

Course code and name:	F20PA - Research Methods and Requirements Engineering
Type of assessment:	Individual
Coursework Title:	Year 4 Dissertation
Student Name:	Cameron Riley
Student ID Number:	H00374199

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.
- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the [University's website](#), and that I am aware of the penalties that I will face should I not adhere to the University Regulations.
- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on [Academic Integrity and Plagiarism](#)

Student Signature (*type your name*): Cameron Riley

Date: 25/03/2025

Copy this page and insert it into your coursework file in front of your title page.
For group assessment each group member must sign a separate form and all forms must be included with the group submission.

Your work will not be marked if a signed copy of this form is not included with your submission.

Neural Network-Based Intrusion Detection and Prevention System for
Securing Medical Databases

Year 4 Dissertation

Cameron Thomas Riley

H00374199

BSc (Hons.) Computer Science Honours Dissertation

Supervised by Prof. Oleksandr Letychevskyi



Heriot-Watt University

School of Mathematical and Computer Sciences

September 2024

The copyright in this dissertation is owned by the author. Any quotation from the dissertation or use of any of the information contained in it must be acknowledged as the source of the quotation or information.

ABSTRACT

The purpose of the dissertation project is to explore and improve AI-based methods for detecting intrusions into medical databases, with a particular focus on the identification and mitigation of threats such as sensitive data theft, unauthorised access into medical databases, and unauthorised alterations to patient treatment plans, furthering complications and disruptions to healthcare services. These intrusions represent a growing challenge, as healthcare institutions rapidly adopt a digital approach to storing sensitive patient data. This research will compare and build upon traditional Intrusion Detection and Prevention Systems (IDPS) through a software system integrating Deep Neural Networks (DNNs) and a firewall software wrapper, which will be responsible for perceiving network traffic before it is classified by the DNN. Using the CIC-IoMT-2024 dataset, the project evaluates the system's capability to detect and mitigate intrusions in real-time, addressing the challenges posed by traditional systems. By utilising synthetic databases for testing, the research ensures compliance with ethical standards while improving the security of sensitive medical data.[5][7]

DECLARATION

I, Cameron Riley, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Name: Cameron Riley

Signed: 

Date: 23/03/25

ACKNOWLEDGEMENTS

I would firstly like to thank Professor Oleksandr Letychevskyi for overseeing this project, his knowledge and guidance have been fundamental to the completion of this project.

I would also like to thank my family, girlfriend, and friends for their unwavering support and motivation.

Contents

1.1	List Of Figures	9
1.2	List Of Tables	9
1.3	Acronyms And Abbreviations	9
2	INTRODUCTION	10
2.1	Motivation	10
2.2	Aim and Objectives.....	10
2.3	Contributions	11
3	BACKGROUND	11
3.1	Intrusions in Medical Databases	11
3.2	Prevention and Detection	14
3.3	Summary	16
4	METHOD AND REQUIREMENTS ANALYSIS	17
4.1	Research Methodology	17
4.1.1	Overview of Methodology.....	17
4.1.2	Data Collection and Selection.....	18
4.1.3	Data Preprocessing	18
4.1.4	Model Design and Architecture	18
4.1.5	Training and Testing.....	19
4.1.6	Creation of Synthetic Database	19
4.1.7	Integration and Deployment Considerations	19
4.2	Requirement Analysis	20
4.2.1	Functional Requirements	20
4.2.2	Non-Functional Requirements	21
4.3	Evaluation Strategy	22
5	IMPLEMENTATION	23
5.1	Version 1: Initial Implementation	24
	Attack Classification and Labelling.....	25
	Data Inspection and Validation	25
	Phase 2: DNN Training	25
	Model Architecture Design	25
	Model Completion and Optimisation.....	26
	Training Process and Hyperparameter Selection.....	28
	Model Evaluation and Performance Metrics	28
	Model Persistence and Deployment Preparation.....	29

Phase 3: Evaluation through Synthetic Data Generation and Firewall Integration	29
Creation of the Synthetic Medical Database	29
Phase 4: Firewall Integration for Real Time Intrusion Prevention	30
5.2 Version 2: Improved Implementation:	32
5.3 Testing Implementation & Set Up.....	38
5.3.1 Virtualized Environment Setup	38
5.3.2 Attack Simulation Plan.....	39
6 RESULTS ANALYSIS	41
6.1 Evaluation Metrics	41
6.2 Binary Classifier	42
6.3 Multi-Class Classification Model.....	44
6.4 Testing within Virtual Environment.....	47
7 CONCLUSION.....	48
8 Appendix	49
8.1 REFERENCES	49
8.2 Professional, Legal, Ethical and Social (PLES) Issues.....	51
8.2.1 Legal and Professional Issues.....	51
8.2.2 Ethical and Social Issues.....	51
8.3 Project Management.....	51
8.3.1 Risk Form	51
8.3.2 Mitigation Strategy	52
8.4 Schedule.....	54
8.5 Code Snippets and Scripts	56
8.5.1	56
8.5.2	56
8.5.3	56
8.5.4	56
8.5.5	56
8.5.6	57
8.5.7	57
8.5.8	57
8.5.9	57
8.5.10	58
8.5.11	59
8.5.12	59
8.5.13	59

8.5.14	60
8.5.15	60
8.5.16	61
8.5.17	61
8.5.18	61
8.5.19	62
8.5.20	62
8.5.21	63
8.5.22	64
8.5.23	datasetFilter.py	65
8.5.24	binaryClassifier.py	68
8.5.25	multiclassClassifier.py	70
8.5.26	DNNIDPS.py	72

1.1 List Of Figures

Figure number	Description
Figure 1	Signature and Anomaly-based detection architecture
Figure 2	A Flowchart diagram depicting the abstracted view of dataflow through the IDPS
Figure 3	Original DNN Iterations Per-class performance metric output
Figure 4	Commad terminal output detailing class imbalance in initial dataset
Figure 5	Virtual Environment Set Up, Proof of Internal Isolated Network
Figure 6	Binary Classifier Classification Report
Figure 7	Binary Classifier Confusion Matrix
Figure 8	Binary Classifier Training Vs. Validation Accuracy Graph
Figure 9	Multi-Class Classification Report
Figure 10	Multi-Class Classifier Confusion Matrix
Figure 11	Multi-Class Classifier Training Vs. Validation Accuracy Graph
Figure 12	Gantt chart depicting key milestones, dependencies and deadlines for the project.

1.2 List Of Tables

Table Name (In Order as they are presented)	Description
Functional Requirements	A table of all functional requirements for the project
Non-Functional Requirements	A table of all nonfunctional requirements
Packet Capture Feature Table	A table depicting some of the features available in live packet capture
Attack Simulation Tables	A number of tables highlighting the tools used and expected outputs of orchestrating the simulated attacks
Risk Form Table	A table highlighting the risks and mitigations for the project

1.3 Acronyms And Abbreviations

Abbreviation	Meaning
DNN	Deep Neural Network
IDPS/IDS	Intrusion Detection and Prevention System
AI	Artificial Intelligence

2 INTRODUCTION

The increasing digitisation of healthcare systems has led to the widespread use of databases to store sensitive patient information, treatment plans and other critical medical data. Whilst the introduction of these systems has enhanced the efficiency, and accessibility of medical services, it has also led to these healthcare providers now prime targets to a number of cyber-attacks. Due to the sensitive nature of the data stored, cybercriminals are now looking to exploit medical systems to gain access to the mentioned data for either profit or to cause disruption to the provided services. Intrusions to these systems not only compromise patient privacy but can also lead to alterations to treatment plans, unauthorised access to medical networks and devices as well as service disruptions, posing a significant risk to patient safety.

This dissertation focuses on leveraging current AI technology to detect and mitigate intrusions into medical databases, specifically exploring how these AI techniques can improve pre-existing Intrusion Detection Systems (IDS). By comparing AI-based methods to more conventional techniques, this research aims to highlight the potential of AI in providing more robust solutions for the security of medical data.

2.1 Motivation

The growing reliance on digital healthcare solutions has opened the door to vulnerabilities that can be exploited by attackers, leading to data theft, unauthorised data modifications, and system disruptions. While traditional intrusion detection systems are effective in identifying known threats, they struggle to keep up with evolving attack vectors. As cyber threats become increasingly sophisticated, there is a need for more adaptive solutions that can detect anomalies and protect these systems. AI-based methods offer a promising solution to this challenge by learning from data to identify malicious behaviour that may go undetected by signature-based systems.

2.2 Aim and Objectives

This dissertation aims to address the shortcomings of traditional intrusion detection systems on medical databases by exploring the potential of AI techniques. I aim to achieve the following objectives:

- Develop and implement an DNN-based IDPS by combining methods such as Deep Neural Networks (DNN) with preexisting methods such as firewalls, to create a detection and prevention system.
- Compare the effectiveness of traditional IDS with the created DNN-based solution.
- Analyse the impact of existing detection systems that combine elements of pre-existing security solutions with AI-based anomaly detection.
- Create a synthetic medical database containing entirely artificial data, generated to mimic the

structure and characteristics of real medical records whilst ensuring no data relates to any real individual. The synthetic database provides a controlled testing environment for the DNN model and Firewall, ensuring the project is in line with any ethical and privacy concerns.

2.3 Contributions

This dissertation makes the following contributions:

- (1) Development of an DNN based IDPS solution for detecting and preventing intrusions into medical databases.
- (2) A comprehensive evaluation of traditional and AI-based methodologies, using benchmark datasets such as CIC-IDS-2024.
- (3) Further insight into the advantages and challenges of using AI solutions in cybersecurity within the healthcare sector, with a focus on real-world applications.

3 BACKGROUND

The digitisation of healthcare has brought about numerous benefits, such as enhanced patient care, improved data accessibility, and streamlined administrative processes. However, it has also exposed medical databases to increasing risks from cyberattacks. In this chapter, we explore the existing literature and key concepts surrounding intrusion detection systems (IDS) within the healthcare sector, focusing particularly on the use of artificial intelligence (AI) techniques to detect and mitigate intrusions. We will examine both traditional IDS approaches and AI-driven models, setting the stage for the contributions of this dissertation.

In Section 2.1, we will explore the nature of intrusions into medical databases, identifying the various types of cyberattacks and their impact on healthcare systems.

In Section 2.2, we will review existing methods of preventing and detecting intrusions, comparing traditional signature-based systems with AI-based anomaly detection methods.

3.1 Intrusions in Medical Databases

As healthcare institutions increasingly digitise their operations, medical databases have become essential for storing sensitive patient information, treatment plans, and other administrative records. However, this reliance on digital systems has made them attractive targets for cybercriminals. Intrusions into these databases can result in data theft, unauthorised modifications, or disruptions to critical healthcare services, all of which can severely impact patient care and the operation of healthcare facilities.

Several types of attacks are commonly observed in intrusion datasets such as CICIDS, NSL-KDD, and CIC IoMT. These include:

Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) Attacks: DoS and DDoS attacks are designed to overwhelm a network or system, making services unavailable or inaccessible to legitimate users. In the context of healthcare, this could include attacks on services that manage patient records or network-connected medical devices, greatly disrupting essential patient services.[1][2]

Examples of DoS attacks can be found in the selected CIC-IoMT-2024 dataset, here we see several Flood attacks, including SYN Flood and UDP Flood attacks. In a SYN Flood attack, the attacker sends a flood of SYN requests to a target's system but does not complete the TCP handshake, leaving the

system with half-open connections and unable to handle legitimate requests. These types of attacks can cripple healthcare databases that need to be accessible at all times. The implications caused by such attacks can be devastating for both healthcare providers as well as patients, preventing operations that are critical to patient care.

Reconnaissance Attacks: Reconnaissance Attacks involve gathering information about a system's structure, vulnerabilities, and active services to prepare for more severe attacks like data theft or DoS. These attacks are usually the precursor to more damaging intrusions. [1][2] These attacks have previously been identified by the detection of Port scanning and Ping sweeps, notably in datasets such as the CICDS2017.

Port scanning is a method used to identify open or active ports on a host system. With each port corresponding to a specific service, i.e. HTTPS (Port 80) or FTP (Port 21). Any open ports that may be discovered can potentially be used to form an attack, especially if the related services are deemed vulnerable. Attacks will typically use port scanning to gain previously unknown insights into a network's configuration, identifying both active and possibly vulnerable ports. Port scanning tools such as Nmap, systematically test a range of IP addresses and ports, providing insight into the types of services running on a target network and highlights possible points of system entry.

Ping Sweep, or ICMP sweep, is used to determine the active hosts within a range of IP addresses. By sending ICMP (Internet Control Message Protocol) echo requests to multiple addresses, attackers can identify and verify which IP addresses respond, thus revealing live systems within a network. Ping sweeps are commonly the first set in network reconnaissance, as it enables attackers to 'map' which hosts are active and may warrant further investigation. As Lippmann et al. (2020) describe ping sweeps are often followed by port scans on responsive hosts to gain deeper insights into accessible, and potentially vulnerable, services. This provides justification for the number of Ping Sweep and Port Scans which are visible within the mentioned datasets (CICIDS, NSL-KDD, CIC-IoMT-2024).

Man-in-the-Middle (MITM) Attacks: MITM attacks are a form of cyber-attacks in which the attacker intercepts and potentially modifies communications between two oblivious parties. This form of attack is particularly dangerous as it allows the attacker to monitor, modify or inject false or dangerous information into a communication stream, often without either parties' knowledge.

In a typical MITM attack, the attacker positions themselves between the intended victim and a legitimate service (e.g. NHS digital record service and User). They will then intercept and control the flow of data, typically via IP or ARP Spoofing, which causes the victims traffic to be routed through the attacker's system. Conti et al. (2021) identified that through this approach, attacks can steal sensitive information (Patient data/Treatment plans etc. within the healthcare sector) or alter data that is being transmitted by the other two parties to disrupt services and deceive users. Common techniques within MITM attacks include, ARP (Address resolution protocol) spoofing and DNS (Domain Name System) spoofing, both of which deceive network components into sending data to the routed system rather than the true intended recipient.[3]

More recent research by Muhanna Saed and Aljuhani. Ahamed highlights the evolving threat of MITM attacks, particularly within unsecure and poorly configured wireless networks. The study suggests that machine learning techniques can and will enhance the detection of these threats by improving anomaly detection within network traffic patterns, indicative of MITM attacks, such as unusual packet delays or altered traffic paths. [4]

Ransomware Attacks: Ransomware attacks are a type of cyberattack in which malicious software, usually downloaded without the victim's knowledge, encrypts the victims' sensitive files and data, usually leaving the system and affected files unusable and inaccessible until a ransom is paid. Ransomware is particularly damaging in critical sectors such as healthcare, where data accessibility

is essential to carry out services as normal. Attackers will often demand payment with cryptocurrency, making these transactions and attacks harder to trace than ever.

The WannaCry ransomware attack in 2017 is one of the most notable and well documented cases of a ransomware attack, it especially demonstrates the severe impact of ransomware on healthcare systems. This attack exploited a known vulnerability in the Windows SMB protocol, which allowed this malware to spread rapidly across effected networks, effecting over 300,000 vulnerable devices. The NHS (National Health Service) was significantly affected, with nearly 20,000 appointments cancelled, emergency services delayed, and critical systems across hospitals rendered unusable. WannaCry underscored the vulnerabilities within the infrastructure of healthcare and the urgent need for stronger cyber security protocols. According to J. Ferdous, the attack highlighted the risk often associated with outdated systems and underscored the necessity of regular software updates and security patches to prevent related incidents.[5]

Recent literature by S. Ghafur suggests that the integration of AI and machine learning in ransomware detection can improve real-time identification of suspicious file encryption indicative of ransomware attacks. By monitoring behavioural patterns within a network, AI-driven models are shown to improve the identification and mitigation of ransomware before it spreads across an infected network, although these systems still need improvements to minimise false positives, especially within sensitive environments such as healthcare.[6]

Insider Attacks: These attacks are carried out by individuals within the healthcare institution who have authorised access but misuse it to steal or tamper with sensitive data. Insider threats are difficult to detect compared to external attacks and can be initiated by legitimate users who hold the required access credentials. Individuals who have inside access may chose to perform these attacks due to human factors or by mistake, resulting in a high risk for confidential data and systems.

The concern of insider attacks grows in the healthcare sector because they are not only targeted to steal organisational, operational information but also exploit and gain access to sensitive patient data. For instance, employees who are authorised to such data, like health care officers or IT administrators, may leak patient information intentionally or unintentionally that can endanger patient confidentiality. Vasko et al. (2021) explained that insider attacks represent a significant number of breaches within the healthcare sector , often caused by insufficient access control policies and monitoring systems that do not detect abnormal behaviour from insiders.[7]

Newer techniques in behavioural analytics and machine learning show great potential for detecting insider attacks. According to the findings published by M. Bishop , AI frameworks that response to user trends, flagging behaviours off course from baseline activity patterns have proven valuable in pre-empting potential insider risks before any serious damage takes place. Research also highlights the need to balance privacy versus effective monitoring due to the over-surveillance of legitimate employees posing both ethical and legal complications. In addition, these systems are notorious for producing false positives, necessitating sensitive tuning to keep precision at an acceptable level.[8]

SQL Injection: SQL Injection is a type of cybersecurity attack in which an attacker inserts or "injects" malicious SQL code into an input field, aiming to manipulate the backend database. This technique allows attackers to access, modify, or delete data, bypass authentication, or even gain administrative control over the database. SQL injection attacks exploit vulnerabilities in applications that fail to properly validate user input, making them especially prevalent in web applications and other systems that handle user-generated input.

SQL injection attacks are particularly dangerous in sectors that store sensitive data, such as healthcare. For instance, an attacker could use SQL injection to retrieve confidential patient records from a hospital's database or alter records without authorisation. J.O Okesola (2023) highlights that SQL injection remains a top security threat due to its simplicity and effectiveness, especially in

environments where software updates and code reviews are infrequent.[9]

Example of an SQL Injection Attack

Consider a basic SQL query in a login form that checks for valid user credentials:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'password';
```

An attacker might enter the following input into the password field:

```
' OR '1'='1
```

This would alter the query to:

```
SELECT * FROM users WHERE username = 'admin' AND password = " OR '1'='1';
```

This injected code (' OR '1'='1') modifies the SQL query so that it always returns true for the password condition, bypassing authentication and granting unauthorised access. This example demonstrates the vulnerability caused by failing to sanitise inputs, using parameterised queries or prepared statements is crucial to prevent SQL injection by ensuring user input is treated as data rather than executable code.

Research by Rihab Bouafia (2023) suggests that in addition to parameterised queries, web applications should implement strong input validation, employ Web Application Firewalls (WAFs), and routinely review code for potential vulnerabilities. The authors also note that automated vulnerability scanning tools can assist in identifying and mitigating SQL injection risks early in the development lifecycle.[10][11]

3.2 Prevention and Detection

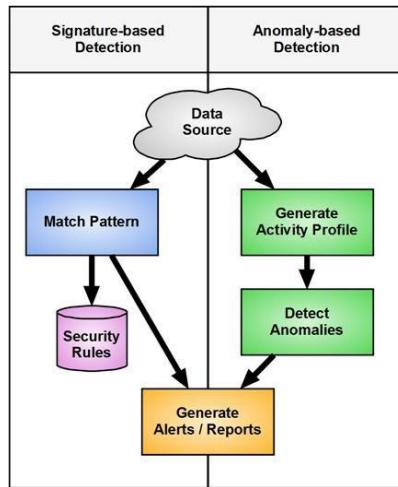
To mitigate the risks posed by these cyberattacks, a variety of prevention and detection methods have been developed. Traditional intrusion detection systems (IDS) and intrusion prevention systems (IPS) are widely used in healthcare. These systems monitor network traffic and system activity for signs of malicious behaviour. Signature-based IDS relies on recognising known attack patterns to identify threats. While this approach is effective against previously encountered threats, it struggles with new or zero-day attacks, which often exhibit patterns not seen before

In contrast, AI-based methods, particularly deep neural networks (DNNs), have shown promise in detecting novel attack patterns by learning from data. DNNs can identify subtle deviations from normal behaviour that might indicate an intrusion. This ability to detect unknown threats makes AI a valuable tool for securing medical databases

Both IDS and IPS play a critical role in monitoring network traffic for signs of malicious activity. Traditional IDS and IPS systems are typically signature-based, meaning they compare incoming traffic to a database of known attack patterns (signatures) to identify threats. However, as Nguyen et al. (2020) note, signature-based systems are limited in their ability to detect new or "zero-day" attacks,

which lack established patterns in the database. Signature-based IDS has been effectively used to detect common threats like SQL injection or buffer overflow attacks but struggles with sophisticated and evolving threats.[12]

A complementary approach is anomaly-based IDS, which establishes a baseline for normal network behaviour and flags deviations that could indicate intrusions. This approach allows for the detection of previously unknown attack types. However, anomaly-based IDS has higher false positive rates than signature-based systems, often requiring additional fine-tuning and contextual analysis to achieve operational efficiency.



[Fig.1 Signature and Anomaly-based detection architecture

AI-based techniques, particularly those based on Deep Neural Networks (DNNs), offer a new class of approaches to detect zero-day or complex cyber-attacks. This is due to DNNs ability to analyse large datasets and learn complex patterns, enabling them to identify small deviations from the normal behaviour of the network which might be overlooked by more traditional IDS. Nguyen et al. (2021) demonstrated that DNNs can achieve high accuracy in identifying zero-day attacks by recognising anomalies that diverge from normal traffic patterns[12]

DNNs are especially effective when trained on healthcare-specific datasets, such as the CICIDS2017 or CIC-IoMT-2024, which include data on healthcare-related cyber threats. However, DNNs require significant computational resources and time for training, making their real-time deployment challenging. Despite these constraints, their capacity to improve detection rates and reduce false positives has made DNN-based IDS increasingly popular in healthcare security solutions.

Firewalls serve as the first line of defence in many security frameworks. They monitor and control both incoming and outgoing network traffic based on a set of predetermined security rules. In

healthcare, firewalls are often configured to allow only specific traffic patterns, thereby reducing the risk of unauthorised access. S. Arunprasath, and Suresh Annamalai (2019) emphasise that while firewalls effectively block many unauthorised access attempts, they are still limited in detecting threats from inside the network. [14]

Advanced firewalls, known as Next-Generation Firewalls (NGFWs), integrate features like deep packet inspection and intrusion prevention capabilities. Making them more effective in identifying sophisticated threats when combined with IDS/IPS and AI-based systems.

Data protection strategies like encryption and data masking are essential for securing sensitive medical data. Encryption ensures that even if a database is breached, the stolen data is unreadable without the decryption key. Advanced Encryption Standard (AES-256), for instance, is widely used in healthcare for its strength and compliance with industry regulations like HIPAA and GDPR. Data masking complements encryption by anonymising data during transmission and storage which further safeguards patient privacy.

Recent studies by Manili et al. (2022) highlight that while encryption is highly effective in protecting data integrity, it does not prevent attacks; thus, it is best combined with detection systems to identify breaches proactively. [15]

Behavioural analytics involves monitoring user behaviour to detect abnormal activities that could signify either insider threats or compromised credentials. In healthcare, insiders, such as employees with legitimate access, are a notable threat. Behavioural analytics systems analyse login times, access patterns, and system interactions to flag unusual activity. Wahab, Fazel, et al. (2023) demonstrated that when combined with AI, behavioural analytics can help distinguish between normal and malicious behaviour with high accuracy, though maintaining privacy and minimising false positives remain challenges. [16]

Integrated detection systems integrate both signature-based and anomaly-based detection methods to achieve a balanced and robust defence. By combining these approaches, integrated systems benefit from the low false-positive rates of signature-based systems and the adaptive threat-detection capabilities of anomaly-based systems. Studies by Al-Mhiqani reveal that integrated systems are particularly effective in healthcare environments, where both known and emerging threats must be monitored [17]

3.3 Summary

In this chapter, we have reviewed the current state of intrusion detection in medical databases, highlighting the most common types of attacks and the methods used to prevent and detect them. While traditional systems such as signature-based IDS are effective at identifying known threats, the increasing complexity of cyberattacks necessitates more adaptive solutions, such as AI-based anomaly

detection systems. These AI-driven models, especially when integrated into integrated detection systems, offer promising advancements in securing sensitive medical data.

4 METHOD AND REQUIREMENTS ANALYSIS

In the Method chapter, clearly explain your research methodology, including the theoretical basis and practical execution. Justify your choice of methods, discussing how they align with the objectives of your study. Consider other methods, explaining why you preferred your chosen approach. This chapter should also address ethical aspects, ensuring your research adheres to ethical standards. Discuss both the strengths and limitations of your methodology to provide a balanced perspective, reinforcing the reliability and validity of your research outcomes.

4.1 Research Methodology

This section details the methodology employed in the design, development and evaluation of the Deep Neural Network (DNN) based IDPS focused on the safeguarding of medical networks and databases. The approach leverages DNNs to enhance the intrusion detection and prevention capabilities of existing software solutions such as Azure Firewall, Comodo and Checkpoint by introducing AI-detection methods which can classify and prevent malicious network traffic. This methodology section outlines the data selection, preprocessing, model architecture, training and testing approaches and ethical implications ensuring compliance with relevant standards and regulations such as GDPR

4.1.1 Overview of Methodology

Traditional IDS rely on signature-based methods that are often ineffective against novel cyberattacks, as noted in studies by Nguyen et al. (2020), who emphasise the limitations of static, signature-based IDS within rapidly evolving attack landscapes [12]. Therefore, this project adopts a DNN approach capable of identifying and analysing deviations from normal access patterns. Anomaly detection and behavioural analytics will be employed to recognise unusual activity in network traffic, addressing the shortcomings of traditional methods and adapting to dynamic threats within the healthcare environment.

The methodology integrates components for continuous learning and improvement, allowing the DNN to adapt to new intrusion patterns over time. This adaptability is especially important in healthcare, where attackers may exploit vulnerabilities in medical databases to access, alter, or disrupt patient records and treatment plans

The key advantages of DNNs within intrusion and detection include their ability to:

1. Recognise complex patterns in network traffic
2. Adapt to evolving threats through model retraining
3. Achieve higher accuracy rates in anomaly detection over traditional systems.

4.1.2 Data Collection and Selection

The dataset chosen for the training and evaluation of the IDPS is the Canadian Institute of Cybersecurity IoMT 2024. This dataset includes a broad range of simulated cyber-attacks targeting medical devices and databases, capturing realistic attack scenarios relevant to healthcare environments. CIC IoMT 2024 is particularly suitable as it provides a variety of attack vectors commonly observed in medical environments, such as but not limited to, Distributed Denial-of-Service attacks (DDoS), reconnaissance attacks, man-in-the-middle (MITM) attacks, ARP Spoofing and Bluetooth Low Energy (BLE) attacks. By focusing on a dataset with simulated attacks, the model can be effectively trained to detect these forms of intrusions whilst maintaining patient care and data confidentiality. This dataset's detailed features, such as timestamps, source and destination IP addresses, action types, and threat classifications allow the model to recognise attack patterns and distinguish legitimate and malicious access, supporting the IDPS's goal of real-time, accurate threat detection in medical settings.

4.1.3 Data Preprocessing

Effective preprocessing is essential to prepare the data for the neural network model. This stage involves cleaning, transforming, and structuring the data to enhance model accuracy and ensure reliable intrusion detection. Initially, all instances of missing or inconsistent data are addressed to maintain data integrity. Duplicate entries are removed, and features with missing values are either imputed or excluded if irrelevant. Continuous variables, such as timestamps and network traffic volumes, are normalised to ensure uniform scaling across features. Categorical variables, including action types and outcomes, are one-hot encoded to facilitate efficient processing within the neural network. Features relevant to intrusion detection, such as IP addresses, action types, and threat classifications, are selected for model input. This selection ensures that the neural network is provided with critical indicators of malicious activity.

Johnson and Lang (2023) highlight that robust preprocessing is essential in cybersecurity models, as unstructured or irrelevant data can degrade model performance by introducing noise, particularly in anomaly-based systems

Preprocessing is performed using the following python libraries; Pandas, NumPy, and Scikit-learn. Chosen for their compatibility with the subsequent model training [11]

4.1.4 Model Design and Architecture

A deep neural network (DNN) is designed to classify network traffic as either benign or malicious, identifying specific types of intrusions that may threaten medical databases. The DNN consists of multiple hidden layers, each employing activation functions (such as ReLU) to model complex, non-linear relationships within the data. The architecture's depth is chosen to ensure the network can

capture subtle patterns associated with various attack types, from DoS attacks to data theft. The model includes dropout layers to mitigate overfitting, a common issue in deep learning, particularly with imbalanced datasets. Additionally, batch normalisation is applied to stabilise and speed up training, further improving the model's performance in classifying rare but critical intrusion events. The architecture is implemented using TensorFlow framework, ensuring scalability and extensibility for future improvements.

4.1.5 Training and Testing

The dataset is divided into training, validation, and test sets to evaluate the model's performance and generalisation capabilities. The CIC IoMT 2024 dataset is split into 70% training data, 15% validation data, and 15% test data. This split ensures a representative sample for model training while reserving sufficient data for unbiased evaluation. The DNN model is trained using the Adam optimiser, known for its adaptive learning rate properties, which accelerates convergence while reducing training time. Cross-entropy loss is chosen as the loss function, as it performs well in binary and multiclass classification tasks. Key performance metrics include accuracy, precision, recall, F1-score, and the area under the ROC curve (AUC-ROC). These metrics are particularly important for evaluating intrusion detection performance, where maintaining high precision and low false-positive rates is essential to prevent unnecessary alerts in a medical environment. Wahab et al. (2023) [16] emphasise the importance of achieving high precision and low false-positive rates, as false alerts can strain healthcare operations.

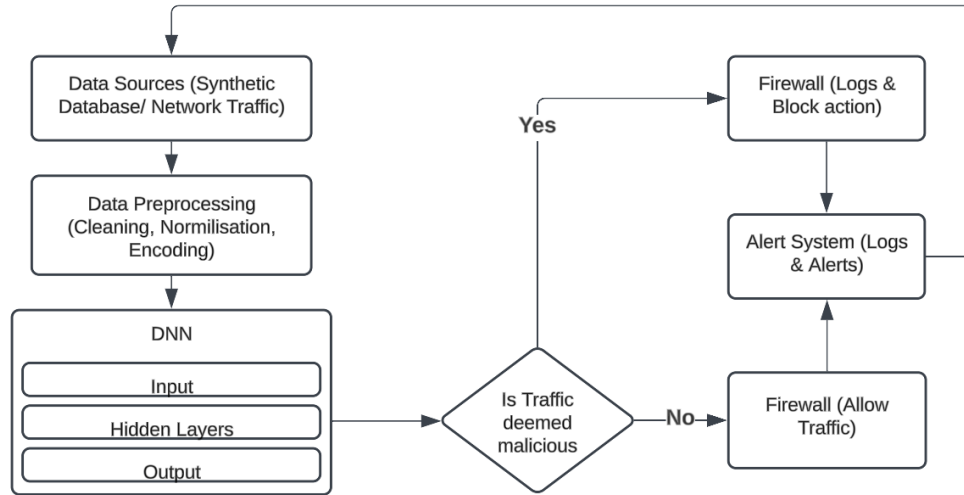
4.1.6 Creation of Synthetic Database

A synthetic database is created to serve as a controlled testing environment for the IDPS. This database is populated with entirely artificial data, designed to mimic the structure and characteristics of real-world medical records. It is to include fields for synthetic data such as patient details and records, treatment histories, etc. The use of the synthetic database is integral to ensure compliance with ethical standards and privacy/data regulations whilst still providing a robust environment for system testing.

4.1.7 Integration and Deployment Considerations

To implement real-time threat detection, the trained DNN will be integrated with an existing security framework, such as a firewall, to form a fully functional IDPS. The DNN model will be deployed within a software wrapper that facilitates its integration into a live network environment. This wrapper enables the IDPS to continuously monitor network traffic, detect threats, and trigger alerts or initiate preventive actions as needed. Given the evolving nature of cyber threats, the DNN model will be designed to accommodate regular updates. Periodic retraining with new data will ensure the IDPS's adaptability to novel attack patterns, maintaining its effectiveness as threats

evolve. The use of deep learning, specifically a DNN, is chosen due to its demonstrated ability to detect complex, evolving attack patterns that traditional signature-based systems may miss. The DNN's capacity to generalise from diverse examples of attack scenarios enables the IDPS to detect both known and novel intrusions. This adaptive approach is especially critical in healthcare, where the protection of sensitive data and the uninterrupted function of services is paramount. A rudimentary diagram of the systems basic architecture is pictured below:



[Fig.2. A Flowchart diagram depicting the abstracted view of dataflow through the IDPS]

4.2 Requirement Analysis

This section outlines the requirements for developing a neural network-based Intrusion Detection and Prevention System (IDPS) designed to secure medical databases. The analysis is divided into functional and non-functional requirements to provide a clear understanding of system capabilities and performance standards. A MoSCoW prioritisation framework is applied to prioritise requirements, ensuring that essential features are implemented to meet the project's core objectives.

4.2.1 Functional Requirements

The functional requirements define the specific actions that the IDPS must perform to protect medical databases. Each requirement is linked to the project's objectives of detecting unauthorised access, identifying anomalies in database interactions, and mitigating potential threats in real-time

Requirement No.	Requirement Description	MoSCoW
-----------------	-------------------------	--------

FR01	The system shall monitor all access to the synthetic medical database and identify any unauthorised attempts to access the data stored	Must
FR02	The system shall detect and alert when met with abnormal patterns in database access that could indicate potential intrusions	Must
FR03	The system shall monitor detect unauthorised modifications to data within the database	Must
FR04	The system shall log all access to the database, including an access identifier, timestamp and action performed	Must
FR05	The system shall utilise a DNN model to predict potential intrusions based on pre-existing data patterns.	Must
FR06	The system shall integrate with a firewall, enabling real-time monitoring of network traffic.	Must
FR07	The system shall include a synthetic database that simulates realistic medical records for testing and validation purposes.	Must
FR08	The system must classify the following attacks: DoS & DDoS Attacks (SYN/UDP Flood), Reconnaissance Attacks (Port Scanning/Ping Sweeps), MITM Attacks (ARP/DNS Spoofing), Ransomware Attacks, Insider Threats (Anomalous access patterns) and SQL Injections	Must
FR09	The system shall automatically restrict users based on identifying suspicious activity or restrict processes that have triggered an alert metric	Should
FR10	The system shall enable real-time updating of intrusion detection rules without downtime or system reboot.	Could
FR11	The system shall provide role-based access control, allowing only authorised personnel to modify system settings, access logs, and intrusion detection configurations.	Could
FR12	The system shall provide a function for manual override, allowing security personnel to whitelist specific activities or users that are wrongly flagged as suspicious.	Could

4.2.2 Non-Functional Requirements

The non-functional requirements specify the quality attributes, performance standards, and compliance criteria the IDPS must meet to be effective in a healthcare environment. These requirements are essential for maintaining the system's reliability, usability, and regulatory alignment

Requirement No.	Requirement Description	MoSCoW
NFR01	The system shall process and analyse database access logs with minimal latency to ensure real-time intrusion detection.	Must
NFR02	The system shall maintain high availability to ensure uninterrupted monitoring and detection.	Must
NFR03	The system shall integrate a synthetic database capable of replicating real-world medical database structures and traffic.	Must
NFR04	The system shall include a scalable firewall module that handles increased network traffic while maintaining performance.	Should
NFR05	The system shall support encryption of sensitive data both in transit and at rest	Must
NFR06	The system shall support modularity, allowing for the addition of new components, such as updated intrusion detection rules.	Could
NFR07	The system shall provide a user-friendly interface for monitoring the system status and responding to alerts.	Should
NFR08	The system shall be designed within Python utilising libraries such as NumPy, SciPy, Scikit-learn, TensorFlow	Must
NFR09	The system shall be robust against false positives, maintaining an accuracy rate of 95% or higher in distinguishing between legitimate access and intrusions	Must
NFR10	The system shall allow for easy retraining of the AI model to incorporate new types of intrusions or changes in attack patterns.	Could

4.3 Evaluation Strategy

The evaluation of the IDPS will rely on comprehensive testing against real-world scenarios in simulated environments. Each functional and non-functional requirement will be assessed based on the performance metrics defined in the training and testing section, ensuring that the model can reliably detect a broad spectrum of intrusions with minimal false positives.

Additionally, AL-Mhiqani et al. recommends that evaluation strategies for cybersecurity systems should include both technical performance (e.g., accuracy, latency) and usability metrics, given the high stakes in healthcare environments [17]

The following sections detail the specific metrics and testing phases that will be employed to effectively evaluate the systems performance and usability.

Evaluation Metrics

- Accuracy: Measures the overall percentage of correctly classified intrusions or benign events.
- Precision: Evaluates the system's ability to correctly identify positive detections whilst avoiding false positives.
- Recall (Detection Rate): Measures the proportion of actual intrusion events that the system correctly detects.
- F1-Score: Provides a mean of precision and recall, offering a balance performance metric.
- AUC-ROC Curve: Assesses the system's ability to distinguish between benign and malicious events across the varying thresholds.
- Latency: Measures time taken for the system to detect and respond to intrusions, ensuring real-time performance
- False Positive Rate (FPR): Tracks the percentage of benign events incorrectly flagged as intrusions, provides insight to usability in real-world scenarios

Evaluation Phases

- Unit Testing: Each module, including data pre-processing, DNN model, Synthetic database, and Firewall integration will be tested independently.
 - Focus is to be placed on ensuring individual components meet the functional requirements.
- Integration Testing: Evaluate the operation of the DNN, Firewall, and synthetic database as a unified system.
 - Scenarios will include detecting known attack patterns and handling benign traffic
- System Testing: Use the CIC-IoMT-2024 dataset to validate system functionality utilising the synthetic database to test for simulated real-world conditions.

Testing Environments

- Simulated Environment: The synthetic database will be used as a controlled testing environment, simulating real-world medical database structures and interactions. Network emulation tools such as Mininet can be adopted to mimic real-time network traffic for testing intrusion detection capabilities.
- Benchmark Dataset Environment: The CIC-IoMT-2024 dataset will serve as the primary benchmark for evaluating the DNN Model's performance. Data subsets will simulate diverse attack vectors, including reconnaissance, DDoS, and MITM Attacks.

5 IMPLEMENTATION

The implementation of the Deep Neural Network (DNN) based Intrusion Detection and Prevention System (IDPS) was structured into two distinct versions; Version 1, which represented the initial system development and deployment, and Version 2, which represents an enhanced iteration of the system developed to address critical shortcomings of Version 1

5.1 Version 1: Initial Implementation

The initial DNN based IDPS implementation consisted of four distinct phases; dataset cleaning and preprocessing, DNN configuration and training, the creation of a synthetic medical database, and finally integration with firewall rules

Phase 1: Dataset Cleaning and Preprocessing

Initially, the preprocessing included basic normalisation and straightforward feature extraction from the CIC-IoMT-2024 Dataset. Numeric Columns from 51 training and 21 test .csv files were identified, scaled using MinMax scaler, and finally combined using multithreading to optimise computational efficiency. Despite these processes, several inefficiencies emerged:

- Redundant feature extraction due to insufficient feature selection strategies
- Misclassification issues arising from improper string handling when categorising attack types, specifically confusing DoS with DDoS attacks due to both containing the string 'DoS'

Found below is a detailed explanation of the steps taken in this first phase:

The script begins by identifying and loading all .csv files within the specified directory, each file is identified using the 'os' library, ensuring only relevant files are considered:

```
files = [os.path.join(inputFolder, f) for f in os.listdir(inputFolder) if f.endswith(".csv")]
```

The use of `os.listdir(inputFolder)` ensures that all files within the specified directory are detected, this is filtered to detect only the .csv files via 'if `f.endswith(".csv")`' this ensures only usable files are loaded and used in training, also preventing unnecessary computation if no .csv files are found.

Since it was possible that the differing .csv files may have had slightly varying structures, the script extracts all unique numeric columns across the dataset, specifically those of type 'float64' and 'int64' [See code excerpt 8.5.1]

By dynamically extracting numeric columns rather than relying on predefined feature lists, this method enhances the pipelines robustness. This feature standardisation ensures each file contributes to the final dataset in a consistent manner.

Feature scaling is an essential step in preprocessing, especially in machine learning applications that rely on gradient based optimisation. The script employs MinMax scaling, which translates numerical values into a fixed range of [0,1]. This mitigates the impact of features with vastly different scales, preventing dominant features from skewing bias in model training: [See code excerpt 8.5.2]

The decision to fit the chosen scaler to the entire dataset, rather than individual files, ensures global consistency in the feature scaling process. Before the scaler is fitted, missing or nullified values are also replaced with the mean of the column to prevent bias in normalisation. This ensures all numeric data is transformed on a common scale, particularly beneficial in algorithms that are sensitive to feature magnitude variations

Given the large volume of network traffic data, the large computational overhead was pre-emptively avoided by utilising multithreading via 'concurrent.futures.ThreadPoolExecutor' enabling parallel file processing. Each file is processed as an independent task, where each step of preprocessing is executed. This significantly reduces processing time, making it a suitable choice for handling such large volumes of data.

Following the concurrent processing of all the files within the dataset, the processed DataFrames are combined into a single unified database. The panda's 'pd.concat()' function is used to merge the DataFrames while maintaining continuity [See code excerpt 8.5.3]

This final step merges the DataFrames and saves it to the specified output file in .csv format, suitable for model training later in the projects implementation.

Attack Classification and Labelling

In the original CIC-IoMT-2024 dataset, each form of attack was designated to its own .csv file, with the following data columns; Header_Length, Protocol Type, Duration Rate, Srate, Drate, fin_flag_number, syn_flag_number, rst_flag_number, psh_flag_number, ack_flag_number, ece_flag_number, cwr_flag_number, ack_count, syn_count, fin_count, rst_count, HTTP, HTTPS, DNS, Telnet, SMTP, SSH, IRC, TCP, UDP, DHCP, ARP, ICMP, IGMP, IPv, LLC, Tot sum, Min, Max, AVG, Std, Tot size, IAT, Number, Magnitude, Radius, Covariance, Variance and Weight.

To facilitate accurate classification of intrusion types, attack categories must be assigned to each .csv file before they could be combined for model training. A mapping was created for the varying attack types, including DDoS, DoS, MQTT-based attacks, reconnaissance scans, and benign traffic. This process was automated using a python script I created titled "add_attactypes_train.py", in which an "Attack_category" column was appended to each file after determining the category of attack based on each .csv file's name. The first error encountered was the misclassification of DDoS attacks as DoS attacks, this was due to incorrect string handling in which the string 'DoS' was found in both DDoS and DoS .csv files. To rectify this, DDoS attacks were given priority in the string search which prevented DDoS attacks being classified as DoS. The complete and correct .py script can be found in the appendix of this report.

Data Inspection and Validation

The final step of data preprocessing, in this case, was to verify the structure and integrity of the combined dataset. To do this another python script was created, trainset_search.py, in which numerous rows of the combined dataset were displayed. This included the list of all column names to identify potential target columns by analysing value distributions. This step in validation was critical in confirming the completeness of the preprocessing scripts, ensuring the combined dataset was ready to be used for model training.

Phase 2: DNN Training

The second phase of the development of the Intrusion Detection and Prevention System focused on the training of a Deep Neural Network to detect and classify live network intrusions based on the preprocessed traffic data. Deep learning algorithms, particularly DNNs, have been widely adopted within cybersecurity applications due to their ability to model and identify complex relationships in high dimensional data. In this study, a fully connected feedforward DNN was employed to classify attack types, using the numerical features extracted from the network traffic.

Model Architecture Design

The architecture of the proposed DNN was carefully designed to balance computational efficiency and

model complexity, ensuring optimal learning of attack patterns where possible. The implemented network consists of multiple fully connected layers that capture the hierarchical representations of network traffic features. The model was developed using TensorFlow and Keras, which is illustrated in the below code excerpt 8.5.4 [See appendix]:

The network is structured as follows:

- **Input Layer:** The input dimensions correspond to the number of preprocessed features extracted from the combined dataset.
- **Hidden Layers:** Two fully connected layers (128 and 64 neurons respectively) with ReLU activation functions. ReLU was chosen due to its computational efficiency and its ability to mitigate the vanishing gradient problem, which commonly affects deep networks by preventing gradient updates from flowing through the network during backpropagation. Unlike sigmoid or tanh functions that saturate and cause gradients to approach zero for large input values, ReLU simply outputs the input if it's positive or zero otherwise, maintaining a constant gradient of 1 for all positive inputs. This property allows deeper networks to learn more effectively, as the gradient signal remains strong even when propagating through multiple layers. Additionally, the sparsity induced by ReLU activations (where approximately 50% of neurons are inactive with output 0) creates more disentangled representations and improves model generalization while reducing computational demands during both training and inference phases.
- **Dropout Regularisation:** A dropout rate of 0.2 was applied between the hidden layers to prevent overfitting. Dropout is an effective regularisation technique that randomly disables or deactivates a proportion of the neurons (In our case 0.2 or 20% of the neurons) during training, improving the generalisability of the model
- **Output Layer:** Finally, a SoftMax activated layer was used for multi-class classification. The number of output neurons corresponds to the number of unique attack categories present in the preprocessed dataset (In our case 45)

The architecture was chosen after iterative experimentation, where multiple hyperparameters were tuned to experiment in various set-up scenarios for the DNN. The final approach demonstrates a suitable trade-off between computational cost and classification performance

Model Completion and Optimisation

To facilitate efficient and stable training, the Adam optimiser was chosen due to its ability to adaptively adjust learning rates for each parameter during gradient updates. Adam, which stands for Adaptive Moment Estimation, combines the advantages of both the momentum method and RMSProp, allowing it to accelerate convergence in deep networks while maintaining stability. The mathematical formulation of Adam includes first-order moment estimation (β_1) and second-order moment estimation (β_2), which dynamically regulate the learning rate based on the magnitude of past gradients:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

where g_t represents the gradient at time step t , and m_t , v_t are the biased first and second moment

estimates, respectively. These moment estimates are then used to compute the parameter updates. The choice of loss function was particularly crucial due to the multi-class classification nature of the intrusion detection task. The model was required to classify network traffic into distinct attack categories, necessitating a loss function that effectively handles categorical target labels. Sparse categorical cross-entropy was selected as the loss function due to its efficiency in handling integer-encoded class labels, as opposed to one-hot encoded vectors.

Sparse Categorical Cross-Entropy:

Sparse categorical cross-entropy is a variant of categorical cross-entropy that operates directly on integer-encoded class labels rather than one-hot encoded vectors. The standard categorical cross-entropy loss function is computed as:

$$L = - \sum_{i=1}^C y_i \log(\hat{y}_i)$$

where:

y_i is the true one-hot encoded label for class i ,
 \hat{y}_i is the predicted probability for class i ,
 C represents the total number of classes.

However, in the case of sparse categorical cross-entropy, instead of using a one-hot encoded representation of class labels, the target variable remains as an integer index, reducing memory overhead and computational complexity. The loss function is then computed as:

$$L = - \log(\hat{y}_c)$$

where \hat{y}_c is the predicted probability for the true class c , corresponding to the integer-encoded label.

The key advantages of using sparse categorical cross-entropy over categorical cross-entropy include:

- Reduced memory usage: Since sparse categorical cross-entropy operates on integer labels instead of one-hot encoded vectors, it significantly reduces memory consumption, especially in datasets with many unique classes.
- Computational efficiency: Eliminating the need for one-hot encoding simplifies computations, leading to faster loss computation.
- Improved numerical stability: Sparse categorical cross-entropy avoids unnecessary matrix operations associated with one-hot encoding, which can sometimes introduce floating-point precision issues in deep networks.

Given the above considerations, the final model was compiled using Adam as the optimiser and sparse categorical cross-entropy as the loss function:

```
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

For initial performance monitoring during training, classification accuracy was used as the primary metric. Accuracy is defined as:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

where TP and TN represent true positives and true negatives, and FP and FN correspond to false positives and false negatives, respectively.

Although accuracy provides a straightforward measure of model performance, it can be misleading in imbalanced datasets, where certain attack categories may be underrepresented. Therefore, additional evaluation metrics such as precision, recall, and F1-score were computed post training to provide a more comprehensive assessment of classification effectiveness.

Training Process and Hyperparameter Selection

Training was conducted using a 80/20 split between training and validation data. The dataset was first preprocessed, where features were normalised using the StandardScaler and target labels were encoded into numerical values [See code snippet 8.5.5]

Once preprocessed, the dataset was then split into the training and testing subsets. Key hyperparameters were tuned to optimise the model's learning efficiency:

- Batch Size: Set to 32, balancing gradient stability with computational efficiency
- Epochs: Initially Set to 20, though an early stopping mechanism was implemented to halt training once validation performance plateaued. This was observed in initial runs to be on Epoch 10/20. It was later decided to reduce the number of Epochs to 10, to reduced computational overhead and training runtimes during development.
- Early Stopping: Monitored the validation loss to prevent overfitting, restoring the best performing model check point:

Model training was then executed using the Keras fit() function [See code excerpt 8.5.6]

This approach ensured the model had received sufficient training without excessive epochs which was observed to lead to overfitting and accuracy loss.

Model Evaluation and Performance Metrics

A comprehensive framework was implemented to assess the DNN's performance beyond the simple accuracy metric. The trained model was evaluated on the test set, and key performance metrics such as precision, recall, and F1-score were calculated [See 8.5.7]

A classification report was then generated to provide 'per-class' analysis:

```
print("\nClassification Report:")
```

```
print(classification_report(yTest, predictedLabels, target_names=label_encoder.classes_))
```

Which results in the following output in the terminal:

	precision	recall	f1-score	support
ARP Spoofing	0.00	0.00	0.00	3142
Benign	0.77	1.00	0.87	38591
DDoS ICMP	0.80	0.99	0.88	307462
DDoS SYN	0.85	0.93	0.89	160442
DDoS TCP	0.68	0.99	0.81	160377
DDoS UDP	0.80	0.97	0.88	327180
DoS ICMP	0.58	0.87	0.12	83006
DoS SYN	0.85	0.70	0.77	88738
DoS TCP	0.84	0.82	0.84	76263
DoS UDP	0.79	0.28	0.41	113465
MQTT DDoS Connect Flood	0.99	1.00	0.99	34592
MQTT DDoS Publish Flood	1.00	0.96	0.98	5615
MQTT DoS Connect Flood	0.99	0.83	0.91	2551
MQTT DoS Publish Flood	0.98	1.00	0.99	8839
MQTT Malformed Data	0.00	0.00	0.00	1028
OS Scan	0.00	0.00	0.00	3375
Ping Sweep	0.00	0.00	0.00	130
Port Scan	0.84	0.89	0.86	16922
Recon VulScan	0.00	0.00	0.00	449
accuracy			0.79	1432167
macro avg	0.62	0.56	0.55	1432167
weighted avg	0.79	0.79	0.74	1432167

[fig3: 'per-class' performance metric output]

The model demonstrated strong performance across multiple attack categories, with particularly high precision in detecting DDoS and MQTT-based attacks. However some attack classes exhibited lower recall, indicating potential challenges in distinguishing similar intrusion patterns.

Model Persistence and Deployment Preparation

Upon successful training and validation, the finalized DNN model was serialized using TensorFlow's built-in model-saving functionality. This allowed the trained model to be deployed without requiring re-training. The trained model, along with the fitted scaler and label encoder, were saved [See code excerpt 1.8] This step ensured that the model's learned parameters were preserved and could be seamlessly integrated into the Intrusion Detection and Prevention System (IDPS). The inclusion of the label encoder mapping was particularly important to maintain consistency between training and deployment, ensuring that numerical classifications were accurately translated back into human-readable attack categories.

Phase 3: Evaluation through Synthetic Data Generation and Firewall Integration

This phase of the project focuses on two key points of development (1) testing the intrusion detection capabilities on the trained DNN model against simulated attacks on a synthetic medical database and (2) integrating the model with real-time Intrusion Detection and Prevention capabilities that monitor live network traffic and block detected threats. These steps were crucial in evaluating the models capabilities and the practical applicability of the DNN based IDPS in the context of cyber security for the healthcare industry.

Creation of the Synthetic Medical Database

To ensure ethical integrity within this study whilst effectively evaluating the performance of the model,

a synthetic medical records database was created. Unlike conventional intrusion detection datasets that focus on simulated network traffic data, this database was designed to simulate real world *electronic health records* (EHRs), thereby reflecting the types of sensitive data that attackers may target when orchestrating attacks within the healthcare sector

Database Structure and Implementation

The synthetic medical database was implemented as an SQLite relational database, ensuring that it closely mimics the structure of real hospital records. The schema consisted of multiple tables:

- Patients Table: Contains demographic information (e.g. Name, DoB, Gender, etc.)
- Visits Table: Stores details of patient visits, including timestamps and reasons for consultations.
- Diagnoses Table: Maps a number of common medical conditions to patient visits
- Medications Table: Tracks prescribed drug dosages and duration of prescriptions.

The Python script `create_synthetic_medical_records.py` was used to generate realistic patient data using the Faker library, a large code snippet of this process is available in [8.5.9 of the appendix]

This approach ensured that the database maintained structural realism without using real patient data, thus strictly adhering to privacy laws and standards of ethical research.

Phase 4: Firewall Integration for Real Time Intrusion Prevention

The next phase of the study focused on real-time response mechanisms. Rather than simply detecting and classifying intrusions, it was important to add design considerations to actively block or prevent attackers using firewall-based prevention techniques.

Live Network Traffic Monitoring

To implement real time monitoring, the system employed PyShark, a Python wrapper for Wireshark, enabling continuous network packet sniffing. The network interface used for live capture was explicitly defined and can be changed depending on the application scenario:

```
import pyshark
capture = pyshark.LiveCapture(interface="WiFi")
for packet in capture.sniff_continuously():
    process_packet(packet)
```

This setup allows the system to continuously process incoming network traffic. Each captured packet was then passed to the `process_packet()` function, where feature extraction and classification were performed.

Feature Extraction and Threat Classification

For intrusion detection, the most informative network features were extracted from each packet, ensuring consistency with the input format expected by the trained DNN model. These extracted features included:

Feature	Description
Packet Length	Total size of the packet in bytes

Transport Protocol	Encoded as 1 for TCP, 2 for UDP, 0 otherwise
Source Port	Identifies the originating port of the packet
Destination Port	Identifies the target port of the packet

The feature extraction process was implemented [See excerpt 8.5.10] Each packet was converted into a structured numerical vector matching the feature space used during model training.

Classification using the DNN

Once numerical features were extracted, they were normalised using the pre-trained StandardScaler to ensure numerical consistency with the expected inputs of the DNN.

```
features_scaled = scaler.transform(features)
```

The pre-trained DNN intrusion detection model was then used to classify the packet as benign or malicious and to also classify the predicted attack type if applicable.

```
prediction = model.predict(features_scaled)
predicted_class_index = np.argmax(prediction, axis=1)[0]
predicted_label = label_encoder.inverse_transform([predicted_class_index])[0]
```

Here the DNN outputs a probability distribution over all attack categories, and the argmax (np.argmax(x)) identifies the attack category with the highest probability. Finally, the label encoder converts the predicted index into a human-readable attack type. The final predicted output for this classification problem defines each packet as either benign or belonging to a specific attack category learnt in model training.

Although the DNN outputted class probabilities, a confidence threshold was defined to reduce false positives. Only predictions about a 50% confidence rate were treated as valid attack classifications:

```
if prediction[0][predicted_class_index] > 0.5:
    logging.warning(f'Detected {predicted_label} attack with high confidence.")
```

Automated Firewall Enforcement for Intrusion Prevention

Once a packet was classified as malicious, the system responds by blocking the source IP address. The firewall integration was designed to work across platforms, adaptively selecting the correct blocking mechanism based on operating system that is detected

On Windows, the netsh command was used to add a rule blocking the malicious IP:

```
command = f'netsh advfirewall firewall add rule name="Block {sourceIP}" dir=in action=block
remoteip={sourceIP}'
```

```
subprocess.run(command, shell=True)
```

On Linux, iptables was used to drop packets from the attackers IP:

```
command = f"iptables -A INPUT -s {sourceIP} -j DROP"
subprocess.run(command, shell=True)
```

To prevent redundant firewall rules, a global set was also maintained to track previously blocked Ips [See 8.5.11] This ensures that each IP is blocked only once, preventing excessive firewall rule proliferation.

Version 1 was fully developed and tested up to the synthetic data generation and initial integration stages. However, when deploying the software on the Virtual Machine Testing Environment, significant issues were encountered that prevented successful progression to final testing. Primarily, these issues were centred around compatibility with the DNN's saved '.h5' format on the virtual environment, alongside extensive inefficiencies due to redundant and mismatched feature sets when integrating PyShark for live network packet capture, which was essential to the functionality of the project. These issues ultimately prevented Version 1 from reaching the intended testing phase

5.2 Version 2: Improved Implementation:

In response to the critical shortcomings of Version 1, Version 2 underwent redevelopment proceeding the initial combined dataset, emphasising feature compatibility, computational efficiency, and reliable deployment within the Virtual Environment.

Phase 1: Data Preprocessing

The initial data preprocessing phase was fundamental in ensuring the efficiency and accuracy of the machine learning process further along in the development pipeline. Given the problems faces in Version 1 of development, such as mismatched features during live packet capture and substantial data redundancy, a full restructuring of this process was necessary. This was done via python script 'datasetFilter.py', which was designed to prepare the data explicitly for compatibility with PyShark's real-time packet analysis

Protocol Inference Based on Attack_Category

One major issue with the earlier iteration was the misalignment of protocol features between the training dataset and the live packet capture. To resolve this a custom function was created to identify each attack based on the contents of the 'Attack_Category' from the earlier combined training dataset. This can be seen in section 8.5.12 of the appendix of this report

This inference approach provides multiple advantages:

- Eliminates any dependency on external or additional data sources for protocol information, thus reducing the complexity and computational demand of preprocessing
- Ensures the inferred protocols are consistent with the format and types that PyShark is able to capture during live monitoring
- Significantly reduces errors stemming from incorrect or missing protocol labels, which was present in Version 1 and ultimately improved model accuracy and reliability

Feature Selection for Real-time Packet Capture

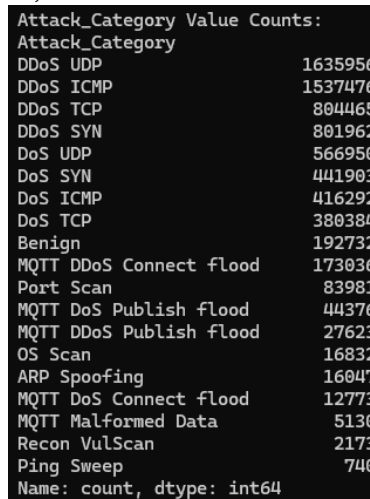
Version 1 faced significant discrepancies between the features present during model training and those captured by PyShark in testing. To address this, a precise selection of compatible features was chosen, directly corresponding to those available via PyShark. The refined list of features deemed critical for model training included [See excerpt 8.5.13]

By selecting these features, the following was achieved:

- A precise match between training data and live network data, significantly reducing the occurrence of runtime errors and mismatches present in Version 1
- Improved computational efficiency by removing redundant features, streamlining the DNN training process

Dataset Balancing

Another substantial issue from Version 1 was the significant imbalance between various attack categories, causing the Version 1 DNN to have extreme bias, this subsequently caused the model to 'over-predict' certain attack categories, this is illustrated below:



Attack_Category	Value Counts
DDoS UDP	1635956
DDoS ICMP	1537476
DDoS TCP	804465
DDoS SYN	801962
DoS UDP	566950
DoS SYN	441903
DoS ICMP	416292
DoS TCP	380384
Benign	192732
MQTT DDoS Connect flood	173036
Port Scan	83981
MQTT DoS Publish flood	44376
MQTT DDoS Publish flood	27623
OS Scan	16832
ARP Spoofing	16047
MQTT DoS Connect flood	12773
MQTT Malformed Data	5130
Recon VulScan	2173
Ping Sweep	740

Name: count, dtype: int64

[Fig4: Cmd output detailing class imbalance in initial dataset]

To rectify this, a robust dataset balancing strategy was added to the script, to equalise the number of samples across all classes. [See code excerpt 8.5.14]

Other techniques were tested such as SMOTE, which filled the lower classes with synthetically created data, however this was removed due to the technique providing inadequate and false packet data, decreasing the accuracy of the model.

The improved balancing process provided the following benefits:

- It equalised representation across all classes, ensuring that the model would not become biased towards overrepresented classes, such as the previously dominant DDoS UDP attacks
- Improved overall model performance and reliability in recognising a wider range of attack types, enhancing the practical utility of the intrusion detection system.

Reindexing Dataset Features

Ensuring a consistent order and complete representation of features within the dataset was crucial for maintaining alignment with training and prediction environments. Reindexing of the DataFrame with a defined set of expected columns was employed: [See 8.5.15]

Key advantages of this process included:

- Prevention of unexpected feature mismatches during training, critical for maintaining reliability in live packet detection
- Standardisation of datasets, enhancing the replicability and stability of the preprocessing process

Separation of Binary and Multi-Class Datasets

To further streamline training and improve clarity, datasets were separated into binary and multi-class datasets. This separation was vital for the training of two machine learning models; a binary classifier to detect benign vs. malicious traffic, and a more detailed multiclass classifier (DNN) to categorise specific attack types [See 8.5.16]

This categorisation provided the following practical benefits:

- Allowed specialised optimisation strategies for binary vs. multiclass training processes
- Improved computational efficiency and simplified the management of training workflows

Phase 2: DNN Configuration and Training

Version 2 introduces substantial advancements in the architecture of the DNN as well as training procedures to address the performance and compatibility challenges identified from Version 1. This phase provided enhancements to accuracy, computational efficiency, reduced training times, and reliability for both binary and multiclass classification tasks

Model Architecture

The revised DNN architectures were designed for both binary and multiclass classification, each tailored to maximise both performance and stability within the computational constraints.

Binary Classification Model:

A binary classifier was added and structured for binary classification between benign and malicious network traffic. The architecture employed densely connected layers with a Rectified Linear Unit (ReLU) activations for improved gradient flow, also added were two dropout layers to reduce overfitting by randomly disabling a proportion of neurons during training: [See 8.5.17]

Key considerations for this architecture include:

- Dense layers provided sufficient complexity to model relationships of benign vs. malicious traffic without increasing computational overhead
- Dropout layers (rate of 0.3) significantly reduced the risk of overfitting, which was particularly relevant due to the binary nature of this classification task, where model bias can significantly

skew predictions

- The sigmoid output layer facilitated clear probabilistic interpretation, providing a threshold mechanism for identifying malicious traffic

Multi-Class Classification Model:

The multiclass classification task required a more robust and complex model architecture to effectively classify network traffic between multiple attack categories. Hence, a deeper network structure with LeakyReLU activations and Batch Normalisation was developed, this can be seen in section [8.5.18]

This multiclass model offered again several advantages:

- Batch Normalisation to improve training stability by normalising layer inputs, mitigating internal covariate shift and improving convergence speeds
- LeakyReLU activations prevent the previous issue of dying neurons, encountered with the previous implementation (standard ReLU), this maintains gradient flow especially for negative inputs
- L2 regularisation (set at 0.001) and varying dropout rates (0.4 and 0.2 respectively) further prevented overfitting, improving the models generalisation capabilities to unseen data

Training improvements:

To further ensure efficient and effective training, more advanced mechanisms such as early stopping and adaptive learning rate scheduling were implemented:

```
early_stopping = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True, verbose=1)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6, verbose=1)
checkpoint = ModelCheckpoint("temp_multiclass_best.weights", monitor='val_loss', save_best_only=True, save_weights_only=True, verbose=1)
```

- Early stopping monitored the models validation loss, ensuring training ceased upon convergence before overfitting could occur, this preserves the best-performing model parameters
- ReduceLROnPlateau adaptively adjusted the learning rate when validation loss plateaus, significantly improving convergence speed and preventing divergence
- Checkpointing ensured that the optimal model weights were retained throughout training, providing robust fail-safes against potential training instabilities.

Training Pipeline and Data Handling:

The comprehensive training process integrated many preprocessing steps, explicit dataset splitting strategies and rigorous model evaluation techniques: [See section 8.5.19]

- Explicit stratified splitting ensured that both training and test sets adequately represented all attack categories, critical for valid performance assessments.
- StandardScaler was employed to normalise features, ensuring consistent numerical ranges for stable and efficient neural network training.
- Validation data was continually monitored throughout training to identify performance degradation or overfitting, with mechanisms in place for immediate mitigation.

Model Persistence and Deployment:

To overcome previous compatibility issues found in Version 1. TensorFlow's 'SavedModel' format was adopted for model serialisation, providing more robust cross-platform compatibility with long term maintainability in mind:

```
# Saving the trained model, scaler, and encoder
model.save(multiclassModelPath)
with open(multiclassScalerPath, "wb") as f:
    pickle.dump(scaler, f)
with open(labelEncPath, "wb") as f:
    pickle.dump(label_encoder, f)
```

- Approach ensures integration with real-time deployment environments without concern for compatibility found with the previous versions .h5 format
- Facilitates updates and enhancements due to the widely supported SavedModel format, thus significantly improving the operational longevity of the system

Phase 3: Synthetic Medical Database

The synthetic medical database was retained from Version 1. It continues to provide as an essential component for comprehensive ethical testing within the virtual environment. The database closely mimics realistic healthcare data scenarios through structured tables representing patients, visits, diagnoses and medications. The maintained structure provided a stable environment, crucial for testing the IDPS and validating the improved DNN model's performance.

Phase 4: Firewall Integration

Phase 4 of the DNN IDPS development concentrated on embedding firewall rules that enable automatic real-time responses to identified cyber threats. The system integration provides strong support for the Intrusion Detection and Prevention System (IDPS) enabling firewalls to automatically adjust their settings to block detected threats as soon as advanced DNN classifiers identify them without requiring human input.

Real-time network traffic monitoring played an essential role during this phase and utilised PyShark which functions as an effective Python interface for Wireshark. PyShark was chosen as it provides comprehensive packet-level data capture functionality from network interfaces. Through PyShark's ability to track inbound and outbound network traffic it enabled quick packet processing which minimised the time between threat discovery and firewall action. The capability for instant analysis played a crucial role in greatly reducing the vulnerability window during cyber-attack incidents. [See

8.5.20] This approach provided several key benefits:

- Enabled immediate detection of threats as packets traverse the chosen network interface
- Ensures minimal latency between live packet capture and threat mitigation actions, critical for effective live response
- Utilisation of a widely supported tool (Wireshark via PyShark), which is reliable, thoroughly tested and robust for use in network traffic analysis

The real-time packet capture required a detailed technical process for feature extraction because it needed to match the criteria used by deep neural networks which were trained during previous stages. The procedure required an exact extraction of network packet attributes including the IP header length, protocol identification, TCP flags (FIN, SYN, RST, PSH, ACK, ECE, CWR) and the entire packet length. Accurate feature extraction remained essential because any mismatches between training and operational features could lead to significant drops in model performance. The extraction process for each attribute of network packets converted them into numerical arrays which matched the existing training dataset schema. [See code excerpt 8.5.21]

The classification stage was performed as a hierarchical two tier process for optimal computational load and accuracy in classification. To begin with, packets were classified in two classes only, as normal and attack packets based on a probability of malice. Any packet that was suspected to be malicious was subjected to a detailed multiclass classification in order to determine the exact kind of cyber threat. This is because the layered classification scheme disclosed in this paper disposed the expensive computational tasks only to real suspicious traffic in order to enhance the system's efficiency and the time to respond. [See code excerpt 8.5.22]

Finally, the integration with Windows Advanced Firewall rules was added to nearly immediately counter identified threats. This component automatically updated the systems firewall rules to block IP addresses contributing to malicious network traffic via the command line 'utility netsh'. Automation of the firewall management was essential in achieving near instantaneous threat detection and mitigation. By dynamically adding these rules, the system can not only detect malicious activity but also isolate these threats and prevent further malicious behaviour.

```
def blockIp(ip_address):  
    try:  
        command = f"netsh advfirewall firewall add rule name='Block IP {ip_address}' protocol=TCP  
dir=in remoteip={ip_address} action=block"  
        subprocess.run(command, shell=True)  
        print(f"Blocked IP Address: {ip_address}")  
    except Exception as e:  
        print(f"Error blocking IP address {ip_address}: {e}")
```

Compared to Version 1, Version 2 of the DNN-based IDPS system offers significantly improved performance, reliability, and operational robustness. Data preprocessing, improved machine learning models, and live firewall integration were used to address the limitations of Version 1. A comprehensive feature alignment, newly filtered datasets, and adaptive training techniques have collectively contributed to the development of an effective cybersecurity solution suitable for deployment in

healthcare environment.

5.3 Testing Implementation & Set Up

To evaluate the effectiveness of the Deep Neural Network Intrusion Detection and Prevention System (DNN IDPS), a controlled Windows virtual testing environment was established using Oracle VirtualBox 7.16. This set up ensures safe and realistic cyberattack simulations against a hospital server whilst analysing the DNN IDPS's performance in real time.

5.3.1 Virtualized Environment Setup

The test bed consists of two Virtual Machines (VM's), designed to simulate a hospital server and an attackers machine

Virtual Machine 1, HospitalServer:

OS: Windows 11

CPU Cores: 2

RAM: 4GB

Storage: 40GB Virtual SSD

Network Mode: Internal Network – ('Hospital Network', IP: 192.168.1.11 [Static IPv4])

Installed Software:

- DNN_IDPS Software: real time live network monitoring and attack prevention
- Synthetic Medical Database: Simulated hospital records
- Wireshark & PyShark: Live packet analysis
- Windows Defender Firewall: Integrated with IDPS for blocking malicious Ips

Virtual Machine 2, AttackHost:

OS: Kali Linux

CPU Cores: 2

RAM: 4GB

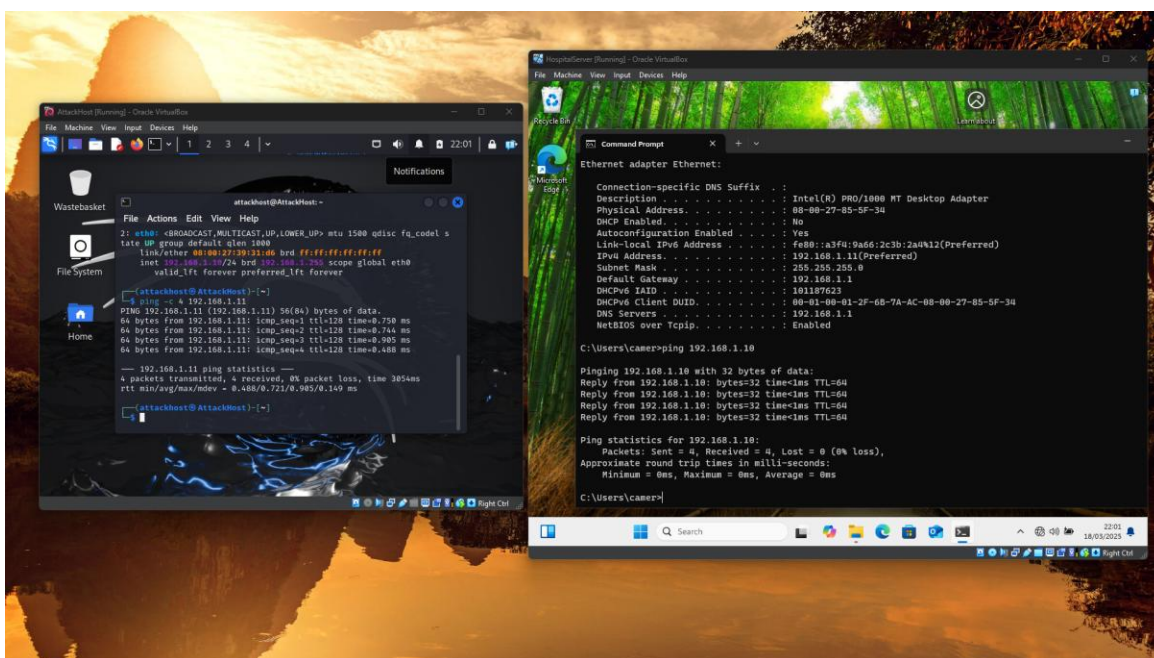
Storage: 40GB Virtual SSD

Network Mode: Internal Network – ('HospitalNetwork', IP: 192.168.1.10 [Static IPv4])

Installed Attack Tools:

- Hping3: DoS/DDoS attacks
- Metasploit Framework (Reconnaissance & exploits)
- Nmap (port scanning & OS fingerprinting)
- Ettercap (ARP spoofing & MITM Attacks)
- MQTT Pwn (MQTT-based attacks)

The Hospital Server and Attack Host communicate via an internal network, simulating an environment where an attacker has gained access to a hospital wireless network. This ensures attack traffic is isolated from external networks.



[Fig5: Virtual Environment Set Up, Proof of Internal Isolated Network]

In the image above, we can see the Kali Linux VM: 192.168.1.11 sending a ping request to the Windows 11 VM: 192.168.1.10. This demonstrates the internal network has been set up correctly and is isolated from any other network environment.

The details of the simulated attack/testing plan are detailed on the page below.

5.3.2 Attack Simulation Plan

The DNN IDPS is tested against 5 major attack categories, covering: DoS, DDoS, Reconnaissance, Spoofing, and MQTT-based Attacks

The below tables identify the variations in attacks, the tools which will be utilised to orchestrate them, a description of the attack and the expected system response.

Attack	Tool	Description	Expected System Response
SYN Flood	Hping3	Overwhelms server with half-open TCP connections	IDPS detects traffic spike, blocks host IP

TCP Flood	Hping3	Sends excessive TCP packets to exhaust resources	IDPS blocks flooding IP
ICMP Flood	Hping3	Overloads server with ping requests	IDPS detects high ICMP traffic and blocks source IP
UDP Flood	Scapy	Sends large amounts of UDP packets	IDPS Blocks source IP of excessive UDP traffic

Distributed Denial of Service (DDoS) Attacks

Denial of Service (DoS) Attacks

Attack	Tool	Description	Expected System Response
SYN Flood	Hping3	Overwhelms server with half-open TCP connections	IDPS detects traffic spike, blocks host IP
TCP Flood	Hping3	Sends excessive TCP packets to exhaust resources	IDPS blocks flooding IP
ICMP Flood	Hping3	Overloads server with ping requests	IDPS detects high ICMP traffic and blocks source IP
UDP Flood	Scapy	Sends large amounts of UDP packets	IDPS Blocks source IP of excessive UDP traffic

Spoofing Attacks

Attack	Tool	Description	Expected response
ARP Spoofing	Ettercap	Intercepts hospital network packets	IDPS detects spoofing attempts and blocks source IP

Reconnaissance Attacks

Attack	Tool	Description	Expected System Response
Ping Sweep	Nmap	Identifies active hosts via IP pings	IDPS detects abnormal pings and blocks source IP
Vulnerability Scan	Metasploit	Scans for known security weaknesses	IDPS flags for unauthorized scans
OS Scan	Nmap	Identifies OS via TCP/IP fingerprinting	IDPS detects scan patterns and blocks source IP
Port Scan	Nmap	Detects open network ports	IDPS Blocks source IP detected scan attempts

MQTT-Based Attacks

Attack	Tool	Description	Expected Response
Malformed Data Injection	MQTT Pwn	Sends corrupted MQTT Packets	IDPS detects and blocks IP
DoS Connect Flood	MQTT Pwn	Overloads broker with connection requests	IDPS detects and blocks IP
DDoS Connect Flood	MQTT Pwn	Distributed attack on MQTT service	IDPS detects and blocks IP
DoS Publish Flood	MQTT Pwn	Exhausts server resources through publishing	IDPS detects and blocks IP
DDoS Publish Flood	MQTT Pwn	Distributed publish attack	IDPS detects and blocks IP

6 RESULTS ANALYSIS

6.1 Evaluation Metrics

The evaluation of the Deep Neural Network based Intrusion Detection and Prevention system is primarily based on accuracy as the key performance metric. Although others have been used in the development process (i.e. F1-Score, Value Loss, etc.). The accuracy metric is calculated by dividing the number of correct predictions made by the system divided by the total number of predictions that the system has made

The following metrics are used to assess the models performance:

- TP – True Positive: The number of correctly identified malicious actions by the system
- TN – True Negative: The number of correctly identified benign actions by the system

- FP – False Positive: The model incorrectly labels benign traffic as malicious
- FN – False Negative: The model fails to identify true malicious traffic

The accuracy formula is represented as:

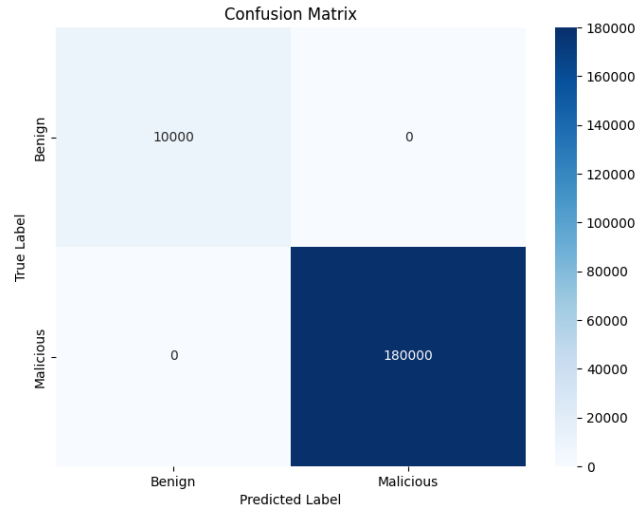
$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

Other evaluation metrics such as precision, recall, and F1-Score will also be considered to provide a further understanding of the system performance.

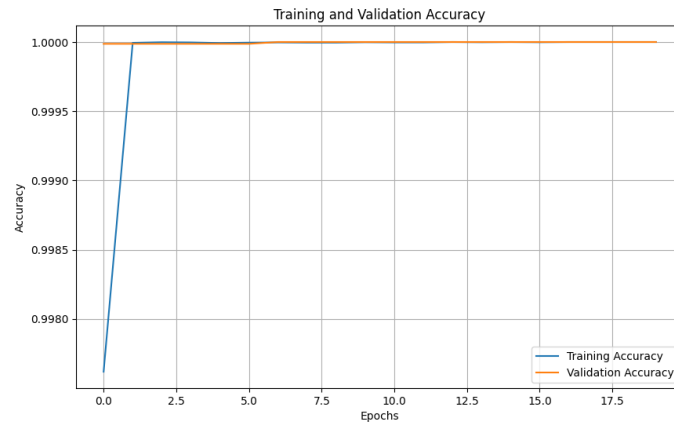
6.2 Binary Classifier

Classification Report:				
	precision	recall	f1-score	support
Benign	1.00	1.00	1.00	10000
Malicious	1.00	1.00	1.00	180000
accuracy			1.00	190000
macro avg	1.00	1.00	1.00	190000
weighted avg	1.00	1.00	1.00	190000

[Fig6: Binary Classifier Classification Report]



[Fig7: Binary Classifier Confusion Matrix]



[Fig8: Binary Classifier Training Vs. Validation Accuracy Graph]

The binary classifier model was trained and evaluated on the `binary_filtered_dataset.csv`, comprising of 10,000 benign samples and 180,000 malicious samples. The following evaluation metrics have been observed:

Classification Report

- Precision: The classifier made no incorrect positive predictions, correctly identifying all malicious samples within the dataset
- Recall: The classifier accurately detected all instances of both benign and malicious samples.
- F1-Score: The harmonic mean of precision and recall is also 1.00, confirming the model's capability of achieving ideal performance.
- Accuracy: The overall accuracy of the model is observed as 1.00 (100%)
- Macro Avg & Weighted Avg: Both again are 1.00 (100%), indicating the binary classifiers capability of perfectly predicting network traffic over both classes

Confusion Matrix:

The confusion matrix provided further supports the findings above:

- True Positives (Malicious traffic detected correctly): 180,000
- True Negatives (Benign traffic detected correctly): 10,000
- False Positives: 0
- False Negative: 0

The model can successfully distinguish between benign and malicious network traffic without making any errors. The perfect classification rate demonstrates the model's robustness in handling binary classification when distinguishing the intent of network packets.

Training and Validation Accuracy:

The training and validation accuracy graph indicates:

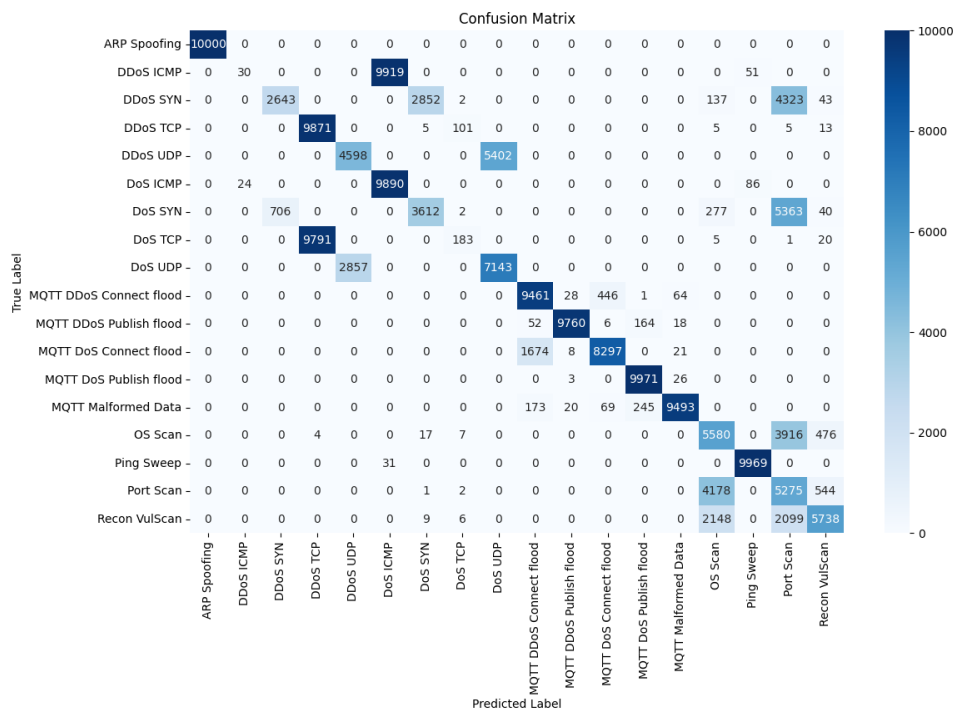
- The model achieves near-perfect accuracy within the first few epochs and maintains it throughout the training process.
- Both training and validation accuracy curves are very closely aligned. This indicates excellent generalisation with zero signs of overfitting.

The binary classifiers perfect performance can be attributed to the clear distinction between both malicious and benign network traffic within the filtered dataset. Its ability to achieve 100% accuracy, precision, recall, and F1-score demonstrates the classifiers effectiveness in detecting malicious activity in a binary classification setting. This performance is highly desirable for detecting generic intrusion attempts however it does not account for identifying specific attack types which will be evaluated in the multi-class classification task.

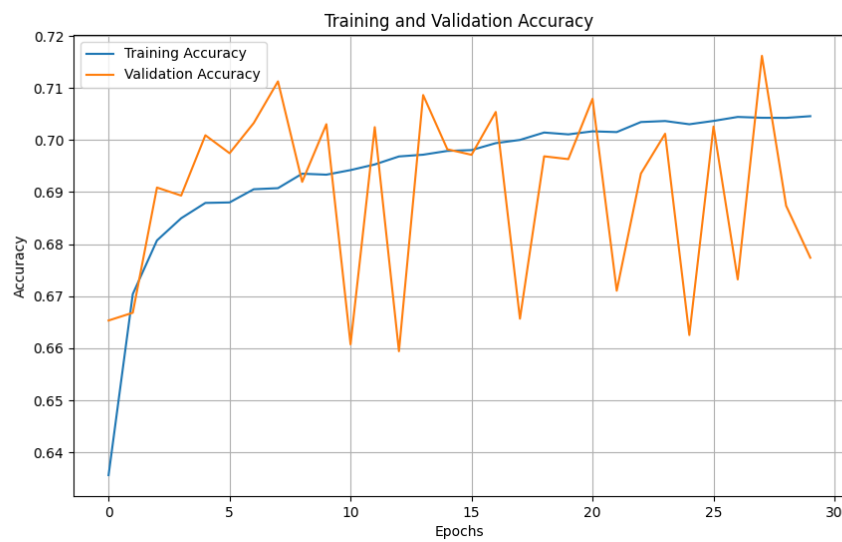
6.3 Multi-Class Classification Model

Classification Report:				
	precision	recall	f1-score	support
ARP Spoofing	1.00	1.00	1.00	10000
DDoS ICMP	0.56	0.00	0.01	10000
DDoS SYN	0.79	0.26	0.40	10000
DDoS TCP	0.50	0.99	0.67	10000
DDoS UDP	0.62	0.46	0.53	10000
DoS ICMP	0.50	0.99	0.66	10000
DoS SYN	0.56	0.36	0.44	10000
DoS TCP	0.60	0.02	0.04	10000
DoS UDP	0.57	0.71	0.63	10000
MQTT DDoS Connect flood	0.83	0.95	0.89	10000
MQTT DDoS Publish flood	0.99	0.98	0.98	10000
MQTT DoS Connect flood	0.94	0.83	0.88	10000
MQTT DoS Publish flood	0.96	1.00	0.98	10000
MQTT Malformed Data	0.99	0.95	0.97	10000
OS Scan	0.45	0.56	0.50	10000
Ping Sweep	0.99	1.00	0.99	10000
Port Scan	0.25	0.53	0.34	10000
Recon VulScan	0.83	0.57	0.68	10000
accuracy			0.68	180000
macro avg	0.72	0.68	0.64	180000
weighted avg	0.72	0.68	0.64	180000

[Fig9: Multi Class Classification Report]



[Fig10: Multi Class Confusion Matrix]



[Fig11: Multi Class Training vs. Validation Accuracy]

The multi-class classifier was trained and evaluated on the filtered dataset comprising of 18 specific attack types, denoted by `attack_category`, each with 10,000 network traffic samples for a total of 180,000 samples across the combined dataset

Classification Report:

- Overall Accuracy: The model achieved an overall accuracy of 0.68 (68%)
- Macro Avg: Precision – 0.72, Recall – 0.68, F1-Score – 0.64
- Weighted Avg: Precision - 0.72, Recall - 0.68, F1-Score - 0.64.
- While some attack types such as ARP Spoofing (Precision, Recall, F1-Score: 1.00) and Ping Sweep (Precision: 0.99, Recall: 0.97, F1-Score: 0.98) are perfectly or near-perfectly detected, other classes such as Port Scan (Precision: 0.25, Recall: 0.53, F1-Score: 0.34) and OS Scan (Precision: 0.45, Recall: 0.56, F1-Score: 0.50) have poor performance
- The variability in performance across classes highlights difficulty in detecting certain types of attacks, particularly reconnaissance-related ones.

Confusion Matrix Analysis:

The confusion matrix analysis shows varying levels of performance across classes, with significant misclassification between the following Attack types:

- Reconnaissance attacks (OS Scan, Port Scan, VulScan)
- DDoS and DoS attacks with similar packet structures (e.g., SYN, TCP, UDP floods).
- Some MQTT-based attacks are better identified than others, with MQTT Publish Flood and MQTT Malformed Data showing strong detection rates.

Classes like ARP Spoofing are correctly classified with no confusion; however, it is observed that certain attack types, especially those with similar network traffic patterns, are heavily misclassified.

Training and Validation Accuracy:

Observed in the provided graph above we see the model's training and validation accuracy both stabilise at around 0.70 (70%). The validation accuracy fluctuates significantly throughout training, indicating a level of overfitting or instability when generalising unseen data. Despite this fluctuation, the overall trend shows gradual improvement for the model, suggesting the model is capable of learning relevant network traffic features from the filtered dataset

The multi-class classifier's performance is significantly lower than that of the binary classifier (Benign vs. Malicious), which is to be expected to some degree based on the added complexity of differentiating specific attack types. Whilst some classes are perfectly detected, others do suffer from poor recall and precision scores. The confusion matrix provides further insight into the overlap of certain categories, specifically DoS/DDoS based attacks as well as reconnaissance categories. Further improvements are needed to increase the model's robustness in handling diverse attack types, these potential solutions could include:

- Further fine tuning of the model's architecture and hyperparameters
- Using data augmentation to improve balancing methods within the filtered dataset
- Implementing further feature engineering to improve distinguishability against the lacking attack signatures

6.4 Testing within Virtual Environment

To validate the effectiveness of these models in conjunction with the added firewall rules, a series of controlled experiments were conducted within the simulated hospital network environment, detailed in 4.3. This environment, built using Oracle VirtualBox 7.16, consisted of two Virtual Machines assigned to their own isolated network.

Simulated Attack Plan:

The IDPS system was tested against five major attack categories: DoS, DDoS, Reconnaissance, Spoofing, MQTT-based attacks. Each form of attack was conducted 5 times in line with section 4.3. The results were recorded as follows:

Distributed Denial of Service (DDoS) & Denial of Service (DoS) Attacks:

Tools utilised: Hping3, Scapy

Attack Types: SYN flood, TCP flood, ICMP flood, UDP flood.

Results:

- The DNN IDPS system successfully detected and blocked 23 of the 25 conducted attack attempts across all four attack types
- Average detection rate: 92%

Common Failure points: Some UDP Flood attempts went undetected, likely due to atypical traffic patterns when observing live packet data or packet fragmentation

Note: Although the multi-class classifier achieved variable performance, the high detection rate it attributed to the success of the binary classifier in detecting benign versus malicious network traffic

Reconnaissance Attacks:

Tools Used: Nmap, Metasploit

Attack Types: Ping Sweep, Vulnerability Scan, OS Scan, Port Scan

Results:

- The IDPS detected 18 out of 20 attempts successfully.
- Average Detection Rate: 90%.
- OS Scan and Port Scan: Lower detection rates of 50-70%, attributed to similarities in legitimate network traffic and attack traffic.
- Note: Despite the low performance of these attack types in the multi-class classification report, the high success rate in blocking attempts was due to the binary classifier effectively flagging the traffic as malicious.

Spoofing Attacks:

Tools Used: Ettercap

Attack Types: ARP Spoofing

Results:

- The IDPS identified and blocked all five ARP Spoofing attempts.
- Detection Rate: 100%.
- Note: The high detection rate of this attack type is consistent with the results obtained in both the binary and multi-class classifiers.

MQTT-Based Attacks:

Tools Used: MQTT Pwn

Attack Types: Malformed Data Injection, DoS Connect Flood, DDoS Connect Flood, DoS Publish Flood, DDoS Publish Flood

Results:

- The IDPS detected and blocked attacks in 22 out of 25 attempts.
- Average Detection Rate: 88%.
- Failure Points: Certain malformed data injections bypassed detection due to packet size and formatting issues.
- Note: The high detection rate is largely due to the effectiveness of the binary classifier, which accurately identified malicious activity despite lower performance from the multi-class model.

Evaluation Summary:

The result of the virtual environment simulation testing reveals that the DNN IDPS performs well against varying types of attacks, particularly against DoS/DDoS, Spoofing, and MQTT-based attacks.

The multi-class classifier performs satisfactorily despite having lower accuracy than the binary classifier, which performs very well in distinguishing between malicious and benign traffic.

Improvement areas involve further improved model performance in Reconnaissance attack detection, particularly OS Scans and Port Scans. Additional training data, improved model structures, or domain-specific techniques for identifying harder to detect traffic patterns may lead to improved results. Overall performance reflects the usability and effectiveness of the model for real-world application in a hospital network environment.

7 CONCLUSION

The findings of this dissertation highlight the potential of Deep Neural Networks (DNNs) in strengthening the detection and prevention of cyber-attacks targeting medical databases. By integrating a DNN-based Intrusion Detection and Prevention System (IDPS) with conventional firewall mechanisms, the proposed solution presents a robust approach to safeguarding sensitive medical data. The binary classifier achieved remarkable accuracy, recording a perfect detection rate of 100% across both benign and malicious traffic samples. Such high performance underscores the model's capacity to distinguish legitimate network traffic from potentially harmful data — a critical feature for any practical intrusion detection system. Additionally, the system's ability to integrate with firewall rules and dynamically block malicious IP addresses in real-time further demonstrates its applicability for real-world deployment.

Despite this success, the multi-class classifier designed to differentiate between various attack types exhibited comparatively lower performance. Achieving an accuracy rate of approximately 68%, it

effectively detected some attacks, such as ARP Spoofing and Ping Sweep, with near-perfect accuracy. However, its ability to identify other categories, particularly reconnaissance attacks involving OS Scans and Port Scans, was noticeably limited. This discrepancy reveals the inherent difficulty in distinguishing between attack types that produce similar network traffic patterns. Testing within the simulated virtual environment provided valuable insights into the practical efficacy of the system. The high success rate in blocking malicious traffic can be primarily attributed to the efficiency of the binary classifier, which accurately identified and blocked IP addresses associated with attacks. Notably, this indicates that while the multi-class classifier requires further optimisation, the binary classifier alone offers substantial protection against a broad range of threats. The experimentations further revealed that attack types identified effectively by the multi-class classifier were accurately detected and blocked during virtual environment testing. In essence, the strong performance of the binary classifier compensated for the multi-class classifier's limitations, enhancing the overall effectiveness of the IDPS. Future improvements to this system should prioritise enhancing the performance of the multi-class classifier. This could be achieved through advanced feature engineering, expanding dataset diversity, and employing alternative neural network architectures better suited to differentiating between challenging attack types. Furthermore, incorporating reinforcement learning or ensemble learning techniques may prove beneficial in improving classification accuracy across all categories. In conclusion, this research has successfully established a functional DNN-based IDPS capable of detecting and mitigating cyber threats in medical environments. While certain areas still require improvement, the system's performance across both simulated and real-time testing environments demonstrates its potential for deployment within healthcare infrastructures to enhance the protection of sensitive medical data.

8 APPENDIX

8.1 REFERENCES

- [1] Lippmann, Richard, et al. "The 1999 DARPA Off-Line Intrusion Detection Evaluation." *Computer Networks*, vol. 34, no. 4, Oct. 2000, pp. 579–595, wenke.gtisc.gatech.edu/ids-readings/1998_eval_discex00_paper.pdf, [https://doi.org/10.1016/s1389-1286\(00\)00139-0](https://doi.org/10.1016/s1389-1286(00)00139-0).
- [2] Burgess, Jonah. "Modern DDoS Attacks and Defences -- Survey." *ArXiv.org*, 2022, arxiv.org/abs/2211.15404. Accessed 20 Nov. 2024.
- [3] Conti, Mauro, et al. "A Survey of Man in the Middle Attacks." *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, 2016, pp. 2027–2051, [orbit.dtu.dk/en/publications/a-survey-of-man-in-the-middle-attacks\(6f5d4b72-8258-4747-a08c-1c2f1b31d85e\).html](http://orbit.dtu.dk/en/publications/a-survey-of-man-in-the-middle-attacks(6f5d4b72-8258-4747-a08c-1c2f1b31d85e).html), <https://doi.org/10.1109/comst.2016.2548426>. Accessed 21 Oct. 2019.
- [4] Saed, Muhanna, and Ahamed Aljuhani. "Detection of Man in the Middle Attack Using Machine Learning." *IEEE Xplore*, 1 Jan. 2022, ieeexplore.ieee.org/document/9711555. Accessed 10 Aug. 2022.
- [5] Ferdous, Jannatul , et al. "AI-Based Ransomware Detection: A Comprehensive Review." *Ieee.org*, 2017, ieeexplore.ieee.org/document/10681072?, <https://doi.org/10.1109/ACCESS.2024.3461965>. Accessed 22 Sept. 2024.
- [6] Ghafur, S., et al. "A Retrospective Impact Analysis of the WannaCry Cyberattack on the NHS." *Npj Digital Medicine*, vol. 2, no. 1, 2 Oct. 2019.

- [7] Vasko , Josh , and Carrie Gardner. "Insider Threats in Healthcare (Part 7 of 9: Insider Threats across Industry Sectors)." SEI Blog, 21 Feb. 2019, insights.sei.cmu.edu/blog/insider-threats-in-healthcare-part-7-of-9-insider-threats-across-industry-sectors/?. Accessed 20 Nov. 2024.
- [8] Bishop, Matthew, et al. "The Insider Threat in Healthcare: Risk Factors and Mitigation Strategies." *Journal of Healthcare Information Management*, vol. 39, no. 4, 2021, pp. 272–285, <https://www.hhs.gov/sites/default/files/insider-threats-in-healthcare.pdf>. Accessed 20 Nov. 2024..
- [9] Okesola, J O, et al. "Securing Web Applications against SQL Injection Attacks - a Parameterised Query Perspective)." *Securing Web Applications against SQL Injection Attacks - a Parameterised Query Perspective*, 5 Apr. 2023, <https://doi.org/10.1109/seb-sdg57117.2023.10124613>. Accessed 4 Jan. 2024.
- [10] Rihab Bouafia, et al. "Automatic Protection of Web Applications against SQL Injections: An Approach Based on Acunetix, Burp Suite and SQLMAP." *Automatic Protection of Web Applications against SQL Injections: An Approach Based on Acunetix, Burp Suite and SQLMAP*, 5 Oct. 2023, <https://doi.org/10.1109/icoa58279.2023.10308827>. Accessed 28 Mar. 2024.
- [11] Zachos, Georgios, et al. "An Anomaly-Based Intrusion Detection System for Internet of Medical Things Networks." *Electronics*, vol. 10, no. 21, 20 Oct. 2021, p. 2562, <https://doi.org/10.3390/electronics10212562>. Accessed 15 Nov. 2021.
- [12] Nguyen, Thanh Thi, and Vijay Janapa Reddi. "Deep Reinforcement Learning for Cyber Security." *IEEE Transactions on Neural Networks and Learning Systems*, 2021, pp. 1–17, <https://doi.org/10.1109/tnnls.2021.3121870>. Accessed 20 Nov. 2021.
- [13] Anwar, Raja Waseem, et al. "Firewall Best Practices for Securing Smart Healthcare Environment: A Review." *Applied Sciences*, vol. 11, no. 19, 2 Oct. 2021, p. 9183, www.mdpi.com/2076-3417/11/19/9183/htm.
- [14] S. Arunprasath, and Suresh Annamalai. "Improving Patient Centric Data Retrieval and Cyber Security in Healthcare: Privacy Preserving Solutions for a Secure Future." *Multimedia Tools and Applications*, 31 Jan. 2024, <https://doi.org/10.1007/s11042-024-18253-5>.
- [15] Krunal Manilal. "The Role of AI in Detecting Insider Threats in Healthcare Organizations." *International Journal of Computer Applications & Information Technology*, vol. 7, no. 2, 21 Oct. 2024, pp. 239–248, www.researchgate.net/publication/385449097_The_Role_of_AI_in_Detecting_Insider_Threats_in_Healthcare_Organizations, <https://doi.org/10.5281/zenodo.13960643>.
- [16] Wahab, Fazal, et al. "An AI-Driven Hybrid Framework for Intrusion Detection in IoT-Enabled E-Health." *Computational Intelligence and Neuroscience*, vol. 2022, 21 Aug. 2022, pp. 1–11, <https://doi.org/10.1155/2022/6096289>. Accessed 14 Nov. 2022.
- [17] Al-Mhiqani, Mohammed Nasser, et al. "A Review of Insider Threat Detection: Classification, Machine Learning Techniques, Datasets, Open Challenges, and Recommendations." *Applied Sciences*, vol. 10, no. 15, 28 July 2020, p. 5208, www.mdpi.com/2076-3417/10/15/5208, <https://doi.org/10.3390/app10155208>.

8.2 Professional, Legal, Ethical and Social (PLES) Issues

This section outlines potential issues related to the development and deployment of this project, along with the steps taken to address them to ensure adherence to professional, legal, ethical, and social standards.

8.2.1 Legal and Professional Issues

Libraries used in the project are publicly available open-source libraries and use of code is within the limits of licensing terms and standards. All libraries, tools and methodologies will be well documented alongside all datasets used for training/testing, such as the CIC IoMT 2024 dataset which clearly state permission for academic use. Thus, there are no legal issues because the project is for academic purposes only. If the system were to commence commercial operation, then licensing and data use agreements would need to be re-negotiated consistent with the standards for commercial software. This ensures that all academic integrity is maintained as a result of citation on any research papers and external sources referenced in the development process. The code and documentation will be released to a public GitHub repository with an appropriate open-source license supporting transparency, potential collaboration, peer review, and improvements.

8.2.2 Ethical and Social Issues

This project involves no human testing or research and no PII is handled throughout all stages of development, testing or deployment, so there are few ethical concerns. A. This dataset, composed of synthetically generated network traffic and simulating the real-world scenario, is used during model training and evaluation steps while ensuring no actual patient/user information has been compromised. Such synthetic data solves the problem of privacy and bias that typically occurs with real-world personal data bias in the used synthetic dataset itself, largely containing types of attack that would be expected in medical network traffic. However, since research on this project is focused on network data and not demographic measures, it weighs away at larger societal biases.

As the project progresses and if this system starts analysing real-time network traffic from live environments, privacy and ethical implications will become more prevalent. The utilisation of open source as a part of the project mitigates this because it ensures that no one besides authorised third-party parties can see the data. Furthermore, any real time use of the model would require to be ran on a local server/device so that there are no data sent to third party servers which also guarantees data privacy.

8.3 Project Management

8.3.1 Risk Form

Risk	Impact	Probability
Illness or Personal Issues	Delays project milestones, impacting development time and reducing focus, which may result in inadequate testing or rushed implementation.	Medium
Data Privacy Breach	Unauthorised access or exposure of sensitive data and	Low

	information may lead to ethical and legal violations, potentially halting the project	
Dataset Complications	Dataset may remove access or lack diverse simulated scenarios which in turn may reduce the DNN's accuracy, leading to a less effective IDPS	Medium
Overfitting in DNN	Overfitting could cause the model to perform well on training data but poorly on real data, affecting the IDPS's accuracy and reliability in production.	Medium
Software or Library Compatibility	Issues with library compatibility (e.g., TensorFlow, Scikit-learn) could require additional debugging, impacting time for other development activities.	Medium
Hardware Limitations	Insufficient hardware resources may slow model training, especially for DNNs, extending the project timeline or limiting model complexity.	High
Model Convergence Issues	Training instability or failure to converge during DNN training may lead to extended training times and potential model inefficacy.	Medium
High False Positives	Frequent false positives may reduce the system's usability, causing alarm fatigue and risking the IDPS's credibility with end-users.	Medium
Software Bugs or Code Errors	Bugs in code or incorrect algorithm implementation can lead to project delays, compromised functionality, or incorrect model behaviour.	High
Loss of Data or Corrupted Files	Data loss due to system failure or file corruption may require retraining the model from scratch, delaying project progress.	Medium
Ineffective Model Tuning	Failure to properly tune model parameters (e.g., learning rate, batch size) could reduce detection accuracy, compromising project success.	Medium
Model Updating & Retraining Issues	Difficulty in updating the model with new attack data may reduce its effectiveness against emerging threats, leading to a less robust IDPS.	Low

8.3.2 Mitigation Strategy

This section identifies key risks associated with the development and deployment of the Intrusion Detection and Prevention System (IDPS) and outlines strategies to mitigate their potential impact.

1. Illness or Personal Issues

Mitigation: To reduce the impact of unexpected delays, a project buffer timeline is included, allowing for flexibility in milestone completion. Additionally, regular progress tracking and

early completion of key tasks minimise the effects of unforeseen absences.

2. Data Privacy Breach

Mitigation: The project exclusively uses synthetic and publicly accessible datasets that do not contain personally identifiable information, lowering privacy risks. Data handling protocols align with GDPR requirements to ensure compliance and ethical data usage.

3. Dataset Complications

Mitigation: Alternative datasets with similar attack patterns, such as CICIDS2017, are identified to supplement data if access issues arise. Additionally, synthetic data generation tools are considered to create diverse training scenarios, maintaining the DNN's accuracy and robustness.

4. Overfitting in DNN

Mitigation: Regular validation testing with cross-validation and dropout layers in the model's architecture help prevent overfitting. Data augmentation techniques may also be applied to improve generalisation on unseen data.

5. Software or Library Compatibility

Mitigation: Compatibility testing is conducted early in the development process to identify any issues with key libraries like TensorFlow and Scikit-learn. This reduces the risk of last-minute conflicts that could affect project timelines.

6. Hardware Limitations

Mitigation: Cloud-based resources and computational clusters are considered to augment local hardware limitations. Additionally, model complexity can be adjusted by tuning hyperparameters to balance performance and training time within hardware constraints.

7. Model Convergence Issues

Mitigation: Advanced optimisers like Adam and adaptive learning rates are used to improve convergence stability. Hyperparameter tuning and regular monitoring of training logs further help identify and address convergence issues early in the training process.

8. High False Positives

Mitigation: The model is fine-tuned using evaluation metrics such as precision and recall minimising false positives. A feedback loop with healthcare professionals could also help in fine-tuning the model's thresholds in real-world deployments.

9. Software Bugs or Code Errors

Mitigation: A structured testing and debugging plan, including unit tests, integration tests, and code reviews, is established to identify and fix bugs promptly. Using version control systems like Git helps in tracking and managing code effectively.

10. Loss of Data or Corrupted Files

Mitigation: Regular backups of datasets, models, and code are scheduled to prevent data loss.

Important files are stored in secure cloud storage, ensuring redundancy and quick recovery in case of data corruption.

11. Ineffective Model Tuning

Mitigation: A systematic approach to hyperparameter tuning, including grid search and random search methods, is used to identify optimal model settings. Validation testing ensures that tuning changes improve model performance.

12. Model Updating & Retraining Issues

Mitigation: The system is designed with modularity, allowing for straightforward updates to the model when new data or attack patterns emerge. Scheduled retraining sessions will be implemented to ensure the model remains effective against new threats.

8.4 Schedule

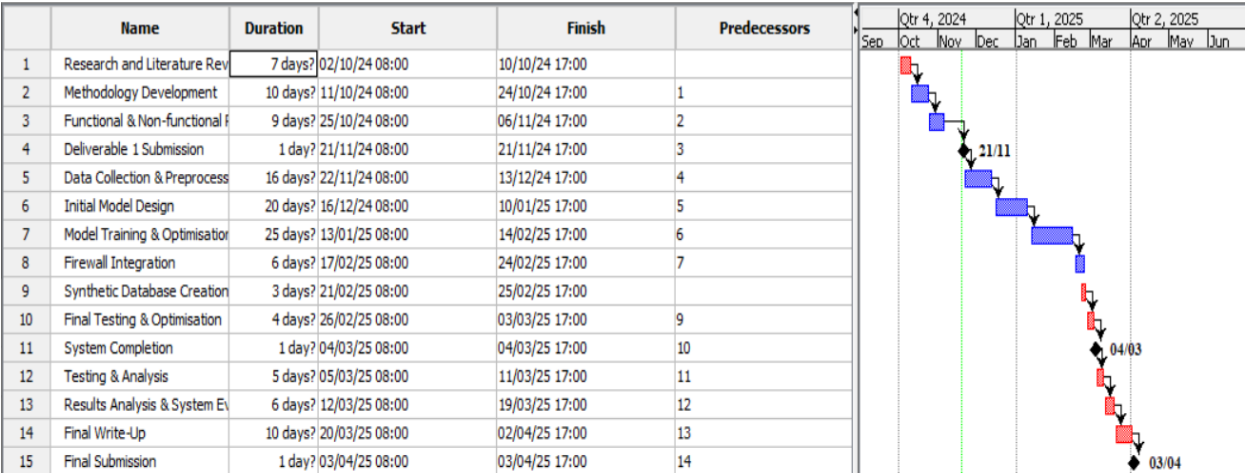
The Gantt chart found below outlines the timeline and key milestones for the project, providing a visual representation of both task dependencies and deadlines across both semesters.

The project has been broken into 3 sections for effective completion:

Deliverable 1, which includes Research and Literacy Review, Methodology Development, Functional & Non-functional requirements, and the final Deliverable 1 Submission (21st November 2024)

System Development, which includes Data Collection & Preprocessing, Initial Model Design (DNN Design & Implementation), Model Training & Optimisation, Firewall Integration, Synthetic Database Creation, Final Testing & Optimisation and the final milestone of System Completion.

Final Dissertation, which includes the following tasks for the final submission for this project, Testing & Analysis, Results Analysis and System Evaluation, Final Write-up and the Final Submission.



[Fig.12 Gantt chart depicting key milestones, dependencies and deadlines for the project]

8.5 Code Snippets and Scripts

8.5.1

```
allNumericCols = set()
for file in files:
    df = pd.read_csv(file)
    numericCols = df.select_dtypes(include=["float64", "int64"]).columns
    allNumericCols.update(numericCols)
numericColumns = list(allNumericCols)
```

8.5.2

```
scaler = MinMaxScaler()
allData = pd.concat([pd.read_csv(f)[numericColumns] for f in files], ignore_index=True)
scaler.fit(allData.fillna(allData.mean()))
```

8.5.3

```
combinedDataFrame = pd.concat(combinedData, ignore_index=True)
combinedDataFrame.to_csv(outputFile, index=False)
print(f"Combined dataset saved to: {outputFile}")
```

8.5.4

```
model = keras.Sequential([
    keras.layers.Dense(128, activation="relu", input_shape=(input_dim,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dense(numClasses, activation="softmax")
])
```

8.5.5

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
```


8.5.6

```
history = model.fit(xTrain, yTrain,
                    epochs=20,
                    batch_size=32,
                    validation_data=(xTest, yTest),
                    callbacks=[earlyStop],
                    verbose=1)
```

8.5.7

```
estLoss, testAcc = model.evaluate(xTest, yTest, verbose=0)
print(f"Test Accuracy: {testAcc:.4f}")
predictions = model.predict(xTest)
predictedLabels = np.argmax(predictions, axis=1)
accuracyVal = accuracy_score(yTest, predictedLabels)
precisionVal = precision_score(yTest, predictedLabels, average="weighted")
recallVal = recall_score(yTest, predictedLabels, average="weighted")
f1Val = f1_score(yTest, predictedLabels, average="weighted")
```

8.5.8

```
model.save("DNN_attack_classifier.h5")
with open("training_scaler.pkl", "wb") as file:
    pickle.dump(scaler, file)
with open("label_encoder.pkl", "wb") as file:
    pickle.dump(label_encoder, file)
```

8.5.9

```
from faker import Faker
import sqlite3
import random
fake = Faker()
conn = sqlite3.connect("synthetic_medical.db")
cursor = conn.cursor()
# Create the Patients table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Patients (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT,
```

```

        last_name TEXT,
        date_of_birth DATE,
        gender TEXT
    )
    ""
for _ in range(50):
    cursor.execute("INSERT INTO Patients (first_name, last_name, date_of_birth, gender) VALUES (?, ?, ?, ?)",
                    (fake.first_name(), fake.last_name(), fake.date_of_birth(), random.choice(["Male", "Female"])))
conn.commit()
conn.close()

```

8.5.10

```

def extract_features(packet):
    packetLen = 0
    protocolCode = 0
    sourcePort = 0
    destinationPort = 0
    try:
        packetLen = float(packet.length)
    except AttributeError:
        logging.debug("Packet length not found; defaulting to 0.")
    try:
        proto = packet.transport_layer
        protocolCode = 1 if proto == "TCP" else 2 if proto == "UDP" else 0
    except AttributeError:
        logging.debug("Transport layer not found; defaulting to 0.")
    try:
        if hasattr(packet, 'tcp'):
            sourcePort = float(packet.tcp.srcport)
            destinationPort = float(packet.tcp.dstport)
        elif hasattr(packet, 'udp'):
            sourcePort = float(packet.udp.srcport)
            destinationPort = float(packet.udp.dstport)
    except AttributeError:
        logging.debug("Port information not found; defaulting to 0.")
    return np.array([packetLen, protocolCode, sourcePort, destinationPort]).reshape(1, -1)

```

8.5.11

```
blockedIPs = set()
blockedIpsLock = threading.Lock()

def block_ip(ipAddress):
    with blockedIpsLock:
        if ipAddress in blockedIPs:
            logging.info(f"IP {ipAddress} is already blocked. Skipping.")
            return
        blockedIPs.add(ipAddress)
```

8.5.12

```
def inferProtocol(row):
    category = str(row["Attack_Category"]).lower()
    if "mqtt" in category:
        return "MQTT"
    elif "arp" in category:
        return "ARP"
    elif "icmp" in category or "ping" in category:
        return "ICMP"
    elif "udp" in category:
        return "UDP"
    elif ("tcp" in category or "syn" in category or "port scan" in category or
          "os scan" in category or "recon" in category):
        return "TCP"
    else:
        return "Unknown"
```

8.5.13

```
columnsNeeded = [
    "Header_Length",
    "Protocol Type",
    "fin_flag_number",
    "syn_flag_number",
    "rst_flag_number",
    "psh_flag_number",
    "ack_flag_number",
    "ece_flag_number",
```

```

    "cwr_flag_number",
    "Tot size",
    "Attack_Category"
]

```

8.5.14

```

def balanceDataset(df, target_samples=50000):
    balanced_dfs = []
    for label, group in df.groupby("Attack_Category"):
        if len(group) < target_samples:
            balanced_group = group.sample(n=target_samples, replace=True, random_state=42)
        elif len(group) > target_samples:
            balanced_group = group.sample(n=target_samples, random_state=42)
        else:
            balanced_group = group.copy()
        balanced_dfs.append(balanced_group)
    balanced_df = pd.concat(balanced_dfs).sample(frac=1, random_state=42).reset_index(drop=True)
    return balanced_df

```

8.5.15

```

expected_columns = [
    "Header_Length",
    "Protocol Type_ARP",
    "Protocol Type_ICMP",
    "Protocol Type_MQTT",
    "Protocol Type_TCP",
    "Protocol Type_UDP",
    "Protocol Type_Unknown",
    "fin_flag_number",
    "syn_flag_number",
    "rst_flag_number",
    "psh_flag_number",
    "ack_flag_number",
    "ece_flag_number",
    "cwr_flag_number",
    "PacketLength",
    "Attack_Category",
    "Binary_Label"
]

```

```
df_balanced = df_balanced.reindex(columns=expected_columns, fill_value=0)
```

8.5.16

```
benign_samples = df_balanced[df_balanced["Binary_Label"] == "benign"]
malicious_samples = df_balanced[df_balanced["Binary_Label"] == "malicious"]
target_samples = min(len(benign_samples), len(malicious_samples))
benign_balanced = benign_samples.sample(n=target_samples, random_state=42, replace=True)
malicious_balanced = malicious_samples.sample(n=target_samples, random_state=42, replace=True)
df_binary_balanced = pd.concat([benign_balanced, malicious_balanced]).sample(frac=1,
random_state=42).reset_index(drop=True)
```

8.5.17

```
model = Sequential([
    Dense(64, activation='relu', input_dim=input_dim),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

8.5.18

```
model = Sequential([
    Dense(1024, kernel_regularizer=l2(0.001), input_dim=input_dim),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.4),
    Dense(512, kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.4),
    Dense(256, kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.3),
    Dense(128, kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    LeakyReLU(alpha=0.1),
    Dropout(0.2),
```

```

        Dense(num_classes, activation='softmax')
    ])

    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

```

8.5.19

```

# Data loading and preprocessing
X, y, scaler, label_encoder = loadNPreprocess(filteredDataPath)
# Dataset splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
# Model training
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=30,
    batch_size=256,
    callbacks=[early_stopping, reduce_lr, checkpoint],
    verbose=1
)
# Model evaluation
loss, acc = model.evaluate(X_test, y_test, verbose=1)
print(f"Improved Multiclass Classifier Accuracy: {acc * 100:.2f}%")

```

8.5.20

```

def startLiveCapture(interface_name):
    print(f"Starting live packet capture on interface: {interface_name}")
    capture = pyshark.LiveCapture(interface=interface_name)
    try:
        for packet in capture.sniff_continuously():
            if hasattr(packet, 'ip'):
                print(f"Processing Packet: Source IP: {packet.ip.src}, Destination IP: {packet.ip.dst}")
            else:
                print("Processing Non-IP Packet")
            processPacket(packet)
    except KeyboardInterrupt:
        print("Live capture stopped by user.")

```

8.5.21

```
def extractFeatures(packet):
    features = []
    try:
        header_length = int(packet.ip.hdr_len)
    except AttributeError:
        header_length = 0
    features.append(header_length)
    protocol = packet.highest_layer.upper() if hasattr(packet, 'highest_layer') else "UNKNOWN"
    protocol_onehot = {
        "TCP": [0, 0, 0, 1, 0, 0],
        "UDP": [0, 0, 0, 0, 1, 0],
        "ICMP": [0, 1, 0, 0, 0, 0],
        "MQTT": [0, 0, 1, 0, 0, 0],
        "ARP": [1, 0, 0, 0, 0, 0]
    }.get(protocol, [0, 0, 0, 0, 0, 1])
    features.extend(protocol_onehot)
    tcp_flags = ["flags_fin", "flags_syn", "flags_rst", "flags_psh", "flags_ack", "flags_ece", "flags_cwr"]
    if 'TCP' in packet:
        for flag in tcp_flags:
            try:
                flag_str = getattr(packet.tcp, flag)
                flag_val = int(flag_str, 0)
            except Exception:
                flag_val = 0
            features.append(flag_val)
    else:
        features.extend([0] * len(tcp_flags))

    try:
        packet_length = int(packet.length)
    except AttributeError:
        packet_length = 0
    features.append(packet_length)
    features_df = pd.DataFrame([features], columns=ordered_feature_names)
    return features_df
```

8.5.22

```
def processPacket(packet):
    try:
        features_df = extractFeatures(packet)
    except Exception as e:
        print("Error extracting features from packet:", e)
        return

    try:
        features_bin = binScaler.transform(features_df)
    except Exception as e:
        print("Error occurred scaling features for binary classifier:", e)
        return

    pred_bin = binModel.predict(features_bin)
    malicious_prob = pred_bin[0][0]
    print(f"[Binary] Malicious probability: {malicious_prob:.2f}")
    if malicious_prob >= malThreshold:
        try:
            features_multi = multiScaler.transform(features_df)
        except Exception as e:
            print("Error scaling features for multiclass classifier:", e)
            return

        pred_multi = multiModel.predict(features_multi)
        attack_class_idx = np.argmax(pred_multi)
        attack_class = labelEnc.inverse_transform([attack_class_idx])[0]
        print(f"ALERT: Malicious packet detected, Classified attack: {attack_class}")
        src_ip = getattr(packet, 'src', None)
        if src_ip:
            block_ip(src_ip)
    else:
        print("Packet classified as benign.")
```


8.5.23 datasetFilter.py

```
import pandas as pd

#Defined file paths used for saving and loading relevant material
datasetPath = r"D:\DNNIDPS2\combined_train_dataset.csv"
outputDirectory = r"D:\DNNIDPS2"
filteredDatasetPath = outputDirectory + "filtered_train_dataset.csv"
binaryDatasetPath = outputDirectory + "binary_filtered_dataset.csv"
maliciousDatasetPath = outputDirectory + "malicious_filtered_dataset.csv"

#Inference function for protocol type based on attack category, inference comes from string in
Attack_Category Column
def findProtocol(row):
    category = str(row["Attack_Category"]).lower()
    if "mqtt" in category:
        return "MQTT"
    elif "arp" in category:
        return "ARP"
    elif "icmp" in category or "ping" in category:
        return "ICMP"
    elif "udp" in category:
        return "UDP"
    elif ("tcp" in category or "syn" in category or "port scan" in category or
        "os scan" in category or "recon" in category):
        return "TCP"
    else:
        return "Unknown"

#Filter function to get contents suitable to PyShark capabilities
def filterForPyshark(df):
    neededColumns = [
        "Header_Length",
        "Protocol Type",
        "fin_flag_number",
        "syn_flag_number",
        "rst_flag_number",
        "psh_flag_number",
        "ack_flag_number",
        "ece_flag_number",
        "cwr_flag_number",
        "Tot size",
        "Attack_Category"
```

```

]
keepColumns = [col for col in neededColumns if col in df.columns]
df_filtered = df[keepColumns].copy()

if "Tot size" in df_filtered.columns:
    df_filtered.rename(columns={"Tot size": "PacketLength"}, inplace=True)

df_filtered["Protocol Type"] = df_filtered.apply(findProtocol, axis=1)
return df_filtered

#Balancing function for attack categories - limits to 50000, uses resampling for lacking classes
def datasetBalancer(df, target_samples=50000):
    balanced_dfs = []
    for label, group in df.groupby("Attack_Category"):
        if len(group) < target_samples:
            balanced_group = group.sample(n=target_samples, replace=True, random_state=42)
        elif len(group) > target_samples:
            balanced_group = group.sample(n=target_samples, random_state=42)
        else:
            balanced_group = group.copy()
        balanced_dfs.append(balanced_group)
    balanced_df = pd.concat(balanced_dfs).sample(frac=1, random_state=42).reset_index(drop=True)
    return balanced_df

def main():
    df = pd.read_csv(datasetPath)
    df_filtered = filterForPyshark(df)
    df_filtered["Binary_Label"] = df_filtered["Attack_Category"].apply(
        lambda x: "benign" if str(x).strip().lower() == "benign" else "malicious"
    )

    df_balanced = datasetBalancer(df_filtered, target_samples=50000)

    #Onehot encoding for protocol type
    df_balanced = pd.get_dummies(df_balanced, columns=["Protocol Type"])

    #define the correct column order expected by the DNN model
    columnsExpected = [
        "Header_Length",
        "Protocol Type_ARP",
        "Protocol Type_ICMP",
        "Protocol Type_MQTT",

```

```

"Protocol Type_TCP",
"Protocol Type_UDP",
"Protocol Type_Unknown",
"fin_flag_number",
"syn_flag_number",
"rst_flag_number",
"psh_flag_number",
"ack_flag_number",
"ece_flag_number",
"cwr_flag_number",
"PacketLength",
"Attack_Category",
"Binary_Label"
]

# Reindexing the DataFrame to match the expected order of columns
df_balanced = df_balanced.reindex(columns=columnsExpected, fill_value=0)

# Balance benign and malicious samples
benignSample = df_balanced[df_balanced["Binary_Label"] == "benign"]
malSamples = df_balanced[df_balanced["Binary_Label"] == "malicious"]
targetSample = min(len(benignSample), len(malSamples))

balancedBenignDS = benignSample.sample(n=targetSample, random_state=42, replace=True)
balancesMalDS = malSamples.sample(n=targetSample, random_state=42, replace=True)

balacnedBinaryDS = pd.concat([balancedBenignDS, balancesMalDS]).sample(frac=1,
random_state=42).reset_index(drop=True)

#save the binary balanced dataset to earlier directory
balacnedBinaryDS.to_csv(binaryDatasetPath, index=False)

#save the filtered balanced dataset to specified path
df_balanced.to_csv(filteredDatasetPath, index=False)

# Save the malicious-only dataset for multiclass classification
df_malicious = df_balanced[df_balanced["Binary_Label"] == "malicious"].copy()
df_malicious.to_csv(maliciousDatasetPath, index=False)

print("Datasets created successfully")

if __name__ == "__main__":

```

```
main()
```

8.5.24binaryClassifier.py

```
import pandas as pd
import numpy as np
import pickle
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

binDatasetPaths = r"D:\DNNIDPS2\binary_filtered_dataset.csv"
binaryModel = r"D:\DNNIDPS2\binary_classifier" #SavedModel
binScalerPath = r"D:\DNNIDPS2\binary_scaler.pkl"

# preprocessing Function for the Binary Dataset
def binaryPreprocess(df):
    df = pd.get_dummies(df, columns=["Protocol Type"])
    columnsExpectedd = [
        "Header_Length", "Protocol Type_ARP", "Protocol Type_ICMP", "Protocol Type_MQTT", "Protocol
Type_TCP",
        "Protocol Type_UDP", "Protocol Type_Unknown", "fin_flag_number", "syn_flag_number",
"rst_flag_number",
        "psh_flag_number", "ack_flag_number", "ece_flag_number", "cwr_flag_number", "PacketLength"
    ]
    X = df.reindex(columns=columnsExpectedd, fill_value=0)
    y = df["Binary_Label"].apply(lambda x: 0 if str(x).strip().lower() == "benign" else 1).values
    return X, y

#train the binary classifier
def main():
    print("Loading binary dataset from:", binDatasetPaths)
    df = pd.read_csv(binDatasetPaths)
    X, y = binaryPreprocess(df)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

```

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

input_dim = X_train_scaled.shape[1]

model = Sequential([
    Dense(64, activation='relu', input_dim=input_dim),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()

print("Starting training of binary classifier...")
history = model.fit(X_train_scaled, y_train, epochs=20, batch_size=256, validation_split=0.1,
verbose=1)

loss, acc = model.evaluate(X_test_scaled, y_test, verbose=1)
print("Binary Classifier Accuracy: {:.2f}%".format(acc * 100))

model.save(binaryModel)
with open(binScalerPath, "wb") as f:
    pickle.dump(scaler, f)

# plot accuracy into visual
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title("Training and Validation Accuracy")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

#confusion Matrix
y_pred = (model.predict(X_test_scaled) > 0.5).astype(int)
cm = confusion_matrix(y_test, y_pred)

```

```

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Benign', 'Malicious'],
yticklabels=['Benign', 'Malicious'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

#classification Report
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=['Benign', 'Malicious']))

if __name__ == "__main__":
    main()

```

8.5.25multiclassClassifier.py

```

import pandas as pd
import numpy as np
import pickle
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

binDatasetPaths = r"D:\DNNIDPS2\binary_filtered_dataset.csv"
binaryModel = r"D:\DNNIDPS2\binary_classifier" #SavedModel
binScalerPath = r"D:\DNNIDPS2\binary_scaler.pkl"

# preprocessing Function for the Binary Dataset
def binaryPreprocess(df):
    df = pd.get_dummies(df, columns=["Protocol Type"])
    columnsExpectedd = [
        "Header_Length", "Protocol Type_ARP", "Protocol Type_ICMP", "Protocol Type_MQTT", "Protocol
Type_TCP",
        "Protocol Type_UDP", "Protocol Type_Unknown", "fin_flag_number", "syn_flag_number",
        "rst_flag_number",

```

```

    "psh_flag_number", "ack_flag_number", "ece_flag_number", "cwr_flag_number", "PacketLength"
]
X = df.reindex(columns=columnsExpectedd, fill_value=0)
y = df["Binary_Label"].apply(lambda x: 0 if str(x).strip().lower() == "benign" else 1).values
return X, y

#train the binary classifier
def main():
    print("Loading binary dataset from:", binDatasetPaths)
    df = pd.read_csv(binDatasetPaths)
    X, y = binaryPreprocess(df)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    input_dim = X_train_scaled.shape[1]

    model = Sequential([
        Dense(64, activation='relu', input_dim=input_dim),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dropout(0.3),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    model.summary()

    print("Starting training of binary classifier...")
    history = model.fit(X_train_scaled, y_train, epochs=20, batch_size=256, validation_split=0.1,
        verbose=1)

    loss, acc = model.evaluate(X_test_scaled, y_test, verbose=1)
    print("Binary Classifier Accuracy: {:.2f}%".format(acc * 100))

    model.save(binaryModel)
    with open(binScalerPath, "wb") as f:
        pickle.dump(scaler, f)

```

```

# plot accuracy into visual
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title("Training and Validation Accuracy")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

#confusion Matrix
y_pred = (model.predict(X_test_scaled) > 0.5).astype(int)
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Benign', 'Malicious'],
yticklabels=['Benign', 'Malicious'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

#classification Report
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=['Benign', 'Malicious']))

if __name__ == "__main__":
    main()

```

8.5.26DNNIDPS.py

```

import pyshark
import tensorflow as tf
import pickle
import numpy as np
import pandas as pd
import subprocess

# binary classifier files
binModelPath = r"D:\DNNIDPS2\binary_classifier"
binScalerPath = r"D:\DNNIDPS2\binary_scaler.pkl"

```



```

# multiclass classifier files
multiModelPath = r"D:\DNNIDPS2\multiclass_classifier"
multiScalerPath = r"D:\DNNIDPS2\multiclass_scaler.pkl"
labelEncPath = r"D:\DNNIDPS2\label_encoder.pkl"

print("Loading binary classifier and scaler:")
binModel = tf.keras.models.load_model(binModelPath)
with open(binScalerPath, "rb") as f:
    binScaler = pickle.load(f)

print("Loading multiclass classifier, scaler, and label encoder:")
multiModel = tf.keras.models.load_model(multiModelPath)
with open(multiScalerPath, "rb") as f:
    multiScaler = pickle.load(f)
with open(labelEncPath, "rb") as f:
    labelEnc = pickle.load(f)

malThreshold = 0.9

# The order of features as the scaler was trained on
ordered_feature_names = [
    "Header_Length",
    "Protocol Type_ARP", "Protocol Type_ICMP", "Protocol Type_MQTT",
    "Protocol Type_TCP", "Protocol Type_UDP", "Protocol Type_Unknown",
    "fin_flag_number", "syn_flag_number", "rst_flag_number", "psh_flag_number",
    "ack_flag_number", "ece_flag_number", "cwr_flag_number", "PacketLength"
]

def extractFeatures(packet):
    features = []

    try:
        header_length = int(packet.ip.hdr_len)
    except AttributeError:
        header_length = 0
    features.append(header_length)

    protocol = packet.highest_layer.upper() if hasattr(packet, 'highest_layer') else "UNKNOWN"
    protocol_onehot = {
        "TCP": [0, 0, 0, 1, 0, 0],
        "UDP": [0, 0, 0, 0, 1, 0],

```

```

        "ICMP": [0, 1, 0, 0, 0, 0],
        "MQTT": [0, 0, 1, 0, 0, 0],
        "ARP": [1, 0, 0, 0, 0, 0]
    }.get(protocol, [0, 0, 0, 0, 0, 1])

    features.extend(protocol_onehot)

    tcp_flags = ["flags_fin", "flags_syn", "flags_rst", "flags_psh", "flags_ack", "flags_ece", "flags_cwr"]
    if 'TCP' in packet:
        for flag in tcp_flags:
            try:
                flag_str = getattr(packet.tcp, flag)
                flag_val = int(flag_str, 0)
            except Exception:
                flag_val = 0
            features.append(flag_val)
    else:
        features.extend([0] * len(tcp_flags))

    try:
        packet_length = int(packet.length)
    except AttributeError:
        packet_length = 0
    features.append(packet_length)

    features_df = pd.DataFrame([features], columns=ordered_feature_names)
    return features_df

def processPacket(packet):
    try:
        features_df = extractFeatures(packet)
    except Exception as e:
        print("Error extracting features from packet:", e)
        return

    try:
        features_bin = binScaler.transform(features_df)
    except Exception as e:
        print("Error occurred scaling features for binary classifier:", e)
        return

    pred_bin = binModel.predict(features_bin)

```

```

malicious_prob = pred_bin[0][0]
print(f'[Binary] Malicious probability: {malicious_prob:.2f}')

if malicious_prob >= malThreshold:
    try:
        features_multi = multiScaler.transform(features_df)
    except Exception as e:
        print("Error scaling features for multiclass classifier:", e)
        return

    pred_multi = multiModel.predict(features_multi)
    attack_class_idx = np.argmax(pred_multi)
    attack_class = labelEnc.inverse_transform([attack_class_idx])[0]
    print(f'ALERT: Malicious packet detected, Classified attack: {attack_class}')

    src_ip = getattr(packet.ip, 'src', None)
    if src_ip:
        blockIP(src_ip)
    else:
        print("Packet classified as benign.")

def blockIP(ip_address):
    try:
        command = f'netsh advfirewall firewall add rule name="Block IP {ip_address}" protocol=TCP'
        dir=in remoteip={ip_address} action=block'
        subprocess.run(command, shell=True)
        print(f'Blocked IP Address: {ip_address}')
    except Exception as e:
        print(f'Error blocking IP address {ip_address}: {e}')

def liveCapPyshark(interface_name):
    print(f'Starting live packet capture on interface: {interface_name}')
    capture = pyshark.LiveCapture(interface=interface_name)
    try:
        for packet in capture.sniff_continuously():
            if hasattr(packet, 'ip'):
                print(f'Processing Packet: Source IP: {packet.ip.src}, Destination IP: {packet.ip.dst}')
            else:
                print("Processing Non-IP Packet")
            processPacket(packet)
    except KeyboardInterrupt:
        print("Live capture stopped by user.")

```

```
if __name__ == "__main__":  
    interfaceName = "WiFi" #this can be changed depending on the network interface  
    liveCapPyshark(interfaceName)
```