

Machine Learning-based Intrusion Detection for Virtual Infrastructures

Mayank Arora, BAI

A Dissertation

Presented to the University of Dublin, Trinity College
in partial fulfilment of the requirements for the degree of

Integrated Masters in Computer Engineering

Supervisor: Dr. Stefan Weber

April 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Mayank Arora

April 19, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Mayank Arora

April 19, 2022

Machine Learning-based Intrusion Detection for Virtual Infrastructures

Mayank Arora, Integrated Masters in Computer Engineering
University of Dublin, Trinity College, 2022

Supervisor: Dr. Stefan Weber

There has been a shift from running applications in virtual machine-based environments to container-based environments in recent years. Although this shift has provided a better platform for deploying scalable applications, the tools to secure container-based environments from unknown attacks are still being developed.

This research has focused on enhancing security for applications deployed in Kubernetes. In particular, a Network-based Intrusion Detection System (NIDS) was proposed that uses an autoencoder, an unsupervised artificial neural network, to flag potentially anomalous packets. The autoencoder was trained on packets flowing through a Kubernetes node running an application, which created a baseline for normal behaviour. For testing the IDS, various port scans and dictionary attacks were launched against the application deployed in the virtual infrastructure. The IDS accurately detected the deviation from expected behaviour and categorised the packets as anomalous.

As part of the evaluation, we also compared autoencoders against other unsupervised detection algorithms such as IsolationForest and DBSCAN, and a supervised learning approach. Preliminary results show that a combination of signature-based and machine learning approaches is necessary for a comprehensive detection of intrusions in Kubernetes cluster.

Acknowledgments

First and foremost, I would like to thank my supervisor Stefan Weber for his unbelievable support, patience and guidance throughout the project.

The completion of this project could not have been possible without the support of my incredible partner Noor and my friends Abhinav, Piyush, Ian, Iga, Ciara, Conor, Dáire, Mark and David. Thank you for listening to all of my rantings and ravings about Kubernetes and being there to go for coffee at a moment's notice.

Finally, I'd like to thank my family for their unwavering support and belief in me throughout these twenty years of my academic journey. No mum, I'm not doing a PhD.

MAYANK ARORA

*University of Dublin, Trinity College
April 2022*

Contents

Abstract	iii
Acknowledgments	iv
Chapter 1 Introduction	1
1.1 Problem area	1
1.2 Research objectives	2
1.3 Structure of the report	2
Chapter 2 State of the Art	3
2.1 Background	3
2.1.1 Virtual Infrastructure	3
2.1.1.1 Brief history of containers	3
2.1.1.2 Need for containers	4
2.1.1.3 VMs vs containers	4
2.1.1.4 Kubernetes architecture	5
2.1.1.5 Kubernetes networking	7
2.1.2 Machine Learning	8
2.1.2.1 Types of machine learning algorithms	8
2.1.2.2 Classification of anomaly detection	9
2.1.2.3 Anomaly detection using density based clustering	10
2.1.2.4 Anomaly detection using neural network	12
2.1.2.5 Anomaly detection using tree-based technique	12
2.1.3 Intrusion Detection System	13
2.1.3.1 Taxonomy of cyber attacks	13
2.1.3.2 Classification of an IDS based on analysed activity	14
2.1.3.3 Classification of an IDS based on detection method	15
2.1.3.4 Classification of an IDS based on behaviour on detection	15
2.1.3.5 Anomaly-based NIDS	15

2.2	Related projects	17
2.2.1	Monitoring Kubernetes Clusters With Dedicated Sidecar Network Sniffing Containers	17
2.2.2	Application of Machine Learning with Traffic Monitoring to Intrusion Detection in Kubernetes Deployments	18
2.2.3	Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge	18
2.2.4	Unsupervised Packet-based Anomaly Detection in Virtual Networks	19
2.3	Summary	19
Chapter 3 Design		20
3.1	Deployment platform	20
3.1.1	Local deployment	20
3.1.2	Cloud deployment	20
3.1.3	Docker images	21
3.2	Architecture of the NIDS	21
3.3	Collecting data	22
3.4	Machine learning environment	23
3.5	Workflow	24
3.6	Summary	25
Chapter 4 Implementation		26
4.1	Deployment of Kubernetes cluster	26
4.1.1	Local deployment	26
4.1.2	Cloud deployment	27
4.1.3	Differences in local vs cloud deployment	29
4.2	Implementing system architecture	29
4.2.1	Tcpdump container	29
4.2.1.1	Type of pod	29
4.2.1.2	Capturing tcpdump	31
4.2.1.3	Sending pcap file to pcap-service	32
4.2.1.4	Creating tcpdump docker image	33
4.2.1.5	Deploying and collecting data	34
4.2.2	Pcap service	35
4.2.2.1	Flask server	35
4.2.2.2	Processing pcap	36
4.2.3	Model service	37

4.2.3.1	Preprocessing data	37
4.2.3.2	Predicting scores	39
4.2.4	MySQL database	41
4.2.5	Back-end service	42
4.2.5.1	Flask server	42
4.3	Simulating an attack	43
4.3.1	Port scanning	44
4.3.2	Dictionary attack	46
4.4	Creating machine learning models	49
4.4.1	Data collection	50
4.4.2	Data preprocessing	51
4.4.2.1	Processing pcap	51
4.4.2.2	Processing CSV	56
4.4.3	Clustering using DBSCAN	57
4.4.4	Anomaly detection using Isolation Forest	59
4.4.4.1	Anomaly detection using packet flow	59
4.4.4.2	Anomaly detection using packet data	61
4.4.5	Autoencoders	62
4.4.5.1	Training an autoencoder model	62
4.4.5.2	Scoring	63
4.4.6	Supervised Learning	66
Chapter 5	Evaluation	68
5.1	Evaluating machine learning models	68
5.1.1	Clustering using DBSCAN	68
5.1.1.1	Dimensionality reduction	68
5.1.1.2	Evaluating DBSCAN	69
5.1.2	Isolation Forest	70
5.1.3	Autoencoders	70
5.1.3.1	Model 1	71
5.1.3.2	Model 2	73
5.1.3.3	Model 3	74
5.1.3.4	Model 4	75
5.1.3.5	Model 5	76
5.1.3.6	Limitations of autoencoders	77
5.1.4	Supervised learning	77
5.2	Evaluating the architecture	78

5.2.1	Limitations of the prototype	78
5.2.1.1	Tcpdump container	78
5.2.1.2	Pcap service	78
5.2.1.3	Model service	79
5.2.1.4	MySQL database	79
5.2.1.5	Backend service	79
5.2.2	Improved architecture for the prototype	79
Chapter 6 Conclusions & Future Work		82
6.1	Conclusion	82
6.2	Future work	83
6.2.1	NIDS prototype	83
6.2.2	Machine learning	83
6.3	Reflection	84
Bibliography		84
Appendices		89
.1	Difference in sending requests from different pods	90
.2	Code for polling script in tcpdump-container	91
.3	YAML files for MySQL in Kubernetes	92
.4	Console outputs of Nmap and WPScan	96
.5	Falco scanner in Kubernetes	97
.6	Calculating packet flow using destination IP	98

List of Tables

4.1	Final packet features selected for the NIDS	36
4.2	Processing of packet features in table 4.1	38
4.3	Total packets captured for training and testing ML models	50
4.4	All features extracted from a packet for this project	56
4.5	Features of the best autoencoder model	62
5.1	Features selected for different models	71
5.2	Summary of evaluation of autoencoder models	72

List of Figures

2.1	VM based architecture	4
2.2	Container based architecture	5
2.3	Components of a Kubernetes cluster	6
2.4	Networking in a Kubernetes node with Kubenet plugin)	7
2.5	ARP requests captured on cbr0	8
2.6	Taxonomy of machine learning algorithms	9
2.7	Illustration of DBSCAN	11
2.8	Kmeans vs DBSCAN	11
2.9	Architecture of an autoencoder	12
2.10	Visualisation of Isolation Forest	13
2.11	Mitre attack framework 2020	14
2.12	Classification of different anomaly based intrusion detection techniques . .	17
3.1	Micro-services based architecture of the protoype	21
3.2	Workflow of a Kubernetes developer	24
4.1	Console output of minikube start	27
4.2	Console output of kubectl	27
4.3	Deploying configuration for AKS	28
4.4	Environment variables in tcpdump-container	32
4.5	Features extracted from the pcap being stored in CSV	37
4.6	Protocol in packet being one hot encoded	38
4.7	Strcuture of the data frame being stored in MySQL	40
4.8	Screenshot of webpage displaying top 1000 anomalous packets	43
4.9	Tcpdump output of the nmap scan in Wireshark	46
4.10	Wireshark output of packets from attacker’s IP during WPSan attack . .	47
4.11	Output of the WPSan	49
4.12	Normal vs Attack packet flow	50
4.13	Azure networking IP ranges	55
4.14	Data points created for visualising DBSCAN	57

4.15	Predicted anomalies by DBSCAN for figure 4.14	58
4.16	Output of DBSCAN on normal data flow	59
4.17	Isolation Forest for predicting attack data flow	61
4.18	Score distribution of packets in training data	64
4.19	Score distribution of packets during attack	65
4.20	Score distribution of packets with attacker's IP	65
5.1	Performance comparison of various dimensionality reduction algorithms	69
5.2	Score distribution of packets during attack (model 1)	72
5.3	Score distribution of packets with attacker's IP (model 1)	72
5.4	Score distribution of packets during attack (model 2)	73
5.5	Score distribution of packets with attacker's IP (model 2)	73
5.6	Score distribution of packets during attack (model 3)	74
5.7	Score distribution of packets with attacker's IP (model 3)	74
5.8	Score distribution of packets during attack (model 4)	75
5.9	Score distribution of packets with attacker's IP (model 4)	75
5.10	Score distribution of packets during attack (model 5)	76
5.11	Score distribution of packets with attacker's IP (model 5)	76
5.12	Confusion matrix for supervised learning	78
5.13	Performance comparison of different packet analysing libraries	79
5.14	A better data processing model	80
1	Sending request from a pod to a service	90
2	Sending request from tcpdump-container pod	90
3	Console output of the WPScan scan	96
4	Output of the Nmap scan script	97
5	Console output of Falco scanner while running NIDS	98

Acronyms

AKS Azure Kubernetes Services.

API Application Programming Interface.

ARP Address Resolution Protocol.

CIDR Classless Inter-Domain Routing.

DBSCAN Density-Based Spatial Clustering of Applications with Noise.

HIDS Host-based Intrusion Detection System.

IDS Intrusion Detection System.

ML Machine Learning.

NIDS Network-based Intrusion Detection System.

NPDF Normal Packet Data Flow.

PAIP Packets with Attacker's IP.

PDA Packets During Attack.

VM Virtual Machine.

Listings

4.1	YAML file creating a deployment for a privileged pod	31
4.2	Kterm alias for getting a bash shell into a pod	31
4.3	Bash script to capture packets on bridge cbr0 using tcpdump	32
4.4	Snippet of polling code to send files using requests library in Python	33
4.5	Dockerfile for tcpdump-container image	34
4.6	Bash commands for multi-platform build on docker	34
4.7	Kubectl command to create a resource using a YAML file	34
4.8	Bash commands for multi-platform build on docker	34
4.9	Python code for receiving files via POST request	36
4.10	Snippet of YAML file for pcap service that shows how to run a command to start scripts	37
4.11	Code for creating packet flow feature from timestamp	39
4.12	Python code to load a CSV file into MySQL database	40
4.13	Creating pcap_table in MySQL to store CSVs	41
4.14	Setting local_infile to true to load files	42
4.15	SQL command to retrieve top 1000 anomalous packets going out of the cluster	42
4.16	SQL command to retrieve data list of IPs trying to contact port 3306, the open port for MySQL	43
4.17	Nmap script to scan open ports on a website	44
4.18	Snippet of console output of the nmap scan script	46
4.19	WPScan command for password attack	46
4.20	Snippet of console output of WPScan	48
4.21	Dockerfile for Pcap service	51
4.22	Console output of above above function	52
4.23	Console output of above above function	52
4.24	Extracting IP.src from from a packet	53
4.25	Extracting flags set in a TCP packet	53
4.26	Extracting raw packet data	54

4.27	Classifying IP as external or internal	55
4.28	creating blobs	58
4.29	Python code to create packet flow feature using a unix timestamp	60
4.30	Creating a model using Isolation Forest implementation in sklearn	60
4.31	Keras code to create the architecture of Figure 2.9	62
4.32	Predicting scores using model	63
4.33	Scores for normal data	64
4.34	Keras code to create the architecture of Figure 2.9	66
4.35	Keras code to create the architecture of Fi	66
5.1	A better way to parse packets using Scapy	78
1	Dockerfile for Pcap service	92
2	YAML file for creating PV and PVC for a MySQL database	93
3	YAML file for a MySQL database	94
4	Python snippet to calculate packet flow using destination IP of the packet	99

Chapter 1

Introduction

This research aims to investigate the performance of various anomaly-based Machine Learning (ML) algorithms that can be used in a Network Intrusion Detection System (NIDS) for virtual infrastructures.

Section 1.1 provides an overview of the current cybersecurity landscape and the need to enhance the security of Kubernetes containers. Section 1.2 defines the objectives of the research, and section 1.3 briefly explains the structure of the report.

1.1 Problem area

In 2021, the total number of internet users stood at 4.9 billion, up from 4.6 billion the previous year (1). Widespread adoption of potentially vulnerable applications among this growing userbase can become a target of cyberattacks. In 2021 alone, there was a 125% increase in the incident volume of cyber-attacks. These trends show that cyberattacks could potentially increase at a much faster rate in the coming years.

In the last two decades, there has been a shift from organizations running applications on physical servers to using cloud providers for managed virtual servers and, now, to deploying applications in container-based virtual infrastructures. This has also given cybercriminals a larger attack surface. Kubernetes, a popular platform for managing containerized workloads, has become a significant target for cybercriminals. It has been seen that cybercriminals may attempt to use Kubernetes to harness a network's underlying infrastructure for computational power for purposes such as cryptocurrency mining (2)(3). Also, misconfigurations in these virtual deployments can lead to vulnerabilities that can be exploited by a cyber attacker (4).

1.2 Research objectives

Given the current state of increasing cyber security attacks and the shift to container-based environments for deploying applications, the primary objective of this project was to determine a way to enhance the security of a Kubernetes cluster against unknown attacks. This research explores several anomaly-based ML algorithms which can be used to create a NIDS for a virtual infrastructure like Kubernetes. This research also explored the feasibility and limitations of an ML-based NIDS against 'real world' threats in a cloud environment.

1.3 Structure of the report

This section describes the chapters in the document.

- Chapter 2 provides the background information for this project and describes the anomaly detection algorithms like IsolationForest, DBSCAN and Autoencoders. It also describes four projects that were related to this project.
- Chapter 3 describes the design decisions taken to create a prototype for anomaly-based NIDS. It also explains the data collection process for machine learning models and the various tools and libraries needed to create a machine learning environment.
- Chapter 4 describes the whole implementation process of the prototype in detail, from creating and deploying NIDS to AKS to implementing various machine learning models.
- Chapter 5 evaluates the machine learning approaches and gives their limitations. It also evaluates the NIDS and gives architecture for a better prototype.
- Chapter 6 gives overall conclusions for the project and highlights the critical findings. It also talks about potential future works regarding the project that would enhance the capabilities of the NIDS.

Chapter 2

State of the Art

This chapter introduces the current state of the art in Kubernetes, machine learning and various kinds of Intrusion detection systems. Section 2.1 provides the background information for the things implemented in the project. Section 2.2 explains the projects that were similar to this research and their limitations, and section 2.3 provides the summary for this chapter.

2.1 Background

This section provides the context for the research done and the prototype implemented. Section 2.1.1 provides the history and needs for container-based applications and moves to Kubernetes and its internal networking. Section 2.1.2 explains in detail the various several anomaly-based ML algorithms, and section 2.1.3 talks about the various types of intrusion detection systems and how they can be classified based on different attributes.

2.1.1 Virtual Infrastructure

2.1.1.1 Brief history of containers

The concept of containers first emerged in 1979 with the Unix V7 operation system. It introduced the `chroot` system call, which changed the root directory of a process and its children to a new location, which was essentially the beginning of process isolation (5). In the early 2000s, Linux VServer (6) introduced kernel level isolation on a physical machine and in 2006, Google introduced the concept of Process Containers(7) which aimed to isolate the resource usage of a collection of processes. In 2015, Docker opened its container format and runtime `libcontainer`, which accelerated the development of containers and container-based solutions(8).

2.1.1.2 Need for containers

In the early 1990s, organizations had to run their application code on physical servers. They did not have any way to define boundaries on a resource, giving rise to resource allocation issues for application code running on the same hardware. A solution to this is scaling vertically, i.e., increasing resources on a machine, and horizontally, i.e., increasing the number of machines. However, this method was rather expensive, and it was difficult for organizations to maintain on-premise servers.

In the early 2000s, virtualization technology had matured, and it was possible to run multiple Virtual Machines on one physical server. It allowed companies better utilization of expensive resources and the ability to scale up and down according to their needs. Around the 2010s, the rise of service-oriented architecture (SOA) and containers changed the way applications we run. Although they were similar to VMs, they were lightweight, which reduced the time to spin up a container, and their reduced size meant one physical machine could handle many containers.

2.1.1.3 VMs vs containers

A Virtual Machine(VM) emulates a physical computer; it has a dedicated amount of memory, CPU, and storage borrowed from the host computer. A VM is always partitioned on a computer to not interfere with the host OS(9). Even if a VM is compromised, other VMs running on the same machine remain unaffected. A hypervisor, which runs on the host OS, acts as a middleman and allows multiple VMs to run on a single machine. It allocates the infrastructure a VM requires to run. VMs may also include a complementary software stack to run on the emulated hardware, providing a complete snapshot of the system(10). A VM image is usually many gigabytes in size.

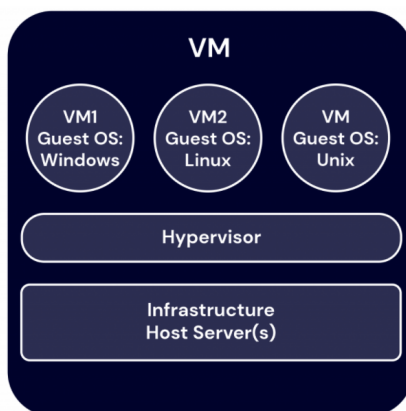


Figure 2.1: VM based architecture (11)

A container is an executable piece of software that runs on a container engine. A

container image, which is executed on container engines to create containers, includes everything needed to run an application: code, runtime, system tools, system libraries and settings.(12). A containerized app perceives that it has the OS—including CPU, memory, file storage, and network connections—all to itself (13). Multiple containers can share the host OS kernel and libraries.

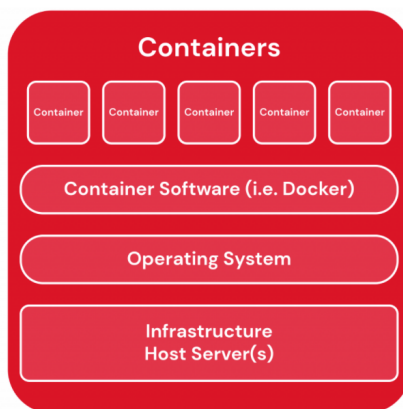


Figure 2.2: Container based architecture(11)

While a VM virtualizes the underlying hardware, a container virtualizes the underlying OS(10). The main argument for using containers instead of VMs is how lightweight the former is. As each VM includes a separate operating system image, it adds a significant amount of overhead on the memory and storage of the host machine. This reduces the speed of the software development lifecycle(14). As containers share the host OS and libraries, a container image is only megabytes in size and thus very lightweight as compared to a VM image. It has a faster spin-up time which improves a developer’s ability to iterate faster. If there is an existing monolithic application to manage and requires complete isolation and enhanced security, VMs are a better choice for that use case. Containers only offer process-level isolation and are more suitable for microservices-based architecture.

2.1.1.4 Kubernetes architecture

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services. (15). It provides a framework to run distributed systems in a resilient manner. It also helps in service discovery and load balancing. If traffic to a container is high, it can automatically horizontally scale up the service, distribute the load and keep the deployment stable. One of the major features is self-healing, i.e. it automatically restarts failed containers and replaces containers that do not respond to health checks. Traffic is not sent to a container until it is ready to serve requests.

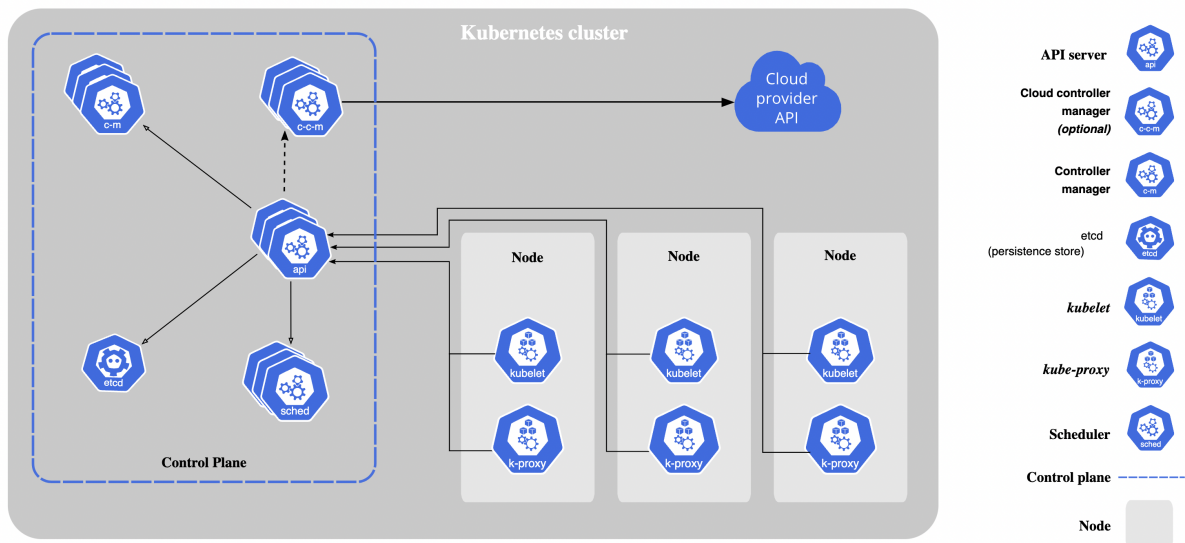


Figure 2.3: Components of a kubernetes cluster(15)

A Kubernetes cluster consists of nodes that can run containerized applications. A worker node hosts pods, the smallest deployable units in Kubernetes. A pod can contain one or more containers. As seen in figure 2.3, a Kubernetes deployment consists of a control plane, which manages the worker nodes.

The components of a control plane are:

- API Server: The API server exposes the Kubernetes API. It acts as the front end of the control plane.
- Etc: It is a highly available and consistent key-value data store that manages all the cluster data.
- Kube-scheduler: It helps with the scheduling of pods on nodes. While scheduling pods, it considers the resource requirements, affinity specifications (a spec to deploy a pod to a particular node) and policy constraints.
- Kube-controller-manager: It is the control plane component that runs the controller process, which can be a node controller, endpoints controller, and job controller. The responsibilities of all the controllers are different; for e.g. a node controller is responsible for responding when a node goes down.

A Kubernetes cluster consists of multiple worker nodes, whose components include:

- Kubelet: It is an agent that runs on each node, ensuring that the pods on the node are healthy.

- Kube-proxy: It is a network proxy that implements part of the Kubernetes service concept. It does that by maintaining network rules which allow communication to and from a pod.
- Container runtime: Container runtime is the software that helps run containers on a node.

2.1.1.5 Kubernetes networking

In Kubernetes, each node has its network namespace, **root netns** as shown in figure 2.4. It has its own ethernet interface **eth0**. Also, each pod has its own IP address where every pod in the cluster can contact it. Hence a pod has its own network namespace, as shown by **pod1 netns** and **pod2 netns**. The namespaces of the two pods are isolated. The network namespaces of pods also have their own **eth0**. To facilitate inter pod communication, a pod must have access to the **root netns**. This is done using a virtual ethernet pipe pair; for example, the **eth0** of the pod is connected to **root netns** via **vethxx**. This virtual ethernet device acts as a tunnel between these two namespaces.

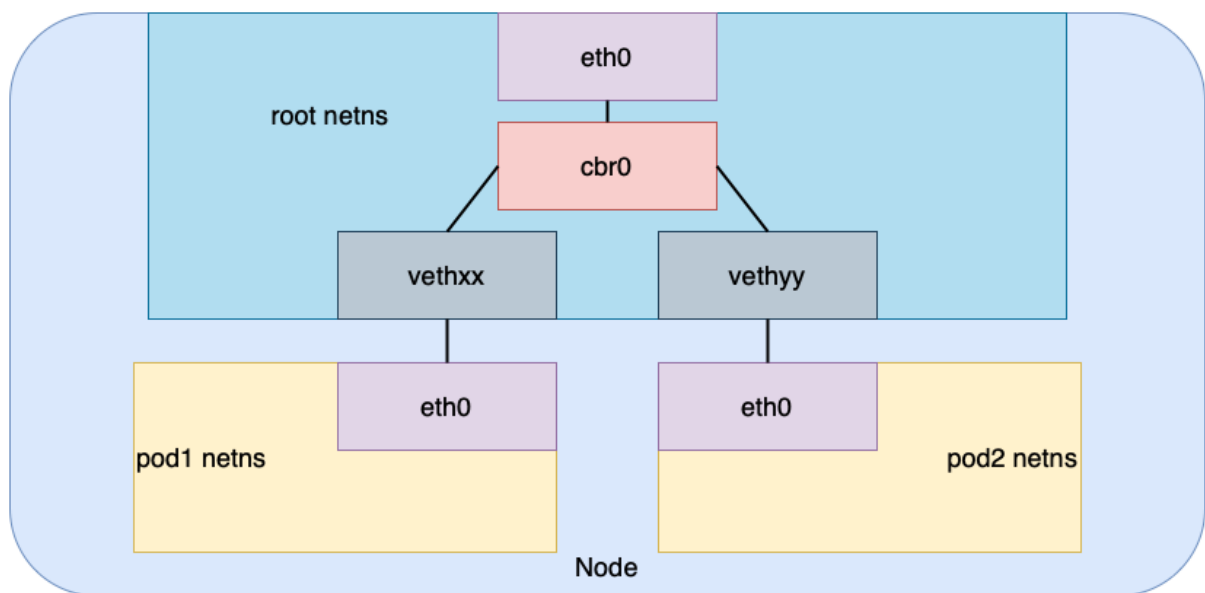


Figure 2.4: Networking in a Kubernetes node with Kubenet plugin

The virtual ethernet device is connected to a bridge **cbr0** to enable inter pod communication. An example of pod 1 communicating with pod 2 on the same node is as follows(16):

1. Pod 1 creates a message with the destination IP of pod2. Since the destination is not one of the containers running in the pod, it is sent to its **eth0**. This interface

is at one end of the virtual ethernet pipe pair and serves as a tunnel. Hence, this packet is forwarded to the root namespace of the node.

2. The bridge `cbr0` resolves the destination pod IP to its MAC address using Address Resolution Protocol (ARP) protocol. A `tcpdump` output of ARP protocol on `cbr0` on a node in Microsoft Azure is shown in figure 2.5.
3. After storing the mapping of IP and MAC address in the ARP cache, the packet is forwarded to pod 2 via the virtual pipe pair, and it reaches `eth0` of the pod 2 namespace.

3731	15.707874	ba:c6:5f:04:4f...	e2:04:5c:48:e6...	ARP	42	Who has 10.244.0.2? Tell 10.244.0.1
3732	15.707920	e2:04:5c:48:e6...	ba:c6:5f:04:4f...	ARP	42	10.244.0.2 is at e2:04:5c:48:e6:48
4002	17.499904	ba:c6:5f:04:4f...	86:41:4c:15:ee...	ARP	42	Who has 10.244.0.6? Tell 10.244.0.1
4003	17.499953	86:41:4c:15:ee...	ba:c6:5f:04:4f...	ARP	42	10.244.0.6 is at 86:41:4c:15:ee:6e
6043	24.155871	3a:e0:26:66:ba...	16:30:0c:42:37...	ARP	42	Who has 10.244.0.11? Tell 10.244.0.9
6044	24.156039	16:30:0c:42:37...	3a:e0:26:66:ba...	ARP	42	10.244.0.11 is at 16:30:0c:42:37:39
111...	33.627870	3a:e0:26:66:ba...	6a:77:c0:85:72...	ARP	42	Who has 10.244.0.15? Tell 10.244.0.9
111...	33.627927	6a:77:c0:85:72...	3a:e0:26:66:ba...	ARP	42	10.244.0.15 is at 6a:77:c0:85:72:c3
133...	44.891863	a2:22:62:00:c9...	e2:04:5c:48:e6...	ARP	42	Who has 10.244.0.2? Tell 10.244.0.4
133...	44.891876	3a:e0:26:66:ba...	e2:04:5c:48:e6...	ARP	42	Who has 10.244.0.2? Tell 10.244.0.9

Figure 2.5: ARP requests captured via `tcpdump` on `cbr0` bridge in Azure. The Pod CIDR for this cluster was 10.244.0.0/16

For a pod that wants to communicate across nodes, the first step is identical. However, when the packet reaches the bridge, the packet is redirected to the default gateway `eth0` of the node as the pod IP is not on the current network. The packet travels through the cloud provider's infrastructure, reaches the second node's `eth0` and is forwarded to the bridge. Using ARP, the bridge determines to which pod the packet will be forwarded.

2.1.2 Machine Learning

Machine learning (ML) is the art and science of creating computer systems that learn and improve with experience. (17). It is one of the most rapidly growing fields, lying at the intersection of computer science and statistics and the core of artificial intelligence (AI).

2.1.2.1 Types of machine learning algorithms

Machine learning algorithms learn patterns in the data provided to them, and usually, the data is considered key to constructing a machine learning model (18). The data can be structured, i.e. stored in a tabular format or unstructured, i.e. has no predefined format like audio files or images.

The machine learning algorithms can be classified into four categories(19):

- **Supervised Learning:** In supervised learning, the data is labelled, and it is the task of the machine learning model to learn the function that maps the given input to the label. Most supervised learning tasks are classification, i.e., separating the data and regression, i.e. fitting the data.
- **Unsupervised Learning:** Unsupervised learning algorithms analyse the unlabelled dataset and try to learn patterns without the need for human interference. Most common unsupervised algorithms include clustering, dimensionality reduction and anomaly detection.
- **Semi-supervised Learning:** Semi-supervised learning is a hybrid approach of supervised and unsupervised learning. It operates on both labelled and unlabelled data. A possible way to use semi-supervised learning is to perform clustering on unlabelled data and apply labels to the identified clusters. This creates labels for the data, and hence the approach can be further used to train a supervised learning model.
- **Reinforcement learning:** This type of algorithm enables a software agent to analyse the environment and, based on it, automatically learn the most optimal behaviour in it. The agent is incentivised to learn good behaviour by providing it with rewards and discouraged to perform actions that are not optimal in an environment by penalising it.

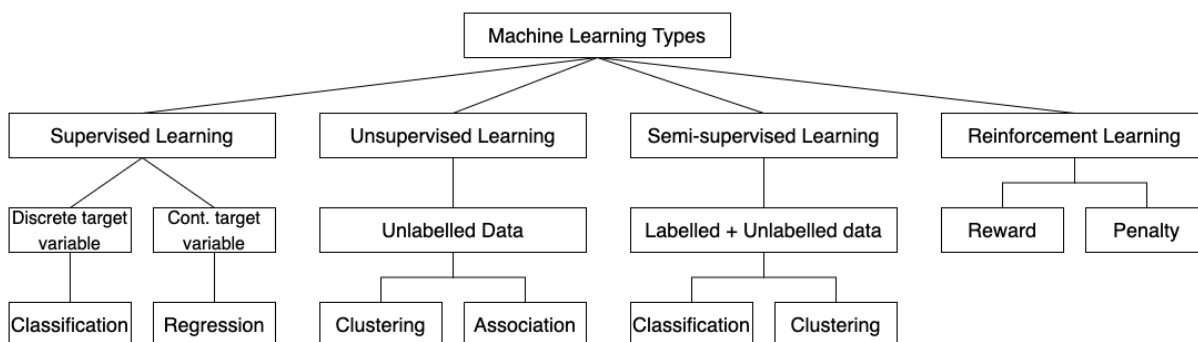


Figure 2.6: Taxonomy of machine learning algorithms: Source: (19)

For this project, the main focus was using unsupervised learning to detect anomalies. Hence the further sections focus on the algorithms used for anomaly detection.

2.1.2.2 Classification of anomaly detection

”Anomalies are patterns in data that do not conform to a well-defined notion of normal behaviour” (20). Anomalies, based on the type of data, can be classified into three categories(20):

- **Point Anomalies:** If a single data point is different from the rest of the dataset, that instance can be considered anomalous. For example, in credit card fraud detection, if a transaction is unusually high than the other transactions ever made by a person, that would classify as a point anomaly.
- **Contextual Anomalies:** If a data instance is anomalous in a particular context and not in itself, then that point is a contextual anomaly. For example, having a temperature of -1°C in the middle of winter in Delhi, the capital of India, is not anomalous, but in May, it is as the average temperature is over 38°C .
- **Collective Anomalies:** If a collection of data points related to each other are anomalous with respect to the rest of the dataset, then those points are considered anomalous. Their individual occurrence might not be anomalous, but their occurrence together can be regarded as anomalous.

Anomaly detection can be done in a supervised, semi-supervised and unsupervised fashion, depending upon data availability. As stated in (20), finding a labelled dataset that is accurate and representative of all kinds of behaviours is expensive. Also, the anomalous behaviour is more dynamic, which means that new kinds of anomalies might arise for which there is no labelled data. Hence, supervised learning was not further explored for this project as it is similar to building predictive models.

The algorithms that operate in an unsupervised fashion are more widely applicable as unlabelled data is far more prevalent than labelled data. These techniques assume that normal data is far more frequent than anomalous data. Unfortunately, these techniques also suffer from a high false-positive rate, especially for those data points on the boundary of 'normal' behaviour. An output of an anomaly detection algorithm can be a label, i.e. normal or anomalous or can be a score, i.e. a degree to which an instance is considered an anomaly.

2.1.2.3 Anomaly detection using density based clustering

Density-based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm that can be used to detect anomalies in a dataset. The DBSCAN algorithm relies on two parameters, i.e. the minimum number of neighbours for a data point $minPts$ and epsilon ϵ , an arbitrary radius for $minPts$ (21). If a data point has more neighbours than $minPts$, that point is considered a *core point*. All the neighbours within radius ϵ of the core point are considered to be part of the same cluster(direct density reachable). If any neighbours of the core point are also a core point, their neighbours are transitively included (density reachable), which in essence means that the clusters

combine to form a bigger cluster. The non-core points, i.e. border points, are part of the reachable density set. Points that are not density reachable from any core point are considered noise or, in this case, anomalies.

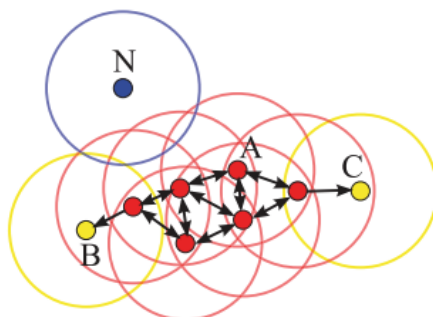


Figure 2.7: Illustration of DBSCAN algorithm. Source: (21)

In figure 2.7, the $minPts$ is set to 4 and ϵ is the radius of the circle around the points. Point A is a core point, while points B and C are border points. Points B and C are part of the same cluster as they are density reachable. Point N is not reachable by any point, hence is an anomaly.

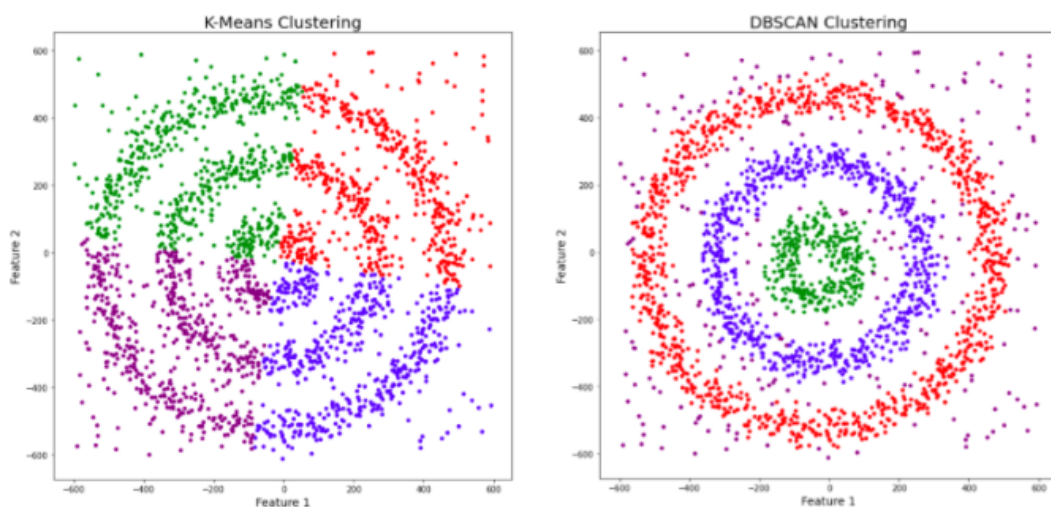


Figure 2.8: Kmeans clustering vs DBSCAN. Source: (22)

Figure 2.8 illustrates the difference between a distance-based clustering algorithm like Kmeans vs DBSCAN. In K-Means, the number of centroids has to be assumed, and the goal of the algorithm is to minimise the sum of distances between the points and their centroid. Whereas in DBSCAN, the numbers of clusters are inferred from the shape of the data.

2.1.2.4 Anomaly detection using neural network

For utilising neural networks for unsupervised anomaly detection, autoencoders can be used(23). An autoencoder is a multilayer feed-forward network with the same number of input and output neurons. The middle layer of an autoencoder is a bottleneck layer, which forces the model to compress data. Once the data is compressed, the model tries to reconstruct the input. A model is trained on normal data points, so it learns to reconstruct them. The error during this reconstruction determines if a data point is anomalous or not.

For example, each data point x_i is reconstructed using the model during the testing phase. The model generates an output o_i with the same number of features n as the input(20). The reconstruction error of a datapoint δ_i can be defined using RMSE. This score can directly be used as an anomaly score for the data point.

$$\delta_i = \sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{x_i - o_i}{\sigma_i} \right)^2}$$

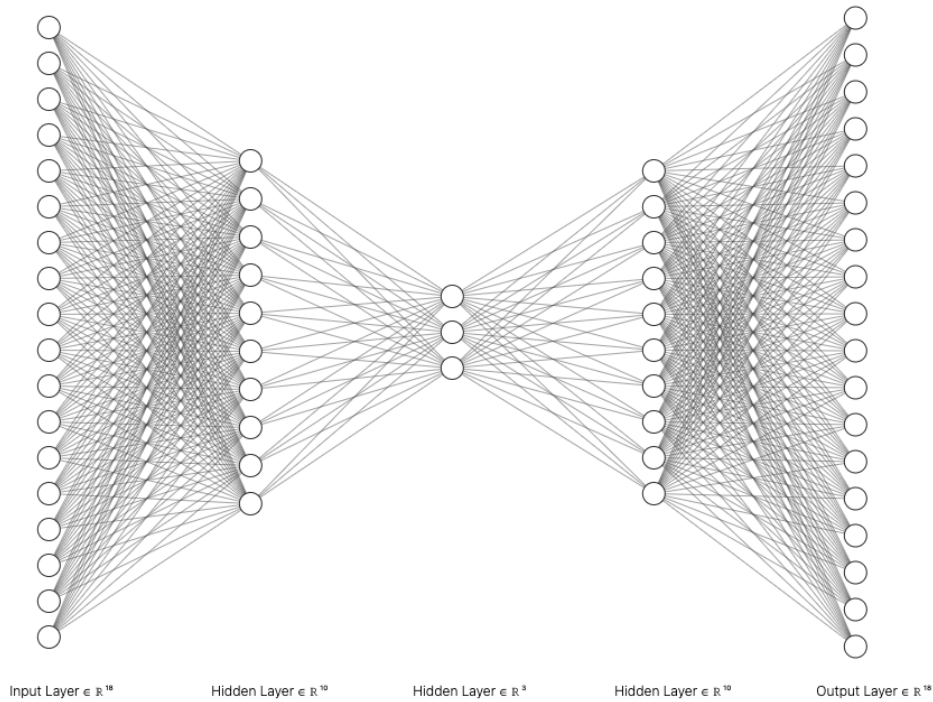


Figure 2.9: Architecture of an autoencoder

2.1.2.5 Anomaly detection using tree-based technique

Isolation forest(24) is an unsupervised anomaly detection technique based on the idea of 'isolating' anomalies. Most model-based approaches build a profile based on normal

instances, then identify the instances not conforming to the model as anomalies. On the other hand, Isolation forest takes advantage of characteristics of an anomaly, which are that they are far less common in a dataset and have attributes much different than that of normal instances.

An Isolation Forest algorithm builds an ensemble of trees, and the instances with a shorter average path from the root node are classified as anomalies. As seen in the figure 2.10, an anomaly denoted by x_0 takes far fewer partitions to isolate, i.e. shorter path in a tree than x_i . The partitions in that example randomly select an attribute and split the value between the min and max values of the selected attribute. It was seen in that using 1000 trees, the average path length of x_0 was 4.02 and for x_i was 12.82 respectively, showing anomalies have shorter path lengths. Also, it was seen that Isolation Forest performs and scales much better than distance-based methods for high dimensional dataset(24).

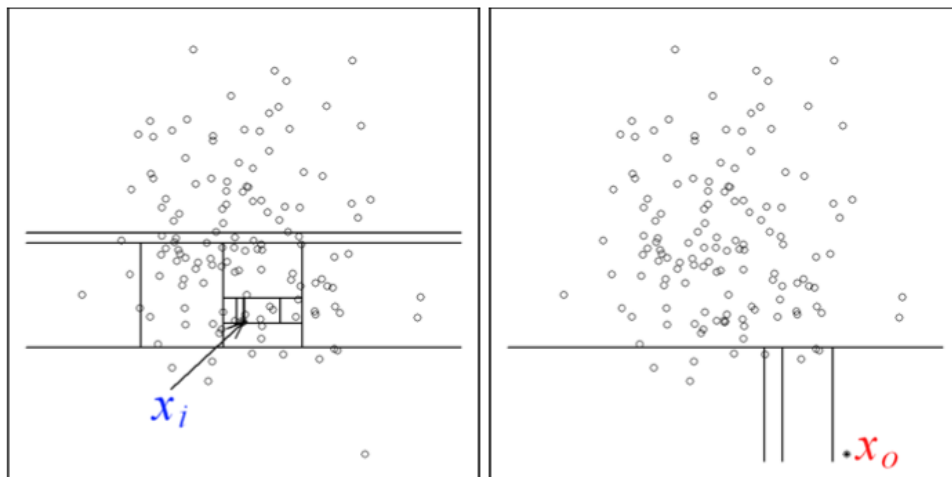


Figure 2.10: Visualisation of Isolation Forest: Source (24)

2.1.3 Intrusion Detection System

To create an IDS, it is first necessary to know the types of cyberattacks, as discussed in section 2.1.3.1. An IDS can be classified into different categories based on analysed activity as explained in section 2.1.3.2, intrusion method as examined in section 2.1.3.3). Furthermore, behaviour on detecting intrusion is discussed in section 2.1.3.4.

2.1.3.1 Taxonomy of cyber attacks

To create an IDS to protect a system against cyberattacks, it is first imperative to know what techniques cyber attackers use. The MITRE ATT&CK (Adversarial Tactics, Techniques, and Common Knowledge) framework is a knowledge base of adversary tactics and techniques based on real-world observations. (25)

Reconnaissance 10 techniques	Resource Development 6 techniques	Initial Access 9 techniques	Execution 10 techniques	Persistence 18 techniques	Privilege Escalation 12 techniques	Defense Evasion 37 techniques	Credential Access 14 techniques	Discovery 25 techniques	Lateral Movement 9 techniques	Collection 17 techniques	Command and Control 16 techniques	Exfiltration 9 techniques	Impact 13 techniques
Active Scanning (0:2)	Acquire Infrastructure (0:6)	Drive-by Compromise (0:3)	Command and Scripting Interpreter (0:3)	Account Manipulation (0:4)	Abuse Elevation Control Mechanism (0:4)	Abuse Elevation Control Mechanism (0:4)	Brute Force (0:4)	Account Discovery (0:4)	Exploitation of Remote Services (0:4)	Archive Collected Data (0:2)	Application Layer Protocol (0:4)	Automated Exfiltration (0:1)	Account Access Removal (0:1)
Gather Victim Host Information (0:4)	Compromise Accounts (0:2)	Exploit Public Facing Application (0:3)	Exploitation for Client Execution (0:3)	BITS Jobs (0:2)	Access Token Manipulation (0:5)	Access Token Manipulation (0:5)	Credentials from Password Stores (0:5)	Application Window Discovery (0:4)	Internal Spearphishing (0:4)	Audio Capture (0:2)	Communication Through Removable Media (0:2)	Data Transfer Size Limits (0:1)	Data Destruction (0:1)
Gather Victim Identity Information (0:3)	Compromise Infrastructure (0:3)	External Remote Services (0:4)	Inter-Process Communication (0:2)	Boot or Logon Autostart Execution (0:2)	Boot or Logon Autostart Execution (0:2)	BITS Jobs (0:2)	Exploitation of Credential Access (0:5)	Brower Bookmark Discovery (0:4)	Lateral Tool Transfer (0:4)	Automated Collection (0:2)	Data Encrypted for Impact (0:1)	Exfiltration Over Alternative Protocol (0:2)	Data Encrypted for Impact (0:1)
Gather Victim Network Information (0:3)	Develop Capabilities (0:4)	Hardware Additions (0:4)	Native API (0:2)	Boot or Logon Initialization Scripts (0:5)	Boot or Logon Initialization Scripts (0:5)	Deobfuscate/Decode Files or Information (0:5)	Forced Authentication (0:5)	Cloud Infrastructure Discovery (0:4)	Remote Service Session Hijacking (0:2)	Clipboard Data (0:2)	Data Encoding (0:2)	Exfiltration Over C2 Channel (0:2)	Data Manipulation (0:2)
Gather Victim Org Information (0:4)	Establish Accounts (0:2)	Phishing (0:2)	Scheduled Task/Job (0:6)	Browser Extensions (0:2)	Boot or Logon Initialization Scripts (0:5)	Direct Volume Access (0:5)	Input Capture (0:4)	Cloud Service Dashboard (0:4)	Remote Services (0:6)	Data from Cloud Storage Object (0:2)	Data Obfuscation (0:2)	Exfiltration Over C2 Channel (0:2)	Disk Wipe (0:2)
Phishing for Information (0:3)	Obtain Capabilities (0:4)	Replication Through Removable Media (0:4)	Shared Modules (0:4)	Compromise Client Software Binary (0:4)	Create or Modify System Process (0:4)	Execution Guardrails (0:1)	Man-in-the-Middle (0:2)	Cloud Service Discovery (0:4)	Replication Through Removable Media (0:4)	Data from Configuration Repository (0:1)	Dynamic Resolution (0:3)	Exfiltration Over Other Network Medium (0:1)	Endpoint Denial of Service (0:4)
Search Closed Sources (0:2)	Supply Chain Compromise (0:3)	System Services (0:2)	Software Deployment Tools (0:2)	Create Account (0:3)	Event Triggered Execution (0:15)	File and Directory Permissions Modification (0:2)	Network Sniffing (0:4)	Domain Trust Discovery (0:4)	File and Directory Discovery (0:4)	Data from Information Repositories (0:2)	Fallback Channels (0:2)	Exfiltration Over Physical Medium (0:1)	Firmware Corruption (0:1)
Search Open Technical Databases (0:5)	Trusted Relationship (0:2)	User Execution (0:2)	Windows Management Instrumentation (0:2)	Create or Modify System Process (0:4)	Exploitation for Privilege Escalation (0:15)	Group Policy Modification (0:2)	OS Credential Dumping (0:8)	Network Service Scanning (0:4)	Software Deployment Tools (0:4)	Data from Local System (0:2)	Ingress Tool Transfer (0:2)	Exfiltration Over Web Service (0:2)	Inhibit System Recovery (0:1)
Search Open Websites/Domains (0:2)	Valid Accounts (0:4)	External Remote Services (0:15)	Hijack Execution Flow (0:1)	Event Triggered Execution (0:15)	Group Policy Modification (0:2)	Hide Artifacts (0:7)	Network Share Discovery (0:8)	Taint Shared Content (0:4)	Taint Shared Content (0:4)	Shared Drive (0:2)	Multi-Stage Channels (0:2)	Network Denial of Service (0:2)	Resource Hijacking (0:1)
Search Victim-Owned Websites (0:2)	Hijack Execution Flow (0:1)	Process Injection (0:1)	Scheduled Task/Job (0:6)	Indicator Removal on Host (0:6)	Indirect Command Execution (0:4)	Impair Defenses (0:7)	Steal Application Access Token (0:1)	Network Sniffing (0:4)	Use Alternate Authentication Material (0:4)	Data from Removable Media (0:2)	Non-Application Layer Protocol (0:2)	Scheduled Transfer (0:1)	Service Stop (0:1)
	Office Application Startup (0:3)	Pre-OS Boot (0:5)	Scheduled Task/Job (0:6)	Server Software Component (0:3)	Masquerading (0:6)	Modify Authentication Process (0:4)	Steal or Forge Kerberos Tickets (0:4)	Password Policy Discovery (0:4)	Peripheral Device Discovery (0:4)	Data Staged (0:2)	Non-Standard Port (0:2)	Transfer Data to Cloud Account (0:1)	System Shutdown/Reboot (0:1)
	Scheduled Task/Job (0:6)				Modify Cloud Compute Infrastructure (0:4)	Two-Factor Authentication Interception (0:4)	Steal Web Session Cookie (0:4)	Permission Groups Discovery (0:3)	Process Discovery (0:4)	Email Collection (0:3)	Protocol Tunneling (0:2)		
						Masquerading (0:6)	Unsecured Credentials (0:6)	Remote System Discovery (0:4)	Query Registry (0:4)	Input Capture (0:4)	Proxy (0:4)		
								Software Discovery (0:1)	Man-in-the-Middle (0:2)	Man-in-the-Middle (0:2)	Remote Access Software (0:1)		
									Screen Capture (0:3)	Screen Capture (0:3)	Web Service (0:3)		
										Video Capture (0:1)			

Figure 2.11: MITRE ATT&CK Matrix for Enterprise 2020. Source: (26)

The behavioural model presented by ATT&CK contains the following core components:

- Tactics: They are short term goals of an adversary during the attack
- Techniques: They are the various techniques employed to achieve those tactics

This framework reflects cyber attackers’ various steps from reconnaissance and finally impact, and the various platforms they target. This framework can be used to create an IDS that targets specific attack vectors and evaluate their performances.

2.1.3.2 Classification of an IDS based on analysed activity

An IDS can be classified into two categories based on what activity it analyses, Network Intrusion Detection System (NIDS) and Host Intrusion Detection System (HIDS).

A NIDS monitors and gathers network traffic information about incoming and outgoing internet traffic for a system at a router or host level. It can detect and log suspicious events like port scanning, policy violations based on specific rules and unknown source and destination traffic (27). One of the points to note about NIDS is that when the network is saturated with traffic, NIDS might drop packets and create a potential ‘hole’ (28). Also, a NIDS might not work when the network traffic is encrypted.

HIDS monitors and analyses the internals of a host machine. It tracks changes made to registry settings and critical system configuration, log and content files, alerting to any unauthorised or anomalous activity.(27) The major drawback of this monitoring system is that it can be quite resource-intensive(29).

2.1.3.3 Classification of an IDS based on detection method

An IDS can be classified by the way it detects malicious activity. It can either be signature-based or anomaly-based.

Signature-based detection tries to find sequences and patterns that match a particular attack signature. An attack signature can be found in network packet headers and in data sequences that match recognised malware or malicious patterns. An attack signature can also be identified in specific sequences of data or series of packets, as well as in destination or source network addresses(30). It uses a list of Indicators of Compromise (IOCs) to match the signature and detect an intrusion. Hence, the major disadvantage of this type of detection method is that it can only detect known signatures. Also, a signature-based method is only as good as its database of IOCs, so keeping the database updated is also an issue.

Anomaly-based detection systems aim at detecting unusual activity within the system. They do this by creating a normal behaviour profile, and if any deviation is detected, an alert is created. Profiles can be either static or dynamic and developed for many attributes, e.g., failed login attempts, processor usage, the count of e-mails sent, etc. (29). One of the significant benefits of this approach is that it is useful in detecting new vulnerabilities. However, this approach also generates a lot of false positives. Also, this approach is dependent on the strength of user profiles; if there is a weak profile on user behaviours, this approach does not work.

2.1.3.4 Classification of an IDS based on behaviour on detection

An Intrusion detection system can be active or passive. An active IDS can activate countermeasures to prevent further escalation of the attack. A passive intrusion detection system only generates alerts when malicious activity is detected. Most IDS in use are passive components due to the complexity of automated countermeasures, and the risk of unintended consequences in case of inappropriate countermeasures (31) (32).

2.1.3.5 Anomaly-based NIDS

As stated in section 2.1.3.3, an anomaly-based intrusion detection system tries to detect abnormal activities in the system by generating a user profile/ behavioural model of the normal baseline behaviour. According to the type of processing on this behaviour, anomaly detection techniques can be broadly classified into three types (33):

- Statistical based: In this technique, the network activity is captured and analysed based on the number of types of packets of each protocol sent/received, traffic rate,

number of IP connections, etc. The current captured data is compared with the previously trained generated profile for the anomaly detection process. An anomaly score is generated based on the comparison of these datasets. Univariate models (34) modelled the parameters into independent variables, and later multivariate models (35) models that considered correlations between two things were proposed.

- Knowledge-based: The knowledge-based approach can be classified further into three categories, Expert systems, which audit data based on certain predefined rules(34), a Finite State Machine (FSM), which models a sequence of states and the transitions between those states(36) and Description Languages that use N-grammars, UML for modelling the FSM.
- Machine learning-based: Machine learning-based techniques use the data captured to create a model that enables recognising and classifying patterns. It can be done in a supervised fashion, i.e. with labelled data and in an unsupervised fashion, i.e. with unlabelled data. The performance of a model can be improved by training it on more data(18). These models are very flexible and adaptable. Hence, this is a very attractive approach, but one downside of this approach is that these models are resource-intensive to train, the quality of data matters a lot, and they do not provide a reason why a particular detection decision has been taken.

The fundamental idea of an anomaly-based IDS is that it is not possible to characterise what malicious activities look like. The new aim should be to model or learn legit activities and treat everything else as potentially hostile.

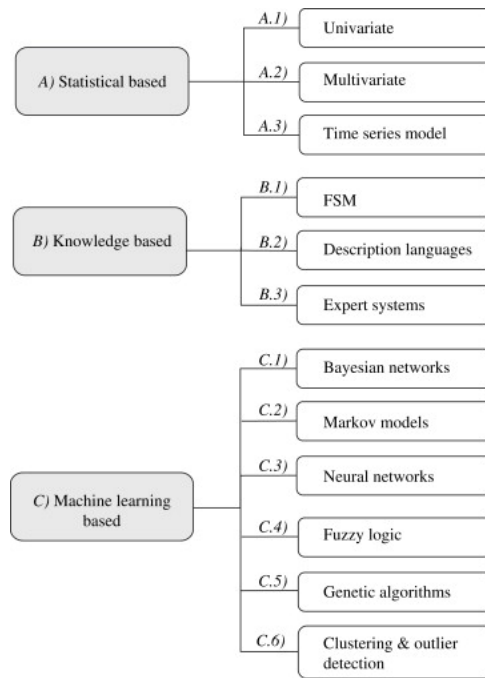


Figure 2.12: Classification of different anomaly based intrusion detection techniques based on nature of creating behaviour model. Figure source:(33)

2.2 Related projects

2.2.1 Monitoring Kubernetes Clusters With Dedicated Sidecar Network Sniffing Containers

This research project(37) uses the concept of sidecars, i.e. a container that runs on the same pod as the application container, shares the same network namespace and can be used for enhancing the capability of the main container. Sidecar containers were used to sniff packets from a pod to perform deep packet inspection. This project used rule-based packet signature analysis to detect anomalous packets and display the output to a dashboard. The main limitation of this approach is that it is rule-based and would not be able to detect an attack that is not in its rules successfully. Another downside of this approach is capturing requests on a pod, not on a Kubernetes node, which includes more information about inter pod and external communication.

2.2.2 Application of Machine Learning with Traffic Monitoring to Intrusion Detection in Kubernetes Deployments

This recently completed dissertation (38) by Irene addresses some of the problems in (37). In this project, the packets were captured on a Kubernetes node instead of pods and a supervised machine learning classifier algorithm was trained to predict normal packets from anomalous ones.

The main downside of this approach was that the anomalous packets were self-generated and were not actual attack data. Also, a supervised machine learning model needs to be trained on actual anomalous packets and getting a labelled dataset is prohibitively expensive, as stated in (20). The implemented solution uses limited packet features and does not take into account the packet flow.

2.2.3 Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge

In this paper(39), the researchers created a UNIDS (Unsupervised Network Intrusion Detection Systems) that uses unsupervised learning to detect previously unknown attacks based on traffic flow. This NIDS does not need any labelled traffic or even training. The novel approach in this paper was using unsupervised clustering outliers detection algorithms on a smaller amount of data. In the KDD99 networks attack dataset, UNIDS was able to detect more than 90% with a very low false-positive rate of less than 3.5%. This was achieved by:

- Detecting an anomalous time slot: The captured packets were aggregated into multi-resolution time flows, and time series were built on these flows. A change detection algorithm is used on volume metrics (number of bytes, number of packets, and number of flows per time slot). The algorithm flags a data flow as anomalous if the derivative of any of these metrics exceeds a threshold computed from the variance of previous anomaly-free flows. This helps limit the frequency of the clustering step as that is much more computationally expensive.
- Outlier detection using an ensemble of multi-clustering algorithms: In a time slot flagged as anomalous, outliers are detected using Sub-Space Clustering, Density-based Clustering, and Evidence Accumulation Clustering techniques. It marks the degree of abnormality and builds an outliers ranking.
- Marking anomalies: Using a threshold detection approach, the top-ranked outliers are marked as anomalies.

This paper shows that unsupervised learning is a viable approach for a NIDS. However, in this research, the features of the packet were not considered.

2.2.4 Unsupervised Packet-based Anomaly Detection in Virtual Networks

This paper (40) analyses Isolation Forest and Local Outlier Factor to perform packet-based anomaly detection. To capture data, the researchers created a VM based cloud environment using OpenStack and deployed a MySQL database and a PHP script for contacting the top 500 websites to simulate real-world traffic. Anomalies were injected into the traffic using scapy, a python framework for packet manipulation which changed the IP version, protocol, set different flags and unused IP addresses. The packet data was captured, and the Ethernet, IP and TCP/UDP features of the packet were extracted to create a dataset. The underlay network of the architecture was also changed to check if the algorithms would detect that as an anomaly.

Anomaly detection algorithms like Isolation Forest and Local Outlier Factor were applied to the dataset. It was observed that changes in the network environment, i.e. addition/deletion of VMs and change in underlay protocols make the algorithm flag normal packets as anomalies.

2.3 Summary

Based on section 2.1 and 2.2, it can be seen that although the fields of machine learning and cyber security have advanced a lot in recent years individually, more work needs to be done to integrate them to defend against cyber attacks. This chapter provided a brief overview of Kubernetes and inter-pod communication. It also describes ML algorithms that can be used for anomaly detection like Isolation forest, DBSCAN and autoencoders. It explains the different kinds of IDSs available and their classification based on analysed activity, detection method and behaviour on detection. In section 2.2, four recent projects were analysed, and their limitations were explained, which this project aims to overcome.

Chapter 3

Design

This chapter describes the various design decisions to create a prototype for a machine learning-based NIDS. Section 3.1 discusses the various deployment platforms for Kubernetes. In section 3.2, the microservices-based architecture of the prototype is described, and section 3.3 explains the data collection process from the deployed application. Section 3.4 provides an overview of the various libraries and packages used for creating a machine learning environment to create a model for the NIDS.

3.1 Deployment platform

Kubernetes, as discussed in section 2.1.1, is a platform that helps run the containerized images. This framework has matured over the years; it has excellent support and can be deployed to test applications locally and in the cloud.

3.1.1 Local deployment

A Kubernetes cluster can be deployed locally using minikube(41). Minikube is a single node cluster that makes it easier to test an application before deploying it to the cloud. It requires Docker or a virtual machine environment running locally. This environment is suitable for testing as the YAML files for local and cloud deployments are almost identical.

3.1.2 Cloud deployment

To test the prototype against real-world threats, deploying the application to the cloud is imperative. Many cloud providers like Amazon Web Services(AWS), Microsoft Azure, Google Cloud and Digital Ocean offer a managed Kubernetes cluster service. Azure Kubernetes Services (AKS) was chosen for the project because, being a student, Azure

gives enough credits to get a cluster started. AKS makes it simple to deploy a Kubernetes cluster with a Kubenet configuration. The AKS Comand Line Interface (CLI) allows creating, upgrading, or deleting a cluster with a single command. Although Amazon Elastic Kubernetes Service (EKS) can run more nodes in a cluster than AKS (42), to test the feasibility of this prototype, running one node in the cluster was sufficient. AKS also has an auto-repair feature that scans and updates unhealthy nodes in a cluster. Hence AKS was considered a suitable platform for this project.

3.1.3 Docker images

A Docker image is a file with an executable code that can create a container on Docker's container runtime system. A docker image can support multiple CPU architectures. As the local machine (M1 based Macbook Pro) has an ARM64 architecture and the cloud deployment is usually Linux/AMD64 based, Docker images needed multiplatform support for this project.

3.2 Architecture of the NIDS

This prototype leverages the micro-services architecture to splits the application into multiple services that perform fine-grained functions.

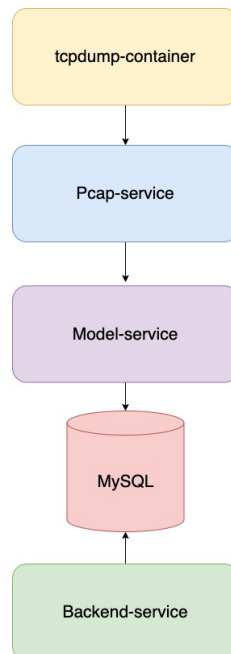


Figure 3.1: Proposed micro-services based architecture of the prototype

The microservices architecture would enable the creation of independently deployable components that allow the ability to scale up/down depending on the network traffic. The NIDS data processing pipeline consists of

- Tcpcap-container: It sniffs the data packets from the cbr0 bridge, saves them to a pcap file, and sends the file via a POST request to the pcap-service.
- Pcap-service: It receives the file, extracts important features from it, and converts it to a CSV. This CSV file was then sent to the model service via a POST request for further processing.
- Model-service: This service contains pods running a TensorFlow container. It pre-processes the CSV and uses a pre-trained model to predict whether a packet is anomalous or not. It then loads the CSV into the MySQL database.
- MySQL database: MySQL database stores the packet information from the CSV and its corresponding scores. This can be within the Kubernetes cluster or can be a managed Azure MySQL database. A SQL database was preferred over a NoSQL database as the data would be structured.
- Backend service: This service fetches the data from the MySQL database and could display the anomalous scores on a webpage. This can be run locally on our machine or in the cluster, independent of the other data processing pipeline.

Python, a high-level programming language, was used to create this pipeline. It has a vast collection of packages that simplify prototyping and rapid development. Flask, a micro-web framework, was used to create servers for receiving files. Unlike other frameworks like Django, Flask does not have much boilerplate code or requires particular tools or libraries. This would have the added benefit of keeping docker images smaller in size.

3.3 Collecting data

WordPress, a free and open-source content management system, was used to collect data and test the NIDS. It has a diverse userbase, from small businesses to Fortune 500 companies. According to their website, as of April 2022, 43% of the web is built on WordPress. The WordPress application uses a PHP backend with a MariaDB database. To deploy the WordPress on the Azure Kubernetes cluster, Helm charts were used. A Helm chart is one single Kubernetes YAML file comprising different Kubernetes resources. Using Helm, an old, vulnerable version of WordPress was deployed. As vulnerabilities were known,

they could be exploited to generate anomalous traffic, which the NIDS should be able to detect.

A pod was created in the root namespace of a node, which ran a custom tcpdump image. This pod was used to collect packets from the cbr0 bridge to generate data for training the machine learning model. After collecting data for 10 minutes, the tcpdump script was stopped, and the data collected in a pcap file was transferred to the local machine. After rerunning the tcpdump script, a scanner for WordPress vulnerabilities like WPScan was used against the application run to generate accurate world anomalous data. This data was then transferred to the local machine, which would be valuable in testing the model.

3.4 Machine learning environment

As the programming language of choice for this project was Python, creating an ML environment using its various libraries was ideal. It has a well documented and vast ecosystem to develop and deploy a machine learning model, which is the core of the NIDS. The various components of the environment include:

- Virtual environments: Creating a virtual environment was necessary as it allows Python packages to be installed in an isolated location for a particular application rather than being installed globally. This isolation ensures that a dependency for one project does not interfere with a dependency for another project. Conda¹, an open-source environment management system, was used to create and manage virtual environments. Specifically, for an M1 based MacBook MiniForge, a conda installer was used as it provided native ARM64 support.
- Scapy²: It is a powerful packet manipulation program written in Python that would allow to read and obtain necessary information from the .pcap files generated in 3.3. After gathering data from packets, they can be written to a CSV for further processing.
- Pandas³: Pandas is a python library that helps with data analysis, preprocessing and manipulation. It has built-in support for reading and writing CSV files. CSV files can be loaded into a Pandas DataFrame object which has numerous inbuilt tools and functionalities that would help extract valuable features and simplify data manipulation.

¹<https://docs.conda.io/en/latest/>

²<https://scapy.net>

³<https://pandas.pydata.org>

- Matplotlib⁴: It is a plotting library used to create visualizations in Python.
- Scikit-Learn ⁵: Scikit-learn is an open-source machine learning library that can be used to create and test various machine learning approaches. It supports various pre-processing, classification, clustering and dimensionality reduction techniques, which would be helpful for this project.
- Keras ⁶: Keras provides a deep learning API for implementing artificial neural nets. It offers a consistent API and useful abstractions, which is ideal for prototyping.
- Jupyter notebooks ⁷: It is an open-source web application that allows users to create documents with code, equations, and visualizations. It allows users to execute Python commands in a cell, the output of the cell is saved locally and is displayed underneath it. This notebook can also be exported as a python file, and the pre-processing functions can be extracted to create the pipeline for NIDS. This process will help keep everything consistent.

3.5 Workflow

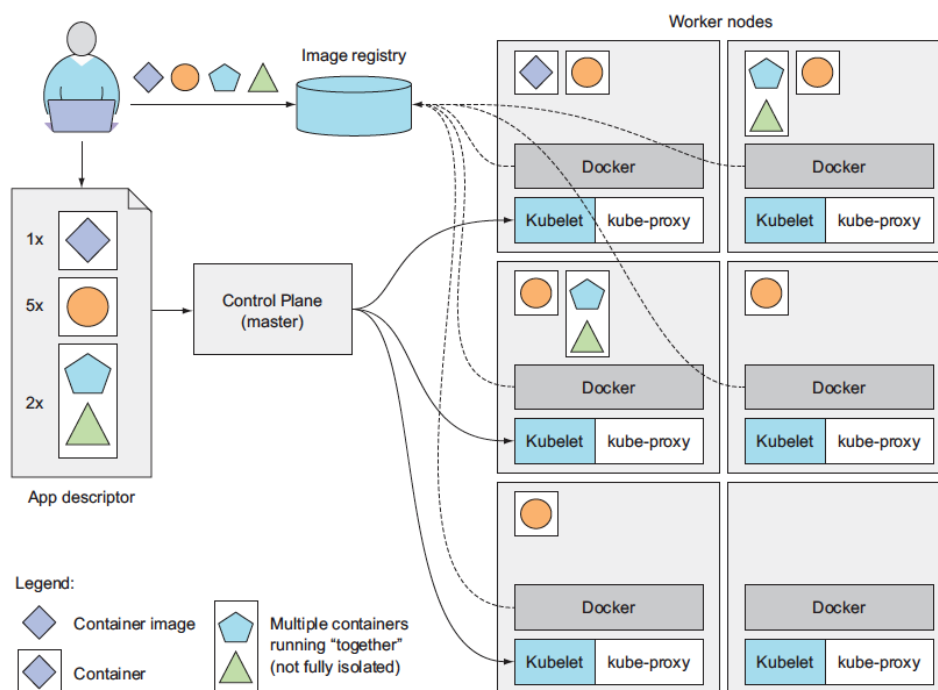


Figure 3.2: Workflow of a developer using kubernetes

⁴<https://matplotlib.org>

⁵<https://scikit-learn.org/stable/>

⁶<https://keras.io>

⁷<https://jupyter.org>

The workflow used for designing and deploying the prototype described in 3.2 is as follows:

- User writes the application code and uses Docker to create a multiplatform image using Dockerfile. This image is then pushed to a container registry like Docker Hub.
- To create a Kubernetes pod using the image, the user can specify the name of the container in the Deployment object and to expose the services running in the pod within the cluster, can create a Service object. These objects are specified in a YAML file.
- The user can connect to a Kubernetes cluster via a terminal and can use `kubectl` to send the YAML file via an API request.
- The resources specified in the YAML file are created in the Kubernetes cluster on worker nodes.

3.6 Summary

This chapter provided an overview of the types of deployments considered for this project and how a local environment can be created to test images locally on minikube, and Microsoft Azure can be used to deploy images on the cloud. Also, as the CPU architecture of the local machine and cloud architectures are different, multiplatform images were needed. Also, the data collection mechanism using WordPress and a tcpdump container for training machine learning models were discussed. Then, a high-level overview of the architecture of NIDS and the workflow used to deploy an application was explained.

Chapter 4

Implementation

In this chapter, the various components of the chapter 3 are described in-depth and implemented in a local as well as a cloud environment¹.

Section 4.1 explains how to create a local and a cloud environment for Kubernetes and the differences between them. Section 4.2 goes in-depth on a data processing pipeline for this project was implemented, and 4.3 describes how an attack was simulated on a cluster to collect anomalous packets. Section 4.4 talks about the various machine learning approaches implemented to detect an attack on a cluster.

4.1 Deployment of Kubernetes cluster

A Kubernetes cluster can be deployed locally for development and testing as explained in section 4.1.1 and on the cloud, specifically on Azure Kubernetes Services (AKS) as described in 4.1.2. Some notable differences between a local and cloud deployment are touched upon in section 4.1.3.

4.1.1 Local deployment

Using Minikube, a single node Kubernetes cluster can be deployed locally. It supports Windows, Linux and macOS and is available in x86-64 and ARM-based architectures. In macOS, the brew package manager can be used to download the minikube tool. It was necessary to have Docker running to start the service. As of April 2022, downloading Docker Desktop also installs Kubernetes command-line tool `kubectl`, which helps interact with the cluster via a command-line interface.

¹The code for this project is available on <https://github.com/hotshot07/thesis>

```

🍏 ~ > minikube start
🍏 minikube v1.25.2 on Darwin 12.3 (arm64)
📄 Kubernetes 1.23.3 is now available. If you would like to upgrade, specify: --kubernetes-version=v1.23.3
🔧 Using the docker driver based on existing profile
👉 Starting control plane node minikube in cluster minikube
🚚 Pulling base image ...
🔄 Updating the running docker "minikube" container ...
🌐 Preparing Kubernetes v1.22.3 on Docker 20.10.8 ...
🔍 Verifying Kubernetes components...
  ■ Using image kubernetesui/dashboard:v2.3.1
  ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
  ■ Using image kubernetesui/metrics-scraper:v1.0.7
🌟 Enabled addons: storage-provisioner, default-storageclass, dashboard
👉 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

```

Figure 4.1: Output of \$minikube start command, running it for the first time will download an image that may take some time.

Running the command \$minikube start for the first time downloads the minikube docker image and starts a cluster locally.

To test if everything was working, \$kubectl get pods -o wide --all-namespaces was run, and it showed a list of all pods running in the kube-system and kube-dashboard namespace. The system components had the status Running as shown in figure4.2; hence the cluster was ready for container deployments.

```

🍏 ~ > k get pods -o wide --all-namespaces

```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	* minikube READINESS GAT
ES									
kube-system	coredns-78fcd69978-d6drr	1/1	Running	7 (13h ago)	7d22h	172.17.0.8	minikube	<none>	<none>
kube-system	etcd-minikube	1/1	Running	7 (13h ago)	7d22h	192.168.49.2	minikube	<none>	<none>
kube-system	kube-apiserver-minikube	1/1	Running	7 (16h ago)	7d22h	192.168.49.2	minikube	<none>	<none>
kube-system	kube-controller-manager-minikube	1/1	Running	7 (13h ago)	7d22h	192.168.49.2	minikube	<none>	<none>
kube-system	kube-proxy-d5gzt	1/1	Running	7 (13h ago)	7d22h	192.168.49.2	minikube	<none>	<none>
kube-system	kube-scheduler-minikube	1/1	Running	7 (13h ago)	7d22h	192.168.49.2	minikube	<none>	<none>
kube-system	storage-provisioner	1/1	Running	10 (13h ago)	7d22h	192.168.49.2	minikube	<none>	<none>
kubernetes-dashboard	dashboard-metrics-scraper-5594458c94-txgfg	1/1	Running	7 (13h ago)	7d22h	172.17.0.5	minikube	<none>	<none>
kubernetes-dashboard	kubernetes-dashboard-654cf69797-9c5zn	1/1	Running	9 (13h ago)	7d22h	172.17.0.4	minikube	<none>	<none>

Figure 4.2: Console output of running the above \$kubectl get pods command. Here 'k' is an alias for kubectl

4.1.2 Cloud deployment

To deploy a Kubernetes cluster in Azure, Kubernetes Service, a service provided by Azure was used. The subscription type and resource group were specified in the 'Create Resource' section. A description of managing Azure resources can be found on the Microsofts Azure setup guide webpage. In the cluster details, the preset cluster configuration was set as Dev/Test. The node count was set to 1, and autoscale was turned off as packet capturing from multiple nodes was out of scope for this project. For this project, the A4.v2 configuration of the node was chosen as it has four cores and eight gigabytes of memory, which was sufficient for compute-heavy and memory-intensive tasks for a prototype.

Create Kubernetes cluster

Subscription * ⓘ Mayank-subscription

Resource group * ⓘ aroram-tcd
[Create new](#)

Cluster details

Cluster preset configuration Dev/Test (\$)
To quickly customize your Kubernetes cluster, choose one of the preset configurations above. You can modify these configurations at any time.
[Learn more and compare presets](#)

Kubernetes cluster name * ⓘ nids-project

Region * ⓘ (Europe) West Europe

Availability zones ⓘ None

Kubernetes version * ⓘ 1.21.9 (default)

API server availability ⓘ

- 99.9%
Optimize for availability. 99.95% is available when at least one availability zone is selected.
- 99.5%
Optimize for cost.
- 99.5% API server availability is recommended for dev/test configuration.

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. [Learn more about node pools in Azure Kubernetes Service](#)

Node size * ⓘ **Standard A4 v2**
4 vcpus, 8 GiB memory
 Standard B4ms is recommended for dev/test configuration.
[Change size](#)

Scale method * ⓘ Manual
 Autoscale

Node count * ⓘ

Figure 4.3: Deployment configuration of the AKS cluster for hosting the prototype

The rest of the parameters were left default; especially in the Network configuration, it was necessary to have `Kubenet` set as default to test the application. `Kubenet` is the basic network plugin for Linux which creates the 'cbr0' bridge and veth pair for each pod with the host end of each pair connected to 'cbr0' as discussed in section 2.1.1.5.

After pressing the `Review + create` button, Azure allocated a VM based on the 'Node size', which acts as the worker node. After a few minutes, the status of the cluster is `Succeeded(Running)`. To connect to the cluster via CLI, `azure-cli` was installed using `$brew install azure-cli` on macOS. After clicking on the `Connect` option on the webpage of the cluster and running the commands specified in the terminal, the local machine could now communicate with the azure cluster. The context for Kubernetes was

set to the cluster name in `~/.kube/config` automatically, and `kubectl` command could now be used to interact with the cluster similarly to a local deployment.

4.1.3 Differences in local vs cloud deployment

Although the local and cloud development environments both have the Kubernetes configuration, the bridge's name is different. On the local machine, the bridge's name that enables inter pod communication is `'docker0'`, while it is `'cbr0'` on cloud deployment. Also, after creating a Loadbalancer resource in the Kubernetes cluster, it is the cloud provider's job to provide the user with an IP address of an external load balancer where the user can interact with the service. For local deployment, the service will have `<pending>` in the External IP section when a user executes `$kubectl get services`. To resolve this, `$minikube tunnel` command can be used. The IP/Port configuration shown can then be used to interact with the service.

As of April 2022, some images like the official tensorflow image and bitnami WordPress helm chart are still not supported on ARM architecture.

4.2 Implementing system architecture

This section goes into depth on how this architecture described in Section 3.2 was implemented. Section 4.2.1 goes into details on how the network traffic was captured on the host using a privileged pod, and section 4.2.2 talks about how a pcap file was converted to a CSV. Section 4.2.3 goes into detail about the steps taken to use a pre-trained machine learning model to predict if a data packet was anomalous or not, and section 4.2.4 explains how to create and store data from a CSV into a MySQL container in Kubernetes. Section 4.2.5 explains how to connect to a database hosted in Azure and display results on a webpage.

4.2.1 Tcpcap container

The motivation behind creating this pod was to capture network traffic from the `'cbr0'` ethernet bridge. After capturing the data in a pcap file, the pod sends the file to pcap-service for further processing. This process continues until the tcpcap script is stopped.

4.2.1.1 Type of pod

To capture the data from the bridge on a Kubernetes node, the pod needs to be in the root namespace of the node. Usually, pods are created in the `default` namespace or a

user-defined namespace and don't have access to the 'cbr0' bridge. Knsiff, a Kubernetes plugin can be used to start a remote capture in any namespace, including the node's namespace using tcpdump and Wireshark. However, there are better ways to capture network traffic in the root namespace.

- Using a privileged pod: Each pod in the Kubernetes has certain assigned privileges. By default, the Kubernetes pod is not allowed to access any devices on the host VM, but a pod's privilege can be elevated using the Pod Security Policies.
- Using debugger pod: A debugger pod can be created in the node's namespace, which has access to all the virtual ethernet ports in the network namespace.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tcpdump-container
  labels:
    app: tcpdump-container
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tcpdump-container
  template:
    metadata:
      labels:
        app: tcpdump-container
    spec:
      hostIPC: true
      hostNetwork: true
      hostPID: true
      containers:
      - image: hotshot07/tcpdump-container:latest
        name: tcpdump-container
        command: ["sleep", "infinity"]
        securityContext:
          privileged: true
          capabilities:
```

```
    add: ["SYS_PTRACE", "SYS_RAWIO",
         "NET_ADMIN", "SYS_ADMIN"]
  restartPolicy: Always
```

Listing 4.1: YAML file creating a deployment for a privileged pod

In listing 4.1 ², a privileged pod with access to `hostIPC`, `hostNetwork` and `hostPID` was created. The image used to create the pod contains the scripts to initialise network traffic capture. To get a shell into a pod, the Lens IDE could be used, but a better approach using a terminal was to create an alias as shown in 4.2 that takes in the name of the pod as an argument to get a shell into it.

```
kterm () {
    kubectl exec --stdin --tty "$1" -- /bin/bash
}
```

Listing 4.2: Kterm alias for getting a bash shell into a pod

4.2.1.2 Capturing tcpdump

To capture packets from the 'cbr0' bridge, the bash script in listing 4.3 was used. The `tcpdump` command in listing 4.3 captures on the 'cbr0' interface for 30 seconds, and the `W` option specifies the maximum number of files after which a file is rotated, set to 1. It does not convert the addresses into names when the `-n` option is specified. This was necessary because the IP of a pod could be used as a feature for the machine learning model. The output was then stored to `pcap-file` directory using the above filename and, after the `tcpdump` was completed, moved to a different directory, `pcap-to-send`. This step was done to avoid accidentally enqueueing a pcap file to send to the pcap service being written by `tcpdump`.

```
trap "exit" INT TERM
trap "kill 0" EXIT

while :
do
    date_time=$(date +"%d-%m-%y-%s")
    filename="pcap-{$date_time}"
    tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/{$filename}.pcap
```

²Due to limitations of the LaTeX package 'minted', captions of the code could not be longer

```
mv pcap-file/${filename}.pcap pcap-to-send/${filename}.pcap
done
```

Listing 4.3: Bash script to capture packets on bridge cbr0 using tcpdump

4.2.1.3 Sending pcap file to pcap-service

On the `pcap-to-send` directory, a polling script is run 1, which looks for new files and sends them to `pcap-service` via a POST request. If the `pcap-service` is down, the file is added back to the queue. As shown in .1, this pod cannot call the service directly by using the service name and port. However, every pod has the name of the service and port in its environment variables, as shown in 4.4. These environment variables are stored in a dictionary when the polling script is started, and the `SERVICE` and `PORT` variables store the name of the `pcap-service`. The `python` requests library enables the script to send a request to the `pcap-service`.

```
root@aks-agentpool-28863865-vmss00000E:/# env | grep 'PCAP'
PCAP_INTERNAL_SERVICE_PORT_5000_TCP_PORT=5000
PCAP_INTERNAL_SERVICE_PORT=tcp://10.0.98.20:5000
PCAP_INTERNAL_SERVICE_SERVICE_PORT=5000
PCAP_INTERNAL_SERVICE_PORT_5000_TCP_PROTO=tcp
PCAP_INTERNAL_SERVICE_PORT_5000_TCP_ADDR=10.0.98.20
PCAP_INTERNAL_SERVICE_PORT_5000_TCP=tcp://10.0.98.20:5000
PCAP_INTERNAL_SERVICE_SERVICE_HOST=10.0.98.20
root@aks-agentpool-28863865-vmss00000E:/# █
```

Figure 4.4: Environment variables in a `tcpdump` container. Note: For these environment variables to be created, it is imperative to deploy the service before creating a pod.

```

env_dict = os.getenv

SERVICE = env_dict["PCAP_INTERNAL_SERVICE_SERVICE_HOST"]
PORT = env_dict["PCAP_INTERNAL_SERVICE_SERVICE_PORT"]

def send_file(path):
    try:
        with open(path, "rb") as file:
            file_dict = {"uploaded_file": file}
            try:
                response = requests.post(
                    f"http://{SERVICE}:{PORT}/file",
                    files=file_dict
                )
                return response.status_code
            except Exception as e:
                logging.error(e)
    except Exception as e:
        logging.error(e)

```

Listing 4.4: Snippet of polling code to send files using requests library in Python

4.2.1.4 Creating tcpdump docker image

To create a docker image containing the tcpdump-script (Listing 4.3) and the polling script (Listing 1), an ubuntu base image was chosen (Listing 4.5). This was because it had the apt-get package manager and bash, which was useful for downloading new packages and debugging while in the container environment. As shown in the docker file, the scripts were copied to the ./scripts directory and two new folders used by the scripts were created. Required packages and python were installed and, for debugging purposes, also had net-tools and vim installed.

```

FROM ubuntu:18.04
COPY tcpdump-script.sh polling.py ./scripts/
RUN mkdir ./scripts/pcap-file
RUN mkdir ./scripts/pcap-to-send
RUN apt-get update && apt-get install -y tcpdump
net-tools python3 python3-pip vim

```



```
RUN pip3 install requests
```

Listing 4.5: Dockerfile for tcpdump-container image

To create a multiplatform image using this Dockerfile, a new builder instance in Docker was instantiated that could run multiple builds in parallel. This command was only run once; the default builder instance is the latest one just created after running the command. After this, the second command in the code snippet 4.6 uses the newly created builder instance and creates an image for Linux/amd64 and Linux/arm64 based architectures. This image was tagged `<dockerhub_username>/tcpdump-container:latest` and pushed to DockerHub.

```
$docker buildx create --use  
  
$docker buildx build --platform linux/amd64,linux/arm64 --push  
-t hotshot07/tcpdump-container:latest .
```

Listing 4.6: Bash commands for multi-platform build on docker

4.2.1.5 Deploying and collecting data

After the image was created, it was used to create a Deployment resource in the Kubernetes cluster using the YAML file in snippet 4.1. This resource was deployed to the cluster using listing 4.7. After the pod was created in the cluster, using `kterm` alias and `tcpdump-containers'` pods name, a shell into the pod was created, and the polling and `tcpdump` script was started. Although both scripts could be started from the YAML configuration, this was not implemented to have more control over the scripts and change the `tcpdump` parameters for testing.

```
$kubectl apply -f <name-of-yaml-file>.yaml
```

Listing 4.7: Kubectl command to create a resource using a YAML file

If the aim was to collect data for training a machine learning model, only the `tcpdump` script could be run, and the files could be copied from the `pcap-to-send` directory to a local directory. This could be done using the `kubectl cp` command as shown in 4.8. This command copies the files from the container and saves them to the local machine in `<local-dir-path>`.

```
$kubectl cp default/<name-of-pod>:/scripts/pcap-to-send <local-dir-path>
```

Listing 4.8: Bash commands for multi-platform build on docker

4.2.2 Pcap service

This service aims to receive pcap files, extract valuable features using the Scapy package, and convert the file into a CSV for further processing. Section 4.2.2.1 explains how a server was created that receives a file in Flask, and Section 4.2.2.2 briefly talks about how a pcap was processed using the Scapy package and converted to CSV.

4.2.2.1 Flask server

To create a server that receives files, Flask was used. As explained in 3.2, Flask can create a server in very few lines of code. Having one route was sufficient for this service, as the only purpose of this server was to receive files. To implement that was relatively straightforward, as shown in the code snippet 4.9.

```
import os
from flask import Flask, request
from werkzeug.utils import secure_filename
import logging

logging.basicConfig(level=logging.DEBUG)

FILE_FOLDER = "./received-files"

app = Flask(__name__)
app.config["FILE_FOLDER"] = FILE_FOLDER

@app.route("/file", methods=["GET", "POST"])
def main():
    if request.method == "POST":
        file = request.files["uploaded_file"]
        filename = secure_filename(file.filename)
        file.save(os.path.join(app.config["FILE_FOLDER"], filename))
        logging.info("File saved: " + filename)
    return "file-received"
```

```

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)

```

Listing 4.9: Python code for receiving files via POST request

It was crucial to run the server on host "0.0.0.0"; otherwise, the requests from outside the container are not received by the service.

4.2.2.2 Processing pcap

This script was similar to the polling script in listing 1. It polls the ./received-files directory to check if any new files were received. If new files were in the directory, the file's path was put into a queue. When the queue was processed, the file was converted into a CSV using the `convert_pcap_to_csv` function.

The `convert_pcap_to_csv` function uses Scapy to extract data from the pcap file. The `rdpcap` function provided by Scapy was used to iterate over the packets in the pcap file. This packet was passed as a constructor argument for the `Packet` class. In Scapy's implementation, packets can be broken down into Layers. A TCP packet, for example, would have an Ethernet layer, IP layer, TCP layer and Raw layer containing data. Features from the different layers can be extracted and stored in a dictionary. Also, some packet features like length and timestamp of capture are extracted and returned in a dictionary. The code for this is explained in section 4.4.2.1. The extracted features shown in table 4.1 were the final features chosen for the ML model. The process of selecting these features is explained in the section 5.1.3.

Feature	Datatype	Description
Length	int	Total length of the packet
Timestamp	float	utctimestamp of packet capture
ip.src	string	Source IP address of packet
source_internal	int	Set to 1 if packet from inside cluster
source_external	int	Set to 1 if packet from outside cluster
ip.dst	string	Destination IP address of packet
destination_internal	int	Set to 1 if packet from inside cluster
destination_external	int	Set to 1 if packet from outside cluster
protocol	string	TCP/UDP
protocol.sport	int	Source port
protocol.dport	int	Destination port

Table 4.1: Final packet features selected for the NIDS

	length	timestamp	ip.src	ip.dst	protocol	protocol.sport	protocol.dport	source_internal	source_external	destination_internal	destination_external
0	180	1.648426e+09	10.240.0.4	10.244.0.6	TCP	10250	36032	1	0	1	0
1	310	1.648426e+09	10.244.0.6	20.73.113.93	TCP	43036	9000	1	0	0	1
2	66	1.648426e+09	20.73.113.93	10.244.0.6	TCP	9000	43036	0	1	1	0
3	198	1.648426e+09	20.73.113.93	10.244.0.6	TCP	9000	43036	0	1	1	0
4	66	1.648426e+09	10.244.0.6	20.73.113.93	TCP	43036	9000	1	0	0	1

Figure 4.5: Features extracted from the pcap are stored in a CSV in the format shown above.

While creating a CSV from the list of dictionaries, the dictionary's keys were the headers, while the values were row items. The CSV file was written to a `./processed-files/` directory. The function returned the path of the newly created CSV. This path was then used as an argument to the `send_file` function, which sends the file to model-service.

Deployment of this and the rest of the services are not discussed as the process to deploy all the services were similar. After creating a multiplatform docker image, the required scripts were run from the deployment configuration specified in the YAML file, as shown in listing 4.10.

```
containers:
  -name: pcap-csv
    image: hotshot07/pcap-service:latest
    imagePullPolicy: Always
    command: ["/bin/sh", "-c"]
    args: ["python3 pcap_server.py & python3 process_pcap.py"]
```

Listing 4.10: Snippet of YAML file for pcap service that shows how to run a command to start scripts

4.2.3 Model service

The model service's function was to receive the CSV generated by the pcap-service, pre-process the CSV's data, and generate a score for the different packets using the pre-trained autoencoder model. It then loaded the CSV into the MySQL database. To create an image of this service, a `tensorflow:2.7.1` base image was used. The implementation of creating the autoencoder model is discussed in section 4.4.5.

4.2.3.1 Preprocessing data

Model service had a structure that is quite similar to the structure of the pcap service. It had the same Flask server as in the pcap service(4.9) for receiving CSV files and had a

similar script to poll for new files. After the CSV file path was dequeued for processing, using `process_and_run_prediction` function the packets are scored.

These preprocessing steps were chosen after many iterations on autoencoder models as discussed in section 5.1.3. The packet features from table 4.1 were the final features chosen, and their preprocessing is shown in table 4.2

Feature	Pre-processing step
Length	Unprocessed
Timestamp	Unprocessed
ip.src	Split into 4 octets
source_internal	Unprocessed
source_external	Unprocessed
ip.dst	Split into 4 octets
destination_internal	Unprocessed
destination_external	Unprocessed
protocol	One hot encoded
protocol.sport	Unprocessed
protocol.dport	Unprocessed

Table 4.2: Processing of packet features in table 4.1

The IPs were split into four octets, and the protocol, which consisted of TCP or UDP values, was one-hot encoded. Many features were left unprocessed as they were already in the proper format; for example, the `source_internal` was created in such a way in that it was already in a categorical format as it had only 1 or 0 values.

protocol	
0	TCP
1	UDP
2	TCP
3	TCP
4	UDP

Before one hot encoding

	TCP	UDP
0	1	0
1	0	1
2	1	0
3	1	0
4	0	1

After one hot encoding

Figure 4.6: asdfasdasd

After initial preprocessing, another feature `packet flow` was created. This feature was engineered as it was seen in 4.4.4 that packet flow is an excellent feature for anomaly detection. This feature is the number of packets flowing through the node every 10 seconds. If 2000 packets were flowing between time t and $t+1$, every packet during the

time is assigned a value of 2000. Another approach was also considered where the packet flow from individual IP source and destination was created, as shown in the appendix .6, but that feature did not perform as well as expected for this dataset.

```
# time_df is the new dataframe which  
# has timestamp from unprocessed_df  
  
time_df = pd.DataFrame(unprocessed_df["timestamp"])  
  
time_df["packet"] = 1  
time_df["timestamp"] = time_df["timestamp"].apply(lambda x:  
                                                    datetime.fromtimestamp(x))  
time_df = time_df.set_index("timestamp")  
  
time_df = time_df.resample("10s").sum()  
  
packet_flow_list = []  
  
for x in list(time_df["packet"]):  
    packet_flow_list += [x] * x  
  
processed_df["packet_flow"] = packet_flow_list
```

Listing 4.11: Code for creating packet flow feature from timestamp

4.2.3.2 Predicting scores

After initial preprocessing, the features were converted into `float64` datatype. Using `StandardScaler` method in `sklearn`, the features were standardised by removing the mean and scaling to unit variance. The pre-trained autoencoder model, which was trained on normal data to create a baseline for normal behaviour was loaded using `Keras`. A prediction vector was created using `model.predict` function. Every array in this vector was the reconstruction of the packet according to the autoencoder model.

For determining if a packet is anomalous, a score was assigned to each packet. Root Mean Square Error (RMSE) was used for creating a score. This score represents the error between the actual packet and its reconstruction. A higher score means the model could not generate a reasonable reconstruction of the packet, as it had not seen a packet like this before. After assigning a score, a Universally Unique Identifier (UUID) was generated for

each packet to differentiate packets when being stored in a database. This CSV was then saved to a `processed_csvs` folder, and the path for the CSV was returned (see figure 4.7 for structure of the CSV).

uuid	timestamp	length	ip.src	source_internal	source_external	ip.dst	destination_internal	destination_external	protocol	protocol.sport	protocol.dport	score
cd3a3440edbf456ca2b2b6abc008734d	1.645211e+09	96	10.240.0.4	1	0	10.244.0.11	1	0	TCP	10250	49190	0.296678
582d01e2622e4b299295f5da53f64fe4	1.645211e+09	172	10.240.0.4	1	0	10.244.0.11	1	0	TCP	10250	49190	0.295463
2abe6186ba714408b6669e452328e953	1.645211e+09	230	10.244.0.11	1	0	52.143.28.136	0	1	TCP	54822	9000	0.612088
19de619fed59441f97b04f5880fec907	1.645211e+09	66	52.143.28.136	0	1	10.244.0.11	1	0	TCP	9000	54822	0.509443
ca5709bea49f4cc682df0257a9113fd5	1.645211e+09	198	52.143.28.136	0	1	10.244.0.11	1	0	TCP	9000	54822	0.503236

Figure 4.7: The structure of the data frame being stored in MySQL database

After the CSV was saved, it was loaded into the database using python's MySQL connector package. If the loading of data was successful, the file was deleted from the server. The snippet to load a CSV file to a MySQL database is shown in listing 4.12.

```
# create a connection, using config dictionary
cnx = mysql.connector.connect(**config)

cursor = cnx.cursor()

logging.info(f"Connected to MySQL database} \
             sending {path_to_processed_csv}")

csv_import = ( f"""LOAD DATA LOCAL INFILE '{path_to_processed_csv}'
                INTO TABLE pcap_table
                FIELDS
                TERMINATED BY ','
                ENCLOSED BY '"'
                LINES TERMINATED BY '\n'
                """)

cursor.execute(csv_import)
cnx.commit()
```

Listing 4.12: Python code to load a CSV file into MySQL database

4.2.4 MySQL database

To create a MySQL database, the `mysql:oracle` docker image was used. To store the data in a database inside the Kubernetes cluster, Kubernetes PersistentVolume (PV) and PersistentVolumeClaim (PVC) resources were needed. PVs are resources in the cluster, while PVCs are requests for those resources and act as claim checks to the resource. The code to create a PV, PVC and Deployment resource in Kubernetes using is given in Appendix .3.

After shelling into the running pod and launching a container, the MySQL credentials were used for logging into the database. The table `pcap_table` was created in the `pcap` database using the commands as shown in 4.13

```
CREATE DATABASE pcap;

USE pcap;

CREATE TABLE pcap_table(
  uuid VARCHAR(36) NOT NULL,
  utctimestamp DOUBLE PRECISION,
  packet_length VARCHAR(100),
  ip_src VARCHAR(50),
  source_internal VARCHAR(10),
  source_external VARCHAR(10),
  ip_dst VARCHAR(50),
  destination_internal VARCHAR(10),
  destination_external VARCHAR(10),
  protocol VARCHAR(10),
  protocol_sport VARCHAR(10),
  protocol_dport VARCHAR(10),
  score DOUBLE,
  PRIMARY KEY (uuid)
);
```

Listing 4.13: Creating `pcap_table` in MySQL to store CSVs

After creating the table, it was imperative to run the command shown in 4.14 to allow files to be loaded into the database.

```
SET GLOBAL local_infile = true;
```


Listing 4.14: Setting local_infile to true to load files

4.2.5 Back-end service

The back-end service was created to interact with the database and display the scores. As this service is not a part of the data processing pipeline, it could be hosted locally. By exposing the MySQL service as a LoadBalancer, the backend service was able to connect and interact with the DB using the external IP of the MySQL service. Due to the project's time constraints, the functionalities of this service could not be fully implemented.

4.2.5.1 Flask server

The Flask server connects with the MySQL database using the `mysql connector` Python package. To display the top 1000 anomalous packets, it then queries the database with listing 4.15

```
SELECT utctimestamp, packet_length, ip_src, source_internal, source_external,
ip_dst, destination_internal, destination_external,
protocol, protocol_sport, protocol_dport, score
FROM pcap_table
WHERE protocol = 'TCP'
AND destination_external = 1
AND score > 1
ORDER by score desc, packet_length desc
LIMIT 1000
```

Listing 4.15: SQL command to retrieve top 1000 anomalous packets going out of the cluster

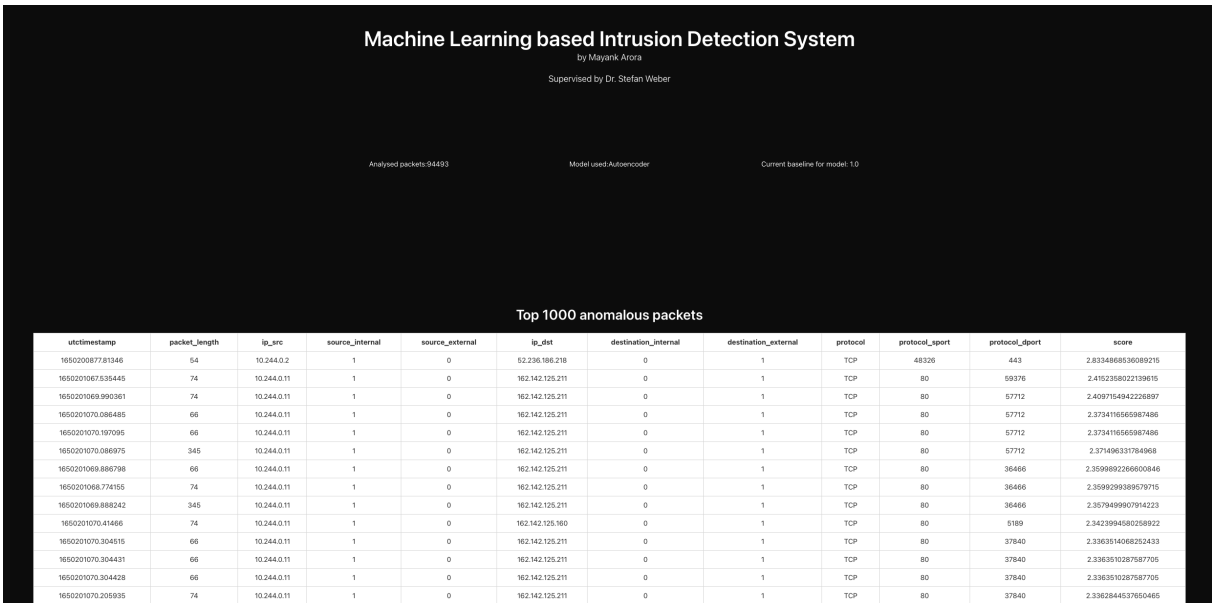


Figure 4.8: Screenshot of webpage displaying top 1000 anomalous packets. The SQL query for populating the table is shown in listing 4.15

The value returned from the query above was turned into a dictionary and served on a webpage using a dynamically resizable table. This statement was run to check if any packets had been sent from the exposed MySQL database to another IP.

```
Select ip_dst, count(*) as count
from pcap_table
where protocol_sport = 3306
and destination_external = 1
group by ip_dst
order by count desc
```

Listing 4.16: SQL command to retrieve data list of IPs trying to contact port 3306, the open port for MySQL

4.3 Simulating an attack

To get anomalous data and test the machine learning model, an attack was simulated on the WordPress application. Section 4.3.1 talks about various port scanning techniques using nmap and how they can be used to detect a host OS. Section 4.3.2 describes how WPScan, a WordPress site scanner, was used to launch a dictionary attack against the website hosted on Azure Kubernetes Services.

4.3.1 Port scanning

A port scanning tool is a standard tool in an attacker's library. Port scanners help an attacker identify any open ports on an IP that can potentially be exploited.

To run a port scan, Nmap, a very popular open-source port scanning tool, can be used. It provides several techniques like Ping scan, TCP half-open scan, and Xmas scan. It can even launch aggressive scans to detect a Hosts OS. For this project, a script with multiple nmap scans was created to simulate different kinds of scans possible by the attacker.

```
HOST='<IP-of-deployed-website>'
#TCP SYN scan
nmap -sS $HOST

#TCP connect scan
nmap -sT $HOST

#Probe open ports to determine service/version info
nmap -sV $HOST

#Enable OS detection
nmap -O $HOST

#A: Enable OS detection, version detection, script scanning,
#and traceroute
nmap -A $HOST
```

Listing 4.17: Nmap script to scan open ports on a website

This nmap script was run from a Kali Linux machine, and it was able to figure out the open ports on the deployed WordPress application and the OS type, as shown in snippet of the output 4.18. `proxychains4` can also be used to port scan using a proxy server to hide the IP address, but this was not used in the implementation as using free proxy servers seemed risky. Snippet of the console output is given in 4.18, and the full output is given in appendix .4.

```
Nmap done: 1 IP address (1 host up) scanned in 11.04 seconds
Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-07 17:28 IST
Nmap scan report for 20.31.228.177
Host is up (0.020s latency).
Not shown: 998 filtered tcp ports (no-response)
```

```

PORT      STATE SERVICE  VERSION
80/tcp    open  http     Apache httpd 2.4.48 ((Unix) OpenSSL/1.1.1d PHP/7.4.21)
|_http-server-header: Apache/2.4.48 (Unix) OpenSSL/1.1.1d PHP/7.4.21
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
|_http-title: Mayank's Blog! ; Just another WordPress site
|_http-generator: WordPress 5.7.2
443/tcp   open  ssl/http Apache httpd 2.4.48 ((Unix) OpenSSL/1.1.1d PHP/7.4.21)
|_http-title: Mayank's Blog! ; Just another WordPress site
| ssl-cert: Subject: commonName=example.com
| Not valid before: 2012-11-14T11:18:27
|_Not valid after:  2022-11-12T11:18:27
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
|_http-server-header: Apache/2.4.48 (Unix) OpenSSL/1.1.1d PHP/7.4.21
|_http-generator: WordPress 5.7.2
|_ssl-date: TLS randomness does not represent time
Warning: OSScan results may be unreliable because we could not find at least
1 open and 1 closed port
Device type: general purpose
Running (JUST GUESSING): Linux 4.X|5.X|2.6.X (87%)
OS CPE: cpe:/o:linux:linux_kernel:4.0 cpe:/o:linux:linux_kernel:5
cpe:/o:linux:linux_kernel:2.6.32
Aggressive OS guesses: Linux 4.0 (87%), Linux 4.15 - 5.6 (86%),
Linux 5.0 (86%), Linux 5.0 - 5.4 (86%), Linux 5.3 - 5.4 (85%),
Linux 2.6.32 (85%), Linux 5.0 - 5.3 (85%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 25 hops

TRACEROUTE (using port 80/tcp)
HOP RTT      ADDRESS
1   5.40 ms  10.10.0.1
2   5.45 ms  089-100-107150.ntlworld.ie (89.100.107.150)
3   6.20 ms  089-101-115225.ntlworld.ie (89.101.115.225)
4   6.21 ms  089-100-182198.ntlworld.ie (89.100.182.198)
5   7.25 ms  ie-dub01a-rc1-ae-15-0.aorta.net (84.116.238.249)
6   6.20 ms  ie-dub02a-ri1-ae-73-0.aorta.net (84.116.134.110)
7   17.50 ms ae68-0.ier02.dba.ntwk.msn.net (104.44.198.115)
8   6.25 ms  ae25-0.icr02.dub07.ntwk.msn.net (104.44.239.35)
9   19.46 ms be-122-0.ibr02.dub07.ntwk.msn.net (104.44.11.73)
10  87.63 ms be-7-0.ibr01.ams30.ntwk.msn.net (104.44.17.57)
11  18.27 ms be-1-0.ibr02.ams30.ntwk.msn.net (104.44.16.147)
12  ... 24
25  21.85 ms 20.31.228.177

```

Listing 4.18: Snippet of console output of the nmap scan script

4483	23.192471	10.244.0.19	89.100.107.148	TLsv_	337 Application Data
4528	23.225449	10.244.0.19	89.100.107.148	TCP	2962 8080 -> 52816 [PSH, ACK] Seq=1 Ack=19 Win=65152 Len=2896 TSval=2628493174 TSecr=2618198626 [TCP segment of a reassembled PDU]
4529	23.225478	10.244.0.19	89.100.107.148	TCP	2962 8080 -> 52816 [PSH, ACK] Seq=2897 Ack=19 Win=65152 Len=2896 TSval=2628493174 TSecr=2618198626 [TCP segment of a reassembled PDU]
4530	23.225486	10.244.0.19	89.100.107.148	TCP	2962 8080 -> 52816 [PSH, ACK] Seq=5793 Ack=19 Win=65152 Len=2896 TSval=2628493174 TSecr=2618198626 [TCP segment of a reassembled PDU]
4531	23.225494	10.244.0.19	89.100.107.148	TCP	123 8080 -> 52816 [PSH, ACK] Seq=8689 Ack=19 Win=65152 Len=57 TSval=2628493174 TSecr=2618198626 [TCP segment of a reassembled PDU]
4539	23.226045	10.244.0.19	89.100.107.148	TCP	676 8080 -> 52816 [PSH, ACK] Seq=8746 Ack=19 Win=65152 Len=610 TSval=2628493175 TSecr=2618198626 [TCP segment of a reassembled PDU]
4540	23.226111	10.244.0.19	89.100.107.148	HTTP	66 HTTP/1.0 200 OK (text/html)
5834	29.199654	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40680 [ACK] Seq=1783 Ack=638 Win=64768 Len=0 TSval=2628499237 TSecr=2618204697
5228	29.288350	10.244.0.19	89.100.107.148	TCP	2962 8443 -> 40680 [PSH, ACK] Seq=1783 Ack=638 Win=64768 Len=2896 TSval=2628499237 TSecr=2618204697 [TCP segment of a reassembled PDU]
5221	29.288378	10.244.0.19	89.100.107.148	TCP	2962 8443 -> 40680 [PSH, ACK] Seq=4679 Ack=638 Win=64768 Len=2896 TSval=2628499237 TSecr=2618204697 [TCP segment of a reassembled PDU]
5222	29.288386	10.244.0.19	89.100.107.148	TLsv_	2488 Application Data
5233	29.289048	10.244.0.19	89.100.107.148	TLsv_	1273 Application Data
5234	29.289129	10.244.0.19	89.100.107.148	TLsv_	90 Application Data
5235	29.289238	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40680 [FIN, ACK] Seq=11228 Ack=638 Win=64768 Len=0 TSval=2628499238 TSecr=2618204697
5252	29.328563	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40680 [ACK] Seq=11229 Ack=662 Win=64768 Len=0 TSval=2628499269 TSecr=2618204818
5256	29.321444	10.244.0.19	89.100.107.148	TCP	74 8080 -> 52820 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2628499270 TSecr=2618204819 WS=128
5258	29.321627	10.244.0.19	89.100.107.148	TCP	74 8080 -> 52822 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2628499271 TSecr=2618204820 WS=128
5268	29.323857	10.244.0.19	89.100.107.148	TCP	74 8080 -> 52818 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2628499272 TSecr=2618204819 WS=128
5262	29.323194	10.244.0.19	89.100.107.148	TCP	74 8443 -> 40682 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2628499272 TSecr=2618204819 WS=128
5264	29.323452	10.244.0.19	89.100.107.148	TCP	74 8443 -> 40686 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2628499272 TSecr=2618204820 WS=128
5270	29.344208	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40682 [ACK] Seq=1 Ack=518 Win=64768 Len=0 TSval=2628499293 TSecr=2618204842
5273	29.344959	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40686 [ACK] Seq=1 Ack=518 Win=64768 Len=0 TSval=2628499294 TSecr=2618204842
5274	29.345683	10.244.0.19	89.100.107.148	TLsv_	1386 Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
5275	29.345368	10.244.0.19	89.100.107.148	TLsv_	1386 Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
5277	29.367930	10.244.0.19	89.100.107.148	TCP	66 8080 -> 52820 [ACK] Seq=1 Ack=19 Win=65152 Len=0 TSval=2628499316 TSecr=2618204866
5279	29.367218	10.244.0.19	89.100.107.148	TCP	66 8080 -> 52822 [ACK] Seq=1 Ack=618 Win=64640 Len=0 TSval=2628499316 TSecr=2618204866
5283	29.368183	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40682 [ACK] Seq=1241 Ack=598 Win=64768 Len=0 TSval=2628499317 TSecr=2618204866
5284	29.368371	10.244.0.19	89.100.107.148	TLsv_	337 Application Data
5285	29.368450	10.244.0.19	89.100.107.148	TLsv_	337 Application Data
5293	29.368620	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40686 [ACK] Seq=1241 Ack=598 Win=64768 Len=0 TSval=2628499318 TSecr=2618204866
5296	29.368779	10.244.0.19	89.100.107.148	TLsv_	337 Application Data
5298	29.368845	10.244.0.19	89.100.107.148	TCP	66 8080 -> 52818 [ACK] Seq=1 Ack=176 Win=65024 Len=0 TSval=2628499318 TSecr=2618204866
5300	29.368942	10.244.0.19	89.100.107.148	TLsv_	337 Application Data
5542	29.388627	10.244.0.19	89.100.107.148	TCP	66 8443 -> 40682 [ACK] Seq=1783 Ack=795 Win=64640 Len=0 TSval=2628499338 TSecr=2618204886

Figure 4.9: Tcpdump output of the nmap scan in Wireshark

4.3.2 Dictionary attack

To launch a password dictionary attack against the deployed WordPress site, WPScan was used. WPScan is an open-source security scanner to detect vulnerabilities in a WordPress website. It can detect the kind of WordPress theme, the plugins installed and if they are out of date, etc. It can also find users on a website via scanning blog posts. Against these users, a password dictionary attack can be launched.

For a password dictionary, a common dictionary `rockyou.txt` was used, which contains the most commonly used passwords in plaintext. Using the command below shown in listing 4.19, a dictionary attack can be launched for `$HOST`, which enumerates all the users on the WordPress website and tries the username-password combination. For the purposes of generating the figure, the password was added to `rockyou.txt`. WPScan was successfully able to find the username-password combination of the WordPress application deployed. During this attack, this anomalous data was captured via tcpdump as shown in figure 4.10. Snippet of the console output is given in 4.20, and the full output is given in appendix .4.

```
HOST='<IP-of-deployed-website>'
wpscan --url $HOST --enumerate u --passwords rockyou.txt --max-threads 50
```

Listing 4.19: WPScan command for password attack

606.	27.105447	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
606.	27.118514	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
607.	27.160298	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
612.	27.295465	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
612.	27.310799	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
613.	27.330286	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
613.	27.334111	10.244.0.19	89.100.107.148	HTTP_	693	HTTP/1.1	403	Forbidden
614.	27.350567	10.244.0.19	89.100.107.148	HTTP_	607	HTTP/1.1	200	OK
912	2.214518	10.244.0.19	89.100.107.148	TCP	74	8080 - 53336	[SVN, ACK]	Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2629350067 TSecr=2619055610 WS=128
916	2.233899	10.244.0.19	89.100.107.148	TCP	66	8080 - 53336	[ACK]	Seq=1 Ack=255 Win=65024 Len=0 TSval=2629350087 TSecr=2619055630
1108	2.311726	10.244.0.19	89.100.107.148	TCP	2962	8080 - 53336	[PSH, ACK]	Seq=1 Ack=255 Win=65024 Len=2896 TSval=2629350165 TSecr=2619055630 [TCP segment of a reassembled PDU]
1109	2.311754	10.244.0.19	89.100.107.148	TCP	2962	8080 - 53336	[PSH, ACK]	Seq=2897 Ack=255 Win=65024 Len=2896 TSval=2629350165 TSecr=2619055630 [TCP segment of a reassembled PDU]
1110	2.311762	10.244.0.19	89.100.107.148	TCP	2962	8080 - 53336	[PSH, ACK]	Seq=5793 Ack=255 Win=65024 Len=2896 TSval=2629350165 TSecr=2619055630 [TCP segment of a reassembled PDU]
1111	2.315643	10.244.0.19	89.100.107.148	TCP	94	8080 - 53336	[PSH, ACK]	Seq=0609 Ack=255 Win=65024 Len=28 TSval=2629350165 TSecr=2619055630 [TCP segment of a reassembled PDU]
1214	3.204869	10.244.0.19	89.100.107.148	TCP	66	8080 - 53336	[ACK]	Seq=8745 Ack=609 Win=64768 Len=0 TSval=2629351058 TSecr=2619056601

Figure 4.10: Wireshark output of packets from attacker's IP during WPScan attack

```

-----
--
\ \      / /  _ _ \ / ____|
\ \ / \ / / | |_) | (___ ___ _ _ _ _ _ ®
\ \ \ / / | ___/ \___ \ / ___/ _ ' | ' _ \
\ / \ / | | | ___) | (___ ( | | | | |
\ \ \ | | | ___/ \___ \ \_ _ , | | | |
\ \ \ | | | ___/ \___ \ \_ _ , | | | |

WordPress Security Scanner by the WPScan Team
Version 3.8.20
Sponsored by Automattic - https://automattic.com/
@_WPScan_, @ethicalhack3r, @erwan_lr, @firefart
-----

[+] URL: http://20.31.228.177/ [20.31.228.177]
[+] Started: Thu Apr 7 17:42:42 2022

Interesting Finding(s):

[+] Headers
| Interesting Entries:
| - Server: Apache/2.4.48 (Unix) OpenSSL/1.1.1d PHP/7.4.21
| - X-Powered-By: PHP/7.4.21
| Found By: Headers (Passive Detection)
| Confidence: 100%

[+] robots.txt found: http://20.31.228.177/robots.txt
| Interesting Entries:
| - /wp-admin/
| - /wp-admin/admin-ajax.php
| Found By: Robots Txt (Aggressive Detection)
| Confidence: 100%

[+] XML-RPC seems to be enabled: http://20.31.228.177/xmlrpc.php

```

```

.
.
.

[+] Enumerating Users (via Passive and Aggressive Methods)
    Brute Forcing Author IDs - Time: 00:00:00 <=====
===== > (10 / 10) 100.00% Time: 00:00:00

[i] User(s) Identified:

[+] mayank
    | Found By: Author Posts - Author Pattern (Passive Detection)
    | Confirmed By:
    |   Rss Generator (Passive Detection)
    |   Wp Json Api (Aggressive Detection)
    |     - http://20.31.228.177/wp-json/wp/v2/users/?per_page=100&page=1
    |   Rss Generator (Aggressive Detection)
    |   Author Sitemap (Aggressive Detection)
    |     - http://20.31.228.177/wp-sitemap-users-1.xml
    |   Author Id Brute Forcing - Author Pattern (Aggressive Detection)
    |   Login Error Messages (Aggressive Detection)

[+] Performing password attack on Xmlrpc against 1 user/s
[SUCCESS] - mayank / arora
Trying mayank / arora Time: 00:00:07 <=          > (195 / 59385) 0.32% ETA: ??:?:??

[!] Valid Combinations Found:
    | Username: mayank, Password: arora

```

Listing 4.20: Snippet of console output of WPScan

```

<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>
            faultCode
          </name>
          <value>
            <int>
              403
            </int>
          </value>
        </member>
        <member>
          <name>
            faultString
          </name>
          <value>
            <string>
              Incorrect username or password.
            </string>
          </value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>

```

XML output of server responding with invalid credentials

```

<name>
  isAdmin
</name>
<value>
  <boolean>
    1
  </boolean>
</value>
</member>
<member>
  <name>
    url
  </name>
  <value>
    <string>
      http://20.31.228.177/
    </string>
  </value>
</member>
<member>
  <name>
    blogid
  </name>
  <value>
    <string>
      1
    </string>
  </value>
</member>
<member>
  <name>
    blogName
  </name>
  <value>
    <string>
      Mayank&amp;#039;s Blog!
    </string>
  </value>
</member>
<member>
  <name>
    xmlrpc
  </name>
  <value>
    <string>
      http://20.31.228.177/xmlrpc.php
    </string>
  </value>
</member>

```

XML output of server when credentials are found

Figure 4.11: The difference between server response when incorrect credentials are entered vs when credentials are found

4.4 Creating machine learning models

This section goes into detail about how various machine learning models were implemented. To create a machine learning environment on M1 Mac that utilises the GPU, a video ³ by Jeff Heaton was followed, which explains in detail how to set up an environment locally.

Section 4.4.1 briefly explains how the data was collected for training and testing the machine learning models. Section 4.4.2 describes in detail how the data was cleaned and processed using Scapy and Pandas. Section 4.4.3 goes into detail about how SVD and DBSCAN were tested for unsupervised clustering, and section 4.4.5 goes into detail about how an autoencoder was implemented. Section 4.4.4 explains how packet flow was a helpful feature by implementing Isolation Forest, an unsupervised machine learning technique. Section 4.4.6 talks about how a supervised learning model was implemented to test the feasibility of using a supervised model.

³video link

4.4.1 Data collection

Two data collection sessions were done to collect data for machine learning models. In the first session, the normal baseline behaviour of the WordPress application was observed for 10 minutes. All the packets were captured via tcpdump on 'cbr0'. A total of 140538 packets were received in this time frame. Also, there are 1549 packets in this data with the attacker's IP (browsing the blog, not attacking currently).

In the second session, to collect anomalous data for testing the model, The port scanning and dictionary attacks as described in section 4.3 were run. This time, the password was removed from the password dictionary so that the enumeration would go on for a longer duration of time. A total of 1070554 packets were received. Out of these packets, 41824 packets had 'ip.dst' of the Kali Linux machine from where the attack was launched. Summary of the total packets captured in shown in table 4.3.

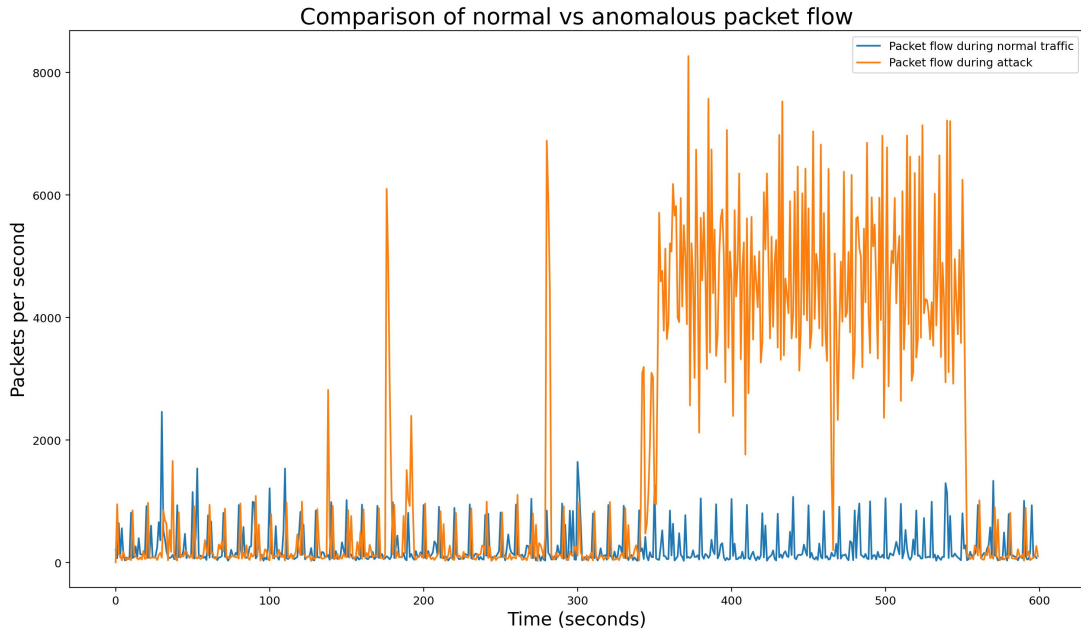


Figure 4.12: Packet flow during normal time vs attack. The first two spikes observed are the Nmap Scans, and from 350 seconds onwards, the dictionary attack was launched

Type of data	Total packets captured
Normal data flow	140538
Attack data flow	1070554
Packets with attackers IP	41824

Table 4.3: Total packets captured for training and testing ML models

4.4.2 Data preprocessing

This section details how a pcap file was converted to CSV for training machine learning models.

4.4.2.1 Processing pcap

A Packet class was created to get all of these features from all the packets and maintain consistency with a `extract_data` function. This function gets the data from the packet and writes it to a CSV. To extract useful features from a pcap, Scapy was used. `rdpcap` can be used to iterate over all the packets in the pcap file. To see the structure of the first packet in the list, `.show()` can be called on the packet variable, which shows the different layers of the packet.

```
def preprocess(path):
    pcap = rdpcap(path)

    for packet_var in pcap:
        print(packet_var.show())
        break

if __name__ == '__main__':
    preprocess_path('<name_of_file>.pcap')
```

Listing 4.21: Dockerfile for Pcap service

```
###[ Ethernet ]###
dst      = a6:4d:66:07:0e:f8
src      = 16:78:bc:af:56:36
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 82
id       = 23589
flags    = DF
frag     = 0
ttl      = 64
```

```

    proto      = tcp
    chksum     = 0xc88e
    src        = 10.240.0.4
    dst        = 10.244.0.11
    \options   \
###[ TCP ]###
    sport      = 10250
    dport      = 49190
    seq        = 166982416
    ack        = 1524731990
    dataofs    = 8
    reserved   = 0
    flags      = PA
    window     = 501
    chksum     = 0x1637
    urgptr     = 0
    options    = [('NOP', None), ('NOP', None),
                  ('Timestamp', (164084236, 3145284487))]
###[ Raw ]###
    load       = '\x17\x03\x03\x00\x19\xff\x9c\xc1\xe#IH
                  \xf7q+m\xbb}\xa2\xbdQ\xbb\xa2\xb71
                  \xb0\x13\tp\r'

```

Listing 4.22: Console output of above above function

A packet in scapy is in the form of layers as shown in 4.22, above the packet has Ethernet, IP, TCP and Raw layers. Every layer has its fields that can be extracted. In 4.7, printing `packet.layers()` would generate a list of classes of layers

```

[<class 'scapy.layers.l2.Ether'>, <class 'scapy.layers.inet.IP'>,
<class 'scapy.layers.inet.TCP'>, <class 'scapy.packet.Raw'>]

```

Listing 4.23: Console output of above above function

The names of these layers can be extracted by using `layer.__name__`. This can be stored in a variable `layer_name`, and while iterating over layer names, features of a particular layer can be extracted, as shown in 5.1. Running this code for the packet in Listing 4.22 would 10.240.0.4.

```

for layer in self.packet.layers():
    packet_layer = self.packet[layer.__name__]

```

```

layer_name = layer.__name__

if layer_name == "IP":
    self.packet_dict["ip.src"] = packet_layer.fields.get("src")
...

```

Listing 4.24: Extracting IP.src from from a packet

Flags from a packet in scapy can be extracted using code in snippet 4.31.

```

# The flags set in the packet are returned in character
# format, which is the key in the flag_dict dictionary. The common
# name of the flag is returned by self.flag_dict.get(str(flag))

flag_dict = {
    "F": "FIN",
    "S": "SYN",
    "R": "RST",
    "P": "PSH",
    "A": "ACK",
    "U": "URG",
    "E": "ECE",
    "C": "CWR",
    "?": "UNK",
}

for flag in self.packet.sprintf("%TCP.flags%"):
    flag_type = self.flag_dict.get(str(flag))

```

Listing 4.25: Extracting flags set in a TCP packet

The data from the packet can also be extracted from the Raw packet layer. The packets are in a byte string format but can be converted to hex and further converted to integer processing. In the code snippet, the length of the packet data and the first 10 bytes of the packets are

```

#packet_layer.fields.get('load') is of the form
#b '\x17\x03\x03\x00\x19\xf5\x9c\xc1\x0e#IH\xf7q+m\xbb}
# \xa2\xbdQ\xbb\xa2\xb71\xb0\x13\tp\r'

```

```

#first 10 bytes extracted and converted to hex
#1703030019f59cc10e23

# hex are then converted to integers
# using hex lambda and stored in different key_value pairs,
# resulting in 20 features

hex_lambda = lambda x: int(x,16)

if layer_name == 'Raw':
    self.packet_dict['load.count'] = len(packet_layer.fields.get('load'))
    current_load = str(packet_layer.fields.get('load')[:10].hex())

    if current_load != str(0):
        for i, ch in enumerate(current_load):
            self.packet_dict[f'load_{i}'] = hex_lambda(ch)

```

Listing 4.26: Extracting raw packet data

Another function was created to check if the IP.src and IP.dst came from within the cluster or outside to provide context to a machine learning model about the IP addresses. Azure provides the IP ranges of pods, services and other internal services.



Networking

API server address	mayank-thesis-test-dns-e0ff2b94.hcp.westeurope.azmk8s.io
Network type (plugin)	Kubenet
Pod CIDR	10.244.0.0/16
Service CIDR	10.0.0.0/16
DNS service IP	10.0.0.10
Docker bridge CIDR	172.17.0.1/16
Network Policy	None
Load balancer	Standard
HTTP application routing	Enabled
Private cluster	Not enabled
Authorized IP ranges	Not enabled
Application Gateway ingress controller	Not enabled

Figure 4.13: Azure networking IP ranges

```
def check_if_ip_is_internal_or_external(self, ip, kind = None):
    if not ip:
        return None

    split_ip = ip.split('.')

    if split_ip[0] == '172' and split_ip[1] == '17':
        self.packet_dict[kind + '_internal'] = 1
        return

    if split_ip[0] == '10' and split_ip[1] == '0':
        self.packet_dict[kind + '_internal'] = 1
        return

    if split_ip[0] == '10' and split_ip[1] == '244':
        self.packet_dict[kind + '_internal'] = 1
        return
    else:
        self.packet_dict[kind + '_external'] = 1
        return
```

Listing 4.27: Classying IP as external or internal

The total number of features that were extracted from the CSV are shown in the table 4.4

Packet feature	Data extracted
General	length,timestamp
Ethernet	eth.src, eth.dst, eth.type
IP	ip.src, ip.dst, ip.version,ip.proto,ip.len, ip.ihl, ip.tos, ip.ttl
Protocol	protocol, protocol.sport, protocol.dport
Flags	FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, UNK
Source_ip_type	source_pod, source_external
Destination_ip_type	destination_pod, destination_external
Raw	load.count, first 10 bytes of data

Table 4.4: All features extracted from a packet for this project

4.4.2.2 Processing CSV

After the CSV was generated from the pcap, this CSV needed to be further processed before being used for machine learning purposes.

The Ethernet address was split across `:` and converted into hex using a lambda function `lambda x: int(x,16)`. The IP address was split across `.` to get four octets as features. Here, the IP and Ethernet features are not considered categorical features but rather numerical features. Hence, One hot encoding was not used. This was done to keep the number of input features the same across all inputs. Also, as this was being tested for a real-world environment, the number of new IPs hitting the service could change; hence having IPs and Ethernet addresses as categorical features and one hot encoding them could result in a massive dataset.

The rest of the columns were not further processed as they were already in the required format. A packet flow feature was also generated, as shown in 4.11. This resulted in a total of 73 features per packet. Initially, all of the features were used, but it was seen that reducing the features created a better-unsupervised machine learning model.

After this, all the columns except categorical ones were scaled using `StandardScaler` and converted to `float64` datatype so that the data could be fed into the machine learning model.

4.4.3 Clustering using DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised density-based clustering technique that can be used to detect outliers. To visualise this algorithm, the `make_blobs` function in scikit-learn was used to generate isotropic gaussian blobs(43). For this example, 1000 sample data points were generated with one centre and a standard deviation of 0.4 as shown in 4.14.

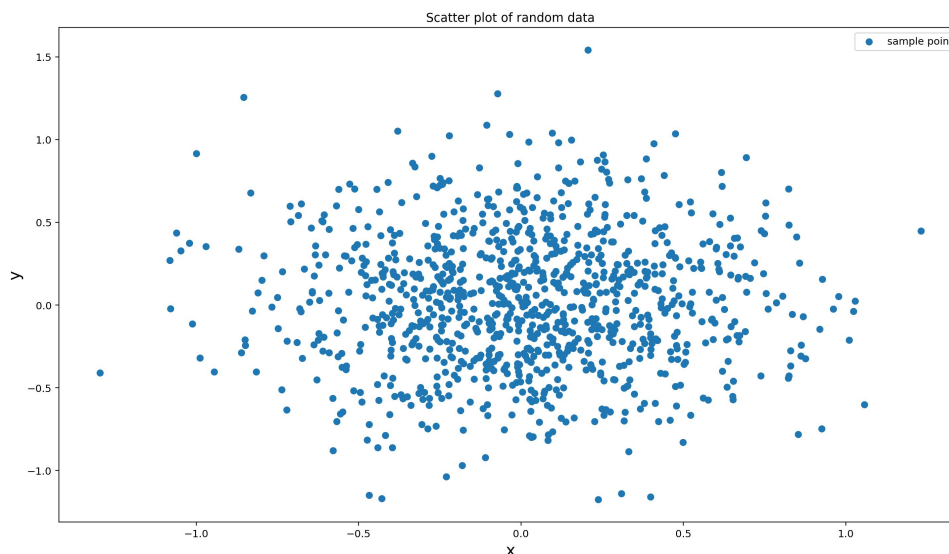


Figure 4.14: Data points created for visualising DBSCAN using `make_blobs` function

Then a `dbscan` object was instantiated using the scikit-learn implementation of DBSCAN with an epsilon of 0.2 and minimum samples of 20. Epsilon is the parameter for choosing the distance between two samples for one to be considered as in the neighbourhood of the other, while minimum samples are the number of samples in a neighbourhood to be considered a core point. A point is a core point if there are at `[number-of-samples]` points in its surrounding area with radius epsilon.

```
x, y = make_blobs(n_samples=1000, centers=1, cluster_std=.4,
                  center_box=(0,0))

dbscan = DBSCAN(eps = 0.2, min_samples = 20)

predictions = dbscan.fit_predict(x)
anomaly_indices = np.where(predictions == -1)
anomaly_values = x[anomaly_indices]
```


Listing 4.28: creating blobs

After running the `fit_predict` function provided by DBSCAN on the generated test data, it classified the data points into two categories, 0 for data points that were part of a cluster and -1 for anomalies. The index of features with -1 was plotted with red colour as shown in 4.15.

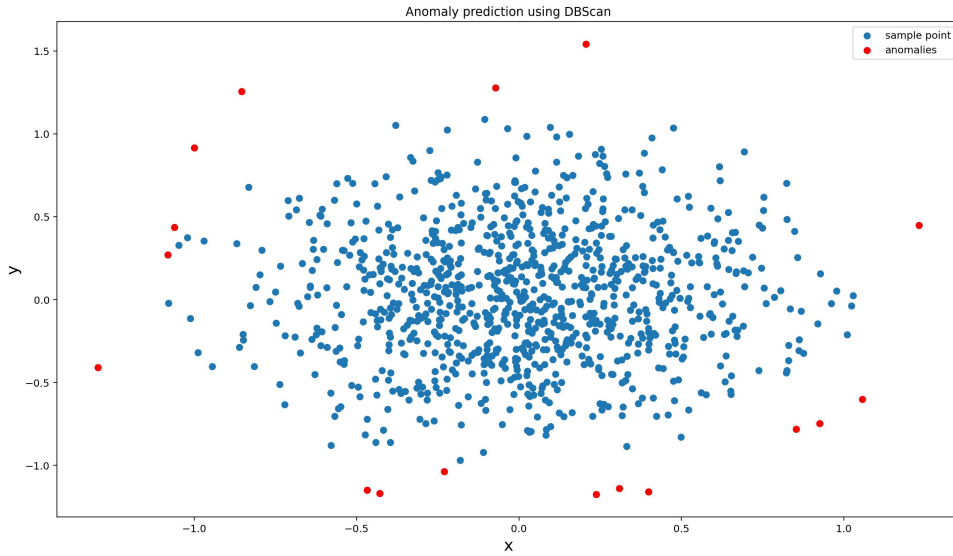


Figure 4.15: Predicted anomalies (in red) by DBSCAN for figure 4.14

This example shows that DBSCAN can be used for anomaly detection. To fit the model on normal data, the number of features of the dataset had to be reduced. This is because it would be easier to visualise the outliers on a 2D or 3D plot and easier for the algorithm to compute the outliers. For dimensionality reduction, TruncatedSVD was used. This transformer performs linear dimensionality reduction and is efficient with sparse matrices. Many features in the dataset were categorical and were one hot encoded; hence, SVD was seen as a better algorithm. Hence, TruncatedSVD, an implementation of SVD by scikit-learn, was used for feature reduction.

For the first iteration, two output features were chosen. The DBSCAN was implemented on the normal training dataset to detect outliers. The normal dataset consisted of 123K samples. It could find 16 anomalies and was able to categorise everything into a cluster as shown in figure 4.16. The total time taken for SVD and clustering was less than 10 seconds.

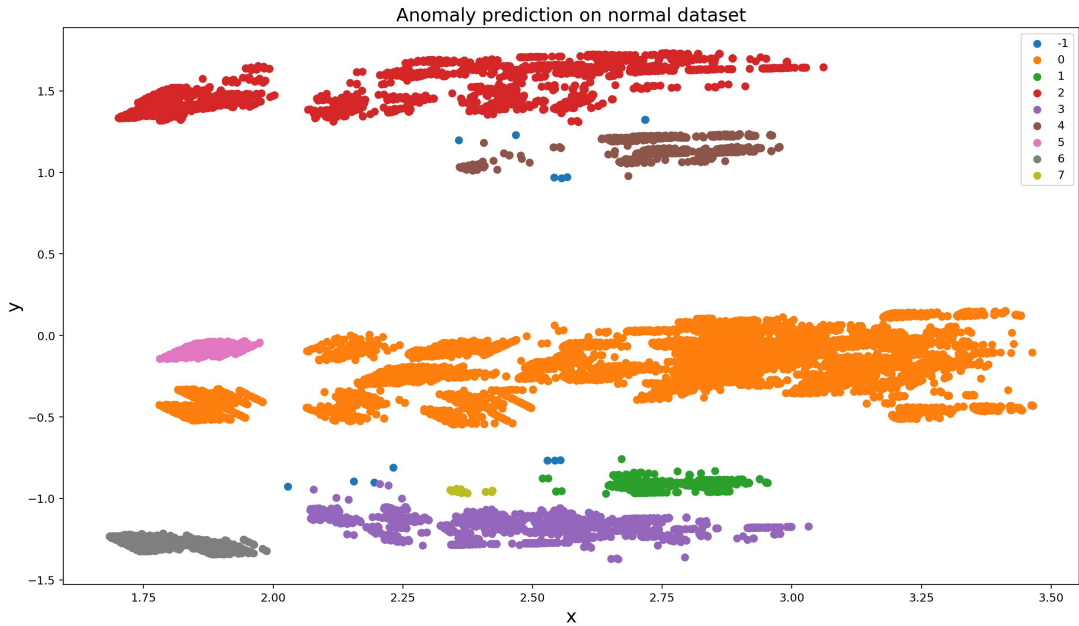


Figure 4.16: Output of DBSCAN on normal data flow. Blue data points (-1) are predicted anomalies in this dataset

For anomaly prediction on the attack dataset, which consisted of about 1 million samples, TruncatedSVD was completed in less than 5 seconds. However, even after 1 hour, the DBSCAN fit_prediction was not complete. It could be evaluated that this model does not scale well; hence a clustering-based approach was abandoned as this approach would not be able to deal with the high volume of traffic flow.

4.4.4 Anomaly detection using Isolation Forest

4.4.4.1 Anomaly detection using packet flow

After the first few iterations of autoencoders were not able to give a high score to anomalous packets, this approach was considered. Here, the features of the packets were not considered; only the number of packets flowing through 'cbr0' per second was calculated. This was done by creating a new pandas series from the timestamp feature of the normal_data data frame and creating a packet column with each row = 1, signifying one packet. The index was then set to the DateTime column, and the df was resampled and summed on it, with the time period equal to 1 second. As the total time of the tcpdump capture was 10 minutes, the total rows in the data frame were 600. The same process was also done for the dataset that contains the anomalous data, which was also captured for 10 minutes as described in 4.4.1.

```

from datetime import datetime
resampled_time = normal_time['timestamp'].
    apply(lambda x:datetime.fromtimestamp(x))
resampled_time_series = pd.to_datetime(resampled_time,
    format='%Y-%m-%d %H:%M:%S')

df = pd.DataFrame(resampled_time_series)
df['packet'] = 1
df = df.set_index('timestamp')
df = df.resample('1S').sum()

```

Listing 4.29: Python code to create packet flow feature using a unix timestamp

Isolation forest, an unsupervised outlier detection technique, was trained on this normal data. For training this dataset, the default values of the IsolationForest model were used, with a change in contamination value. Contamination represents the proportion of outliers in the dataset. As the number of outliers in the training dataset is expected to be really low, this was set to 0.01.

```

from sklearn.ensemble import IsolationForest
model=IsolationForest(n_estimators=100, max_samples='auto',
    contamination=float(0.01), verbose = 0)
model.fit(df.values)

```

Listing 4.30: Creating a model using Isolation Forest implementation in sklearn

After training the model on normal data, a prediction was run on data with anomalous values. As the packet flow during that time was much higher, the model was expected to categorise those points as anomalies, which it successfully does. The normal vs anomalous data flow can be found in figure 4.12 described earlier.

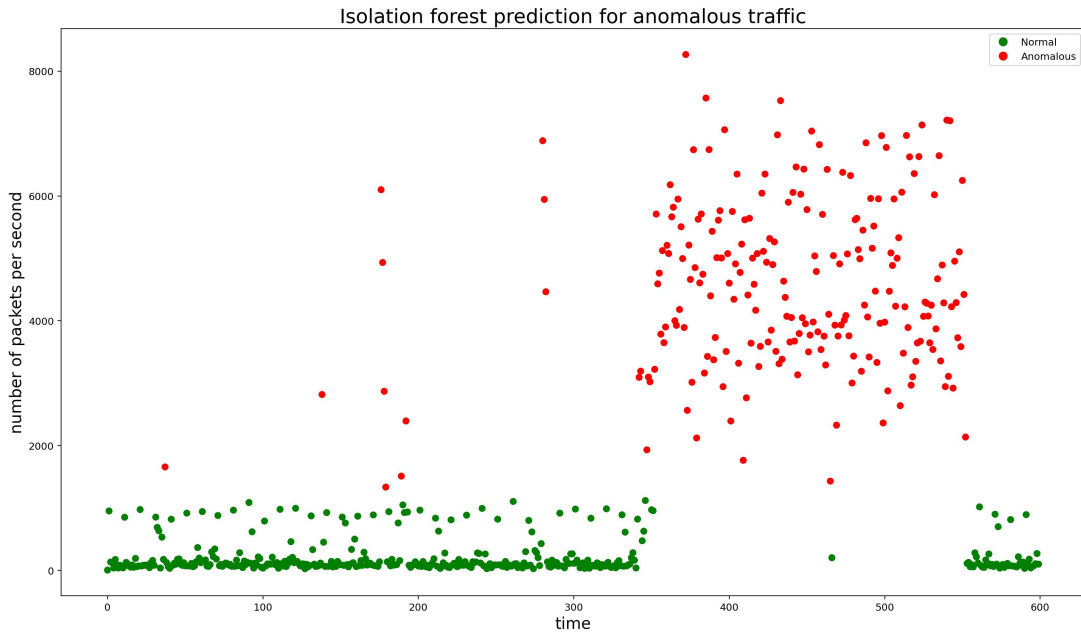


Figure 4.17: Isolation Forest bring used to predict attack packet data flow

As the approach of this project focused more on packet data rather than packet flow, this approach was not further analysed, but it can be seen that predicting anomalies in traffic data can be done and, in practice, is a much better approach. This approach also gave the idea of using packet flow as a feature in autoencoders, which significantly increased the scores of the anomalous data packets, hence improving the model.

4.4.4.2 Anomaly detection using packet data

For this approach⁴, the model was trained on the subset of data as shown in table 4.5. This reason for choosing this subset of features is explained in evaluations of the autoencoder model in section 5.1.3.

Every parameter was set to default values, and this model was trained on the 'normal data flow' dataset and predictions were run on the 'attack data flow' dataset. There were 1,070,554 packets in the dataset, out of which the model predicted that there were 186545 outliers. Interestingly, all the 41824 packets with the destination IP of the attack machine were in these 186545 predictions. To further test the Isolation Forest on this data, another model was created, but this time the contamination was set to 0.04.⁵ Out of the 53639 packets it predicted as outliers, it was seen that all the 41824 packets were again in this dataset. Further evaluation is done in section 5.1.2

⁴This model was implemented few days before the submission, hence it could not be analysed in depth. However, this is an interesting approach hence included in this dissertation

⁵proportion of outliers calculated by approximate value of $41824/1070554$

Packet data type	Feature
General	length,timestamp
IP	ip.src, ip.dst,
Protocol	protocol, protocol.sport, protocol.dport
Source_ip_type	source_pod, source_external
Destination_ip_type	destination_pod, destination_external
Time	packet_flow

Table 4.5: Features of the best autoencoder model

4.4.5 Autoencoders

Autoencoder is a neural network-based approach; hence, to implement an autoencoder model, Keras was used. After the initial data preprocessing, no more preprocessing was required. Various autoencoders with different architectures and input data were implemented throughout the project. Only the model with the best classification of packets from the attacker's IP is shown for this section with features shown in table 4.5 . Evaluation of the most different models used is in the section 5.1.3

4.4.5.1 Training an autoencoder model

After the data preprocessing step, the values from a data frame can be extracted to an array form using its `df` to `train` an `autoencoder model.values` attribute. The shape of the input vector and the output vector of the model was the number of features, i.e. `df.values[1]`. The architecture of an autoencoder model is shown in figure 2.9.

```

model = Sequential()
model.add(Dense(10, input_dim=df.values.shape[1], activation='LeakyReLU'))
model.add(Dense(3, activation='LeakyReLU'))
model.add(Dense(10, activation='LeakyReLU'))
model.add(Dense(df.values.shape[1]))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x_normal_train,x_normal_train,verbose=1,epochs=25)
model.save("autoencoder_model")

```

Listing 4.31: Keras code to create the architecture of Figure 2.9

After initialising the model, it is compiled using the mean squared error loss. Adam optimiser was chosen as it is regarded as one of the best optimisers that work out of the box(44). As the input and output vectors were the same, hence in `model.fit()`, the `x` and `y`

parameters are the same. For this dataset, after 20 to 25 epochs, the loss function usually converged; hence 25 epochs were chosen. After the training, an autoencoder model was saved so that it could be used in the data processing pipeline.

4.4.5.2 Scoring

After a model was trained, it was used to generate predictions of the normal dataset. The distributions of this score over all the packets were then used to generate a baseline.

```
# normal_df -> unprocessed dataframe
# processed_normal_df -> processed dataframe
normal_values = processed_normal_df.values
normal_predictions = model.predict(normal_values)

normal_score_list = []

for index , x in enumerate(normal_predictions):
    normal_score_list.append(np.sqrt(metrics.mean_squared_error
                               (normal_values[index],normal_predictions[index])))
```

Listing 4.32: Predicting scores using model

In this example, a histogram of the scores is generated using the normal scores list. This list was also added as a column to the training data frame, further used to generate statistics for the scores. These scores can help in describing a baseline for the model. In this example, the 99.9th quantile is 0.90. So any packet with a score of more than that could be possibly classified as anomalous.

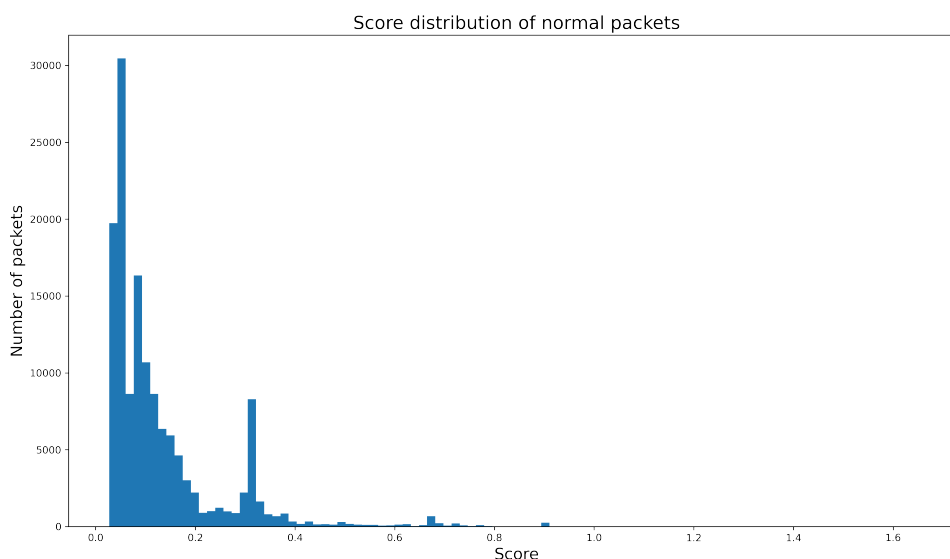


Figure 4.18: Score distribution of the packets in training data based on scores generated by the autoencoder model

```

>>> normal_df['score'] = normal_score_list
>>> normal_df['score'].describe().apply(lambda x: format(x, 'f'))
count      140538.000000
mean         0.130965
std          0.124563
min          0.027543
25%         0.049564
50%         0.087578
75%         0.155795
max          1.662013

>>>normal_df['score'].quantile(0.995)
0.90918444430714451

>>>attack_df['score'].quantile(0.932)
>>>1.0046321075064268

```

Listing 4.33: Scores for normal data

Similarly, scores for the dataset containing the attack can be tested. As this is a prediction, the mean of the predictions is slightly higher, so the distribution was shifted towards the right. Some packets were scored as high as 6, but 93.2% of all the scores were less than 1.004. Hence, the baseline of 1 from the initial dataset was a good indicator.

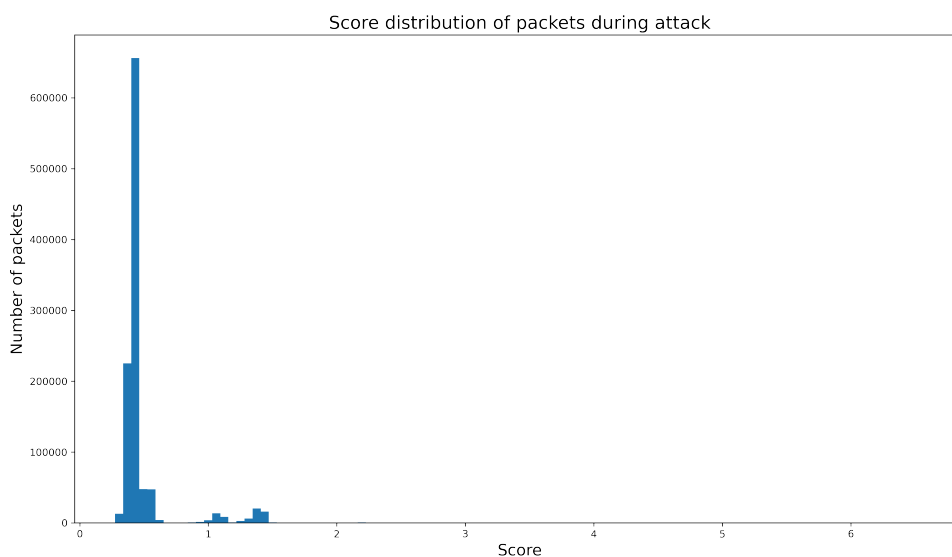


Figure 4.19: Score distribution of packets during attack based on scores generated by the autoencoder

After this, the scores of the attacker’s IP were isolated. If the model could predict a high enough score distribution for these packets, that model was classified as a ”model of interest”; if not, it was discarded. For this model, the mean was 1.38, and no packet was classified as less than the initial baseline defined of 1.

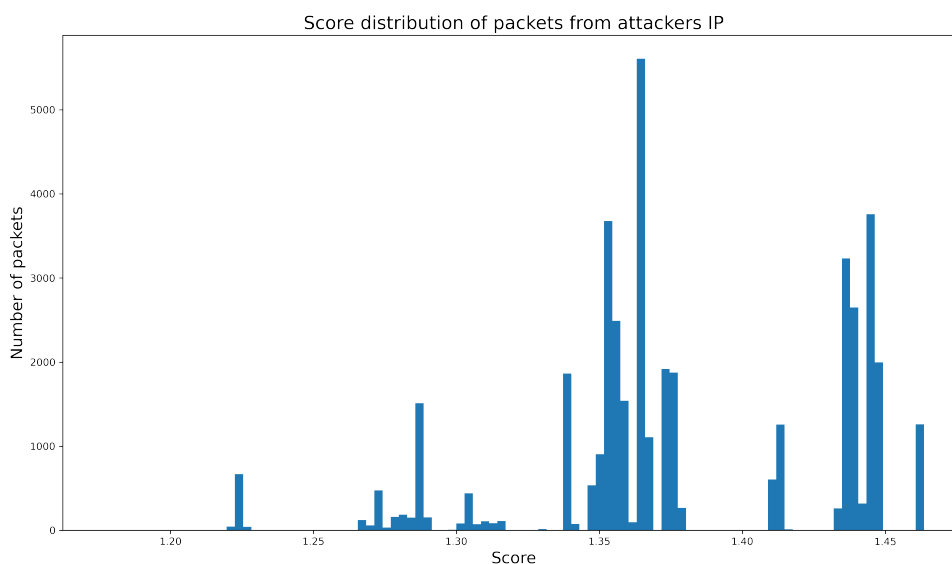


Figure 4.20: Score distribution of packets during the attack with attacker’s IP, based on scores generated by the autoencoder

```
>>>attack_machine_packets=attack_df[attack_df["ip.dst"]
== '<ip-of-attackers-machine>']
```



```
>>>attack_df['score'].describe().apply(lambda x: format(x, 'f'))
count      41824.000000
mean        1.380329
std         0.053284
min         1.176635
25%         1.354040
50%         1.365800
75%         1.437380
max         1.463493
```

Listing 4.34: Keras code to create the architecture of Figure 2.9

4.4.6 Supervised Learning

A supervised learning approach was also implemented to check if that is viable for classifying attack vectors. To create the training dataset, twenty thousand samples from the data frame with the attacker's IP were taken and merged with the normal df. The normal df was given a class of 0, while packets with the attacker's IP were given a class of 1.

After preprocessing the data frame, they were split using StratifiedShuffleSplit class in sklearn, which helps maintain the proportion of samples in the training and testing dataset. If there were 160K samples(140K normal + 20K malicious) and the proportion of test size was 0.2, then the training dataset would consist of 112K of normal data with 16K of malicious data.

All the data features except raw load from the table 4.4 were included in this approach. This prediction was also made using a neural net as well.

```
model = Sequential()
model.add(Dense(30, input_dim=train_set.shape[1], activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])

model.fit(train_set, labels, epochs=10, batch_size=64)
```

Listing 4.35: Keras code to create the architecture of Fi

The model was easy to train but had an unusually high accuracy of 99.8% just after training on ten epochs. Further evaluation of the model is discussed in section 5.1.4.

Chapter 5

Evaluation

5.1 Evaluating machine learning models

5.1.1 Clustering using DBSCAN

5.1.1.1 Dimensionality reduction

After preprocessing the data, there were 72 features present in the dataset. For visualizing the clusters on a 2D/3D plane, the dimensions of this dataset had to be reduced to 2/3, respectively. Four algorithms from the `sklearn.decomposition` were selected and tested on the 'Attack data flow' dataset to choose a dimensionality reduction algorithm. The four algorithms chosen were:

- Principal Component Analysis (PCA)
- Truncated singular value decomposition (TruncatedSVD)
- Fast Independent Component Analysis (ICA)
- Kernel PCA (with RBF kernel)

This test was to check which algorithm can reduce dimensions of the dataset from 72 to 3 fastest ¹. Every algorithm had to reduce the first 50,100,200,500, 1000, 5000, 10000, and 20000 data samples in the dataset. The graph for the comparison is given in figure 5.1

¹Comparison done on 2021 Apple M1 Pro MacbookPro with ARM architecture 10 Core CPU @ 3.2GHz

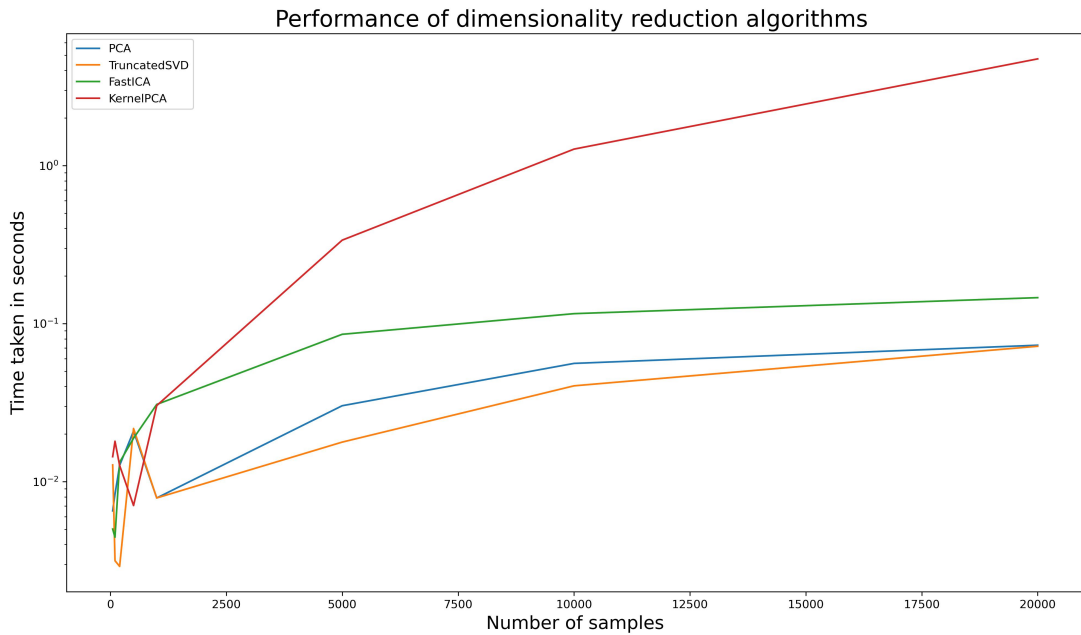


Figure 5.1: Performance comparison of various dimensionality reduction algorithms

It could be seen that for a small dataset, i.e. under 1000 samples, all algorithms can reduce the size in 10^{-1} range. However, as the dataset size increased, it could be seen that the TruncatedSVD algorithm was the fastest. Even for a million samples with more than 72 features, it took the algorithm 2.43 seconds. It benefited from being efficient with sparse datasets, which many columns of the 'Attack data flow' datasets were. Hence TruncatedSVD algorithm was chosen for this project.

5.1.1.2 Evaluating DBSCAN

After reducing the dataset's features using TruncatedSVD, DBSCAN was used for unsupervised clustering on the dataset to detect anomalies. Although clustering can be used to detect anomalies in an unsupervised fashion as described in (45) and as shown in figures 4.14 and 4.15, for this project, this approach was not ideal. For a dataset where the number of clusters was already known, anomaly detection might be more accurate and a more viable approach. However, analysing clusters is hard for this approach and is more time and memory intensive with respect to this project. Also, it is possible that the clusters would be different for each dataset; hence this approach was not a viable option here.

5.1.2 Isolation Forest

Isolation Forest was used on packet flow and packet data to find anomalies. A model was trained on normal data flow, and predictions were run on attack data flow. As the number of packets during the attack increased significantly, Isolation Forest could find the anomalies easily, as seen in figure 4.17. One thing to note is that this analysis was done on 10 minutes of data. In real-world environments, the data flow changes significantly; for example, a website might have heavy traffic from 9 AM to 6 PM but little to no traffic from 12 AM to 6 AM. So, a model needs to be trained to learn this pattern of behaviour. Clustering techniques can also be used to detect anomalous behaviour in packet flow, as seen in (39). Nevertheless, this gave the idea that packet flow is an important feature to consider; this feature was part of the best autoencoder model.

When Isolation Forest was used to predict anomalies using packet data, preliminary results show that it performs exceptionally well, having fewer false positives than the best autoencoder model. It was seen that when the contamination factor was set to 'auto', it predicted a lot more packets as anomalous. However, when the correct contamination value was given, it identified all the packets with the attacker's IP as anomalous. Perhaps, this might be because it flagged every new IP seen as anomalous. Further analysis needs to be done.

5.1.3 Autoencoders

For this project, more than 12 autoencoder models were tested. The critical difference in all these iterations is the features selected, as shown in table 5.1. The last row, 'TruncatedSVD', represents if the features were reduced using the algorithm. The key to evaluating these models is the baseline created from the distribution of regular packets and how far the score distribution of packets with the attacker's IP is with respect to the rest of the 'attack data flow'.

To evaluate these models, a few parameters were chosen. As these scores were on distribution, it was tricky to define a baseline. After evaluating the data, it was seen that a score at 99.5% percentile of the Normal Packet Data Flow (NPDF) would be an ideal choice. A high percentile was chosen because little to no packets in the NPDF data frame were anomalous, but some scores generated by the model were anomalously high, so this score disregards the top 0.5% scores.

The goal of the autoencoder model was to create a high enough score for packets with the destination IP of the attacker while keeping most of the packets below the baseline created. Hence, The mean and standard deviation of the scores from Packets During Attack (PDA) was taken as a metric. Scores of the Packets with the Attacker's

Features	Model 1	Model 2	Model 3	Model 4	Model 5
Packet length	✓	✓	✓	✓	✓
Ethernet source	✓	✓			
Ethernet destination	✓	✓			
Ethernet type		✓			
IP source	✓	✓		✓	✓
IP destination	✓	✓		✓	✓
IP version		✓			
IP protocol		✓	✓		
IP length	✓	✓	✓		
IP ihl	✓	✓	✓		
IP tos	✓	✓	✓		
IP ttl		✓	✓		
Protocol	✓	✓	✓	✓	✓
Protocol source port	✓	✓	✓	✓	✓
Protocol destination port	✓	✓	✓	✓	✓
Flags		✓	✓		
Origin of IP.src		✓	✓		✓
Origin of IP.dst		✓	✓		✓
Load: Number of bytes	✓	✓	✓		
Load: First 20 bytes		✓	✓		
Packet flow					✓
TruncatedSVD		✓	✓		

Table 5.1: Features selected for different models

IP (PAIP) were also isolated, and their mean was recorded. Also, the percentile of the mean of PAIP was recorded. This was done to see how many packets in PDA were below the majority of PAIP.

A good model for anomaly detection would have the mean of scores of PDA much below the baseline, while the mean of PAIP would be much higher than the baseline. Also, if the percentile of the mean of PAIP was above the 96th percentile, that would be ideal as there are 4% of the packets in PDA are anomalous. A table summarizing this analysis can be seen in table 5.2.

5.1.3.1 Model 1

For the first model, the features that were different in each of the packets and the total number of bytes in the packet data were chosen. A model with 3 hidden layers was chosen, the first, second and third layers having 10, 3 and 10 neurons, respectively. After training the model on normal packet data, this model was able to differentiate the attacks' IP, placing the mean in the 95th percentile of scores predicted. This was a good start, but

	Model 1	Model 2	Model 3	Model 4	Model 5
Baseline(99.5 percentile of NPDF)	3.268	1.3562	0.3	0.16	0.72
Mean of scores (PDA)	1.256	1.167	0.21	0.017	0.48
Std. deviation of scored (PDA)	0.647	0.607	0.12	0.013	0.23
Mean of scores (PAIP)	1.984	1.83	0.31	0.05	1.38
Percentile of mean of scores (PAIP)	95th	91st	77th	96th	98th

Table 5.2: Summary of evaluation of autoencoder models

this model did not include the packet's flags data and the packet's load.

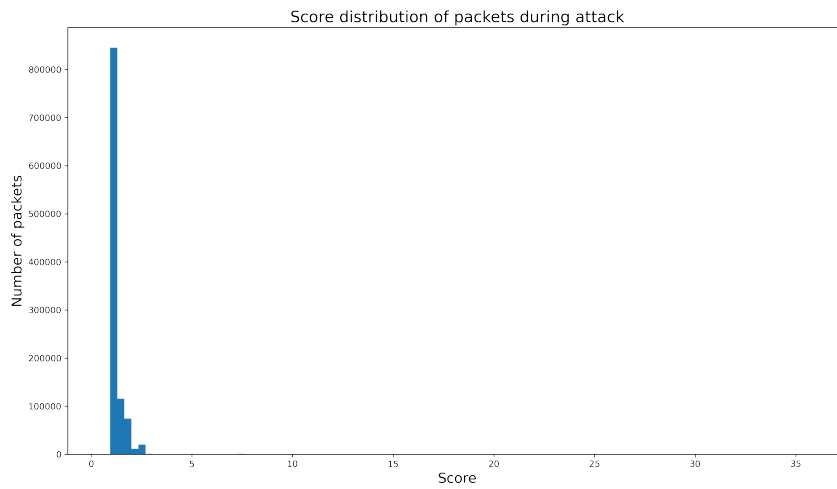


Figure 5.2: Score distribution of packets during attack based on scores generated by autoencoder-model 1

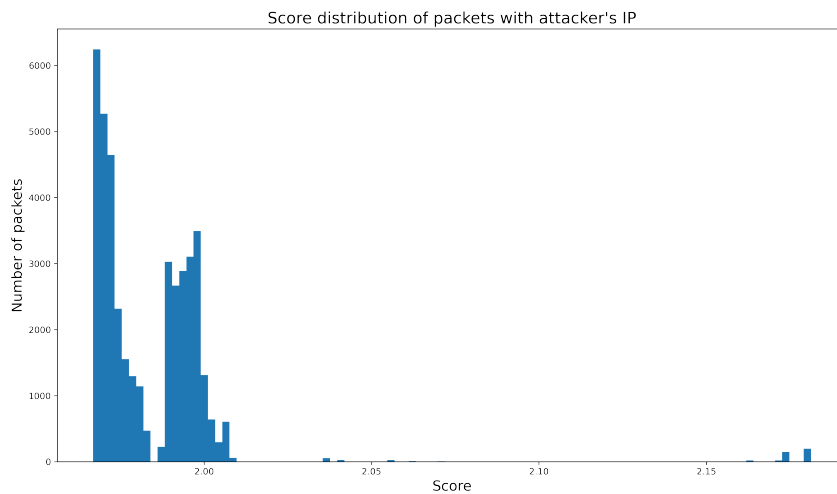


Figure 5.3: Score distribution of packets during the attack with attacker's IP, based on scores generated by autoencoder-model 1

5.1.3.2 Model 2

After including all the possible features except packet flow, as it had not been analyzed yet, there were a total of 72 features. Having 72 features made the training of the model and processing speed of generating scores significantly slow; hence the number of features had to be reduced. For feature reduction, TruncatedSVD was used because of its high performance, as shown in figure 5.1. After reducing the number of features to 30, the model was trained, and predictions were made on attack data. Here, the model performs worse than Model 1. This might be because of the choice of dimensionality reduction algorithm and the introduction of many features that did not suit the model.

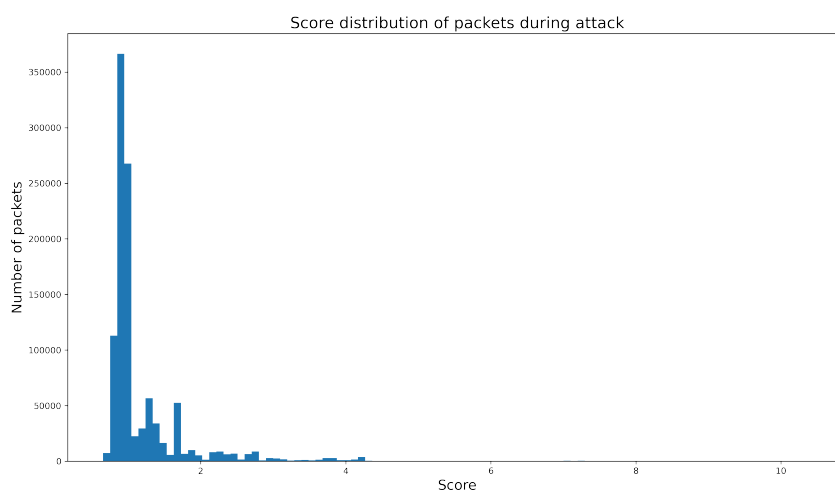


Figure 5.4: Score distribution of packets during attack based on scores generated by autoencoder-model 2

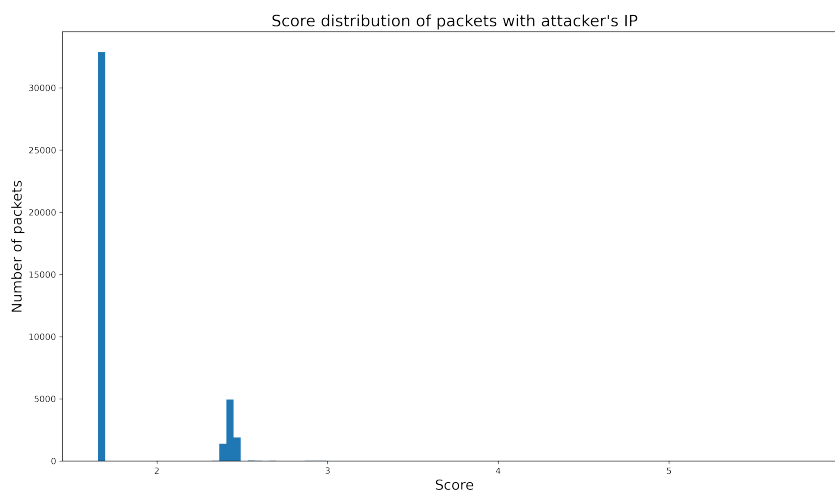


Figure 5.5: Score distribution of packets during the attack with attacker's IP, based on scores generated by autoencoder-model 2

5.1.3.3 Model 3

In the next iteration, to test the significance of IP and Ethernet features of the packets, they were removed, and the model was just trained on the rest of the packet features. The total number of features was 49; hence Truncated SVD was used again for feature reduction. Here, the features were also normalized between 0 and 1 instead of using Standard Scaler, hence the lower overall scores of the model. After training the model, it was seen that it performed significantly worse than the previous iterations, placing the mean scores of the packet in the 72nd percentile.

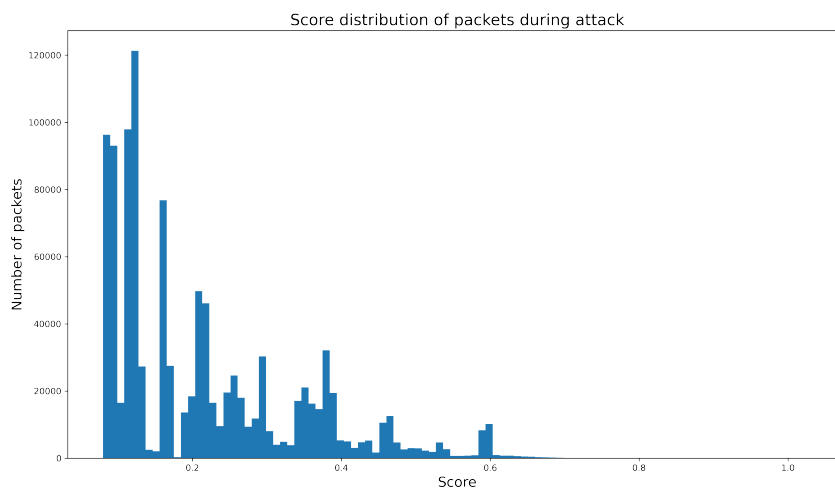


Figure 5.6: Score distribution of packets during attack based on scores generated by autoencoder-model 3

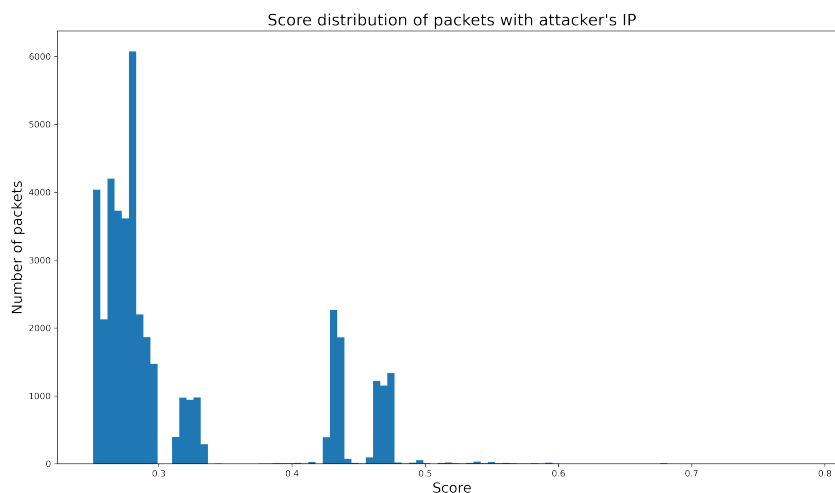


Figure 5.7: Score distribution of packets during the attack with attacker's IP, based on scores generated by autoencoder-model 3

5.1.3.4 Model 4

Not seeing any increase in models performance with many features or changing the normalization technique, a model was created on the most basic features: IP source, IP destination, type of protocol, ports, and packet length. Interestingly, this model performed significantly better than all the other iterations before it, placing the mean of PAIP in the 96th percentile. It was observed that having fewer features is much better.

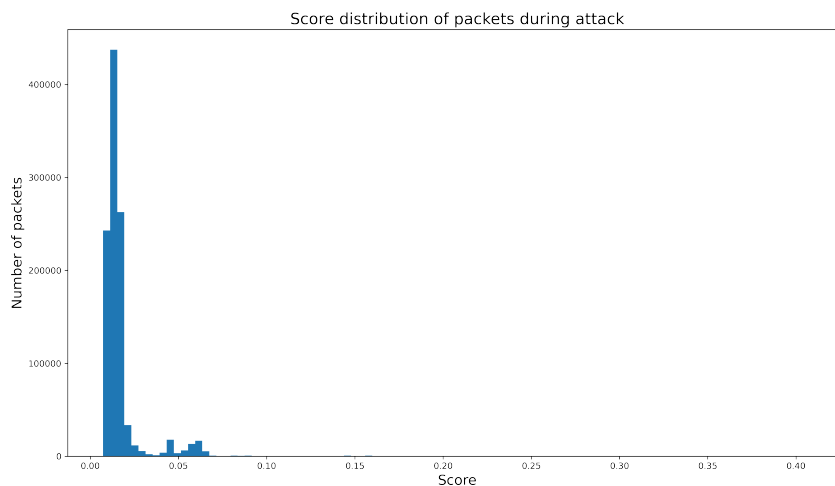


Figure 5.8: Score distribution of packets during attack based on scores generated by autoencoder-model 4

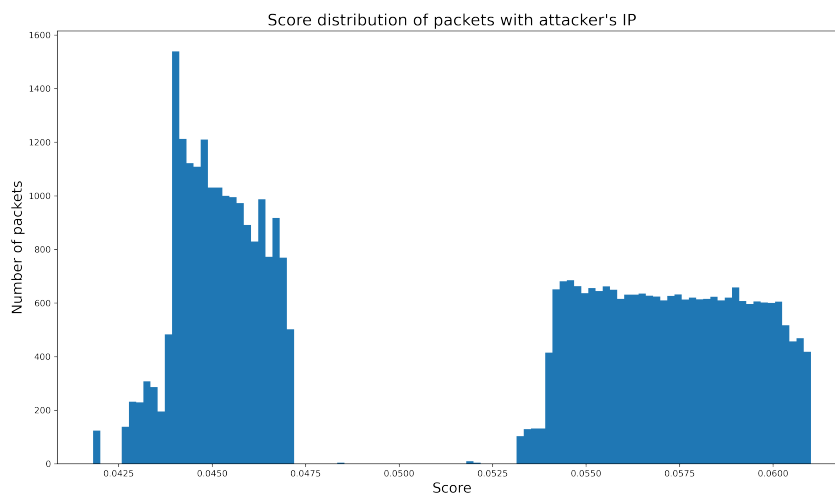


Figure 5.9: Score distribution of packets during the attack with attacker's IP, based on scores generated by autoencoder-model 4

5.1.3.5 Model 5

After creating the model in 5.1.3.4, features were added one by one to create the best possible model for the given data. The origin of IP.src/dst as a feature and introducing packet flow gave the best results. It had 18 features; hence no feature reduction was required. This model gave sufficiently higher scores of PAIP, placing the mean of the packets in the 98th percentile of PDA.

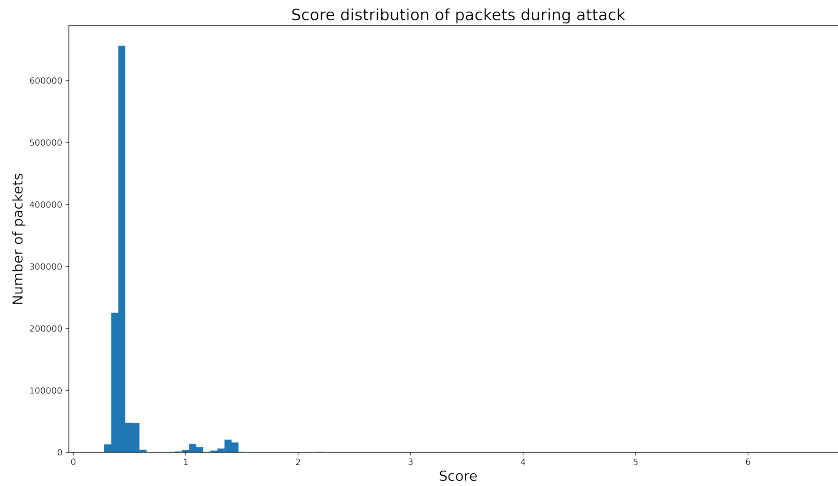


Figure 5.10: Score distribution of packets during attack based on scores generated by autoencoder-model 5

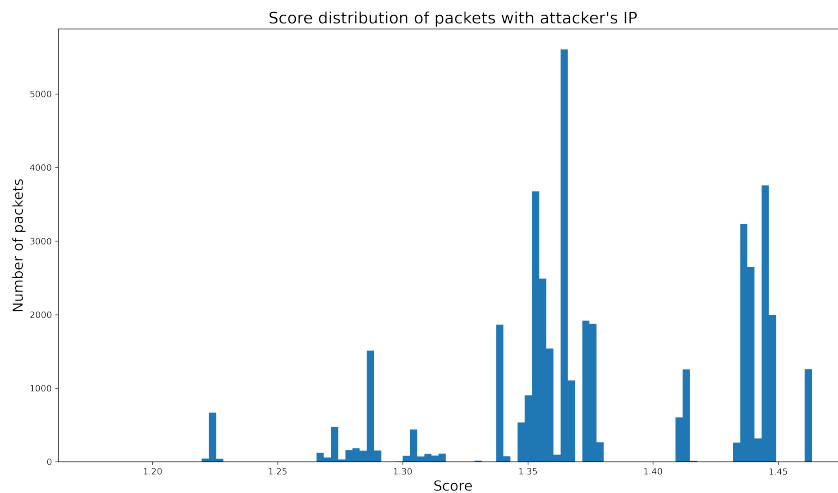


Figure 5.11: Score distribution of packets during the attack with attacker's IP, based on scores generated by autoencoder-model 5

5.1.3.6 Limitations of autoencoders

As with any other unsupervised learning technique, this approach produces a lot of false positives. Even the best model(section 5.1.3.5) gave a score of above 0.72 (the selected baseline) to 77007 packets. That makes it 35883 false positives.² During testing, it was also noted that it gave a high score to every new IP it saw. There were many requests with destination IP of Azure Datacentre located in Netherlands³ which had a high score. To verify this, it was seen that only 66 packets with 'source_internal' == 1 and 'destination_internal' == 1 had a score greater than the baseline. The rest of the packets had at least one external IP as source or destination.

That is the major limitation of using autoencoders for NIDS. It needs to be trained and retrained in an online fashion on a lot of data for creating a good baseline so that IP values from expected data sources like Azure do not get a high score. Using just an autoencoder model for a NIDS is not ideal. However, this model, in combination with others like Isolation Forest or a supervised model trained on attack dataset, might give much better accuracy in detecting intrusion.

5.1.4 Supervised learning

After training the dataset and evaluating it on the dataset containing the attack, the model had an unusually high accuracy of 99.8 on the test dataset%. This was very suspicious, as the model was able to get this accuracy in less than ten epochs. Labels for the PDA were then predicted using the model, and it had an accuracy of 96.25%. This was interesting, as only the 4% of the packets were actually anomalous, so even if the model predicted every packet as normal, it would still have an accuracy of 96%. When precision and recall of the model were calculated using the confusion matrix in 5.12, it was seen that the model had a precision of 74% and recall of just 6%.

This model might show improvements if the number of features was reduced, as seen in section 5.1.3.5. Due to the project's time constraints and as the focus was on using unsupervised learning to classify anomalous packets, this approach was not evaluated further.

²Another thing to note was that the external IP of my local machine was very close to the IP of the Kali Linux machine as they were on the same network. This also might have made making good predictions harder for the model.

³Some IPs include 40.113.176.128, 13.69.106.212, 13.69.106.208

	Predicted		
		Negative	Positive
Actual	Negative	1027829	901
	Positive	39280	2644

Figure 5.12: Confusion matrix for the supervised learning implementation. There were 39280 packets actually positive predicted as negative, which brings the recall to just 6%

5.2 Evaluating the architecture

5.2.1 Limitations of the prototype

5.2.1.1 Tcpcap container

Initially, for sending packets sniffed by the tcpcap, a message streaming service like Kafka was considered⁴. However, this approach was not thought to be ideal then, as it would generate a lot more network traffic. A possible way to mitigate that is to capture only the TCP/UDP packets via tcpcap/tshark filters. Due to the time constraints of the project and unfamiliarity with configuring Kafka in a Kubernetes cluster, this was not implemented.

5.2.1.2 Pcap service

The pcap service receives the pcap files and converts them into a CSV. Scapy, the python package used for reading and analysing the data from pcap, is relatively slow (see figure 5.13) and memory intensive compared to other packet analysers. It was realised that `rdpcap`, the method used to read the pcap files, first stores the file in memory. This can be prevented by using the `sniff` command in offline mode.

```
def method_filter_HTTP(pkt):
    #Your processing

sniff(offline="your_file.pcap",prn=method_filter_HTTP,store=0)

#https://stackoverflow.com/questions/10800380/scapy-and-rdpcap-function
```

Listing 5.1: A better way to parse packets using Scapy

⁴This can be implemented, as shown in a tutorial by Robin Moffatt

A better solution might be to use Libtins, a C++ library designed with packet sniffing efficiency in mind. Dpkt, a Python package, can also be used as it is faster than scapy. It was not used for this project as scapy had better documentation and features.

Library	Time taken(seconds)	Packets per second
libpcap	0.141	3546099
libtins	0.273	1831501
pcapplusplus	0.38	1315789
dpkt	8.132	61485
libcrafter	12.209	40953
impacket	18.5	27027
scapy	187.082	2672

Figure 5.13: The time taken for a library to parse 500000 packets from a pcap file(46). Source: Libtins benchmark

5.2.1.3 Model service

The model service receives the CSV file generated, preprocesses it for machine Learning and generates scores for the packets. Managing data frames within this cluster can be optimised to reduce memory. Also, a separate node inside this cluster that uses GPU acceleration can be used to further increase the speed of scoring packets.

5.2.1.4 MySQL database

It was sufficient to have a MySQL database inside the cluster for a prototype. However, as the number of services in a cluster increases, the tcpdump data would increase exponentially and having a managed SQL DB is a better solution.

5.2.1.5 Backend service

The backend service currently has limited functionality because of the project's time constraints. A better UI that shows the packet data flow and a dashboard that can filter out packets based on IP Addresses and automatically create a list of suspicious IPs can be created.

5.2.2 Improved architecture for the prototype

This prototype currently works for a single node. Although this solution can be scaled to multiple nodes, given the memory-intensive tasks, the cluster would have issues deal-

ing with a random increase in packet flow. When the password dictionary attack was launched, there were more than 1 million packets collected in less than 10 minutes. Even when scaled up with multiple replicas of each service, the current architecture could not handle the increased load, and the pods failed, resulting in data loss.

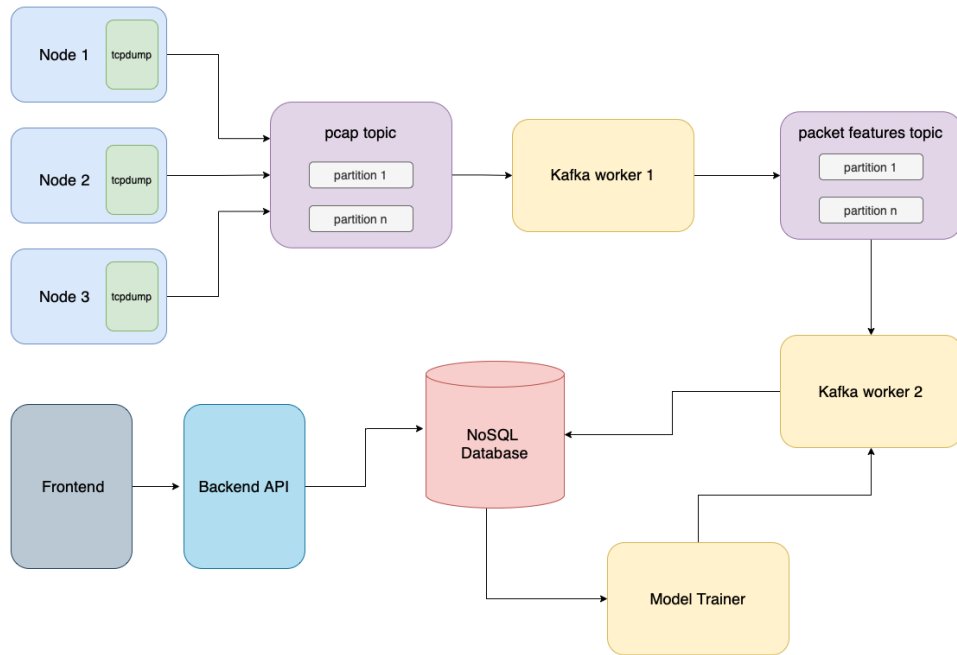


Figure 5.14: High-level architecture for a prototype of a better data processing pipeline

A new architecture is proposed to mitigate this that would handle the load in a much better fashion and is more scalable. Every node can have a `tcpdump` pod running on it, which streams the data to a Kafka topic 'pcap topic'. The pcap topic can have multiple partitions to store data coming in from various streams. This data can be consumed by the `Kafka Worker 1`, which would convert the data captured into the required format. After conversion, this data can then be sent to the `packet features topic`, which the `Kafka worker 2` would consume. This worker can aggregate the data based on either the number of packets or time, and run predictions on the data using one ML model or multiple models with weighted voting.

These packets and their generated scores can then be stored in a distributed NoSQL datastore. NoSQL provides flexibility in defining the structure of the data stored, and if a new feature is added/removed in the data processing pipeline, the database would be easily able to handle it. On this datastore, a job can be run to aggregate data for a day and send it to a Model Trainer VM (not shown in the figure), which has access to GPUs that can train different machine learning models in an online fashion and replace the ones

in `Kafka worker 2` on a periodic schedule. This can be done as the baseline behaviour of an application can change from week to week, and using this setup, it might be able to update the baseline behaviour.

Another aggregation job can be run on the database every K number of minutes, which updates a statistics table that stores the most commonly used stats. A frontend service that a network admin or researcher can see helpful information on can be created, which interacts with the database via the backend API. This can consist of multiple services depending upon the team's needs.

Chapter 6

Conclusions & Future Work

6.1 Conclusion

The primary objective of this research project was to investigate the performance of various unsupervised machine learning algorithms that were trained with network traffic from Kubernetes deployments and to evaluate the trained models against real-world attacks like port scans and dictionary attacks. Anomaly-based ML algorithms like DBSCAN, Isolation Forest and autoencoders were tested for this approach. It was seen that autoencoders could be trained to detect anomalies using various packet features. It was also noticed that packet flow is an essential feature and can be used in conjunction with packet data to increase a models' performance for detecting anomalies.

It was observed that Isolation Forest performed extremely well as an anomaly detection algorithm. Trained on a normal packet flow, it easily identified the data flow as anomalous during the attack. This algorithm was also trained on features of normal packets. Interestingly, if it was modelled with the correct parameter for expected anomalies in the dataset, it outperformed the autoencoder by classifying every packet with the attacker's IP as anomalous, giving 67% fewer false positives than the best autoencoder model.

However, both Isolation Forest and autoencoders suffer from a high number of false positives; 22% of Isolation Forest predictions and 46% of autoencoder predictions were false positives. A common attribute of the Isolation Forest and autoencoder technique was that they classified every new IP that was not in their training dataset as an anomaly.

This hypothesis was tested by deploying a WordPress application and an autoencoder-based NIDS using Azure Kubernetes Services. Various nmap scans and a password dictionary attacks were launched against the WordPress application, and although the NIDS gave high scores to the packets with the attacker's IP, a much higher score was given to

IPs from the Azure datacentre¹. Hence, it is evident that a NIDS based on unsupervised anomaly detection would generate a high number of false positives, leading to alert fatigue if each predicted anomalous packet generates an alert.

DBSCAN, a density-based clustering algorithm, was also analysed for this project. As DBSCAN is much faster for low dimensional datasets, the dataset features were reduced to 2 using SVD, but still, it was seen that it was far too slow to use in a NIDS. Even after an hour, the DBScan could not complete classifying 1 million data instances into clusters. Hence this approach was not further analysed.

6.2 Future work

The future work for this research can be divided into two components, the first being the prototype of NIDS as explained in 6.2.1 and the second being the machine learning approaches described in the section 6.2.2.

6.2.1 NIDS prototype

Currently, the NIDS prototype revolves around sending pcap and CSV files from one service to another. This is very inefficient and not scalable. To solve this, a better NIDS architecture is proposed in section 5.2.2 that uses Kafka message streams and a pub-sub architecture to send data from one microservice to another. Ironically, the current prototype and proposed architecture both suffer from a severe security risk, i.e. privileged pods(see appendix .5). If an attacker gains access to this pod, it has access to the nodes' resources and kernel capabilities(47). To mitigate this, one possible solution is to use network logging services provided by cloud providers; for example, Google's Kubernetes Engine provides network policy logging(48). However, this approach would reduce the number of features captured when compared to the features captured by a tcpdump.

6.2.2 Machine learning

As stated in section 6.1, the number of false positives in an anomaly-based NIDS is very high, which might lead to alert fatigue. A possible solution to this is whitelisting/removing the Azure IPs before being fed into the model and training the model in an online fashion, enabling it to adapt to new data. Another solution is to use multiple machine learning models with weighted voting to predict if a packet is anomalous. In section 5.1.3, it was evaluated that having many features decreases the accuracy of an autoencoder model.

¹The baseline score of the model was around 1, the packets from the attacker's IP for a score of around 1.6 to 1.8 while Azure datacentre IPs got a higher score of around 3-4 consistently

Hence, one model can be trained on just packet load, another on packet features and combining this with a packet flow would possibly reduce the number of false positives(49). This approach also might prevent against "tunneling", i.e. placing the data of a packet of one protocol into the payload of another packet.

6.3 Reflection

This research project was a great learning experience for me. I got the opportunity to learn and deploy applications using containers, both locally on minikube and Azure. I also learned about the basics of network security and how cybercriminals can exploit various vulnerabilities in a website. This research project was my first end to end machine learning project as well, and deploying a prototype of a NIDS gave me an idea of how ML models can be used in real-world applications.

Bibliography

- [1] JOHNSON, Joseph: *Number of internet users worldwide 2005-2021* — Statista. <https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/>. Version: 2021
- [2] TUNG, Liam: *Hackers target kubernetes to steal data and processing power. now the NSA has tips to protect yourself.* <https://www.zdnet.com/article/hacker-target-kubernetes-to-steal-data-and-processing-power-now-the-nsa-has-tips-to-protect-yourself/>. Version: Aug 2021
- [3] TEAM, RedLock C.: *Lessons from the cryptojacking attack at Tesla.* <https://redlock.io/blog/cryptojacking-tesla>. Version: Aug 2018
- [4] GREIG, Jonathan: *Researchers find new attack vector against Kubernetes Clusters via misconfigured Argo workflows instances.* <https://www.zdnet.com/article/researchers-find-new-attack-vector-against-kubernetes-clusters-via-misconfigured-argo-workflows/>. Version: Jul 2021
- [5] OSNAT, Rani: *A brief history of containers: From the 1970s till now.* <https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>. Version: Jan 2020
- [6] *Welcome to linux-vserver.org.* http://linux-vserver.org/Welcome_to_Linux-VServer.org
- [7] CORBET: *Process containers.* <https://lwn.net/Articles/236038/>. Version: May 2007
- [8] WILLIAMS, Alex: *Docker donates container format and runtime code, joins with coreos to form standards group.* <https://thenewstack.io/docker-donates-container-format-and-runtime-code-joins-coreos-to-form-standards-group/>. Version: Feb 2019

- [9] *What is a virtual machine and how does it work: Microsoft azure.* <https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/#overview>
- [10] BUCHANAN, Ian: *Containers vs virtual machines.* <https://www.atlassian.com/continuous-delivery/microservices/containers-vs-vms>
- [11] CLANCY, Molly: *Docker containers vs. VMS: Pros and cons of containers and Virtual Machines.* <https://www.backblaze.com/blog/vm-vs-containers/>. Version: Oct 2021
- [12] *Use containers to Build, Share and Run your applications.* <https://www.docker.com/resources/what-container/>. Version: Mar 2022
- [13] *What is a container?* <https://azure.microsoft.com/en-us/overview/what-is-a-container/>
- [14] JONES, Doug: *Containers vs. Virtual Machines (VMS): What's the difference?* <https://www.netapp.com/blog/containers-vs-vms/>. Version: Mar 2018
- [15] *What is kubernetes?* <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. Version: Jul 2021
- [16] MITEVSKI, Kristijan: *Tracing the path of network traffic in Kubernetes.* <https://learnk8s.io/kubernetes-network-packets>. Version: Jan 2022
- [17] JORDAN, M. I. ; MITCHELL, T. M.: Machine learning: Trends, perspectives, and prospects. In: *Science* 349 (2015), Nr. 6245, 255-260. <http://dx.doi.org/10.1126/science.aaa8415>. – DOI 10.1126/science.aaa8415
- [18] PEREIRA, F. ; NORVIG, P. ; HALEVY, A.: The Unreasonable Effectiveness of Data. In: *IEEE Intelligent Systems* 24 (2009), mar, Nr. 02, S. 8–12. <http://dx.doi.org/10.1109/MIS.2009.36>. – DOI 10.1109/MIS.2009.36. – ISSN 1941–1294
- [19] SARKER, Iqbal H.: Machine learning: Algorithms, real-world applications and research directions. In: *SN Computer Science* 2 (2021), Nr. 3, S. 1–21
- [20] CHANDOLA, Varun ; BANERJEE, Arindam ; KUMAR, Vipin: Anomaly Detection: A Survey. In: *ACM Comput. Surv.* 41 (2009), jul, Nr. 3. <http://dx.doi.org/10.1145/1541880.1541882>. – DOI 10.1145/1541880.1541882. – ISSN 0360–0300
- [21] SCHUBERT, Erich ; SANDER, Jörg ; ESTER, Martin ; KRIEGEL, Hans P. ; XU, Xiaowei: DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. In: *ACM Transactions on Database Systems (TODS)* 42 (2017), Nr. 3, S. 1–21

- [22] SHARMA, Abhishek: *How does DBSCAN clustering work?: DBSCAN clustering for ML.* <https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/>. Version: Oct 2020
- [23] SAKURADA, Mayu ; YAIRI, Takehisa: Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction. In: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. New York, NY, USA : Association for Computing Machinery, 2014 (MLSDA'14). – ISBN 9781450331593, 4–11
- [24] LIU, Fei T. ; TING, Kai M. ; ZHOU, Zhi-Hua: Isolation forest. In: *2008 eighth ieee international conference on data mining IEEE*, 2008, S. 413–422
- [25] MITRE ATT&CK®. <https://attack.mitre.org/>
- [26] MCAFEE: *What is the MITRE ATT&CK framework?* <https://www.mcafee.com/enterprise/en-us/security-awareness/cybersecurity/what-is-mitre-attack-framework.html>
- [27] *Network intrusion detection system: Managed ids.* <https://www.redscan.com/services/managed-intrusion-detection-system/>. Version: Dec 2021
- [28] VACCA, John R.: *Network and system security*. Elsevier, Inc, 2014
- [29] LIAO, Hung-Jen ; RICHARD LIN, Chun-Hung ; LIN, Ying-Chih ; TUNG, Kuang-Yuan: Intrusion detection system: A comprehensive review. In: *Journal of Network and Computer Applications* 36 (2013), Nr. 1, 16-24. <http://dx.doi.org/https://doi.org/10.1016/j.jnca.2012.09.004>. – DOI <https://doi.org/10.1016/j.jnca.2012.09.004>. – ISSN 1084–8045
- [30] REZEK, Michael: *What is the difference between signature-based and behavior-based intrusion detection systems?* <https://accedian.com/blog/what-is-the-difference-between-signature-based-and-behavior-based-ids/>. Version: 2022
- [31] KIENNERT, Christophe ; ISMAIL, Ziad ; DEBAR, Herve ; LENEUTRE, Jean: A Survey on Game-Theoretic Approaches for Intrusion Detection and Response Optimization. In: *ACM Comput. Surv.* 51 (2018), aug, Nr. 5. <http://dx.doi.org/10.1145/3232848>. – DOI 10.1145/3232848. – ISSN 0360–0300
- [32] ANWAR, Shahid ; MOHAMAD ZAIN, Jasni ; ZOLKIPLI, Mohamad F. ; INAYAT, Zakira ; KHAN, Suleman ; ANTHONY, Bokolo ; CHANG, Victor: From Intrusion Detection to


- an Intrusion Response System: Fundamentals, Requirements, and Future Directions. In: *Algorithms* 10 (2017), Nr. 2. <http://dx.doi.org/10.3390/a10020039>. – DOI 10.3390/a10020039. – ISSN 1999–4893
- [33] GARCÍA-TEODORO, P. ; DÍAZ-VERDEJO, J. ; MACIÁ-FERNÁNDEZ, G. ; VÁZQUEZ, E.: Anomaly-based network intrusion detection: Techniques, systems and challenges. In: *Computers Security* 28 (2009), Nr. 1, 18–28. <http://dx.doi.org/https://doi.org/10.1016/j.cose.2008.08.003>. – DOI <https://doi.org/10.1016/j.cose.2008.08.003>. – ISSN 0167–4048
- [34] DENNING, Dorothy ; NEUMANN, Peter G.: *Requirements and model for IDES-a real-time intrusion-detection expert system*. Bd. 8. SRI International Menlo Park, 1985
- [35] YE, Nong ; EMRAN, Syed M. ; CHEN, Qiang ; VILBERT, Sean: Multivariate statistical analysis of audit trails for host-based intrusion detection. In: *IEEE Transactions on computers* 51 (2002), Nr. 7, S. 810–820
- [36] ESTEVEZ-TAPIADOR, Juan M. ; GARCIA-TEODORO, Pedro ; DIAZ-VERDEJO, Jesus E.: Stochastic protocol modeling for anomaly based network intrusion detection. In: *First IEEE International Workshop on Information Assurance, 2003. IWIAS 2003. Proceedings*. IEEE, 2003, S. 3–12
- [37] KARODE, Sameer P.: *Monitoring Kubernetes Clusters With Dedicated Sidecar Network Sniffing Containers*, Trinity College, University of Dublin, Diplomarbeit, 2020
- [38] TONY, Irene A.: *Application of Machine Learning with Traffic Monitoring to Intrusion Detection in Kubernetes Deployments*, Trinity College, University of Dublin, Diplomarbeit, 2021
- [39] CASAS, Pedro ; MAZEL, Johan ; OWEZARSKI, Philippe: Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge. In: *Computer Communications* 35 (2012), Nr. 7, 772–783. <http://dx.doi.org/https://doi.org/10.1016/j.comcom.2012.01.016>. – DOI <https://doi.org/10.1016/j.comcom.2012.01.016>. – ISSN 0140–3664
- [40] SPIEKERMANN, Daniel ; KELLER, Jörg: Unsupervised packet-based anomaly detection in virtual networks. In: *Computer Networks* 192 (2021), 108017. <http://dx.doi.org/https://doi.org/10.1016/j.comnet.2021.108017>. – DOI <https://doi.org/10.1016/j.comnet.2021.108017>. – ISSN 1389–1286

- [41] *Minikube start*. <https://minikube.sigs.k8s.io/docs/start/>
- [42] PERRY, Yifat: *Eks vs AKS: Head-to-head*. <https://cloud.netapp.com/blog/aws-cvo-blg-eks-vs-aks-head-to-head>. Version: Aug 2021
- [43] DATATECHNOTES: *Anomaly detection example with DBSCAN in python*. <https://www.datatechnotes.com/2020/04/anomaly-detection-with-dbscan-in-python.html>. Version: Apr 2020
- [44] SCHMIDT, Robin M. ; SCHNEIDER, Frank ; HENNIG, Philipp: *Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers*. In: *CoRR* abs/2007.01547 (2020). <https://arxiv.org/abs/2007.01547>
- [45] SYARIF, Iwan ; PRUGEL-BENNETT, A. ; WILLS, Gary: *Unsupervised Clustering Approach for Network Anomaly Detection, 2012*. – ISBN 978-3-642-30506-1
- [46] FONTANINI, Matias: *Libtins*. <http://libtins.github.io/benchmark/>
- [47] KAMARA, Or: *Hack my mis-configured Kubernetes - Privileged Pods*. <https://www.cncf.io/blog/2020/10/16/hack-my-mis-configured-kubernetes-privileged-pods/>. Version: Feb 2022
- [48] DOCUMENTATION: *Using network policy logging nbsp;—nbsp; Kubernetes Engine Documentation nbsp;—nbsp; google cloud*. <https://cloud.google.com/kubernetes-engine/docs/how-to/network-policy-logging>
- [49] MIRSKY, Yisroel ; DOITSHMAN, Tomer ; ELOVICI, Yuval ; SHABTAI, Asaf: *Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection*. In: *CoRR* abs/1802.09089 (2018). <http://arxiv.org/abs/1802.09089>

Appendix

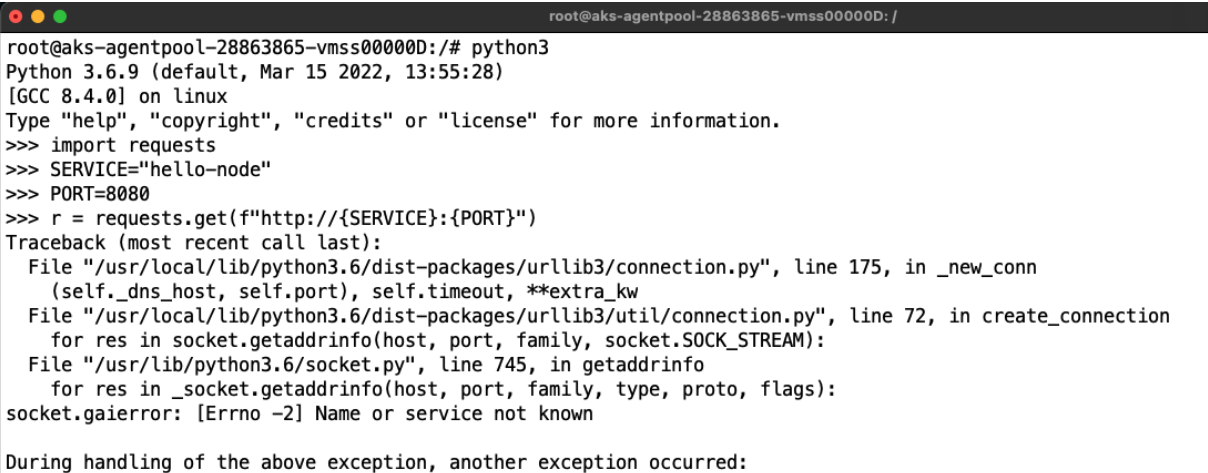
.1 Difference in sending requests from different pods

For testing this, a basic hello-node application was deployed.



```
root@pcap-service-deployment-7ddb5f6648-2x7jg: /
root@pcap-service-deployment-7ddb5f6648-2x7jg:/# python3
Python 3.6.9 (default, Dec 8 2021, 21:08:43)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> SERVICE="hello-node"
>>> PORT=8080
>>> r = requests.get(f"http://{SERVICE}:{PORT}")
>>> r.status_code
200
```

Figure 1: Sending request from a pod to a service



```
root@aks-agentpool-28863865-vmss00000D: /
root@aks-agentpool-28863865-vmss00000D:/# python3
Python 3.6.9 (default, Mar 15 2022, 13:55:28)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> SERVICE="hello-node"
>>> PORT=8080
>>> r = requests.get(f"http://{SERVICE}:{PORT}")
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/urllib3/connection.py", line 175, in _new_conn
    (self._dns_host, self.port), self.timeout, **extra_kw
  File "/usr/local/lib/python3.6/dist-packages/urllib3/util/connection.py", line 72, in create_connection
    for res in socket.getaddrinfo(host, port, family, socket.SOCK_STREAM):
  File "/usr/lib/python3.6/socket.py", line 745, in getaddrinfo
    for res in _socket.getaddrinfo(host, port, family, type, proto, flags):
socket.gaierror: [Errno -2] Name or service not known

During handling of the above exception, another exception occurred:
```

Figure 2: Sending request from tcpdump-container

.2 Code for polling script in tcpdump-container

```
import time
import os
import queue
import requests

jobs = queue.Queue(maxsize=1000)

env_dict = os.environ

SERVICE = env_dict["PCAP_INTERNAL_SERVICE_SERVICE_HOST"]
PORT = env_dict["PCAP_INTERNAL_SERVICE_SERVICE_PORT"]

set_of_files = set()

def check_new_files(path):

    list_of_new_files = []

    for f in os.listdir(path):
        filename = os.path.join(path, f)
        if filename not in set_of_files:
            list_of_new_files.append(filename)
            set_of_files.add(filename)

    return list_of_new_files

def send_file(path):
    try:
        with open(path, "rb") as file:
            file_dict = {"uploaded_file": file}
            try:
                response = requests.post(
                    f"http://{SERVICE}:{PORT}/file",
```

```

        files=file_dict
    )
    return response.status_code
except Exception as e:
    print(e)
except Exception as e:
    print(e)

while True:
    new_files = check_new_files("pcap-to-send/")

    if len(new_files) != 0:
        for file in sorted(new_files):
            if str(file).endswith(".pcap"):
                jobs.put(file)
                print(f"{file} is in queue")

    time.sleep(5)
    if not jobs.empty():
        path = jobs.get()
        status_code = send_file(path)
        if str(status_code) == "200":
            print(f"{path} is being sent to pcap-service")
            os.remove(path)
        else:
            jobs.put(path)
            print(f"{path} is being re-added to queue")

```

Listing 1: Dockerfile for Pcap service

.3 YAML files for MySQL in Kubernetes

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume

```

```

  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi

```

Listing 2: YAML file for creating PV and PVC for a MySQL database

This resource was then mounted to the container, as shown in the snippet for the Deployment YAML.

```

spec:
  containers:
    - image: mysql:oracle
      name: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: <password>
      ports:
        - containerPort: 3306
          name: mysql

```

```
  volumeMounts:
    - name: mysql-persistent-storage
      mountPath: /var/lib/mysql
  volumes:
    - name: mysql-persistent-storage
      persistentVolumeClaim:
        claimName: mysql-pv-claim
```

Listing 3: YAML file for a MySQL database

.4 Console outputs of Nmap and WPScan

```
[+] Headers
| Interesting Entries:
| - Server: Apache/2.4.48 (Unix) OpenSSL/1.1.1d PHP/7.4.21
| - X-Powered-By: PHP/7.4.21
| Found By: Headers (Passive Detection)
| Confidence: 100%

[+] robots.txt found: http://20.31.228.177/robots.txt
| Interesting Entries:
| - /wp-admin/
| - /wp-admin/admin-ajax.php
| Found By: Robots Txt (Aggressive Detection)
| Confidence: 100%

[+] XML-RPC seems to be enabled: http://20.31.228.177/xmlrpc.php
| Found By: Direct Access (Aggressive Detection)
| Confidence: 100%
| References:
| - http://codex.wordpress.org/XML-RPC_Pingback_API
| - https://www.rapid7.com/db/modules/auxiliary/scanner/http/wordpress_ghost_scanner/
| - https://www.rapid7.com/db/modules/auxiliary/dos/http/wordpress_xmlrpc_dos/
| - https://www.rapid7.com/db/modules/auxiliary/scanner/http/wordpress_xmlrpc_login/
| - https://www.rapid7.com/db/modules/auxiliary/scanner/http/wordpress_pingback_access/

[+] WordPress readme found: http://20.31.228.177/readme.html
| Found By: Direct Access (Aggressive Detection)
| Confidence: 100%

[+] The external WP-Cron seems to be enabled: http://20.31.228.177/wp-cron.php
| Found By: Direct Access (Aggressive Detection)
| Confidence: 60%
| References:
| - https://www.iplocation.net/defend-wordpress-from-ddos
| - https://github.com/wpscanteam/wpscan/issues/1299

[+] WordPress version 5.7.2 identified (Insecure, released on 2021-05-12).
| Found By: Rss Generator (Passive Detection)
| - http://20.31.228.177/feed/, <generator>https://wordpress.org/?v=5.7.2</generator>
| - http://20.31.228.177/comments/feed/, <generator>https://wordpress.org/?v=5.7.2</generator>

[+] WordPress theme in use: twentytwentyone
| Location: http://20.31.228.177/wp-content/themes/twentytwentyone/
| Last Updated: 2022-01-25T00:00:00.000Z
| Readme: http://20.31.228.177/wp-content/themes/twentytwentyone/readme.txt
| [!] The version is out of date, the latest version is 1.5
| Style URL: http://20.31.228.177/wp-content/themes/twentytwentyone/style.css?ver=1.3
| Style Name: Twenty Twenty-One
| Style URI: https://wordpress.org/themes/twentytwentyone/
| Description: Twenty Twenty-One is a blank canvas for your ideas and it makes the block editor your best brush. Wi...
| Author: the WordPress team
| Author URI: https://wordpress.org/
|
| Found By: Css Style In Homepage (Passive Detection)
| Confirmed By: Css Style In 404 Page (Passive Detection)
|
| Version: 1.3 (80% confidence)
| Found By: Style (Passive Detection)
| - http://20.31.228.177/wp-content/themes/twentytwentyone/style.css?ver=1.3, Match: 'Version: 1.3'

[+] Enumerating Users (via Passive and Aggressive Methods)
=====
Brute Forcing Author IDs - Time: 00:00:00 <=====
===== > (10 / 10) 100.00% Time: 00:00:00

[i] User(s) Identified:

[+] mayank
| Found By: Author Posts - Author Pattern (Passive Detection)
| Confirmed By:
| Rss Generator (Passive Detection)
| Wp Json Api (Aggressive Detection)
| - http://20.31.228.177/wp-json/wp/v2/users/?per_page=100&page=1
| Rss Generator (Aggressive Detection)
| Author Sitemap (Aggressive Detection)
| - http://20.31.228.177/wp-sitemap-users-1.xml
| Author Id Brute Forcing - Author Pattern (Aggressive Detection)
| Login Error Messages (Aggressive Detection)

[+] Performing password attack on Xmlrpc against 1 user/s
[SUCCESS] - mayank / arora
Trying mayank / arora Time: 00:00:07 < > (195 / 59385) 0.32% ETA: ??:?:??

[!] Valid Combinations Found:
| Username: mayank, Password: arora

[!] No WPScan API Token given, as a result vulnerability data has not been output.
[!] You can get a free API token with 25 daily requests by registering at https://wpscan.com/register

[+] Finished: Thu Apr 7 17:42:52 2022
[+] Requests Done: 209
[+] Cached Requests: 48
[+] Data Sent: 105.949 KB
[+] Data Received: 140.771 KB
[+] Memory used: 169.445 MB
[+] Elapsed time: 00:00:10
```

Figure 3: Output of the WPScan for dictionary attack to generate anomalous data

```

Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-07 17:27 IST
Nmap scan report for 20.31.228.177
Host is up (0.031s latency).
Not shown: 998 filtered tcp ports (no-response)
PORT      STATE SERVICE
80/tcp    open  http
443/tcp    open  https

Nmap done: 1 IP address (1 host up) scanned in 5.14 seconds
Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-07 17:27 IST
Nmap scan report for 20.31.228.177
Host is up (0.020s latency).
Not shown: 998 filtered tcp ports (no-response)
PORT      STATE SERVICE
80/tcp    open  http
443/tcp    open  https

Nmap done: 1 IP address (1 host up) scanned in 4.58 seconds
Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-07 17:27 IST
Nmap scan report for 20.31.228.177
Host is up (0.020s latency).
Not shown: 998 filtered tcp ports (no-response)
PORT      STATE SERVICE VERSION
80/tcp    open  http      Apache httpd 2.4.48 ((Unix) OpenSSL/1.1.1d PHP/7.4.21)
443/tcp    open  ssl/http  Apache httpd 2.4.48 ((Unix) OpenSSL/1.1.1d PHP/7.4.21)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 18.17 seconds
Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-07 17:28 IST
Nmap scan report for 20.31.228.177
Host is up (0.023s latency).
Not shown: 998 filtered tcp ports (no-response)
PORT      STATE SERVICE
80/tcp    open  http
443/tcp    open  https
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running (JUST GUESSING): Linux 4.X|5.X|2.6.X (88%)
OS CPE: cpe:/o:linux:linux_kernel:4.0 cpe:/o:linux:linux_kernel:5 cpe:/o:linux:linux_kernel:2.6.32
Aggressive OS guesses: Linux 4.0 (88%), Linux 4.15 - 5.6 (86%), Linux 2.6.32 (85%), Linux 5.0 (85%), Linux 5.0 - 5.3 (85%), Linux 5.0 - 5.4 (85%), Linux 5.4 (85%)
No exact OS matches for host (test conditions non-ideal).

OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 11.04 seconds
Starting Nmap 7.92 ( https://nmap.org ) at 2022-04-07 17:28 IST
Nmap scan report for 20.31.228.177
Host is up (0.020s latency).
Not shown: 998 filtered tcp ports (no-response)
PORT      STATE SERVICE VERSION
80/tcp    open  http      Apache httpd 2.4.48 ((Unix) OpenSSL/1.1.1d PHP/7.4.21)
|_http-server-header: Apache/2.4.48 (Unix) OpenSSL/1.1.1d PHP/7.4.21
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
|_http-title: Mayank&#039;s Blog! 6#0211; Just another WordPress site
|_http-generator: WordPress 5.7.2
443/tcp    open  ssl/http  Apache httpd 2.4.48 ((Unix) OpenSSL/1.1.1d PHP/7.4.21)
|_http-title: Mayank&#039;s Blog! 6#0211; Just another WordPress site
| ssl-cert: Subject: commonName=example.com
| Not valid before: 2012-11-14T11:18:27
|_Not valid after: 2022-11-12T11:18:27
| http-robots.txt: 1 disallowed entry
|_/wp-admin/
|_http-server-header: Apache/2.4.48 (Unix) OpenSSL/1.1.1d PHP/7.4.21
|_http-generator: WordPress 5.7.2
|_ssl-date: TLS randomness does not represent time
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running (JUST GUESSING): Linux 4.X|5.X|2.6.X (87%)
OS CPE: cpe:/o:linux:linux_kernel:4.0 cpe:/o:linux:linux_kernel:5 cpe:/o:linux:linux_kernel:2.6.32
Aggressive OS guesses: Linux 4.0 (87%), Linux 4.15 - 5.6 (86%), Linux 5.0 (86%), Linux 5.0 - 5.4 (86%), Linux 5.3 - 5.4 (85%), Linux 2.6.32 (85%), Linux 5.0 - 5.3 (85%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 25 hops

TRACEROUTE (using port 80/tcp)
HOP RTT ADDRESS
1 5.40 ms 10.10.0.1
2 5.45 ms 089-100-107150.ntlworld.ie (89.100.107.150)
3 6.20 ms 089-101-115225.ntlworld.ie (89.101.115.225)
4 6.21 ms 089-100-182198.ntlworld.ie (89.100.182.198)
5 7.25 ms ie-dub01a-rc1-ae-15-0.aorta.net (84.116.238.249)
6 6.20 ms ie-dub02a-ri1-ae-73-0.aorta.net (84.116.134.110)
7 17.50 ms ae68-0.iwr02.dba.ntwk.msn.net (104.44.198.115)
8 6.25 ms ae25-0.icr02.dub07.ntwk.msn.net (104.44.239.35)
9 19.46 ms be-122-0.ibr02.dub07.ntwk.msn.net (104.44.11.73)
10 87.63 ms be-7-0.ibr01.ams30.ntwk.msn.net (104.44.17.57)
11 18.27 ms be-1-0.ibr02.ams30.ntwk.msn.net (104.44.16.147)
12 ... 24
25 21.85 ms 20.31.228.177

```

Figure 4: Output of the Nmap script 4.17

.5 Falco scanner in Kubernetes

A Falco security scanner was installed on the node using Helm² with the default ruleset, and it identified the tcpdump container running in a privileged pod as a security risk as shown in figure 5.

²Chart available at: <https://github.com/falcosecurity/charts/tree/master/falco>


```

Informational Privileged container started (user=<NA> user_loginuid=0 command=container:2b514af619a5 k8s.ns=default k8s.pod=tcpdump-container-68dcd
Informational Privileged container started (user=<NA> user_loginuid=0 command=container:8fd2c2185b9e k8s.ns=kube-system k8s.pod=tunnelfront-54879dc
Informational Privileged container started (user=root user_loginuid=0 command=container:28dd0f465ddf k8s.ns=kube-system k8s.pod=omsagent-zwkvz conta
Informational Privileged container started (user=root user_loginuid=0 command=container:9e3930db5d11 k8s.ns=kube-system k8s.pod=omsagent-zwkvz conta
Informational Privileged container started (user=<NA> user_loginuid=0 command=container:cfb67a53c5e9 k8s.ns=kube-system k8s.pod=azure-ip-masq-agent-
Informational Privileged container started (user=root user_loginuid=0 command=container:575a9dc278c1 k8s.ns=kube-system k8s.pod=kube-proxy-122vd con
Informational Privileged container started (user=<NA> user_loginuid=0 command=container:dc2fb9e8a575 k8s.ns=kube-system k8s.pod=csi-azuredisk-node-k
Informational Privileged container started (user=root user_loginuid=0 command=container:3135a94dbfa4 k8s.ns=kube-system k8s.pod=csi-azurefile-node-x
Informational Privileged container started (user=<NA> user_loginuid=0 command=container:d4dab4af8a3b k8s.ns=kube-system k8s.pod=omsagent-rs-58f7dc7b
Informational Container with sensitive mount started (user=<NA> user_loginuid=0 command=container:51022e86b648 k8s.ns=default k8s.pod=kube-prometheu
Notice Network tool launched in container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-22-165
Notice Packet socket was created in a container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-
Notice Packet socket was created in a container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-
Notice Network tool launched in container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-22-165
Notice Packet socket was created in a container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-
Notice Packet socket was created in a container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-
Notice Network tool launched in container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-22-165
Notice Packet socket was created in a container (user=root user_loginuid=-1 command=tcpdump -i cbr0 -G 30 -W 1 -n -vv -tttt -w pcap-file/pcap-17-04-

```

Figure 5: Console output of Falco scanner while running NIDS

.6 Calculating packet flow using destination IP

This packet flow code assigns a packet flow based on the destination IP of the packet.

```

total_packet_list = []

for ts in range(min_timestamp, max_timestamp , 10):

    interval = 9
    if ts + interval > max_timestamp:
        interval = max_timestamp - ts

    # get all the rows for the interval
    time_df = time_analysis[time_analysis['timestamp_int']
                            .between(ts, ts+interval)].copy()

    # create a dictionary from the dataframe
    ip_value_count_dict = time_df['ip.dst'].value_counts().to_dict()

    # apply the dictionary values to the IPs
    time_df['packet_flow_10_s'] = time_df['ip.dst'].apply
        (lambda x: ip_value_count_dict[str(x)])

    # create a list that will be used for appending to the final
    # dataframe
    for x in list(time_df['packet_flow_10_s']):

```

```
total_packet_list.append(x)
```

Listing 4: Python snippet to calculate packet flow using destination IP of the packet