linkage splits that contain fewer points than this will be considered noise, rather than a separate cluster. Additionally, min_samples refers to the number of samples within a neighbourhood for a point to be considered a core point. The effect of increasing these is to make more points be considered noise, or to be put into the -1 cluster. These points have little relation to other core points, either not having enough points close enough to be considered a cluster by themselves, or not having enough points close enough together for a core point (and therefore a cluster) to be defined. In this sense, min_cluster_size can be considered to be the absolute number of points required for a cluster to exist, while min_samples can be considered the minimum density of the points required for a cluster centre to be defined.

There is a balance between having values too high and too low for both values. If values are too low then there will not be enough clusters, as clusters will be merged into each other, leading to the risk scores tending towards 0.5. Alternatively, if the values are too high than too many points will be regarded as noise.

While testing started with low values for both min_cluster_size and min_samples and increased them until the -1 cluster started to increase in size significantly. This led to an optimum min_cluster size of 2000, and a min_sample size of 400, resulting in the risk score to cluster size graph found in Figure 6.3-1:

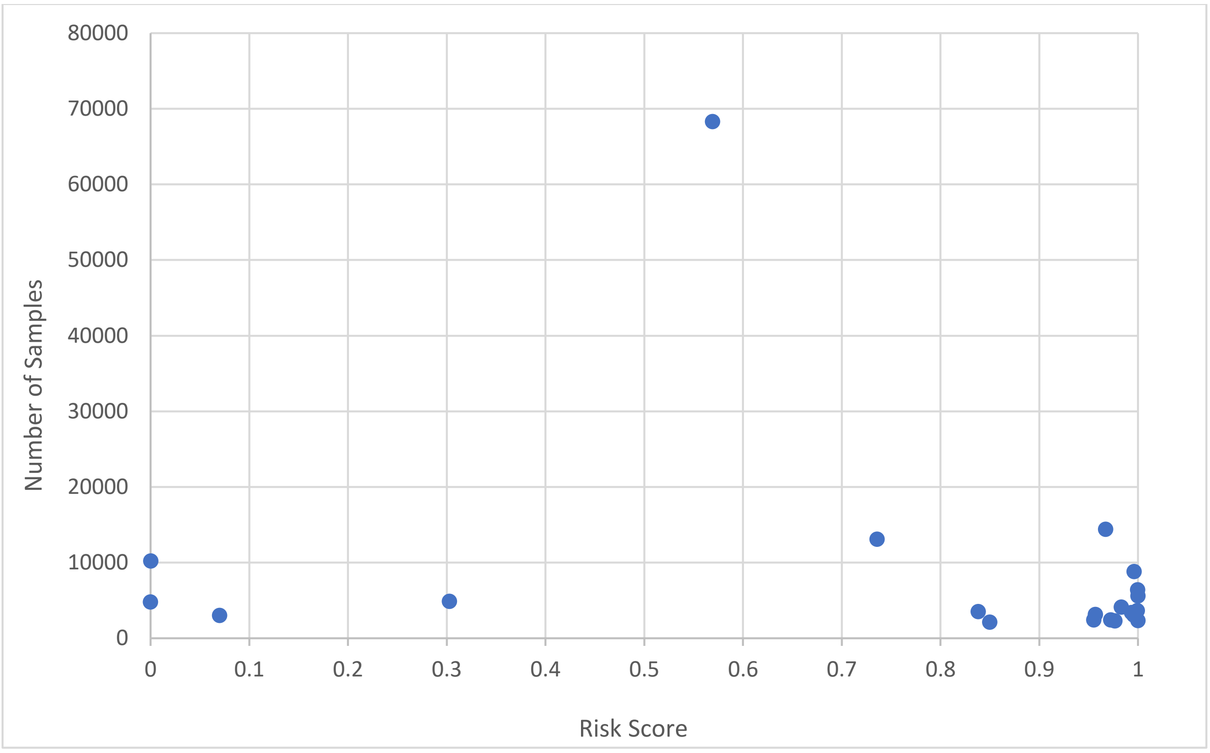


Figure 6.3-1 – Risk score against size of clusters for Min_cluster size of 2000 and min_sample size of 400 for the UNSW-NB15 dataset

The only cluster between the risk score of 0.4 and 0.6 is the unclustered group -1. Additionally, there is only one group between the risk score of 0.3-0.7, and this is a borderline line case with a probability of 0.30273. Five groups have between 2000-2500 samples, and seven have between 3000-4000, with the largest of these being 3658. This means 12 of the 21 groups have less than 4000 samples. If the size of the hyper-parameters is increased beyond this, the size of the -1 cluster increases significantly as the five smaller clusters become unclustered. Smaller values start to have multiple clusters not be well defined in terms of their risk score, with scores between 0.4 and 0.6, additionally it also results in more clusters being created, which adds computational complexity in a system designed to reduce it.

The next step is to ensure that the clustering is effective over multiple datasets using the same hyper-parameters, and so the results for the NetFlow/IPFIX dataset in Figure 6.3-2:

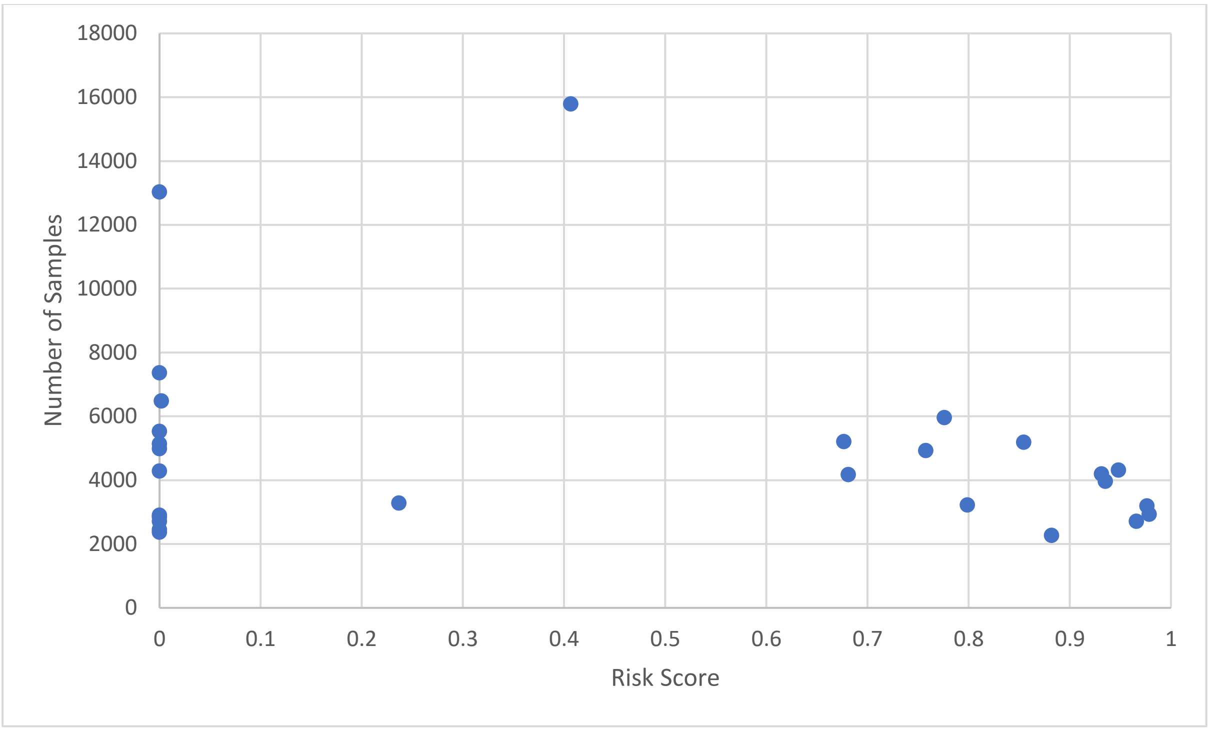


Figure 6.3-2 – Risk score against size of clusters for Min_cluster size of 2000 and min_sample size of 400 for the NetFlow/IPFIX dataset

The same pattern as with the UNSW-NB15 dataset emerges. The only cluster to have a risk score between 0.4-0.6 is the -1 grouping, which has a risk score of 0.4067. Additionally, the only clusters between 0.3-0.7 are two clusters with risk scores 0.67 and 0.68. These indicate good probabilities of any flows being assigned as being benign or malicious, respectively. There is a high number of completely benign clusters, which will significantly aid in reducing the number of flows that require analysis, however the malicious clusters are not as focused as in Figure 6.3-1. In practice, the only difference should be that there is not as high a degree of certainty moving into the Initial Classification stage, as any flows with a risk score above 0.7 should be being analysed with a high priority. With the -1 group score being so low, it is possible there is scope to tune these results further, by increasing the hyper-parameter values, however, it is considered that these are sufficient to proceed with and doing so allows us to keep the hyper-parameters the same between the UNSW-NB15 and NetFlow/IPFIX datasets. As such the final hyper-parameters chosen are a min_cluster size of 2000 and min_sample size of 400.

During this run, it took approximately 7mins to fit the UNSW-NB15 training data using an Intel Core I7-3770 clocked at 3.4GHz, and another 3mins to analyse the testing dataset.

6.3.2 Initial Classification

The first step is the feature reduction, for which PCA is used. PCA has been found to be useful in reducing features in the past, with common values above 0.95 and below 1, or when n_components is above 1, the top n ranked features are selected [100]. PCA works through measuring the variance that can be explained through the feature. This is done through finding new variables that are linear functions of the original dataset, which maximise the variance and are unrelated to each other (which is important).

The binarisation of the protocol feature (explained in Sections 5.4.3 and 6.2) means there is always at least one feature that can be removed without any loss of data, since the sum of the protocol features should always equal 1, and the protocol features will always be labelled 0 or 1 (e.g. UDP 0, TCP 0, ICMP 1). Collectively, this means that if one of the features is removed (for example, the ICMP feature) it can be inferred from the remaining features. This does extend to features outside of the expanded ones created, however. Additionally, several papers have found that some features are more useful than others or explain more variance than others [101], [71], [102], [103].

The goal of feature reduction is to remove only those redundant features that are left after the initial processing, and any features that do not contribute to the overall model. This means using higher values of n_components.

As was stated in Section 5.4.3, a method of balancing the training data is also needed. Section 3.1.3 highlighted how SMOTE based techniques have improved accuracy, and as such comparing both SMOTE and ADASYN is required.

Default "reasonable" values for other hyper-parameters within the model were selected. The model was then run using these reasonable values on a range of values for n_components, using 10-fold cross validation to gather a suitable average of the results.

The results on the UNSW-NB15 dataset are shown in Table 6.3-1 and Table 6.3-2:

PCA Value (Amount of variance explained)	Accuracy	Recall	Precision	F1	Features Selected
0.99	83.81	87.63	83.93	85.63	118/171
0.98	83.05	70.48	98.23	82.08	84/171
0.97	85.06	75.41	96.73	84.75	51/171
0.96	81.66	74.81	90.20	81.79	23/171
0.95	82.05	70.51	95.78	81.22	16/171
0.94	80.44	75.17	87.55	80.89	13/171

Table 6.3-1 – Table to show the effect of PCA n_components value on Accuracy, Precision, Recall, F1 and number of features on the UNSW-NB15 Dataset using ADASYN

PCA Value (Amount of variance explained)	Accuracy	Recall	Precision	F1	Features Selected
0.99	84.87	89.57	84.01	86.70	118/171
0.98	85.34	87.68	85.98	86.82	84/171
0.97	85.10	86.40	86.51	86.46	51/171
0.96	83.92	92.53	80.97	86.37	23/171
0.95	81.60	96.00	76.55	85.18	16/171

0.94	83.02	89.09	81.72	85.25	13/171

Table 6.3-2 – Table to show the effect of PCA n_components value on Accuracy, Precision, Recall, F1 and number of features on the UNSW-NB15 Dataset using SMOTE

It can be seen from the tables that accuracy and F1 score start to be affected negatively below 0.97. At this point, so many features are being removed that overfitting starts earlier. Figure 6.3-3 shows loss against epoch for the PCA value 0.96, which shows this overfitting occurring.

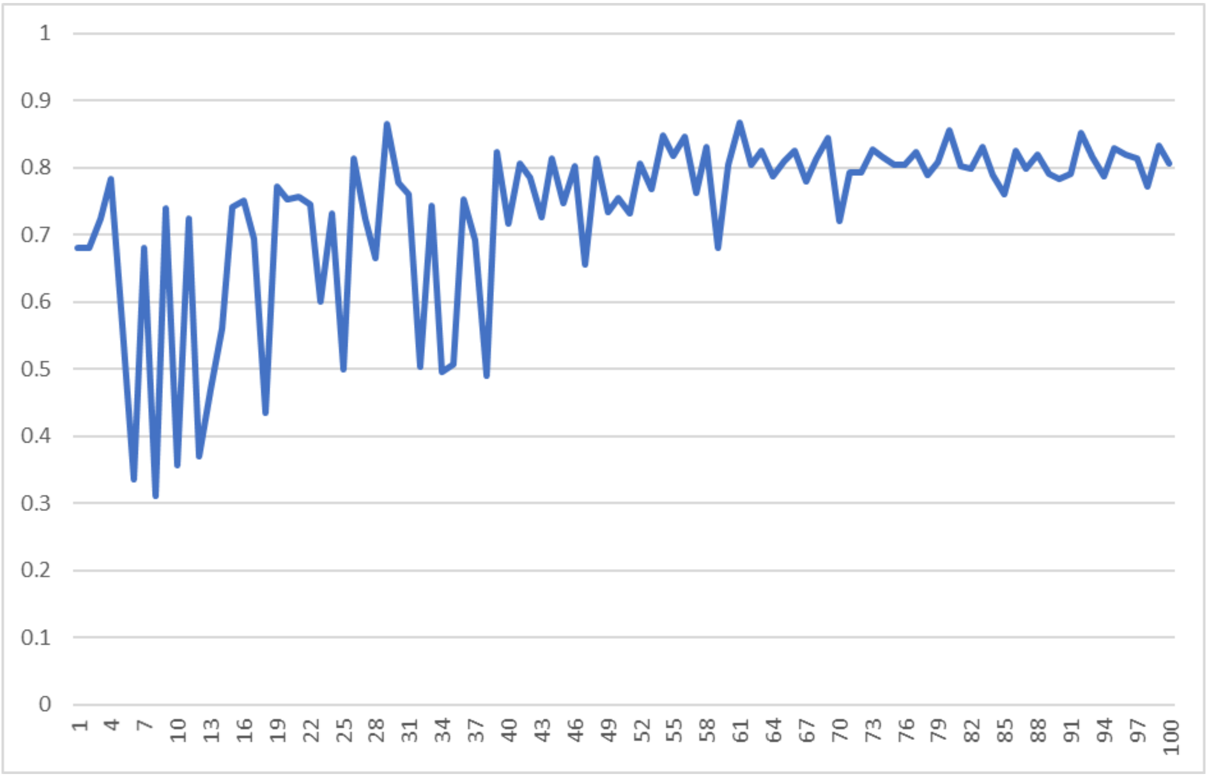


Figure 6.3-3 – Accuracy against epoch for PCA n_component = 0.96

Accuracy peeks at the 61st (86.7%) epoch, and the model settles into a local optimum at about that same point. The other peaks at points 80 and 92 were both lower than the 61st result, with the 92nd epoch still being lower than the 80th (85.67% vs 85.14%). It is likely that even if training had been allowed to continue beyond the 100th epoch, the results would have continued to decline as the model become more and more overfitted. We can compare this to the same graph at 0.98 for the PCA n_component value (Figure 6.3-4).

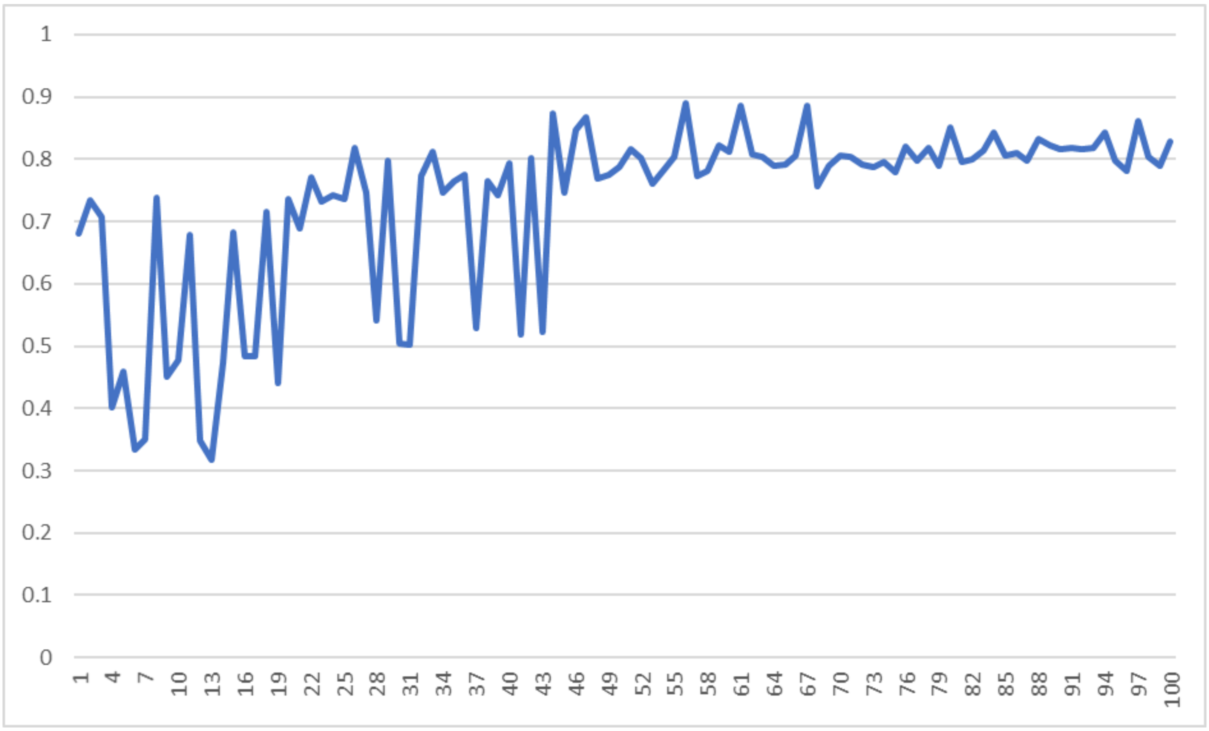


Figure 6.3-4 – Accuracy against epoch for PCA n_component = 0.98

The graph starts to find the local optimum at epoch 44 here (rather than epoch 39 above). It also finds values nearing 90% at epochs 56, 61 and 67. The tail also appears to be flatter, with less variance between epochs. This could indicate that the extra data has given more stability to the optimum solution found by the model.

From the effects on the UNSW-NB15 dataset, it can be seen that SMOTE has the effect of increasing Recall at the expense of Precision. Alternatively, ADASYN has the effect of increasing Precision at the expense of Recall. This makes sense because the aim of ADASYN is to give more training data to more difficult-to-classify records. ADASYN would make more records of the difficult-to-classify malicious activity, whereas SMOTE will just make more malicious examples, without targeting the difficult to classify records. This would naturally have the effect of increasing precision with ADASYN over SMOTE. F1 scores however show a slight preference for SMOTE. This indicates that while ADASYN may be better at ensuring benign records are not misclassified, SMOTE may be more successful at detecting malicious activity overall. SMOTE with a PCA n_components value of 0.98 is chosen.

6.3.3 Binary Classification

The next step is the initial classification itself. The goal of the initial classification is to determine which flows will undergo the mitigation process (i.e. a binary classifier). This is different to determining which flows are malicious, in that even if this initial classification determines a flow is malicious, the mitigation process may later decide the flow is benign or not a threat. A secondary consideration of this is that this stage can be used to add more data to the mitigation process. Therefore, a softmax classifier is used to produce the end classification result. This is unusual, as typically binary classification would be performed with a sigmoid classifier. However, sigmoid gives results as a float that is not constrained and therefore it is difficult to compare different runs of the same model. Softmax alternatively gives results in a float between 0 and 1, where the result is a direct probability of record being that class, and where the sum of the probabilities equal 1.

As a CNN-based model is being used, there are various hyper-parameters that require tuning (identified within Section 5.4.3), such as learning rate, epochs and batch size. The issue with these values is that the optimum for them can change depending upon other hyper-parameters. For the hyper-parameters involved with the PCA analysis and the model structure, it would be expected the same trends to follow regardless of other hyper-parameters. With learning rate, batch size, epochs this is not the case. Changing one can have a marked effect on the optimum value for the other values. There are several ways to manage this. The first is to change each value one at a time, and mark down results for every possible value. While this will work, it is time consuming. The second is to use either a grid or random search.

Grid search can be thought of as exhaustive searching. The programmer specifies several values to test and the model will test every combination of those values. This reduces the time involved in specifying new values for hyper-parameters, recording the results, and restarting the model, but can result in large processing times otherwise. For the three key hyper-parameters identified, if each hyper-parameter is given 5 values, this would result in 5^3 or 125 runs of the model. With 10-fold validation, this becomes 1,250 runs. Assuming one full run of the model were to take 30 minutes, this would result in a run time of 37,500 minutes, or slightly over 26 days. This time can be reduced by configuring fewer hyper-parameters, or running fewer validations.

The optimum learning rate can be estimated to be in the range 0.01-0.0001. This means reasonable values can be 0.01, 0.001 and 0.0001. Likewise, batch size can be estimated to be between 50 and 200 based upon previous works [92][104]–[106]. This means reasonable values can be 50, 125 and 200. Finally, epochs can be estimated using the values 10, 50, and 75. This will cut the processing time from 26 days to approximately 4.5 hours (or 270 runs, using the same estimate on processing time). This is significantly more manageable; however, it also means that the optimum values are somewhere close to the ones selected, rather than being a conclusive estimate.

Random search is an alternative method, where instead of predetermined values being selected, random values are selected to be testing in a range of values that are specified. So, for learning rate, instead of selecting 0.01, 0.001 and 0.0001, a range 0.01-0.0001 can be selected. Then the model will randomly select learning rates within the range to test on. The advantage with this method is that it is possible to select the number of runs that are made, corresponding to how long the model should run for. The model will randomly select values for the hyper-parameters for each run. This randomness makes it difficult to determine whether the optimum value have been obtained, however random search has been shown to obtain good results in the past [107], [108], [109], [110].

One other possibility to reduce the amount of processing time is to use early stopping. Early stopping only varies the number of epochs and works through measuring the results of each epoch. If the results for the epoch do not improve, then it is assumed the model is optimised, and results are shown for that number of epochs. This in itself can be configured a number of ways, including waiting for a specific number of epochs to see if results improve any further, how much a results need to improve by, and importantly which measure to use when comparing results through epochs. Using early stopping could allow us to remove epochs as a hyper-parameter from the grid search, which would result in a squared increase with number of features, rather than cubed. It does however then mean that new parameters need to be selected in order to gather a suitable stopping point (hyper-parameters to configure hyper-parameters).

From the graphs of accuracy compared to epochs shown earlier (Figure 6.3-3 and Figure 6.3-4), it can be seen that accuracy can vary quite dramatically from epoch to epoch. Therefore, parameters to determine when accuracy may have peaked need to are defined. Weights will need to have had time to stabilise before training stops. Again, using the graphs, it appears that broadly if accuracy is to increase at all again, it will increase within 20 epochs. Much of the time, 10 epochs would be enough, however there are edge cases where this is not true. For instance, there was a peak at epoch 26, which was not overtaken until epoch 44. At epoch 26 the model had clearly not had enough time to stabilise and did not start to stabilise until around epoch 50. The patience parameter is therefore set to be 20. With a patience parameter so high, restore_best_weights will be changed to true as the values 20 epochs after the optimum are likely to be different. Other details can be left at default. Min_delta looks for any improvement, and mode defaults to auto which will work for the proposed model. This produces code similar to Code 6.3-1:

```
earlyStopping = EarlyStopping(monitor='val_recall', min_delta=0.1,
patience=20, verbose=50, mode='auto', restore_best_weights=True)
```

Code 6.3-1 – Code to show Early Stopping

A combination of early stopping and grid search is used in order to find the optimum values for the hyper-parameters. While the learning rate options will remain at 0.01, 0.001 and 0.0001, the batch size options can be increased to 50, 100, 150 and 200 in order to home in on the right batch size more quickly. As earlier tests have shown a direct relationship between precision and recall, recall is selected as the parameter to test against with the early stopping. It shall also be tested against the validation set and not the main set in order to prevent overfitting. The test set is the same set as the main training set, but without any of the additional records created by ADASYN or SMOTE. While ideally training and validation sets would be completely separate, in order to allow for comparisons to other works to still be made, the testing and training set of the UNSW-NB15 cannot be changed dataset beyond what has been described in the data pre-processing steps. Reusing the training set allows for the testing set to be completely unseen until testing is required.

Moving on to the structure of the model itself, CNN models generally have become deeper, not wider, with convolutions kernel sizes of 1x1 being used. This size of convolution would not reduce the size of the feature set but allows for the number of layers to be increased. Generally, the kernel sizes should decrease as the number of layers increase, which should allow the model to find increasingly latent features. As such proposed are models with 1x1 kernel sizes in the last few layers, increasing the number of layers in the model. Also proposed are models with the 1x1 kernels earlier and spaced throughout the model to see the effects this will have.

Several model structures will need to be tested, and generally the kernel size should decrease in size as the model progresses. This is based on works such as [47], [46], [111]. Because of the small size of the feature space, and the need for multiple layers, the first layer should be limited to at most a kernel size of 5x5, and 3x3 should be tested. As such the following model sizes will be tested:

1 3x3 kernels stacked
2 3 3x3 kernels, followed by 1x1 kernel, 2 3x3 kernels
3 2 3x3 kernels followed by 1x1 kernels, 3 3x3 kernels
4 2 3x3 kernels, 1 1x1 kernel, 2 3x3 kernels, 1 1x1 kernel, 1 3x3 kernel
5 3 3x3 kernels, 4 2x2 kernels
6 1 5x5 kernel, 3 3x3 kernels

As such code for the final model will be coded as shown in Code 6.3-2:

```
def create_model(dropout_rate=0.2, learn_rate=0.01):
  # create model
    model = Sequential()
    model.add(Dense(121, activation='softmax',
input_dim=int(inputShape)))

    model.add(Reshape((11, 11, 1)))

    model.add(Conv2D(8, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Conv2D(16, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Conv2D(32, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Conv2D(64, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Conv2D(128, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Flatten())
    model.add(Dropout(dropout_rate))

    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='softmax'))

    optimizer = RMSprop(lr=learn_rate, rho=0.9)
    model.compile(optimizer=optimizer,
                  loss='binary_crossentropy',
                  metrics=[km.binary_precision(), 'accuracy']
                 )

    return model
```

Code 6.3-2 – Full model code

The model shapes are tested by only changing the model function. Results for the different model structures are as follows:

Model	Epoch Stopped	Batch Size	Learning Rate	Accuracy	Precision	Recall	F1
1	100	75	0.001	85.39%	97.05%	75.76%	85.10%
2	100	100	0.001	84.06%	98.64%	73.60%	84.30%
3	100	50	0.01	85.03%	90.52%	81.33%	85.68%
4	100	100	0.01	66.78%	67.16%	77.62%	72.01%
5	100	100	0.01	72.37%	76.65%	71.65%	74.07%
6	100	100	0.001	83.82%	99.19%	71.19%	82.90%

Table 6.3-3 – Model accuracy, precision, recall and f-score, as well as grid search metrics for the UNSW-NB15 dataset

Model	Epoch Stopped	Batch Size	Learning Rate	Accuracy	Precision	Recall	F1
1	100	100	0.001	89.7%	91.5%	97.5%	94.4%
2	75	150	0.01	82.1%	72.3%	87.54%	79.22%
3	75	100	0.01	83.82%	70.62%	99.98%	82.77%
4	100	100	0.001	83.81%	71.06%	99.29%	82.84%
5	100	100	0.001	85.5%	73.96%	98.00%	84.29%
6	100	100	0.001	83.2%	72.81%	97.81%	83.51%

Table 6.3-4 – Model accuracy, precision, recall and f-score, as well as grid search metrics for the NetFlow/IPFIX dataset

Models 1, 2 and 3 perform best in terms of accuracy and F-Score. A requirement for precision to be emphasised has been stated, so that as few false positives are passed to the mitigation system as possible. This would lead us to using Model 1 or 2, and since Model 1 has a slight advantage over Model 2 in terms or F1 and accuracy, Model 1 will continue to be used.

Using an Intel I7-3770 with a base clock at 3.4GHz, and 16GB of DDR3 RAM training takes 53secs per epoch. With early stopping on, the amount of time each run takes does vary, however, the models usually stop training before 50 epochs, we can say that one complete run takes approximately 8 hours. Using a GPGPU dramatically decreases this, and runs using a RTX2080 typically took less than 2hrs when combined with an Intel I9-9900 clocked at 3.6GHz.

6.4 Mitigation

This section shall be split into four subsections, each representing one model within the mitigation plane. These are the Network Layer, the Attack Target, the Attack Vector, and the Attack Intended Impact.

6.4.1 Network Layer Targeted

While the model for the first stage of the mitigation process could be based off the model for the binary classification (Section 6.3.3), there are several changes that would need to be made to make it suitable for multi-class classification. Previously the model (shown in Code 6.3-2) used precision to compare the success of it between epochs. This logically does not make sense for a multi-class problem, and so the metric is changed to mean squared error. Rather than attempting to increase precision, the model will now attempt to reduce the error between epochs. The different model shapes and sizes can be tested once more. The reduced size and composition of the datasets (caused by not training on any records that have not been identified as malicious in the initial analysis) means the other model shapes might be more effective. The model structures chosen are shown in Table 6.4-1.

Model Number	Model Structure
Model 1	3x3, 3x3, 3x3, 2x2
Model 2	2x2, 2x2, 2x2, 2x2, 2x2, 2x2, 2x2

Table 6.4-1 – Table of model structures for the Layer Mitigation Model

Additionally, the loss metric should be changed from binary crossentropy to mean squared error, as the problem is no longer a binary problem.

In Chapter 3, it was stated that flows that were marked as being malicious from the classification phase would be used. This gives a better representation of the data the models would actually receive more generally, as if only malicious flows from the dataset were used, then the model would not ever be trained on benign data (or false positives). Part of the reasoning for this is that false positives should be flagged as having no action taken against them, and so should be included in the training set. The code needed to gather the data for training and testing is comparatively simple, as shown in Code 6.4-1:

```
for row, Y in zip(X, classification_results):
              if Y == 1:
                  layerX.append(X)
```

Code 6.4-1 – Code to gather data for mitigation stage

This leads to the results found in Confusion matrix 6.4-1 and Confusion matrix 6.4-2:

	Benign	Network	Application
Benign	88.0%	0.9%	11.1%
Network	4.7%	69.5%	25.8%
Application	16.8%	0.3%	82.9%

Confusion matrix 6.4-1 – Results for the Network Layer model within the Mitigation taxonomy with the first model structure

	Benign	Network	Application
Benign	44.7%	2.1%	53.2%
Network	1.8%	74.3%	23.9%
Application	12.2%	0.4%	87.3%

Confusion matrix 6.4-2 – Results for the Network Layer model within the Mitigation taxonomy with the second model structure

Seen is that the first model has greater success in correctly identifying benign flows, while the second model has greater success in identifying Network and Application level attacks. At this stage, it is still important to minimise the number of misclassified benign flows, and the advantage the Model 2 has in accurately identifying Network and Application malicious activity is minor compared to Model 1 (74.3% vs 69.5% for Network and 87.3% vs 82.9% for Application). Additionally, the advantage for Model 1 in accurately defining benign flows is significant (88% vs 44.7%). For this reason, the Model 1 structure is chosen to continue. However, it is important to note that there are more malicious flows defined as benign for Model 1. This means automatic mitigation will not be undertaken against these flows; however, an alert will still have been raised with an administrator.

6.4.2 Attack Target

The next stage is the Attack Target. This model serves two main functions, the first is to remove any remaining benign activity. The second is to gain a reasonable idea of where the attack is taking place, so that this can be fed into the final decision on the activity to take.

There are not many benign records left within the data set, and this can cause errors with upscaling methods such as SMOTE or ADASYN, since the number of examples can be less than the number of neighbours. Random Oversampling is therefore used in place and is compared to no oversampling at all. This is shown in Confusion matrix 6.4-3 and Confusion matrix 6.4-4:

	Benign	Network	OS	Application
Benign	100%	0%	0%	0%
Network	3.8%	75.5%	1.3%	22.4%
OS	1.2%	0.3%	31.1%	67.3%
Application	0.3%	0%	1.0%	98.6%

Confusion matrix 6.4-3 – Confusion Matrix for the Target in the Mitigation system with Random Oversampling

	Benign	Network	OS	Application
Benign	0%	66.7%	33.3%	0%
Network	0%	75.8%	24.2%	0%
OS	0%	0.3%	99.6%	0%
Application	0%	0%	84.9%	15.1%

Confusion matrix 6.4-4 – Confusion Matrix for the Target in the Mitigation system without Random Oversampling

With oversampling all remaining benign flows are detected as such (with a small number of malicious flows being misclassified as benign). Without oversampling, no benign flows are detected as being classified as benign. Additionally, with the Random Oversampling the results for OS drop significantly (from 96.6% without to 31.1% with), however, this comes at the cost of also misclassifying Application Targets which also have results that drop without Random Oversampling (from 98.6% with Oversampling to 15.1% without). Taken collectively it is concluded that random oversampling should continue.

6.4.3 Attack Vector

The next stage is to determine the Vector for the remaining attacks. Again, the effect of random oversampling can be seen, however, now all benign flows have been classified as such the effects may not be as prominent. The results for these are shown in Confusion matrix 6.4-5 and Confusion matrix 6.4-6.

	Benign	Misconfiguration	Insufficient Validation	Vulnerabilities	Users	System Limitations
Benign	NA	NA	NA	NA	NA	NA
Misconfiguration	0%	78.8%	18.0%	3.2%	0%	0%
Insufficient Validation	0%	65.7%	29.3%	4.2%	0.8%	0%
Vulnerabilities	0%	53.6%	11.4%	32.7%	2.0%	0.4%
Users	0%	0%	0%	50.0%	50.0%	0%
System Limitations	7.2%	18.9%	3.5%	2.1%	0.3%	68.0%

Confusion matrix 6.4-5 – Confusion Matrix for the Vector in the Mitigation system with Oversampling

	Benign	Misconfiguration	Insufficient Validation	Vulnerabilities	Users	System Limitations
Benign	NA	NA	NA	NA	NA	NA
Misconfiguration	NA	0.3%	0%	99.7%	0%	0%
Insufficient Validation	NA	0.2%	26.3%	73.5%	0%	0%
Vulnerabilities	NA	0%	0%	99.6%	0%	0.3%
Users	NA	0%	0%	100%	0%	0%
System Limitations	NA	0%	0%	25.4%	0%	74.6%

Confusion matrix 6.4-6 – Confusion Matrix for the Vector in the Mitigation system without Oversampling

As expected, the case to keep oversampling is far less clear. While it has a positive effect on the accuracy of Misconfiguration, Insufficient Validation and Users, it has a negative effect on all other vectors. 7.2% of System Limitation vector-based attacks are incorrectly classified as benign, meaning those attacks would not be automatically mitigated. However, without oversampling most vectors are being labelled as vulnerabilities. This is potentially showing signs of overfitting, as the model starts to show preference to the majority class. It is chosen to continue with oversampling for now.

6.4.4 Impact Sought from Attack

The final stage is to determine the impact the attack is attempting to achieve. The confusion matrix for impact is shown in Confusion matrix 6.4-7 and Confusion matrix 6.4-8:

	Benign	DoS	Information Disclosure	Access Privilege
Benign	NA	NA	NA	NA
DoS	0.1%	93.6%	0.5%	5.8%
Information Disclosure	0%	12.0%	87.4%	0.6%
Access Privilege	0%	67%	0.3%	32.6%

Confusion matrix 6.4-7 – Confusion Matrix for the Impact in the Mitigation system without Oversampling

	Benign	DoS	Information Disclosure	Access Privilege
Benign	NA	NA	NA	NA
DoS	0%	99.7%	0.3%	0%
Information Disclosure	0%	5.1%	94.9%	0%
Access Privilege	0%	0%	0%	100%

Confusion matrix 6.4-8 – Confusion Matrix for the Impact in the Mitigation system with Oversampling

It can again be seen that oversampling does aid overall accuracy. While both DoS and Information Disclosure attacks have good results without the oversampling, achieving 93.6% on DoS and 87.4% on Information Disclosure. However, only 32.6% of Access Privilege attacks are identified correctly. Additionally, only 0.1% of records are misclassified as benign, meaning that 99.9% of malicious activity that makes it to the final model will have some sort of mitigation applied to it. However, adding in oversampling achieves 100% accuracy on Access Privilege attacks, with 99.7% of DoS attacks correctly classified and 94.9% of Information Disclosure attacks correctly classified. This leads us to believe that Oversampling is still helping with the training process, despite the mixed results for Section 6.4.3 and the Attack Vector.

7 Evaluation and Results

This Chapter will practically demonstrate the extent to which the objectives outlined in Section 1.2 have been achieved. This will be done by listing the objective, the experiment performed to show the objective was achieved and the results of the experiment.

7.1 Develop a method to mitigate the effects of the limited data

While the effects of the limited data on detection accuracy have been mitigated through using techniques such as SMOTE or ADASYN, as well as developing a custom model designed to handle the fewer features, this needs to be put into the context of wider works. As such, this section will compare the initial classification results to other state of the art results. This primarily needs to be done with the UNSW-NB15 dataset, as this is the most widely used. This work is compared to [41], [42], [43], [44].

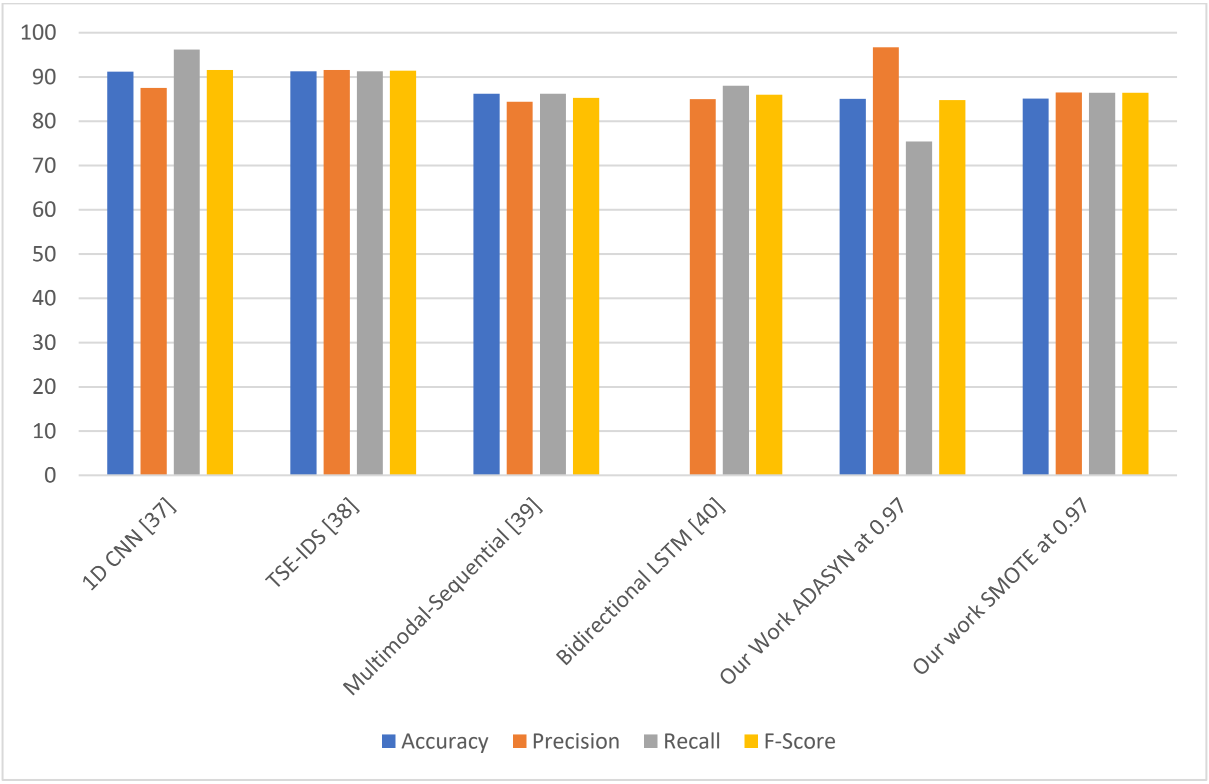


Figure 7.1-1 – A comparison of this work to other state of the art results

This work is broadly comparable to the others listed; however, this needs to be put into the perspective of the features used. All the comparable papers listed use the entire dataset, and all its features to gain their results. Alternatively, these results use a fraction of the features (70% fewer features than the cutting-edge works compared against). While accuracy may be slightly lower (at 85.1% vs an average of 89.5%), precision, and F-score are both competitive. SMOTE reaches a precision of 86.5% vs an average of 87.1%, and ADASYN achieves a higher precision then any of the works mentioned. Similarly, F-Score reaches 86.5% for SMOTE, verses an average of 88.5% for the other papers. The model should run quicker on comparable hardware (simply because there are fewer features to process) and requires less post processing to gather features, while maintaining comparable results.

7.2 Identify Other Potential Data Sources

While results are comparable to other in the space, it is still needed to identify if the extra data extracted from logs has aided this, and to what extent. This can be performed through measuring the success of the initial model both with the extra data collected by logs and without.

In Tables Table 7.2-1 and Table 7.2-2 are the results from models 1 and 3 for the UNSW-NB15 dataset. These models were chosen as they achieved the highest accuracy and F1-scores in Section 6.3.3.

Model	**Epoch Stopped**	**Batch Size**	**Learning Rate**	**Accuracy**	**Precision**	**Recall**	**F1**
1	100	75	0.001	85.39%	97.05%	75.76%	85.10%
3	100	50	0.01	85.03%	90.52%	81.33%	85.68%

Table 7.2-1 – Results with the highest Accuracy and F-score for the UNSW-NB15 dataset with logs

Model	**Epoch Stopped**	**Batch Size**	**Learning Rate**	**Accuracy**	**Precision**	**Recall**	**F1**
1	50	75	0.001	80.96%	88.43%	75.27%	81.32%
3	75	100	0.001	84.06%	99.35%	71.52%	83.17%

Table 7.2-2 – Results with the highest Accuracy and F-score for the UNSW-NB15 dataset without logs

There is a clear difference, with accuracy and F-Score consistently lower for the results without logs. Precision is high for model 3 without logs, however this comes with very low recall, which indicates that overfitting is occurring, and the model is starting to register all activity as malicious. The number of epochs is lower for the results without the log data. This is another indication that overfitting is occurring, as early stopping is stopping the model learning significantly earlier than with the log data as accuracy has peaked.

The results on the NetFlow/IPFIX dataset can be seen in Table 7.2-3 and Table 7.2-4:

Model	**Epoch Stopped**	**Batch Size**	**Learning Rate**	**Accuracy**	**Precision**	**Recall**	**F1**
1	100	100	0.001	89.7%	91.5%	97.5%	94.4%
3	75	150	0.01	82.1%	72.3%	87.54%	79.22%

Table 7.2-3 – Results with the highest Accuracy and F-score for the NetFlow/IPFIX dataset with logs

Model	**Epoch Stopped**	**Batch Size**	**Learning Rate**	**Accuracy**	**Precision**	**Recall**	**F1**
1	75	100	0.01	79.4%	79.3%	99.1%	88.1%
3	50	150	0.01	76.2%	75.7%	99.4%	85.9%

Table 7.2-4 – Results with the highest Accuracy and F-score for the NetFlow/IPFIX dataset without logs

These show similar trends, with lower accuracies and fewer epochs being run. It should also be noted that without logs, it is recall that showcases the overfitting, with results over 99% while precision is less than 80% for both models without logs. The difference between precision and recall is much less pronounced in both models with the extra log data.

7.3 Determine how to run the system at near line speed

While steps have been taken throughout this thesis to ensure that processing complexity is kept to a minimum, it was stated in Sections 1.2 and 2.5 it was likely that some sort of sampling method for the occasions network load exceeds capacity to analyse that load. To that end a method has been established to assess the risk a flow may pose, through the use of HDBScan clustering. The next step is to determine if the clustering was successful. The goal of the clustering was to provide a risk score so that high risk clusters could be analysed first, and low risk clusters could be analysed last, or even ignored if system resources were stretched. In practice, both the UNSW-NB15 and NetFlow/IPFIX datasets that were used contain more malicious activity than would be normal. The NetFlow/IPFIX dataset was taken from a real-world network, however as part of the data preparation the size of the dataset was reduced, and malicious activity oversampled. Effectively this means that malicious data increased from being approximately 4% of the total dataset to being approximately 39%. This means that looking at absolute numbers is not effective, since these would not be representative.

As such the proportion of flows that would have been clustered in a correct flow is analysed. For these four categories of risk are proposed. Clusters within 0%-25% are considered low, clusters within 26%-50% are considered moderately low, clusters within 51%-75% are considered moderately high and clusters within 76%-100% are considered high risk. Ideally benign clusters are labelled as low risk, and malicious clusters being labelled as high risk. Table 7.3-1 shows this.

Cluster risk score grouping	**NetFlow/IPFIX**			**UNSW-NB15**		
	Number of attacks	Proportion of total attacks	Proportion of flows within the class	Number of attacks	Proportion of total attacks	Proportion of flows within the class
0% - 25% Low Risk	789	0.015	0.012	211	0.002	0.012
26% - 50% Moderately Low Risk	6421	0.125	0.406	1481	0.012	0.303
51% - 75% Moderately High Risk	6373	0.124	0.678	48508	0.406	0.596
76% - 100% High Risk	37705	0.735	0.876	68141	0.571	0.959

Table 7.3-1 – Proportion of malicious flows in the super clusters

It can be seen that for both datasets the proportion of flows that is considered low risk (0%-25% risk score) is exceptionally small, with only 1.5% of the NetFlow/IPFIX dataset, and 0.2% of the UNSW-NB15 dataset. This is good since these flows are least likely to be analysed further. Additionally, malicious flows only make up 12.5% of moderately low risk (26%-50%) flows within the NetFlow/IPFIX dataset, and 1.2% for the UNSW-NB15 dataset.

Similarly, most malicious flows are correctly identified as high risk (76%-100%). For the NetFlow/IPFIX dataset, 73.5% of malicious flows are deemed high risk, and 57.1% of flows for the UNSW-NB15 dataset. This means that 87.6% of flows within the high-risk category for the NetFlow/IPFIX dataset are malicious, and 95.9% of high-risk flows within the UNSW-NB15 are malicious. In both cases, even if only high-risk flows were analysed, more than 70% of malicious activity would be analysed. If this analysis is moved to include both high and moderately high risk (51%-75%), then more than 85% of malicious activity will be analysed for the NetFlow/IPFIX dataset, and more than 95% of malicious activity will be analysed for the UNSW-NB15 dataset. This compares to only slightly more than 10% of benign flows being analysed for the NetFlow/IPFIX dataset, however slightly more than 60% of benign activity would also be analysed for the UNSW-NB15. Still, in a real-world system, being able to ignore 40% of flows in exchange for analysing 95% of malicious activity might be considered worthwhile.

7.4 Determine a method to mitigate even 0-day threats

The mitigation method developed is flexible, and rather than attempting to identify the exact attack being performed (something that is logically not always possible for 0-day threats), instead attempts to determine the major aspects of the attack that could lead to a successful mitigation strategy. To determine whether this method has been successful, two factors need to be established. The first is whether the cascading nature (i.e. the feeding of results from lower level models to higher level models) of the mitigation models has been successful. If there is no benefit from the cascading results, then the models could be run in parallel to decrease reaction time. The second is to compare the results to other state of the art results attempting to classify all attack types, to ensure the results are comparable.

The first step is to identify that the cascading model structure is effective through comparing results on the Vector and Impact models, both with and without the previous model's data. Vector and Impact are chosen, as these have the largest effect on what the end mitigation would be.

7.4.1 Vector

The first step is to compare the Vector results with and without the extra data provided by previous models in Confusion matrix 7.4-1 and Confusion matrix 7.4-2:

	Benign	Misconfiguration	Insufficient Validation	Vulnerabilities	Users	System Limitations
Benign	NA	NA	NA	NA	NA	NA
Misconfiguration	NA	0.3%	0%	99.7%	0%	0%
Insufficient Validation	NA	0.2%	26.3%	73.5%	0%	0%
Vulnerabilities	NA	0%	0%	99.7%	0%	0.3%
Users	NA	0%	0%	100%	0%	0%
System Limitations	NA	0%	0%	25.4%	0%	74.6%

Confusion matrix 7.4-1 – Vector Confusion Matrix with taking extra data from earlier models

	Benign	Misconfiguration	Insufficient Validation	Vulnerabilities	Users	System Limitations
Benign	NA	NA	NA	NA	NA	NA
Misconfiguration	NA	0%	0%	99.4%	0%	0.6%
Insufficient Validation	NA	0%	24.4%	75.2%	0%	0.4%
Vulnerabilities	NA	0%	0.1%	99.4%	0%	0.5%
Users	NA	0%	0%	100%	0%	0%
System Limitations	NA	0%	0%	24.6%	0%	75.4%

Confusion matrix 7.4-2 – Vector Confusion Matrix without taking extra data from earlier models

While there are common trends between both confusion matrixes (such as the propensity for malicious activity to be labelled as vulnerabilities) it can be seen that the extra data seems to have had an effect. Insufficient validation, vulnerabilities and misconfiguration all show an increase in accuracy (albeit minor) and misclassified flows are generally higher without the extra data. The exception is system limitations, which has slightly more flows classified as vulnerabilities with the additional data. It is possible that the changes are caused by run to run variance, as they are all within 3%, and frequently within 1%. It should be noted however, that by this point the dataset is growing quite small as all benign flows have been removed. This means that comparatively small numbers of flows changing classes can cause comparatively large shifts in the results. It should also be remembered that the models were run 10 times (through 10-fold cross validation) in an attempt to remove variance. It is concluded therefore that the extra data does have a positive effect, though the effect is small for the Vector itself. Additionally, it should be remembered that the small size of the remaining dataset will be affecting accuracy. Deep learning models require large volumes of information (Section 2.3) and by this point all benign data has been removed from the dataset, as well as some misclassified malicious data, reducing the amount of data available to learn.

7.4.2 Impact

The same comparison is then made, but with Impact rather than Vector. The final model is most important, since it has the largest effect on the end mitigation taken. Quarantining systems under a DoS attack for example would not be effective, and redirection of flows to another legitimate target will not be effective for Access Privilege attacks. Presented are the results of the Impact both with and without the extra data in Confusion matrix 7.4-3 and Confusion matrix 7.4-4:

	Benign	DoS	Information Disclosure	Access Privilege
Benign	NA	NA	NA	NA
DoS	0%	99.7%	0.3%	0%
Information Disclosure	0%	5.1%	94.9%	0%

Access Privilege	0%	0%	0%	100%

Confusion matrix 7.4-3 – *Confusion Matrix for Impact with extra data from earlier models*

	Benign	DoS	Information Disclosure	Access Privilege
Benign	NA	NA	NA	NA
DoS	0%	0.8%	0.7%	98.5%
Information Disclosure	0%	0%	87.6%	12.4%
Access Privilege	0%	0%	0.6%	99.3%

Confusion matrix 7.4-4 – Confusion Matrix for Impact without taking extra data from earlier models

Here the results are much more pronounced than with the Vector. All categories achieve above 90% accuracy with the data from previous models. The difference is clearly in the classification of Access Privilege Impacts, which while only increasing accuracy from 99.3% to 100% also eliminates any misclassification of other Impacts as Access Privilege. Without the extra data there is a tendency for the model to classify as Access Privilege, which is particularly pronounced in DoS attacks. This again could be a sign of overfitting, which the extra data negates.

7.4.3 Comparisons to previous works

While comparisons to previous works are more difficult, owing to the novel way malicious activity is being classified, some comparisons can still be made. Table 7.4-1 shows this work compared to others with categorial accuracy and FAR:

Work	**Accuracy**	**FAR**
This work	88.2%	10.3%
Deep Reinforcement Learning based Intrusion Detection System for Cloud Infrastructure [112]	83.8%	2.6%
A Network Intrusion Detection Method Based on Stacked Autoencoder and LSTM [84]	89.2	10.8%
An Ensemble-based Network Intrusion Detection Scheme with Bayesian Deep Learning [113]	96.9%	0.9%
Hybrid Machine Learning For Network Anomaly Intrusion Detection [114]	95.4%	11.9%

Table 7.4-1 – Comparisons of multi-class classification

It can be seen that this work achieves comparable accuracy to other state of the art works, however FAR is slightly higher than usual. Part of the reasoning for this is that there was an emphasis on ensuring that there were no benign flows being mitigated. As such, all of the misclassified flows are from malicious flows being misclassified as other types of malicious flows. Most of these are from the Vector model, which from Confusion matrix 7.4-1 can be seen did have a number of misclassified flows, notably Users and Insufficient validation.

It should also be noted that the same thing is not strictly being classified. All of the above papers are classifying based on the single attack type (described in Section 2.6). This is potentially more difficult, since there are 9 classes, rather than the (at most) 6 used in this work. At the same time, this work uses far fewer features, and a far smaller dataset with all of the benign records removed by this point. Additionally it should also be noted that [113] and [114] use the full unprepared dataset, which has been stated is slightly easier. At the time of running, the dataset has been reduced to 74,854 records from the original 175,341, a reduction of more than 100,000 records, or 58% of the dataset records. This work is also still only using the data features provided in earlier sections, plus the additional analysis performed as part of the process. We start with only 28% of the features, and less than half of the dataset records, but still achieve the results above.

8 Discussion

The clustering method to create a risk score was successful. While it is possible that some flows may not be analysed due to being placed in a low risk grouping, this is possible with any kind of sampling method. It is unlikely with the method proposed however, with only 1.2% of all attacks for the NetFlow/IPFIX dataset and 0.2% of all attacks for the UNSW-NB15 dataset being placed into the lowest risk grouping. Additionally, 57.1% of malicious flows from the UNSW-NB15 dataset, and 73.5% of malicious flows from the NetFlow/IPFIX dataset are placed in the highest risk class, ensuring they are analysed with a high priority. This extends to over 95% for the UNSW-NB15 dataset and over 85% for the NetFlow/IPFIX dataset if you include any cluster that had a risk score of over 50%.

With regards to the initial classification, it is acknowledged that recall could use improvement, with the best results from SMOTE being around 86%-89%. However, this means that precision is lowered, meaning that more benign flows could be classified as malicious. It is also important to note that these results should be taken into consideration with other results from other authors. The UNSW-NB15 prepared dataset is difficult, and the best state of the art results range between 90%-95%. Chapter 2 shows this with some comparative papers, and the results achieved within this thesis tend to be within 5% of the best. This work also uses only 29% of the whole dataset, and that these comparative works use the entire set. The benefits of using a smaller dataset include faster processing time, as well as it being more applicable to multiple SDN environments and situations.

It should also be noted that that the ADASYN model has lower recall specifically. This is likely due to overfitting starting to occur, combined with ADASYN increasing the number of difficult to classify attacks. SMOTE (which increases all records equally) instead sees an increase in recall n_components equalling 0.96 onwards. The ADASYN model is used in order to reduce the number of false positives, however given the mitigation solution removes false positives as well, it could be considered better to aim for higher recall at this stage and reduce the number of false positives later.

Additionally, the classifier for the model is a DNN with a softmax activation. Other types of shallow classifiers have been found to work particularly well, including more recently random forests. While some experimentation was carried out, the initial results were not significantly better than the DNN after the CNN, and so more experimentation and refinement was not pursued. However, previous studies [103] show that this is potentially something to return to, and random forests can prove effective classifiers after a CNN has been used to reduce the dataset. Potentially the initial lack of results could have been due to the use of PCA at the start of the process, and again this is something that could be experimented with.

The same can be said of PCA itself. There are numerous ways of reducing the feature space. PCA was chosen as it has been shown to be successful in the past [49], [84], and its implementation allows for us to decide how much information to keep, whereas other methods decide how many features to have at the end of the process. It is possible however, that other systems would lead to better classification accuracy. Again, some experimentation was carried out with using random forests as a feature extractor, but the results were not significantly better than PCA and so further experimentation or refinement was not pursued.

The process of “chaining” together multiple models in order to gather a final classification and mitigation proved successful as the results show that data from previous models aids in accuracy of the following models, even though they are not directly classifying the same thing. The fact that there is always an option for benign results mean that any benign flows get removed after the second (Target) model. The number of false negatives also stays consistently low, meaning that most of the initially detected attacks stay within the mitigation system. More to the point, the type of attack (DoS, Information Disclosure or Access privilege) is successfully determined with high degrees of success. 93.6% of DoS attacks are correctly identified, which is important since they have been assigned a unique mitigation measure (sinkhole or redirection to other legitimate source depending upon the certainty of malicious intent). Information disclosure is correctly predicted 87.4% of the time, however Access privilege is only correctly predicted 32.6% of the time. More importantly access privilege is incorrectly identified as DoS 67% of the time. This will cause issues with the mitigation strategies, as it means several access privilege attempts will be mitigated through Sinkhole or redirection. Developing a sinkhole would work, in much the same way blocking the relevant IP would work, however redirection would not be successful at all.

Both methods would remove the opportunity to view the attacker’s actions through the safety of a honeypot, which could be the preferred option as it would give an idea as to what access the malicious user was aiming for, and why. However, it is likely that the sinkhole option would be deployed, as malicious clusters were typically given a risk score of above 0.75. This also means that the option of “Do nothing” should never actually be employed, and all malicious activity should have a mitigation measure employed.

9 Conclusions and Future Works

This chapter shall discuss whether the overall aims of the project succeeded, and what potential future work could consist of. To do this the aims will be discussed, and how they were accomplished, along with any improvements that can be made, and then turn to future work.

9.1 An initial risk assessment mechanism

As discussed in Chapter 8, the initial risk assessment was largely successful. Malicious flows were largely clustered together, giving a good indication as to the level of risk an individual flow within a cluster might possess (Table 7.3-1). Likewise, benign flows are largely clustered together, giving a good indication of low risk. In fact, with the NetFlow/IPFIX dataset there were 11 clusters that contained no malicious flows (making a risk score of 0 (Figure 6.3-2)). Assuming that a system's resources would be stretched to the point of only being able to analyse moderately high risk or greater flows, this would mean that 60.2% of flows would be ignored with the NetFlow/IPFIX dataset, while still analysing over 85% of malicious activity. The method could use improvement for the UNSW-NB15 dataset however, with only 13% of flows being ignored. However, this does come with the caveat that 97.7% of malicious activity is considered moderately high risk or greater, and so would be analysed.

If the criteria were changed to be only high-risk flows were analysed, then 59.5% of data is ignored, while still analysing 57.1% of all malicious activity for the UNSW-NB15 dataset. While still not quite as good as the NetFlow/IPFIX dataset, this does indicate there is room for setting what risk level an administrator is happy to live with, verses how much of the data traversing the network they want to analyse.

It should be remembered that the system proposed would probably not be used in isolation, but as the initial part of a holistic defence that seeks to stop as much malicious activity as possible without disruption to services or users. Unfortunately, for each model the "unclustered" cluster -1 was the largest cluster. In of itself this might not be considered an issue. The fact that the data points were unclustered indicates they were anomalous. The risk score for the -1 cluster was also the closest to being 50% in both datasets, however, even so it did have a risk score of 56.9% for the UNSW-NB15 dataset and 40.7% for the NetFlow/IPFIX dataset, which still indicates a preference for malicious and benign flows respectively. In particular it is believed the result for the NetFlow/IPFIX dataset is particularly good considering the noise inherent in such datasets. The -1 cluster is an intended effect of using HDBScan. Every flow does not need to be clustered, since again the fact that they are a part of the unclustered cluster is significant. Other clustering methods were considered, such as k-means, which will ensure every point is within a cluster. However, instead of producing 1 cluster that had an ambiguous risk score, it produced many smaller clusters with ambiguous risk scores as the -1 grouping was split into smaller groups without any significant change in risk scores.

9.2 An alternative to network packet data

It can be seen that adding the logging data increased accuracy and F-score overall, with increases of almost 10% (Section 7.2 and Table 7.2-3 and Table 7.2-4 in particular). This log data is readily available and allowed the system to compare favourably against other state-of-the-art systems (Section 7.1) that are using much more computationally expensive methods to gather data. While log processing speed has not specifically analysed, it is felt that this is a fast and efficient way to integrate data into future IDSs given that much of the log analysis may be being done, as a separate part of business security processes, and one that should allow real time analysis of network flows in a useful manner. From the results in Section 7.1, this work compares favourably with the results from the most relevant and current state-of the-art works. An important point is that this work uses only 28% of the data other researchers use with the same dataset, moving from 43 features to only 12 in the case of the UNSW-NB15 dataset, and only 11 features being used in the NetFlow/IPFIX dataset. Usually, deep models are trained on images consisting of at least hundreds (in the case of datasets such as MNIST) and more commonly thousands of features. Much of the reason for the comparable results comes from the addition of the few extra features that were added in the form of log analysis, as well as data pre-processing from ADASYN or SMOTE, however shown is that small but deep models can still be successful in analysing these smaller datasets. It is believed that this is the first time that such a small but deep model has been used in this manner to achieve these kinds of results. This is important, as smaller models take less time to train and run (Section 3.1.2, [87], [115]). This is a very time sensitive environment, and for any mitigations to take place, the analysis needs to be completed before the flow has finished. A smaller model helps with this. Often pooling layers are used to aid in reducing computational complexity, with a minimum of effect on accuracy [88], [89]. While not shown, these layers were also tested, and it was found that the reduction in the feature space adversely affected accuracy, precision and recall as the model could not be made as deep, even with the addition of 1*1 sized kernel layers to help. This again shows the limitations of the datasets with so few features.

Having said that the model is not perfect. Recall is lower than would be preferred, which would lead to malicious activity not being flagged. Higher values can be achieved, but this comes at the cost of precision, making accuracy and F-Score typically stay around the same level. Additionally, unsupervised methods could be considered. Unsupervised machine learning is making great strides in the network anomaly research area, as it is seen as being more practical than supervised learning, not required labelled datasets. However, the use of the HDBScan risk assessment is considered cause enough to continue with a supervised approach. Any kind of machine learning is more computationally complex than the signature approach in use now, and even signature approaches struggle with network load, and more efficient sampling or prioritisation techniques are required.

9.3 Mitigation system capable of bespoke responses to different threats

Most works attempt to classify flows according to the grouping they are placed in within the dataset (DoS, Fuzzer, etc). However, this does not help in mitigating an attack. It is true that in order to mitigate an attack you must be able to tell what the attack is, however, the classifications given within datasets do not necessarily help. For instance, DoS attacks can come in many forms. DDoS attacks are typically flood based attacks which attempt to overwhelm a networks capacity (Network Plane, Network Target, System Limitation Vector and DoS Impact). A buffer overflow DoS on the other hand, will attempt to exploit vulnerabilities in the OS or Application which cause it to error and crash (Application Plane, OS or Application Target, Vulnerabilities Vector and DoS Impact). It is fair to say that the datasets in use only offer what the Impact of the attack will be, and not the information required to mitigate it. Within this work inspiration was taken from [68] in building a deep learning system designed to identify what Plane, Target and Vector are being used, and designed a mitigation system that would build a response based upon these. If a DoS impact had been identified, then the system should never completely block a system, since this would achieve the attacker’s goals. Likewise, redirecting to legitimate sources should never be used in situations where a malicious actor is attempting to gain access to restricted information, since this would have no real effect. The implementation of the system is largely successful, with good accuracy rates considering the reduced dataset being used. Additionally, it has been shown that feeding the results from one CNN to the next was particularly helpful, increasing accuracy by up to 10%.

It is believed this is the first time a taxonomy has been used in this way. Typically, once an attack has been identified by name, an appropriate response would be looked up (either in a database or by hand) and the measures implemented (e.g. installing a patch). The method proposed should be more generalisable since it does not rely on previous attacks being analysed and a defence process being formed. Any attack that that the same attributes will have the same mitigation measures put in place, even if they are otherwise unrelated. However, it does suffer in terms of total number of attacks being mitigated, with only 65% of all malicious activity having a mitigation automatically undertaken. This could have been configured to be higher, however you start to get benign activity having mitigations measures automatically undertaken which could have unpredictable consequences.

9.4 Limitations

This work does of course have a number of limitations. The datasets being used are one significant limitation. While the UNSW-NB15 dataset is the state of the art for research in this area, it does not perfectly encapsulate the desired features (such as the sentiment analysis from the logs from the NetFlow/IPFIX dataset). Likewise, the NetFlow/IPFIX dataset is far from perfect. As has been stated, since it consists of raw data, the training, testing, and evaluation datasets had to be generated. This makes comparisons to other works more difficult, even assuming there would be many other works to make comparisons to. Unfortunately, there is no “one size fits all” in this respect. Datasets offering the raw data needed generally do not have prepared datasets to make easy comparisons. One area where this is slightly different is in the area of log analysis itself, where there are some datasets with the raw data separated up to make comparisons easier. However, these do not contain network flow data since the emphasis is on HIDS. Even within the HIDS arena, datasets are somewhat limited by the wide variety of possible hardware implementations. A dataset looking at SSH compromise is inherently different to an Android malware system call dataset for instance.

Moving away from datasets, the choice of OpenFlow as the base for deciding what data is available is not perfect either. OpenFlow is only one SDN implementation, and while many high profile networking companies are using OpenFlow as the base in their solutions [4], they may choose to offer different features than those that have been listed. OpenFlow is the most universally available SDN solution, however it rarely exists as a product in of itself, rather as a product that service providers can offer support for in cross platform network environments. This work takes the same approach, attempting to ensure that any features that are used will always be available, but it is possible that other features are available that were missed as they are not “guaranteed” available. Likewise, not every SDN solution is built on OpenFlow, and for those solutions it is possible features may have been used that do not exist in those implementations.

This moves into a broader limitation, that being those made in Section 4.1 about the network structure. As has been stated, networks are changing, and the conventional view of a network being comprised of servers and clients is changing. On top of that, even superficially similar networks will have different structures and services. A variety of possible log data types have been incorporated, that show what may be possible, however exactly what log features are used will change from network to network. There is no way to fully account for all network types, or even all log types.

The work also makes extensive use of scikit-learn [100], which is a scientific library designed for research in shallow machine learning techniques. It may not be the best tool to use in production environments, and compile times, hyperparameter options and tuning may be different for production tools. These could change even the accuracy of the results. TensorFlow [116] and Keras were also used, however these are much more used to being used in production environments.

9.5 Future Work

The work presented could be applied to different domains and continued in different areas. Some areas for future research involve:

1. Cloud Computing
   As has been stated in Section 9.4, not all network environments have been considered, and cloud computing is an example of one area where this is true. Cloud computing has become more popular over recent years, as individuals and businesses attempt to leverage the scales of economy server farms such as Microsoft’s Azure or Amazon’s AWS provide. This work assumes a traditional network infrastructure with all servers being monitored in house. However, this is not necessarily true. Microsoft and Amazon are responsible for the security of servers placed with them, but do not necessarily have full access to logs, and likewise the administrators of the servers do not have full access to the network flows that interact with their servers. However, this work does pose some possible advantages for cloud environments. Established is that log data can supplement network flow data to aid in detection of anomalies, however the reverse should also be true, i.e. network flow data should be able to aid in the detection of anomalies for HIDS, perhaps supplementing the data for any HIDS Microsoft and Amazon have to increase accuracy and keep customers secure. Additional data could also potentially come from network devices such as routers or switches. These devices do keep their own logs, though usually for a short period of time due to memory concerns. These logs could potentially be used to increase accuracy on HIDS through monitoring anomalous network events (short flows, burst flows etc).
2. IoT

The work could be extended to IoT and sensor networks, potentially as a fog network extension. The sensors in sensor networks are coming under more scrutiny, and methods for securing these networks are still evolving. One of the challenging aspects is how to run IDS systems on such low power devices. However, SDN is being investigated as a potential aid to help manage these low power networks, using fog systems to run the SDN controller. The same process could happen with this system. Instead of attempting to run security systems on these low powered devices, move the processing to the fog, with the SDN flows serving as an indicator of malicious activity. Additionally, the sensor information itself could potentially serve as an invaluable source of data in the security process, taking the place of log data in the proposed solution. While sensor data monitoring temperature (for example) is clearly not a detailed log, combining the temperature reading with flow data at the time of malicious activity may give a better indication of that malicious activity (i.e. does the temperature fluctuate at the time of the malicious activity, or is the temperature reading missing entirely).

3. Traffic Shaping

   A change of focus could lead this research to being used as a way to balance legitimate traffic. Generally, SDN helps administrators balance network traffic through dynamically changing the flow rules based up measures such as traffic volume or packet size. However, using the NHFs provided in this work, along with logs consisting of server resource usage, it could be envisioned that rather than responding to malicious activity the system could predict loads that are going to be computationally expensive, and dynamically redirect them to servers that have a low load. Essentially the models would be predicting traffic that is going to be computationally expensive and directing it dynamically to ensure QoS requirements are maintained.

4. Semi-supervised or Unsupervised deep learning IDS

   The initial risk assessment could be expanded up on to create its own IDS. While the HDBScans results were not sufficient to rely upon as an IDS itself, it has been shown how adding additional small amounts of data can aid with results. It is possible that adding other small amounts of data could allow the risk assessment to perform well as an unsupervised IDS. This is something that was briefly looked into, however timing restrictions prevented a full analysis.

5. Other Network Types

   This work has been carried out on an assumption of it being placed into a midsized corporate network, large enough to realise the benefits of SDN, but also have access to traditional server logs. However, it could be expanded to other network types with different logs types. Backbone internet companies are frequently transitioning to using SDN, and though not looked at in this work, router logs could be a source of valuable information for this kind of tool (packets dropped due to age, or invalid destinations etc). The limitations however would still be the load required to run the service. As traffic increases, so will the load of the IMS and SDN controller, and so the viability of moving to backbone internet structure would need to be assessed. Digital Twins are an evolving area of research, and some of the work regarding logs could be transferred to there, potentially to detect attacks on physical assets by monitoring associated meta data and changes to that data.

References

[1] Cisco and I. Cisco Systems, “Cisco Visual Networking Index: Forecast and Trends, 2017–2022,” pp. 2017–2022, 2019, [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf.

[2] Cisco, “Cisco Visual Networking Index: Forecast and Methodology, 2015-2020,” 2015. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf.

[3] Open Networking Foundation, “ONF SDN Evolution,” 2016.

[4] I. Cisco Systems, “Software-Defined Networking : Why We Like It and How We Are Building On It,” 2013.

[5] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015, doi: 10.1038/nature14539.

[6] M. Ahmed, A. Naser Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” *Journal of Network and Computer Applications*, vol. 60. pp. 19–31, 2016, doi: 10.1016/j.jnca.2015.11.016.

[7] Verizon, “2018 Data Breach Investigations Report,” p. 7, 2018.

[8] Symantec Corporation, “ISTR Internet Security Threat Report Volume 24 | 02/19,” 2019.

[9] Verizon, “2016 Data Breach Investigations Report,” 2016.

[10] Symantec, “Internet Security Threat Report ISTR,” no. April, 2018, [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.

[11] IBM Services, “SDN Verses Traditional Networking Explained | IBM,” 2020. https://www.ibm.com/services/network/sdn-versus-traditional-networking (accessed Nov. 26, 2020).

[12] Cloudflare, “What is BGP? | BGP Routing Explained.” available: https://www.cloudflare.com/learning/security/glossary/what-is-bgp/.

[13] I. Cisco Systems, “Understanding Rapid Spanning Tree Protocol.” https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/24062-146.html.

[14] V. G. Ivakhnenko, A. G.; Lapa, *Cybernetic predicting devices*. United States. Joint Publications Research Service., 1965.

[15] L. Deng and D. Yu, “Deep Learning: Methods and Applications,” *Found. Trends® Signal Process.*, vol. 7, no. 3–4, pp. 197--387, 2013, doi: 10.1136/bmj.319.7209.0a.

[16] M. Z. Alom *et al.*, “A state-of-the-art survey on deep learning theory and architectures,” *Electron.*,

vol. 8, no. 3, 2019, doi: 10.3390/electronics8030292.

[17] J. E. van Engelen and H. H. Hoos, “A survey on semi-supervised learning,” *Mach. Learn.*, vol. 109, no. 2, pp. 373–440, 2020, doi: 10.1007/s10994-019-05855-6.

[18] G. Wilson and D. J. Cook, “A Survey of Unsupervised Deep Domain Adaptation,” *ACM Trans. Intell. Syst. Technol.*, vol. 11, no. 5, 2020, doi: 10.1145/3400066.

[19] G. Zhang, Y. Liu, and X. Jin, “A survey of autoencoder-based recommender systems,” *Front. Comput. Sci.*, vol. 14, no. 2, pp. 430–450, 2020, doi: 10.1007/s11704-018-8052-6.

[20] Google, “AlphaGo | Deepmind,” 2018. https://deepmind.com/research/alphago/.

[21] T.-Y. Mu, A. Al-Fuqaha, K. Shuaib, F. M. Sallabi, and J. Qadir, “SDN Flow Entry Management Using Reinforcement Learning,” vol. 13, no. 2, 2018, [Online]. Available: http://arxiv.org/abs/1809.09003.

[22] S. Hooda and S. Mann, “Distributed synthetic minority oversampling technique,” *Int. J. Comput. Intell. Syst.*, vol. 12, no. 2, pp. 929–936, 2019, doi: 10.2991/ijcis.d.190719.001.

[23] “SMOTE | Azure Machine Learning studio,” [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/smote.

[24] H. He, Y. Bai, E. A. Garcia, and S. Li, “ADASYN: Adaptive synthetic sampling approach for imbalanced learning,” *Proc. Int. Jt. Conf. Neural Networks*, no. 3, pp. 1322–1328, 2008, doi: 10.1109/IJCNN.2008.4633969.

[25] O. E. David and N. S. Netanyahu, “DeepSign: Deep learning for automatic malware signature generation and classification,” in *Proceedings of the International Joint Conference on Neural Networks*, 2015, vol. 2015-Septe, doi: 10.1109/IJCNN.2015.7280815.

[26] K. Gerrigagoitia, R. Uribeetxeberria, U. Zurutuza, and I. Arenaza, “Reputation-based Intrusion Detection System for wireless sensor networks,” *2012 IEEE Work. Complex. Eng. COMPENG 2012 - Proc.*, pp. 128–132, 2012, doi: 10.1109/CompEng.2012.6242969.

[27] Google, “Google Transparency Report.” Accessed: May 04, 2018. [Online]. Available: https://transparencyreport.google.com/safer-email/overview?hl=en.

[28] “NSL-KDD | Datasets | Research | Canadian Institute for Cybersecurity | UNB,” *Canadian Institute for Cybersecurity | University of New Brunswick*. http://www.unb.ca/cic/datasets/nsl.html (accessed Jul. 12, 2018).

[29] N. Moustafa and J. Slay, “The Significant Features of the UNSW-NB15 and the KDD99 Data Sets for Network Intrusion Detection Systems,” *2015 4th Int. Work. Build. Anal. Datasets Gather. Exp. Returns Secur.*, pp. 25–31, 2017, doi: 10.1109/badgers.2015.014.

[30] S. Zargari and T. Janarthanan, “Feature selection in UNSW-NB15 and KDDCUP’99 datasets,” in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, 2017, pp. 10–15, doi: 10.1109/ISIE.2017.8001537.

[31] N. Moustafa and J. Slay, “UNSW-NB15: A Comprehensive Data set for Network Intrusion Detection systems,” *Mil. Commun. Inf. Syst. Conf.*, vol. 8, pp. 1–6, 2015, doi: 10.1109/MilCIS.2015.7348942.

[32] A. Kurakin, I. J. Goodfellow, and S. Bengio, “Adversarial machine learning at scale,” *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, pp. 1–17, 2017.

[33] K. Ren, T. Zheng, Z. Qin, and X. Liu, “Adversarial Attacks and Defenses in Deep Learning,” *Engineering*, vol. 6, no. 3, pp. 346–360, 2020, doi: 10.1016/j.eng.2019.12.012.

[34] C. Szegedy *et al.*, “Intriguing properties of neural networks,” *2nd Int. Conf. Learn. Represent. ICLR 2014 - Conf. Track Proc.*, pp. 1–10, 2014.

[35] E. Alhajjar, P. Maxwell, and N. D. Bastian, “Adversarial Machine Learning in Network Intrusion Detection Systems,” pp. 1–25, 2020, [Online]. Available: http://arxiv.org/abs/2004.11898.

[36] K. Yang, J. Liu, C. Zhang, and Y. Fang, “Adversarial Examples Against the Deep Learning Based Network Intrusion Detection Systems,” *Proc. - IEEE Mil. Commun. Conf. MILCOM*, vol. 2019-Octob, pp. 559–564, 2019, doi: 10.1109/MILCOM.2018.8599759.

[37] ONF, “OpenFlow Switch Specification 1.4.0,” 2013.

[38] Candian Institute for Cyber Security, “No Title.” http://www.unb.ca/cic/datasets/nsl.html (accessed Jun. 07, 2018).

[39] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, “A detailed analysis of the KDD CUP 99 data set,” *IEEE Symp. Comput. Intell. Secur. Def. Appl. CISDA 2009*, no. Cisda, pp. 1–6, 2009, doi: 10.1109/CISDA.2009.5356528.

[40] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, “SSH Compromise Detection using NetFlow/IPFIX,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 20–26, 2014, doi: 10.1145/2677046.2677050.

[41] M. Azizjon, A. Jumabek, and W. Kim, “1D CNN based network intrusion detection with normalization on imbalanced data,” pp. 218–224, 2020, doi: 10.1109/icaiic48513.2020.9064976.

[42] B. A. Tama, M. Comuzzi, and K. H. Rhee, “TSE-IDS: A Two-Stage Classifier Ensemble for Intelligent Anomaly-Based Intrusion Detection System,” *IEEE Access*, vol. 7, pp. 94497–94507, 2019, doi: 10.1109/ACCESS.2019.2928048.

[43] H. He, X. Sun, H. He, G. Zhao, L. He, and J. Ren, “A Novel Multimodal-Sequential Approach Based on Multi-View Features for Network Intrusion Detection,” *IEEE Access*, vol. 7, pp. 183207–183221, 2019, doi: 10.1109/ACCESS.2019.2959131.

[44] Y. Su, “Research on network behavior anomaly analysis based on bidirectional LSTM,” *Proc. 2019 IEEE 3rd Inf. Technol. Networking, Electron. Autom. Control Conf. ITNEC 2019*, no. Itnec, pp. 798–802, 2019, doi: 10.1109/ITNEC.2019.8729475.

[45] M. Al-Zewairi, S. Almajali, and A. Awajan, “Experimental evaluation of a multi-layer feed-forward

artificial neural network classifier for network intrusion detection system," *Proc. - 2017 Int. Conf. New Trends Comput. Sci. ICTCS 2017*, vol. 2018-Janua, pp. 167–172, 2017, doi: 10.1109/ICTCS.2017.29.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017, doi: 10.1145/3065386.

[47] C. Szegedy *et al.*, "Going deeper with convolutions," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 07-12-June, pp. 1–9, 2015, doi: 10.1109/CVPR.2015.7298594.

[48] J. Kim, N. Shin, S. Y. Jo, and S. H. Kim, "Method of Intrusion Detection using Deep Neural Network," *Int. Conf. Big Data Smart Comput.*, pp. 313–316, 2017, doi: 10.1109/BIGCOMP.2017.7881684.

[49] M. Al-Qatf, Y. Lasheng, M. Al-Habib, and K. Al-Sabahi, "Deep Learning Approach Combining Sparse Autoencoder with SVM for Network Intrusion Detection," *IEEE Access*, vol. 6, pp. 52843–52856, 2018, doi: 10.1109/ACCESS.2018.2869577.

[50] G. Andresini, A. Appice, N. Di Mauro, C. Loglisci, and D. Malerba, "Multi-Channel Deep Feature Learning for Intrusion Detection," *IEEE Access*, vol. 8, pp. 53346–53359, 2020, doi: 10.1109/ACCESS.2020.2980937.

[51] S. Phetlasy, S. Ohzahata, C. Wu, and T. Kato, "Applying SMOTE for a sequential classifiers combination method to improve the performance of intrusion detection system," *Proc. - IEEE 17th Int. Conf. Dependable, Auton. Secur. Comput. IEEE 17th Int. Conf. Pervasive Intell. Comput. IEEE 5th Int. Conf. Cloud Big Data Comput. 4th Cyber Sci.*, pp. 255–258, 2019, doi: 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00054.

[52] M. Labonne, A. Olivereau, B. Polve, and D. Zeghlache, "A Cascade-structured Meta-Specialists Approach for Neural Network-based Intrusion Detection," *2019 16th IEEE Annu. Consum. Commun. Netw. Conf. CCNC 2019*, 2019, doi: 10.1109/CCNC.2019.8651856.

[53] B. Dong and X. Wang, "Comparison Deep Learning Method to Traditional Methods Using for Network Intrusion Detection," *8th IEEE Int. Conf. Commun. S oftw are N etw ork s*, pp. 581–585, 2016, doi: 10.1109/ICCSN.2016.7586590.

[54] Akashdeep, I. Manzoor, and N. Kumar, "A feature reduced intrusion detection system using ANN classifier," *Expert Syst. Appl.*, vol. 88, pp. 249–257, 2017, doi: 10.1016/j.eswa.2017.07.005.

[55] N. Praneeth, N. M. Varma, and R. R. Naik, "Principle component analysis based intrusion detection system using support vector machine," *2016 IEEE Int. Conf. Recent Trends Electron. Inf. Commun. Technol. RTEICT 2016 - Proc.*, pp. 1344–1350, 2017, doi: 10.1109/RTEICT.2016.7808050.

[56] M. Bahrololum, E. Salahi, and M. Khaleghi, "Machine learning techniques for feature reduction in intrusion detection systems: A comparison," *ICCIT 2009 - 4th Int. Conf. Comput. Sci. Converg. Inf. Technol.*, pp. 1091–1095, 2009, doi: 10.1109/ICCIT.2009.89.

[57] J. Liu and S. S. Chung, “Automatic feature extraction and selection for machine learning based intrusion detection,” *Proc. - 2019 IEEE SmartWorld, Ubiquitous Intell. Comput. Adv. Trust. Comput. Scalable Comput. Commun. Internet People Smart City Innov. SmartWorld/UIC/ATC/SCALCOM/IOP/SCI 2019*, pp. 1400–1405, 2019, doi: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00254.

[58] S. Shin, L. Xu, S. Hong, and G. Gu, “Enhancing Network Security through Software Defined Networking (SDN),” *2016 25th Int. Conf. Comput. Commun. Networks*, pp. 1–9, 2016, doi: 10.1109/ICCCN.2016.7568520.

[59] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, “FRESCO: Modular Composable Security Services for Software-Defined Networks,” vol. 2, no. February, 2013, doi: 10.1.1.297.7129.

[60] C. Yoon, T. Park, S. Lee, H. Kang, S. Shin, and Z. Zhang, “Enabling security functions with SDN: A feasibility study,” *Comput. Networks*, vol. 85, no. 2015, pp. 19–35, 2015, doi: 10.1016/j.comnet.2015.05.005.

[61] S. Scott-Hayward, G. O’Callaghan, and S. Sezer, “SDN security: A survey,” *SDN4FNS 2013 - 2013 Work. Softw. Defin. Networks Futur. Networks Serv.*, 2013, doi: 10.1109/SDN4FNS.2013.6702553.

[62] J. H. Jafarian, E. Al-Shaer, and Q. Duan, “Openflow random host mutation,” p. 127, 2012, doi: 10.1145/2342441.2342467.

[63] S. Shin and G. Gu, “Attacking software-defined networks: A first feasibility study,” *Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. Netw.*, pp. 165–166, 2013, doi: 10.1145/2491185.2491220.

[64] S. Shin, “AVANT-GUARD : Scalable and Vigilant Switch Flow Management in Software-Defined Networks Categories and Subject Descriptors,” *Proc. 2013 ACM SIGSAC Conf. Comput. Commun. Secur.*, pp. 413–424, 2013.

[65] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, “Deep learning approach for Network Intrusion Detection in Software Defined Networking,” *2016 Int. Conf. Wirel. Networks Mob. Commun.*, pp. 258–263, 2016, doi: 10.1109/WINCOM.2016.7777224.

[66] Q. Niyaz, W. Sun, and A. Y. Javaid, “A Deep Learning Based DDoS Detection System in Software-Defined Networking (SDN),” no. Dl, pp. 1–18, 2016, [Online]. Available: http://arxiv.org/abs/1611.07400.

[67] S. Souissi, “A novel response-oriented attack classification,” *Int. Conf. Protoc. Eng. ICPE 2015 Int. Conf. New Technol. Distrib. Syst. NTDS 2015 - Proc.*, 2015, doi: 10.1109/NOTERE.2015.7293480.

[68] M. Fu, G. Yi, Z. Wang, and L. Zhang, “A security threats taxonomy for routing system intrusion detection,” *Proc. - 12th Int. Conf. Comput. Intell. Secur. CIS 2016*, pp. 267–270, 2017, doi:

10.1109/CIS.2016.67.

[69] Z. Wu, Y. Ou, and Y. Liu, “A taxonomy of network and computer attacks based on responses,” *Proc. - 2011 Int. Conf. Inf. Technol. Comput. Eng. Manag. Sci. ICM 2011*, vol. 1, pp. 26–29, 2011, doi: 10.1109/ICM.2011.363.

[70] C. Simmons, C. Ellis, S. Shiva, D. Dasgupta, and Q. Wu, “AVOIDIT: A Cyber Attack Taxonomy,” 2009, doi: 10.1016/0002-9343(81)90253-9.

[71] A. Binbusayyis and T. Vaiyapuri, “Identifying and Benchmarking Key Features for Cyber Intrusion Detection: An Ensemble Approach,” *IEEE Access*, vol. 7, pp. 106495–106513, 2019, doi: 10.1109/ACCESS.2019.2929487.

[72] M. A. Aydin, A. H. Zaim, and K. G. Ceylan, “A hybrid intrusion detection system design for computer network security,” *Comput. Electr. Eng.*, vol. 35, no. 3, pp. 517–526, 2009, doi: 10.1016/j.compeleceng.2008.12.005.

[73] Z. Chiba, N. Abghour, K. Moussaid, A. El Omri, and M. Rida, “A Cooperative and Hybrid Network Intrusion Detection Framework in Cloud Computing Based on Snort and Optimized Back Propagation Neural Network,” *Procedia Comput. Sci.*, vol. 83, pp. 1200–1206, 2016, doi: 10.1016/j.procs.2016.04.249.

[74] E. Bursztein, “Inside the infamous Mirai IoT Botnet: A Retrospective Analysis,” *CloudFlare*, 2017. https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/ (accessed Jun. 07, 2018).

[75] C. J. Hutto and E. E. Gilbert, “VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. Eighth International Conference on Weblogs and Social Media (ICWSM-14).”,” *Proc. 8th Int. Conf. Weblogs Soc. Media, ICWSM 2014*, 2014, [Online]. Available: http://sentic.net/.

[76] S. Hublikar, V. Eligar, and A. Kakhandki, “Detecting Denial-of-Service Attacks Using sFlow,” in *Inventive Communication and Computational Technologies*, 2020, pp. 483–491.

[77] R. M. A. Ujjan, Z. Pervez, K. Dahal, A. K. Bashir, R. Mumtaz, and J. González, “Towards sFlow and adaptive polling sampling for deep learning based DDoS detection in SDN,” *Futur. Gener. Comput. Syst.*, vol. 111, pp. 763–779, 2020, doi: 10.1016/j.future.2019.10.015.

[78] L. McInnes, J. Healy, and S. Astels, “Benchmarking Performance and Scaling of Python Clustering Algorithms - hdbscan 0.8.1 documentation,” 2020. https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html.

[79] H. Chawla and P. Khattar, “Data Preparation and Training Part I,” in *Data Lake Analytics on Microsoft Azure: A Practitioner’s Guide to Big Data Engineering*, Berkeley, CA: Apress, 2020, pp. 99–142.

[80] M. J. Willemink *et al.*, “Preparing medical imaging data for machine learning,” *Radiology*, vol.

295, no. 1, pp. 4–15, 2020, doi: 10.1148/radiol.2020192224.

[81] T. G. Dietterich and E. B. Kong, “Machine Learning Bias, Statistical Bias, and Statistical Variance of Decision Tree Algorithms,” pp. 0–13, 1995.

[82] D. Jing and H. B. Chen, “SVM based network intrusion detection for the UNSW-NB15 dataset,” *Proc. Int. Conf. ASIC*, pp. 1–4, 2019, doi: 10.1109/ASICON47005.2019.8983598.

[83] R. U. Khan, X. Zhang, M. Alazab, and R. Kumar, “An improved convolutional neural network model for intrusion detection in networks,” *Proc. - 2019 Cybersecurity Cyberforensics Conf. CCC 2019*, no. Ccc, pp. 74–77, 2019, doi: 10.1109/CCC.2019.000-6.

[84] Y. Yan, L. Qi, J. Wang, Y. Lin, and L. Chen, “A Network Intrusion Detection Method Based on Stacked Autoencoder and LSTM,” *IEEE Int. Conf. Commun.*, vol. 2020-June, 2020, doi: 10.1109/ICC40277.2020.9149384.

[85] G. Larsson, M. Maire, and G. Shakhnarovich, “FractalNet: Ultra-Deep Neural Networks without Residuals,” pp. 1–11, 2016, [Online]. Available: http://arxiv.org/abs/1605.07648.

[86] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 2261–2269, 2017, doi: 10.1109/CVPR.2017.243.

[87] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, “Striving for simplicity: The all convolutional net,” *3rd Int. Conf. Learn. Represent. ICLR 2015 - Work. Track Proc.*, pp. 1–14, 2015.

[88] V. Christlein, L. Spranger, M. Seuret, A. Nicolaou, P. Kral, and A. Maier, “Deep generalized max pooling,” *Proc. Int. Conf. Doc. Anal. Recognition, ICDAR*, no. 1, pp. 1090–1096, 2019, doi: 10.1109/ICDAR.2019.00177.

[89] H. Wu and X. Gu, “Max-pooling dropout for regularization of convolutional neural networks,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9489, pp. 46–54, 2015, doi: 10.1007/978-3-319-26532-2_6.

[90] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, “How does batch normalization help optimization?,” *Adv. Neural Inf. Process. Syst.*, vol. 2018-Decem, no. NeurIPS, pp. 2483–2493, 2018.

[91] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *32nd Int. Conf. Mach. Learn. ICML 2015*, vol. 1, pp. 448–456, 2015.

[92] J. Yan, R. Wan, X. Zhang, W. Zhang, Y. Wei, and J. Sun, “Towards Stabilizing Batch Statistics in Backward Propagation of Batch Normalization,” pp. 1–17, 2020, [Online]. Available: http://arxiv.org/abs/2001.06838.

[93] E. Phaisangittisagul, “An Analysis of the Regularization Between L2 and Dropout in Single Hidden Layer Neural Network,” *Proc. - Int. Conf. Intell. Syst. Model. Simulation, ISMS*, vol. 0,

pp. 174–179, 2016, doi: 10.1109/ISMS.2016.14.

[94] Y. Liu, X. Gao, Q. Gao, L. Shao, and J. Han, “Adaptive robust principal component analysis,” *Neural Networks*, vol. 119, pp. 85–92, 2019, doi: 10.1016/j.neunet.2019.07.015.

[95] Z. Cui, F. Li, and W. Zhang, “Bat algorithm with principal component analysis,” *Int. J. Mach. Learn. Cybern.*, vol. 10, no. 3, pp. 603–622, 2019, doi: 10.1007/s13042-018-0888-4.

[96] J. Wu, D. Peng, Z. Li, L. Zhao, and H. Ling, “Network intrusion detection based on a general regression neural network optimized by an improved artificial immune algorithm,” *PLoS One*, 2015, doi: 10.1371/journal.pone.0120976.

[97] S. Bock and M. Weis, “A Proof of Local Convergence for the Adam Optimizer,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2019-July, no. July, pp. 1–8, 2019, doi: 10.1109/IJCNN.2019.8852239.

[98] A. C. Zamfira and H. Ciocarlie, “Developing an ontology of cyber-operations in networks of computers,” *Proc. - 2018 IEEE 14th Int. Conf. Intell. Comput. Commun. Process. ICCP 2018*, pp. 395–400, 2018, doi: 10.1109/ICCP.2018.8516644.

[99] R. Hofstede *et al.*, “Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX,” *IEEE Commun. Surv. Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014, doi: 10.1109/COMST.2014.2321898.

[100] scikit-learn developers, “sklearn.decomposition.PCA - scikit-learn 0.23.2 documentation.” https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html (accessed Nov. 05, 2020).

[101] A. International, P. Reviewed, T. S. Urmila, and R. Balasubramanian, “Journal of Analysis and Computation (JAC) AN EFFICIENT ALGORITHM FOR FEATURE SELECTION IN IDS,” pp. 1–8.

[102] A. A. Salih and M. B. Abdulrazaq, “Combining Best Features Selection Using Three Classifiers in Intrusion Detection System,” *2019 Int. Conf. Adv. Sci. Eng. ICOASE 2019*, pp. 94–99, 2019, doi: 10.1109/ICOASE.2019.8723671.

[103] A. A. Sallam, M. N. Kabir, Y. M. Alginahi, A. Jamal, and T. K. Esmeel, “IDS for Improving DDoS Attack Recognition Based on Attack Profiles and Network Traffic Features,” *Proc. - 2020 16th IEEE Int. Colloq. Signal Process. its Appl. CSPA 2020*, no. Cspa, pp. 255–260, 2020, doi: 10.1109/CSPA48992.2020.9068679.

[104] S.-Y. Zhao, Y.-P. Xie, and W.-J. Li, “Stagewise Enlargement of Batch Size for SGD-based Learning,” no. 1, pp. 1–26, 2020, [Online]. Available: http://arxiv.org/abs/2002.11601.

[105] Y. You, Y. Wang, H. Zhang, Z. Zhang, J. Demmel, and C.-J. Hsieh, “The Limit of the Batch Size,” pp. 1–22, 2020, [Online]. Available: http://arxiv.org/abs/2006.08517.

[106] M. Banton, N. Shone, W. Hurst, and Q. Shi, “Intrusion Detection Using Extremely Limited Data

Based on SDN," in *Proceedings of 2020 IEEE 10th International Conference on Intelligent Systems Intrusion*, 2020, pp. 304–309.

[107] I. Loshchilov and F. Hutter, "CMA-ES for Hyperparameter Optimization of Deep Neural Networks," no. 2001, 2016, [Online]. Available: http://arxiv.org/abs/1604.07269.

[108] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012, doi: 10.1162/153244303322533223.

[109] A. C. Florea and R. Andonie, "Weighted Random Search for hyperparameter optimization," *Int. J. Comput. Commun. Control*, vol. 14, no. 2, pp. 154–169, 2019, doi: 10.15837/ijccc.2019.2.3514.

[110] A. Nugroho and H. Suhartanto, "Hyper-Parameter Tuning based on Random Search for DenseNet Optimization," in *7th International Conference on Information Technology, Computer, and Electrical Engineering, ICITACEE 2020*, 2020, pp. 96–99, doi: https://doi.org/10.1109.

[111] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," pp. 1–18, 2012, [Online]. Available: http://arxiv.org/abs/1207.0580.

[112] K. Sethi, R. Kumar, N. Prajapati, and P. Bera, "Deep Reinforcement Learning based Intrusion Detection System for Cloud Infrastructure," *2020 Int. Conf. Commun. Syst. NETworkS, COMSNETS 2020*, pp. 1–6, 2020, doi: 10.1109/COMSNETS48256.2020.9027452.

[113] J. Zhang, F. Li, and F. Ye, "An Ensemble-based Network Intrusion Detection Scheme with Bayesian Deep Learning," *IEEE Int. Conf. Commun.*, vol. 2020-June, pp. 11–16, 2020, doi: 10.1109/ICC40277.2020.9149402.

[114] Z. Chkirbene, S. Eltanbouly, M. Bashendy, N. Alnaimi, and A. Erbad, "Hybrid Machine Learning for Network Anomaly Intrusion Detection," *2020 IEEE Int. Conf. Informatics, IoT, Enabling Technol. ICIoT 2020*, pp. 163–170, 2020, doi: 10.1109/ICIoT48696.2020.9089575.

[115] Y. Xiao, C. Xing, T. Zhang, and Z. Zhao, "An Intrusion Detection Model Based on Feature Reduction and Convolutional Neural Networks," *IEEE Access*, vol. 7, pp. 42210–42219, 2019, doi: 10.1109/ACCESS.2019.2904620.

[116] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning," *12th USENIX Symp. Oper. Syst. Des. Implement. (OSDI '16)*, pp. 265–284, 2016, doi: 10.1038/nn.3331.