School of Electronic Engineering and Computer Science

Final Report

Programme of study:
BEng Computer Systems Engineering

Project Title:
Network Intrusion Detection System Using Machine Learning Algorithms

Supervisor:
Dr Luk Arnaut

Student Name:
Shan Balendra

190263674

Final Year
Undergraduate Project 2023/24

Date: 30/04/2024

Abstract

Network Intrusion Detection Systems (NIDS) are vital in any modern Internet-connected device, monitoring network traffic for any suspicious network traffic or known attack patterns. However, this threat has become constant with the advancement of AI and automated cyber-attacks. This project was designed to develop and evaluate the capabilities of NIDS before and after incorporating Machine Learning techniques.

Specifically, the project will develop and evaluate the performance of 2 models, a Convolutional Neural Network and a Recurrent Neural Network. Their performance will be evaluated using real network traffic in the CIC-IDS2017 Dataset. (Iman Sharafaldin, January 2018) The project will aim to implement, improve and explore NIDS accuracy, reduce false positives and examine methods of integrating Machine Learning into NIDS's.

This aligns with the goals of my degree as it involves system design, data analysis, and the practical applications of ML. The implementation of a CNN achieved a higher accuracy and lower false positive rate compared to the baseline of a Random Forest Algorithm.

While both neural networks outperformed the algorithm, there is still significant room to optimise weights for both models and train based on larger real-time network traffic. I could also have reduced the dimensionality of the dataset from 85 by specifying specific attacks. i.e., only web attacks are used to increase accuracy and decrease resource usage.

However, this approach did not allow the detection of brand-new types of destructive attacks, highlighting the importance of this system working in tandem with heuristic methods such as signature-based detection, deep packet inspection and behavioural analysis.

Contents

Contents

1.1 Introduction 5

1.2 Background 5

1.3 Problem Statement 6

1.4 Aim 7

1.5 Objectives 7

1.6 Research Questions 8

1.7 Report Structure 9

Chapter 2: Literature Review 11

2.1 Overview 11

2.2 Theoretical Background 11

2.2.1 Artificial Intelligence and Machine Learning in Network Intrusion Detection Systems (NIDS) 13

2.2.2 Related Work and Existing Technologies 13

2.3 Challenges 15

2.3.1 Dataset Bias or Imbalance, False Alarm Rate, and Limited Encrypted Traffic Inspection 15

2.3.2 Model Interpretation and Understanding 16

2.3.3 Evaluation Metrics 17

2.3.4 Future Research 20

Chapter 3: Requirement Analysis and Design 21

3.1.1 Requirement Analysis 21

Chapter 4: Implementation 23

Chapter 5: Conclusion 41

5.1.1 Achievements 41

5.1.2 Challenges Faced 41

Chapter 6: Risk Assessment 43

Chapter 7: References 45

1.1 Introduction

These days, our world is highly connected, and cyber security is constantly changing with more advanced and widespread malware attacks (Verizon, 2023). There are different types of threats that individuals and organizations face. These risks range from targeted strikes on critical infrastructure. (Dragos, 2022) to widespread malware campaigns (AhnLab Security Emergency Response Center (ASEC), 2023). Consequently, better intrusion detection systems must be developed to protect digital assets and uphold network integrity in the era where new sophistication in attackers' technologies keeps growing. (Hung-Jen Liao, 2013). This paper examines how AI-based NIDS can revolutionise threat detection and response amidst the dynamic threat landscape.

Our goal with this project is to meet the learning objectives of the Electrical Engineering and Computer Science programme while contributing to the cybersecurity sector. This entails using cutting-edge system design concepts, utilising AI and ML in real-world applications, and honing data analysis and software development skills. The final objective is to provide a working prototype of an AI-enhanced NIDS and documentation detailing the design, development, and evaluation procedures.

1.2 Background

Due to having an online presence and having digitised infrastructure being indispensable to every business in the modern economy and vital to social interactions, the scale and complexity of cyber threats have also escalated. Having Network Intrusion Detection Systems (NIDS) as part of the cybersecurity toolkit is critical because they can identify unauthorised access attempts and stop network attacks. Historically, these systems have relied heavily on rule-based and signature-based methods, which increasingly are not sufficient against sophisticated cyber threats. These include polymorphic malware, advanced persistent threats (APTs), and zero-day exploits, which can go undetected by conventional defences.

Network security has evolved significantly over the years, driven by the continuous escalation of cyber threats. Traditionally, protection from these threats used to rely on preventive measures such as firewalls, antivirus, and access control mechanisms (Kumar & Nalband, 2022). Undoubtedly, all these tools are still required, but they appear incapable of effectively responding to new threats constantly coming to the fore. Since cybercriminals are using increasingly sophisticated methods, security solutions have become more proactive and intelligent.

Due to this, intrusion detection systems evolved into a significant component of the security concept in a network environment. IDS generally has two major divisions: network-based intrusion and host-based detection systems. In the latter, HIDS monitors events on individual hosts or devices. In contrast, in the

former case, NIDS analyses network traffic for the presence of attack signatures or other malicious indicators (Mukherjee, Todd Heberlein & Levitt, 1994). With these tools, this dissertation is concerned explicitly with providing information on the ability to detect attacks against the network infrastructure through NIDS.

Traditional NIDS solutions are normally signature-based, wherein network traffic is matched against a database of attack signatures. The scheme is exemplary for detecting identified threats but cannot work with zero-day attacks or other new intrusions without prior signatures. Anomaly-based detection, targeting deviations from normal behaviour, is the alternative (Liao et al., 2013). Traditional anomaly detection methods have been impractical due to the high rate of false positives within the dynamic network environment. Machine learning has the potential to learn from the network traffic data itself, which is one such approach for the differentiation of normal and malicious activities. Neural networks, support vector machines, decision trees, and ensemble approaches have been applied as ML algorithms to intrusion detection, and these algorithms have shown promising results in this application (Ahmad et al., 2020). These algorithms can adapt to new threats upon updates, improve with time as more data is received, and understand the network behaviour more accurately.

1.3 Problem Statement

Despite years of research in intrusion detection, network security remains a significant concern as malicious actors constantly find new methods to exploit. The increased volume, velocity, and variety of network traffic, coupled with the sophistication of today's cyber-attacks, demand more sophisticated detection mechanisms. Traditional signature-based Network Intrusion Detection Systems can perform very little in detecting new and unknown attacks. In contrast, conventional anomaly-based systems often produce high false positive rates, leading to alert fatigue and decreased operational efficiency (Sharma, Sharma, and Lal, 2022).

Integrating machine learning algorithms into NIDS brings opportunities to handle such challenges; however, several issues must be attended to for the full potential of ML-based NIDS to be harnessed effectively. One of the critical challenges is feature selection and data representation. Intrusion detection with higher accuracy must identify the most relevant features from network traffic data. The challenge is to select features that capture the essence of ordinary and malicious activities without swamping the system with useless information.

The second most important problem is the choice of appropriate algorithms and their optimisation. The different ML algorithms have diverse limitations and strengths. Selecting the correct algorithm for NIDS use and turning its parameters to optimise its performance is not a trivial exercise and needs careful deliberation and experimentation. High traffic volumes require real-time processing in network intrusion detection. This leads to the conduction of real-time analysis of high traffic volumes. Therefore, it is of great importance that ML-based NIDS can be made to work efficiently in real-time, with no giving in on accuracy and with no bringing in more computational overhead.

Evaluation metrics are essential for the overall effectiveness of the ML-based NIDS. This section provides various metrics: F1-score, precision, false positive rate, recall, and detection rate. They depict an insight into the effective system but must be broken down into a specific network environment and threat landscape. The problems for scalability and adaptability are even more challenging because growth in network infrastructures and continuous changes in them would need a NIDS to scale and be adaptive to growing traffic volumes and new threats. A significant challenge is to ensure this scaling and adaptation is not at the cost of significant performance degradation in the presence of ML-based systems.

1.4 Aim

This project aims to create and implement five NIDS models, using a Convolutional neural network(CNN) architecture and a Recurrent Neural Network(RNN) architecture, to improve the detection of new threats in real time. And conclude on a single AI-enhanced Network Intrusion Detection System to enhance the detection of network intrusions. The aim is to decrease instances of false positives and increase its ability to adjust to real-time threats, ensuring networks are defended against an array of cyber threats and enhancing network safety and security.

1.5 Objectives

1. Algorithm Selection and Model Development
    i. Evaluate a range of ML algorithms, including decision trees, random forests, Support Vector Machines (SVMs), and deep learning techniques (CNNs, RNNs).
    ii. Optimize model performance using feature selection and dimensionality reduction (e.g., Principal Component Analysis, autoencoders).
2. Prototype Development
    i. Design and implement a scalable prototype supporting real-time traffic analysis and intrusion detection.
3. Testing and Optimization
    i. Evaluate the NIDS prototype's accuracy, adaptability, and efficiency under realistic network conditions.
    ii. Create a range of testing scenarios.
    iii. Use cross-validation, hyperparameter tuning, and A/B testing to fine-tune models and system settings.
    iv. Prioritize metrics like accuracy, precision, recall, F1-score, false-positive rates, and computational overhead.
4. Performance Analysis
    i. Analyse the AI-enhanced NIDS against traditional systems.

ii. Establish baseline performance of a traditional signature-based/rule-based NIDS against the same datasets.
iii. Benchmark the AI-enhanced NIDS regarding detection rates, response time, false positives, and adaptability to unseen attacks.
iv. Analyse computational resource requirements and scalability of both the AI-enhanced and traditional systems.

1.6 Research Questions

1. Algorithm Efficacy and Selection:
    a. How do accuracy and false-positive rates of machine learning algorithms (e.g., decision trees, random forests, SVMs, deep learning) compare when applied to network intrusion detection on datasets like CICIDS2017?
    b. How do feature selection and dimensionality reduction techniques (e.g., PCA, autoencoders) impact machine learning models' performance and computational efficiency in network intrusion detection?
2. Prototype Functionality and Adaptability:
    a. How does a modular and scalable NIDS prototype design facilitate real-time network traffic analysis and adaptability to new threats?
    b. What are the operational challenges and benefits of integrating advanced machine learning algorithms into traditional NIDS frameworks?
3. Testing and Optimization:
    a. What testing methodologies best simulate real-world scenarios, especially those involving zero-day and advanced persistent threats, for rigorous NIDS evaluation?
    b. How do hyperparameter adjustments affect system performance under different network conditions?
4. Comparative Performance Analysis:
    a. How does the AI-enhanced NIDS surpass traditional signature-based or rule-based systems in its detection rates, response times, and ability to identify novel threats?
    b. How does the integration of machine learning impact computational resource requirements compare to conventional NIDS systems?
5. Long-term Viability and Scalability:
    a. What are the long-term implications for system maintenance, scalability, and continuous learning when deploying AI-enhanced NIDS solutions?

b. How can AI-enhanced NIDS be designed for seamless updates and integration with evolving cybersecurity technologies and frameworks?

1.7 Report Structure

1. Introduction
    - Overview and Rationale: A brief discussion of network intrusion detection and the motivation to use artificial intelligence to improve them.
    - Objectives: Clearly stated objectives for the project aimed at improving NIDS through AI and machine learning.
2. Background and Literature Review
    - Theoretical Background: Fundamental principles about network intrusion detection and machine learning are discussed.
    - Review of Existing Technologies: Current NIDS technologies, their strengths, and limitations, among others, are examined.
    - Related Work: Recent academic research and industrial efforts in similar fields that give a sense of context while identifying gaps.
3. System Design and Development
    - Algorithm Selection and Model Development: The process of choosing different machine learning algorithms is explained in detail. Some discussions on feature selection techniques like dimensionality reduction will also be made here.
    - Prototype Development: The architectural decisions on integrating selected algorithms into a working NIDS prototype and technology stack have been highlighted here.
4. Testing and Optimization
    - Test Methods: Information on the approaches employed in testing that included simulations of network scenarios common in real life.
    - Methods of Optimization: A discussion on techniques like cross-validation and hyper-parameter tuning that are used to better the performance of a system.
    - Results Analysis
5. Analysis and Evaluation: This section looks at the AI-driven NIDS and traditional systems through measuring detection accuracy and system time gains.

6. Assessment of impact: Evaluating how much the system works in a realistic network environment, its effectiveness, and possible disadvantages.

7. Conclusion and Future Work

    - Summary of Findings: The summary is about what was found from research on network security.

    - Future Research Directions: Further improvements can be made to examine various areas concerning AI-enhanced intrusion detection methods.

    - Implementation Prospects: Here, we'll explain everything that needs to be done for this prototype to become a complete working system.

Chapter 2: **Literature Review**

There has been a significant increase in academic research and studies about implementing machine learning into a network detection system, with several researchers analysing various methods and their efficacy. This chapter describes relevant research that has been done and applies it to different project elements, outlining challenges faced with traditional NIDS systems, ML, and appropriate algorithms and structures. (Farnaaz & Jabbar, 2016)

2.1 Overview

Though they are vital to business and society, the exponential growth of interconnected digital systems has brought about an attack surface on a scale never seen before. Historically speaking, Traditional Network Intrusion Detection Systems (NIDS) have relied on signatures and predefined rules but have proven inadequate against the metamorphic sophistication of cyber threats. Attack vectors now include polymorphic malware, zero-day exploits, and Advanced Persistent Threats (APTs), often designed to evade conventional defences. This vulnerability underscores the urgent need for intelligent NIDS that can adapt to the continually shifting threat landscape. Intrusion detection is an excellent opportunity for improving AI and Machine Learning (ML). By getting them to learn complex patterns, detect anomalies, and respond autonomously, among other things, AI could significantly improve NIDS performance.

2.2 Theoretical Background

Traditional Network Intrusion Detection Systems (NIDS), which depend on predefined rules and signatures to identify malicious activity(without any NIDS built-in), are no longer effective against new and sophisticated attack techniques. Their static nature has made them ill-equipped to handle the ever-changing threat landscape. Thus, considerable steps have been taken in using artificial intelligence and machine learning for intrusion detection. These AI- and ML-based approaches offer better detection capabilities, adaptability, and the possibility that systems can detect unknown threats by learning patterns and behaviours of the network traffic. The paradigm shift has enhanced security posture overall, providing a more robust defence against emergent cyber-attacks.

These are the two fundamental methods upon which Traditional NIDS operate:

Detection based on Signatures: These systems excel in identifying known attack patterns (signatures) from a comprehensive database. They compare all incoming traffic with these signatures and promptly raise an alert when a match is found. Signature-based detection, akin to how antivirus software operates, is particularly effective in scanning files and running processes for patterns that match known malware (Ahmad et al., 2020). Its strength lies in its ability to combat known threats, thanks to the extensive database of signatures that security researchers and software vendors continuously update (Ahmad et al., 2020). The system swiftly flags the event and takes necessary actions, such as quarantining

the file and blocking the network traffic, when a perfect or near-perfect match is found against the malware signatures database.

While signature-based detection is a common method in security systems, it is not foolproof. It is particularly vulnerable to zero-day attacks or changed forms of old malware. These are threats for which no signatures exist, making them undetectable by signature-based systems (Ahmad et al., 2020). The system's major limitation is its reliance on previous knowledge of the signatures. It cannot identify new or previously unknown threats, such as zero-day attacks. Furthermore, malware authors often use polymorphism and other obfuscation methods to alter the 'appearance' of their code, making it unrecognizable to signature-based detection systems.

Detection based on Rules: Rule-based detection, sometimes also referred to as detection by rule, is a technique that involves setting a set of rules or heuristics outlining suspicious behaviour or anomalies in a system. In other words, a set of rules that describe what constitutes normal network behaviour is established. Moreover, alerts are triggered by any deviations from these rules. While signature-based detection seeks to identify predefined patterns, rule-based detection tries to identify deviations from normal behaviour that may indicate malicious activity. The rules could be simple or complex, and usually, they are generated by security analysts who understand the system's regular operation in depth. For example, some rules might assert that a particular application should not communicate to a given external server or that a specific user should avoid accessing one set of files outside working hours. Once an event conforms to the rule, it triggers an alert to indicate that this is an event to be further investigated by the security teams.

A rule-based detection system effectively finds new, sophisticated threats that do not match any known signature (Shafi, Abbass & Zhu, 2006). The system can detect a significantly broader scope of malicious activity by focusing on behavioural anomalies, with zero-day attacks in the list. Rule-based systems are more flexible than signature-based systems but necessitate careful setting up and often miss subtle unknown types of cyber threats (Shafi, Abbass & Zhu, 2006). The development and the question of maintaining an effective rules system are complicated and resource-consuming. It needs constant monitoring and tuning to avoid false positives, where certain activities are incorrectly patterned as being hostile, or, by contrast, false negatives, where certain activities are mistakenly cleared from being hostile. Furthermore, rule-based detection systems can generate many alerts that many organisations struggle to curate if poorly handled. Many organisations couple these with other methods, such as machine learning and behavioural analysis, to increase accuracy and reduce the security team's workload.

A set of rules that describe what constitutes normal network behaviour is established. And alerts are triggered by any deviations from these rules. Rule-based systems are more flexible than signature-based ones but require careful setting up and tend to miss subtle unknown types of attacks.

2.2.1 Artificial Intelligence and Machine Learning in Network Intrusion Detection Systems (NIDS)

Machine learning is changing cybersecurity by bringing adaptability and pattern recognition abilities to surpass conventional methods. Key ML paradigms relevant to NIDS advances are:

Supervised Learning: Algorithms (e.g., decision trees, random forests, Support Vector Machines) learn from labelled training data, classifying traffic as benign or malicious. These methods are good at recognising known attack patterns but may fail to spot new ones.

Unsupervised Learning: This approach (e.g., clustering algorithms) examines unlabelled data, identifying anomalies and potential attacks without prior examples. It is suitable for zero-day attacks but not so good with high false positive rates.

Deep Learning: Neural network architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are particularly adept at discovering complex, unconventional patterns in network traffic. They excel at automatically extracting features and finding hints of subtle threats, although they typically need large datasets and computational resources.

2.2.2 Related Work and Existing Technologies

Many studies demonstrate that the efficiency of NIDS can be enhanced if Machine Learning algorithms are utilised." Divekar et al (2018). conducted benchmarking research using alternate datasets to the traditionally used KDD CUP 99 dataset. The authors criticised the KDD CUP 99 dataset due to the imbalanced response distribution, presence of non-stationarity, and old attacks. Authors have compared the performances of classification models such as K-Means, Naive Bayes, Random Forest, Decision Trees, Support Vector Machines, and Neural Networks over datasets like UNSW-NB15, NSL-KDD, and KDD-99. They balanced the datasets by oversampling the minority class using SMOTE and under-sampling the majority class randomly to get a balanced NSL-KDD. Their approach included training such models on the dataset and testing their performance with the Weighted F1-Score. The findings showed that the UNSW-NB15 classifier significantly outperformed KDD-99 and NSL-KDD in binary classification settings. Notably, this implies that UNSW-NB15 provides a more precise benchmark for A-NIDS, with its higher uniformity of distribution patterns.

Sainis et al. (2018) discussed feature classification and outlier detection, aiming to boost the accuracy of intrusion detection systems. This paper considered five classification machine learning algorithms: Random Forest, Decision Tree, K-Nearest Neighbors, Naive Bayes, and linear SVM. The contrast also opposed the latest datasets and analysis of NSL-KDD and GurKDDcup with the old DARPA KDD Cup 1999 dataset. It was also done to minimise the complexities of the

feature domain using feature reduction techniques. All these classification models have been applied against the datasets by the authors and then evaluated for prediction time, training time, and detection accuracy. The results indicated that the recent datasets, especially NSL-KDD, are accurate and efficient for detecting intrusions within a network. Conversely, Random Forest remained the best classifier, considering the same period's detection accuracy and prediction time.

Belouch et al. (2018) assessed a machine learning-based intrusion detection system through a big data processing tool, Apache Spark. They investigated the most common algorithms in classification, including Decision Tree, Naive Bayes, Support Vector Machine (SVM), and Random Forest. In this work, they utilised the UNSWNB15 dataset, which focused on every classifier in this study in measuring detection, prediction, and build times. The results showed that a random forest classifier can outperform others based on execution time, sensitivity, accuracy, and specificity. The results have indicated the need for modern datasets, such as UNSWNB15, and big-data-processing tools like Apache Spark to improve or augment the performance and efficacy of intrusion detection systems.

On the other hand, Şenol, Talan, and Aktürk (2024) proposed a hybrid feature reduction with Correlation-Based Feature Selection of Association Rule Mining and attribute values. The dataset was further divided into equal portions to save processing time, and the output from the CFS method was used as an input to ARM to decrease the number of features further. The authors applied Naive Bayes algorithms, logistic regression, and expectation maximisation clustering for the decision engine in the detection of network intrusions. Testing was done on the UNSW-NB15 dataset as well as the NSL-KDD dataset. It was demonstrated that this hybrid approach, compared to traditional approaches, offered increased accuracy and lowered false alarm rates and processing time. The study proved that combining feature reduction methods and machine learning algorithms can improve intrusion detection systems' performance.

Random forests and decision trees are among the popular choices for supervised learning because they are relatively computationally efficient and may be interpreted. Decision rules for traffic analysis from a wide range of data make them applicable to various types of input data. However, they can overfit and be less effective against new attacks. For example, SVMs work well with high-dimensional datasets, making it possible for the classifiers to find optimal boundaries between classes using large training materials. However, their performance may fluctuate depending on how the hyperparameters are set in intrusion detection systems.

Deep learning techniques, particularly CNNs and RNNs, have recently received much more attention. The temporal modelling capabilities are well-suited for detecting attack patterns in traffic flows. On the other hand, due to their ability to model time series data, RNNs can be used to detect sequential patterns in cyber-attacks. However, while simpler machine learning algorithms offer transparency, deep learning models need vast training data, which is impractical when training data is not widely available (i.e., medical records).

Many new papers on hybridisation integrate classical NIDS with AI-driven approaches. These systems aim to incorporate the speed and accuracy of signature-based detection for known threats with ML capabilities to identify novel anomalies. Furthermore, combinations of various ML algorithms offer an opportunity to improve accuracy and adaptability even more.

2.3 Challenges

2.3.1 Dataset Bias or Imbalance, False Alarm Rate, and Limited Encrypted Traffic Inspection

Dataset bias or imbalance is a significant challenge in deploying machine learning models for NIDS. Typically, in the case of such systems, attack instances are very few compared with the instances of benign traffic. This has significantly impacted model performance and reliability, opening the door to critical issues. One major problem created by a class imbalance in datasets is that the model begins to bias towards the majority class, which becomes the benign traffic here. Notably, this happens because the algorithms are designed to lower the error rates of the whole by starting to ignore the minority class that bears the actual intrusion attempts (Al-Janabi et al., 2021). This makes models accurate in classifying benign traffic but very poor at detecting malicious activities, causing a high rate of false negatives, missing actual attacks, and posing a security risk to the network.

Moreover, dataset imbalance can raise the problem of false alarms. For instance, the model may not capture the nuanced differences in an imbalanced dataset, increasing the false positives—misrepresenting benign traffic as malicious and increasing the false alarm rate (Al-Janabi et al., 2021). High false alarm rates can thus be disastrous in a real NIDS setting because it flood the security analysts with such a deluge of alerts that it is hard to identify and respond to a real threat in time.

The second critical challenge related to dataset imbalance is the limited encrypted traffic inspection. Considering the advancement in encryption methods for securing data, a large portion of the network traffic becomes encrypted, which the traditional NIDS struggles to analyse and detect malicious activities. Machine learning models trained on imbalanced datasets may not learn effectively regarding the patterns associated with encrypted malicious traffic. This makes the traffic encrypted and, hence, usually inefficient to be analysed using traditional signature-based methods. Machine learning models have to rely on indirect features, which might be inadequate if they are not representative of the dataset of the actual traffic distribution.

Some of the techniques that can address the above problems include:

Resampling Techniques: The method plays a significant role in handling biased or imbalanced datasets. This is because in a raw dataset (especially when working with a cybersecurity scenario), regular activity will often be significantly higher than malicious activity. The imbalance can lead to a model that only predicts the most common class; therefore, it would fail to capture those valuable, rare, but important cases of attacks (Chakravarthy et al., 2019). Techniques such as under-sampling the majority class or oversampling the minority class assist in designing balanced classes again. For example, oversampling entails creating replicate copies of the minority class equal to the size of the majority one, while under-sampling decreases the quantity of information in the significant class (Chakravarthy et al., 2019). They are designed to make the model NumPy for each of the classes during training so that during testing, it will be sensitive to detect the attacks and not overlook them. In other words, these strategies adjust the distribution of classes in a data set. This is to make sure that minority class samples are adequately represented, to correct balance in the dataset, and to improve the model sensitivity to attack detections.

Cost-sensitive Learning: By applying higher weighted costs to misclassifications of the minority class, it prioritises reducing false negatives rather than false positives. In traditional machine learning models, misclassification costs are typically uniform across all classes. However, in security applications, the cost of misclassifying an attack (false negative) can be far more severe than misclassifying normal traffic as an attack (false positive). Cost-sensitive learning assigns higher penalties to false negatives than to false positives. The prioritisation ensures that the model focuses more on correctly identifying attacks, even if it means a slight increase in false positives. By adjusting the misclassification costs, the model becomes more robust in detecting threats, thus significantly reducing the false alarm rate.

Synthetic Data Generation: Methods like SMOTE (Synthetic Minority Over-sampling Technique) allow for increased generation of synthetic cases of the minority class, thereby giving models more training data and helping them learn patterns applicable in a broader range of situations. SMOTE generates synthetic instances of the minority class by interpolating between existing minority class instances (Soltanzadeh & Hashemzadeh, 2021). This method creates new, plausible instances that enrich the training dataset, providing the model with a broader range of examples to learn from. By enhancing the representation of the minority class, SMOTE helps the model better understand and recognise the subtle patterns that distinguish attacks from normal traffic. The comprehensive exposure to diverse data improves the model's ability to generalise and perform accurately in real-world scenarios, where attacks can manifest in various forms.

These methods are crucial to improving the efficacy of NIDS in identifying real-world attacks and ensuring that they work well under different attack scenarios.

2.3.2 Model Interpretation and Understanding

It can be hard to find the reason behind model forecasts when models like deep neural networks are so complicated that they become “black boxes” This is a

critical issue, especially for security analysts who require clear justifications for alerts.

Solutions to these could be Tools such as LIME (Local Interpretable Model-Agnostic Explanations) and SHAP (Shapley Additive Explanations), which are useful to deconstruct and explain which features significantly influence model decisions as Transparentness is necessary for confidence and efficiency in deploying ML in any application.

2.3.3 Evaluation Metrics

NIDS performance has traditionally been measured using metrics like accuracy. However, these metrics always turn out to be deficient when the false negatives and false positives have costly costs. As such, more elaborate metrics like F1-score, recall, precision, and Receiver Operating Characteristic-Area Under Curve are increasingly preferred. According to Khan et al. (2024), these metrics provide a holistic landscape of how well a NIDS performs, especially in high-stakes environments.

F1-score is the harmonic mean of precision and recall and provides a balanced measure of an NIDS's attack detection capability. Precision refers to the ratio between exact positives and total predicted positives, which gives the ratio between real threats and the total attacks detected. On the other hand, recall is the ratio of true positives to total positives, the percentage of actual threats the NIDS detected. The F1 score is a combination that takes both metrics to provide a single measure accounting for both false positives and negatives. This can be particularly useful when errors are costly, such as in critical infrastructures or financial institutions.

In the absence of alarm, recall becomes paramount. For example, one undetected intrusion into safeguarding sensitive governmental or military networks can result in numerous severe security breakups. High recall can detect most such attacks, if not all, reducing the risk that an undetected threat will harm as most are detected (Khan et al., 2024). However, high recall emphasis tends to increase the rate of most false positives, which mistakenly indicate benign activities as malicious ones.

High precision avoids false positives, which may be disruptive and eventually involve high resources for dealing with them. For example, in a corporate environment, many false alarms overwhelm the security people. It causes alarm fatigue, whereby real threats might be ignored. High precision means that alarms raised by the NIDS will likely be actual attacks, optimising the response efforts and avoiding unwarranted investigations.

Another handy metric is the ROC-AUC, which gives a general vision of a NIDS's performance at any classification threshold. The ROC curve graphs the TPR versus the FPR and, therefore, illustrates the trade-off that occurs in the detection between genuine threats and false alarms (Khan et al., 2024). The AUC quantifies the global performance of the NIDS in discriminating between attacks and regular traffic. A value close to 1 in the AUC signifies excellent

performance—the system is good at discriminating malicious activities from benign ones.

2.3.3.1 Hybrid Approaches

Hybrid NIDS architectures fuse conventional rule-based systems with ML-driven techniques to exploit the strengths of both.

Hierarchical Models: These models consist of traditional NIDSs that identify common threats, followed by applying an ML algorithm to the remaining traffic to identify unknown or complex attacks (Saumya & Mohanty, 2021). In these models, original NIDSs are first used to identify known threats on the Internet. The models primarily rely on signature-based systems, which are highly effective in recognising and combating well-known and relatively stable cyber-attack types due to clear rules and distinct identifying characteristics (Şenol, Talan & Aktürk, 2024). However, their effectiveness decreases when encountering new or complex threats that do not belong to a specific group of patterns. The above approach introduces an ML algorithm for traffic that passes through the traditional NIDS. The second layer addresses the remaining traffic to search for previously unnoticed or difficult-to-detect forms of attack that the system based on signatures might have overlooked. Due to their capability to combine deterministic approaches of rule-based systems integrated with the adaptability and prediction possibilities of the ML algorithms, hierarchical models ensure a strong and continuously developing protection against various types of cyber threats.

Parallel Models: In this setup, both traditional and ML-based systems operate simultaneously, achieving full coverage by merging the robust features of signature-based processes with the flexibility of anomaly detection. The basic form of NIDS, as was described before, stays on top of the known threats by using a rule-based approach to classify the intrusions and confirm that the previously identified methods of attacks are detected and eliminated. In parallel with that, the component based on the ML grants the traffic and searches for new and unknown types of attacks using statistical methods and expert tools such as clustering and classification (Qian et al., 2020). Thus, it provides nearly 100% coverage while combining the implementation of signature-based processes with ML's ability to learn anomalies constantly. The parallel operation ensures that one system is always fighting off the known threat. At the same time, the other continually learns and adapts to new and different threats, thus making the system much more reliable and reciprocal.

2.3.3.2 Data Flow and Integration into a Security System

There is a significant impact on accuracy and performance for hybrid NIDS systems depending on how the data pipeline is managed and shared between all the different components of the system.

Data Preprocessing: For effective integration, many preprocessing steps must be made to use the data for traditional and ML-based analysis (Seo et al., 2023). For example, parsing of raw network packets could be required to allow them to fit into a format recognised by machine learning algorithms. Interpreting or decoding the raw network packets is centric to analysing a network, protecting it against threats, and diagnosing problems. It allows gathering and analysing network

traffic information based on packet headers and payloads. They assist in the recognition of abnormalities, the identification of intrusions, and the comprehension of network efficiency. Packet examination allows the identification of connectivity problems, the integrity of data being transmitted over the network, and the quality of service. Moreover, it helps meet any organisation's security policies and legal necessities.

System Coordination: The system will work efficiently if there is no restriction to data moving between different parts without slowing down the whole system's performance. The significant data flow between different system regions determines the system's general functionality. When passing data is not limited by any constraint, the system can perform operations effectively and efficiently, meaning results can be retrieved instantaneously and users' inputs processed quickly. This integration helps avoid clutter and delays in traversing from one level to another, effectively using resources and allowing different system components to transfer information smoothly. Without such efficiency, things can go very wrong, work may slow down, errors may arise, and users may be displeased. In general, the free exchange of data improves system performance and integrity. It enables the system's scalability in delivering complex solutions while dealing with various processes that should run effectively and be reliable and robust.

2.3.3.3 Ethics Considerations

Bias in AI:

Artificial intelligence models can recreate or even increase extremism, bigotry, and hate from biases in their source/training data, leading to AI potentially causing harm by stating human stereotypes or bias as a fact. Machine learning bias mainly occurs when the data used to train models are not representative of the general population or when algorithms strengthen a prejudice (Ntoutsi et al., 2020). Notably, this can lead to unfair outcomes, as biased hiring decisions, lending practices, or access to services could occur. For instance, facial recognition systems have been shown to perform less well on specific demographic groups compared with others, thus increasing the risk of injustices in policing and surveillance. The bias is addressed by multi-dimensionality: diversified data collection, transparency of algorithms, and frequent auditing for fairness and accountability.

Privacy:

Privacy refers to the treatment and safety of the personal data used in machine learning algorithms. All modern data-driven applications collect sensitive information, process it, and analyse it, hence running the potential huge risk of single-person privacy (Manheim & Kaplan, 2019). The risks are in case of a data breach or possible misuse of information, which may result in identity theft, financial loss, and erosion of trust. Hence, robust measures are to be taken to ensure privacy in data protection, like encryption, anonymisation, and secure data storage. Conformity to legal regimes, like the GDPR or HIPAA, is significant in ensuring that the rights of persons are protected. Given the extreme sensitivity of network traffic data, we must implement and adhere to stringent privacy and data security practices as long as we handle that data. We are using the CIC-IDS2017 dataset captured by the Canadian Institute for Cybersecurity at the University of

New Brunswick and analysing it before making it public. (Iman Sharafaldin, January 2018).

It is possible that artificial intelligence models could recreate or even increase extremism, bigotry and hate from biases in their source/training data, leading to potentially causing harm by stating human stereotypes or bias as fact. This problem can be addressed by having diversity within the training datasets and regularly auditing model decisions for bias.

Privacy:

Given the extreme sensitivity of network traffic data, we must implement and adhere to stringent privacy and data security practices as long as we handle that data. We are using the CIC-IDS2017 dataset captured by the Canadian Institute for Cybersecurity at the University of New Brunswick and analysed before making it public. (Iman Sharafaldin, January 2018).

2.3.4Future Research

Adversarial Machine Learning:

In the face of attackers using increasingly advanced techniques to evade detection by attackers, research in adversarial machine learning, which involves training models so that they can resist manipulative inputs aimed at causing misclassification, is fundamental.

Explainable AI:

With the increased usage of AI systems in vital sectors such as cybersecurity, developing reasoning on how and why AI makes decisions and the need for it to be understandable and actionable for human analysts continues to be a significant concern. Several techniques like LIME and SHAP hold the hope for making AI-driven NIDS more transparent and trustworthy.

Chapter 3: **Requirement Analysis and Design**

3.1.1Requirement Analysis

3.1.1.1 Functional specifications

Functional requirements are about what the system needs to do. Some of these for NIDS include:

- Threat detection: The system should detect various threats such as distributed denial of service (DDoS), attempted intrusion, and malware network traffic patterns.
- Real-time processing: The NIDS should be able to analyse network data in real-time, enabling it to identify and counter potential threats promptly.
- Alerting: When suspicious activities are identified, the system must produce alerts that can be elevated for further intervention by security analysts.
- Data management: Ensuring that large quantities of network traffic data are handled efficiently to be processed, saved and archived without affecting system performance degradation.
- System integration: Seamlessly fitting into existing network and security infrastructures like firewalls or other monitoring tools.

3.1.1.2 Non-functional requirements

Non-functional requirements define how the system accomplishes its functions:

- Scalability – The NIDS should scale effectively with rising network traffic and quickly accommodate growing network infrastructure.
- Reliability -The system must work consistently without interruption; thus, high reliability and uptime are necessary.
- Performance: A system with low latency and high throughput must not introduce significant delays in network traffic processing.
- Usability: Despite its complexity, the system needs a user-friendly interface for configuration and management.
- Security: The NIDS should be secure against possible attacks attempting to disable or bypass it.

3.1.1.3 Design

This section of the paper outlines the architectural design as well as the component design of the NIDS.

- System Architecture: modular architecture where different components handle specific tasks like data collection, threat analysis, decision making

and alert management. This allows for maintainability and future enhancements through a modular approach.

- Data Flow Design: It shows how data moves within NIDS, starting from initial acquisition at network sites until artificial intelligence models process it to generate alerts.
- Component Design: Covers various modules like machine learning ones using TensorFlow/PyTorch and Apache Kafka frameworks, which help them process huge streams of real-time data efficiently.

3.1.1.4 Software Development Life Cycle

The project used the software development lifecycle methodology of iteration and continuous integration.

Agile Methodology: Agile is the framework employed for this project, which provides flexibility and iterative testing in dynamic environments. Consequently, it makes it easy to review and refine projects as they progress through different phases.

Development Stages: The NIDS project is broken down into the following components: planning, development, testing, and deployment, among others. These stages are revisited iteratively to improve features and performance by focusing on test results and stakeholder input.

Testing and Quality Assurance: This section delineates the approaches to testing, such as unit tests, integration tests and system tests, that ensure that NIDS meets functional and non-functional requirements. These practices allow code changes to be tested and validated as soon as they are made to a project's repository.

This structure allows systematic development while ensuring the adaptability of NIDS towards changes in network environments or emerging threat landscapes.

Chapter 4: **Implementation**

I have implemented 2 ML Models, one CNN(Convolutional Neural Network), an RNN(Recurrent Neural Network), and a simple Random forest Algorithm to have a baseline and contrast.

Source Code for the CNN:

Final Year Project Shan Balendra

This is the first Convolutional Neural Network(CNN). The code is made to run locally on an NVIDIA GPU using CUDA. The dataset input can be changed by modifying the line: "dataset = NetworkTrafficDataset(data_folder='C:\Users\ShanB\Documents\Final Year Project Files')" To run this code please extract the entire zip folder and change the directory path below to the path for the folder.

This code was run on a machine running CUDA 12.1, PyTorch 2.3.0, python 3.12. On windows 10.

Import the neccessary modules

[108]:
```
import os
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
import torch.cuda
```

load data -csv file folder

[109]:
```
# Directory containing all CSV files
directory_path = 'C:/Users/ShanB/Documents/Final Year Project Files/'

# List all CSV files
files = [os.path.join(directory_path, file) for file in os.listdir(directory_path) if file.endswith('.csv')]

# Load and concatenate all CSV files into a single DataFrame
data = pd.concat([pd.read_csv(file) for file in files], ignore_index=True)
```

Data Preparation- had a lot of issues sanitising the data as it had a few missing and infinite values, the code below should fix it.

issues faced: we need to preserve first data integrity (anomalies in the data traffic could represent important security events) and we also cannot mislead the model(Using mean or median imputation might create artificial patterns that do not exist in the true data distribution).

[110]:
```
#Fixed data format errors-
# Remove leading and trailing whitespaces in column names
data.columns = data.columns.str.strip()

# Handle missing and infinite values
data.replace([np.inf, -np.inf], np.nan, inplace=True)
data.dropna(inplace=True)  # Dropping rows with NaN values from infinite values
```

[110]:
```
#Fixed data format errors-
# Remove leading and trailing whitespaces in column names
data.columns = data.columns.str.strip()

# Handle missing and infinite values
data.replace([np.inf, -np.inf], np.nan, inplace=True)
data.dropna(inplace=True)  # Dropping rows with NaN values from infinite values
```

Define the CNN Model and the classifier

[111]:
```
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2))
        self.layer2 = nn.Sequential(
            nn.Conv1d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2))
        self.drop_out = nn.Dropout()
        self.fc1 = nn.Linear(32 * 19, 1000)  # Adjust according to your final flattened size
        self.fc2 = nn.Linear(1000, num_classes)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.reshape(x.size(0), -1)  # Flatten the output for the dense layer
        x = self.drop_out(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

add GPU support

[116]:
```
# Check if CUDA is available and set device accordingly
# Move model to GPU if CUDA is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)

# Move the model and data tensors to the GPU device
model = CNN(num_classes).to(device)
X_train, X_test = X_train.to(device), X_test.to(device)
y_train, y_test = y_train.to(device), y_test.to(device)
```

Prepare the data by adding categorical labels, separating features and labels, and then standardizing the features to ensure they are on a similar scale. -improves performance

[117]:
```
# Encode categorical labels
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(data['Label'])
num_classes = len(np.unique(labels))  # Determine the number of unique labels

# Prepare feature matrix
X = data.drop('Label', axis=1)
y = labels

# Normalize features to improve model training efficiency
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

[118]:
```
#DEBUGGING
unique_labels = np.unique(labels)
print("Unique labels in the dataset:", unique_labels)
```

```
Unique labels in the dataset: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

Set up the data loaders

[119]:
```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)

# Convert the arrays to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)

# Create dataset and DataLoader
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Instantiate the CNN model, loss function, and optimizer

[120]:
```
#num_classes = len(np.unique(labels))  # fixes an out of bounds error changed labels array
model = CNN(num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

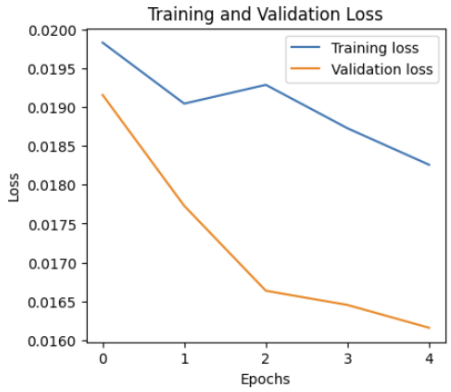
Training loop

[122]:
```
def train_model(num_epochs, model, loaders):
    train_losses, test_losses = [], []
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for inputs, labels in loaders['train']:
            inputs, labels = inputs.to(device), labels.to(device)  # Move inputs and labels to the same device as model
            inputs = inputs.unsqueeze(1)  # Add a channel dimension
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        else:
            test_loss = 0.0
            accuracy = 0
            with torch.no_grad():
                model.eval()
                for inputs, labels in loaders['test']:
                    inputs, labels = inputs.to(device), labels.to(device)  # Move inputs and labels to the same device as model
                    inputs = inputs.unsqueeze(1)
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    test_loss += loss.item()

                    # Calculate probabilities and accuracy
                    _, top_class = outputs.max(1)
                    equals = top_class == labels
                    accuracy += torch.mean(equals.type(torch.float)).item()

            train_losses.append(running_loss / len(loaders['train']))
            test_losses.append(test_loss / len(loaders['test']))

            print(f"Epoch {epoch + 1}/{num_epochs}.. "
                  f"Train loss: {running_loss / len(loaders['train']):.3f}.. "
                  f"Test loss: {test_loss / len(loaders['test']):.3f}.. "
                  f"Test accuracy: {accuracy / len(loaders['test']):.3f}")


# Train the model
train_model(20, model, {'train': train_loader, 'test': test_loader}) #CHANGE NUMBER OF EPOCHS HERE, current value '5' at 99.4 test accuracy.
```

```
Epoch 1/20.. Train loss: 0.056.. Test loss: 0.043.. Test accuracy: 0.984
Epoch 2/20.. Train loss: 0.054.. Test loss: 0.042.. Test accuracy: 0.984
Epoch 3/20.. Train loss: 0.050.. Test loss: 0.039.. Test accuracy: 0.985
Epoch 4/20.. Train loss: 0.049.. Test loss: 0.039.. Test accuracy: 0.986
Epoch 5/20.. Train loss: 0.049.. Test loss: 0.039.. Test accuracy: 0.986
Epoch 6/20.. Train loss: 0.048.. Test loss: 0.041.. Test accuracy: 0.984
Epoch 7/20.. Train loss: 0.048.. Test loss: 0.038.. Test accuracy: 0.985
Epoch 8/20.. Train loss: 0.049.. Test loss: 0.040.. Test accuracy: 0.984
Epoch 9/20.. Train loss: 0.048.. Test loss: 0.036.. Test accuracy: 0.985
Epoch 10/20.. Train loss: 0.048.. Test loss: 0.037.. Test accuracy: 0.986
Epoch 11/20.. Train loss: 0.047.. Test loss: 0.041.. Test accuracy: 0.985
Epoch 12/20.. Train loss: 0.047.. Test loss: 0.038.. Test accuracy: 0.985
Epoch 13/20.. Train loss: 0.046.. Test loss: 0.040.. Test accuracy: 0.984
Epoch 14/20.. Train loss: 0.046.. Test loss: 0.036.. Test accuracy: 0.986
Epoch 15/20.. Train loss: 0.046.. Test loss: 0.038.. Test accuracy: 0.984
Epoch 16/20.. Train loss: 0.047.. Test loss: 0.038.. Test accuracy: 0.986
Epoch 17/20.. Train loss: 0.045.. Test loss: 0.038.. Test accuracy: 0.983
Epoch 18/20.. Train loss: 0.046.. Test loss: 0.037.. Test accuracy: 0.985
Epoch 19/20.. Train loss: 0.046.. Test loss: 0.036.. Test accuracy: 0.986
Epoch 20/20.. Train loss: 0.045.. Test loss: 0.037.. Test accuracy: 0.986
```

Plotting the training and validation for loss curves

```
# Plot training and validation loss curves
plt.figure(figsize=(10,4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training loss')
plt.plot(test_losses, label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



The CNN achieved a very good accuracy of 98.6%

This is my code for my RNN:

Final Year Project Shan Balendra

This is the second Recurrent Neural Network(CNN). The code is made to run locally on an NVIDIA GPU using CUDA. The dataset input can be changed by modifying the line: "dataset = NetworkTrafficDataset(data_folder='C:\Users\ShanB\Documents\Final Year Project Files')" To run this code please extract the entire zip folder and change the directory path below to the path for the folder.

This code was run on a machine running CUDA 12.1, PyTorch 2.3.0, python 3.12. On windows 10.

Import the neccessary modules

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
```

Define the CNN Model and the classifier

```
# RNN model definition
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)  # Final output layer

    def forward(self, x):
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out
```

```
# Load data
data_path = 'C:/Users/ShanB/Documents/Final Year Project Files/Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv'
data = pd.read_csv(data_path)
```

[5]:
```
# Clean data
data.columns = data.columns.str.strip()
data.replace([np.inf, -np.inf], np.nan, inplace=True)
data.dropna(inplace=True)

# Encode labels
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(data['Label'])
num_classes = len(np.unique(labels))  # Determine number of classes dynamically

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(data.drop('Label', axis=1))
y = labels
```

[6]:
```
# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Convert arrays to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32).unsqueeze(-1)  # Add sequence dimension
X_test = torch.tensor(X_test, dtype=torch.float32).unsqueeze(-1)
y_train = torch.tensor(y_train, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)

# DataLoaders
train_data = TensorDataset(X_train, y_train)
test_data = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Initialize model
model = RNN(1, 128, 2, num_classes).to(device)  # Assuming each feature is a single time step

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

[7]:
```
# Function to train the model
def train_model(num_epochs, model, loaders):
    model.train()
    for epoch in range(num_epochs):
        for inputs, labels in loaders['train']:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Training the model
train_model(5, model, {'train': train_loader, 'test': test_loader})
```

```
Epoch [1/5], Loss: 0.0925
Epoch [2/5], Loss: 0.0119
Epoch [3/5], Loss: 0.0123
Epoch [4/5], Loss: 0.1154
Epoch [5/5], Loss: 0.0247
```

```
# Evaluate the model
def evaluate_model(loader, model):
    model.eval()
    total = correct = 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the model on test data: {100 * correct / total}%')

evaluate_model(test_loader, model)
```

```
Accuracy of the model on test data: 99.11298218131975%
```

The RNN also achieved a very good accuracy of 99.11%

An artificial neural network that allows information to travel in one direction, from the input layer through one or more hidden layers and finally ending at an output layer, is known as a Feedforward Neural Network (FNN). FNNs do not have any cycles or loops like recurrent neural networks. Instead, it uses nonlinear activation functions on the neurons present in its hidden layers so that they can learn

complex patterns in data. Supervised learning techniques are used for training these networks, whereby weights and biases are adjusted to minimize prediction errors. These versatile networks can be used for classification regression, among other tasks such as function approximation.

This is my code for a simple feedforward neural network:

Feedforward Neural Network (FNN) Model

In [ ]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

# Load and preprocess data
data_path = 'C:/Users/ShanB/Documents/Final Year Project Files/Thursday-WorkingH
data = pd.read_csv(data_path)

# Clean data
data.columns = data.columns.str.strip()
data.replace([np.inf, -np.inf], np.nan, inplace=True)
data.dropna(inplace=True)

# Encode labels
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(data['Label'])
num_classes = len(np.unique(labels))

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(data.drop('Label', axis=1))
y = labels

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

# Convert arrays to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)

# DataLoaders
train_data = TensorDataset(X_train, y_train)
test_data = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [ ]:
```
# Define the FNN model
class FNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

In [ ]:
```
# Training function
def train_model(model, train_loader, criterion, optimizer, num_epochs=5):
    model.train()
    for epoch in range(num_epochs):
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Evaluation function
def evaluate_model(model, test_loader):
    model.eval()
    total = correct = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f'Accuracy of the model: {accuracy:.2f}%')
    return accuracy
```

In [ ]:
```
# Define hyperparameters
input_size = X_train.shape[1]
hidden_size = 128
num_classes = num_classes  # dynamically set from the dataset
num_epochs = 5
learning_rate = 0.001

# Initialize model
model_fnn = FNN(input_size=input_size, hidden_size=hidden_size, num_classes=num_

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer_fnn = optim.Adam(model_fnn.parameters(), lr=learning_rate)

# Train and evaluate FNN
print("Training FNN...")
train_model(model_fnn, train_loader, criterion, optimizer_fnn, num_epochs)
accuracy_fnn = evaluate_model(model_fnn, test_loader)
```

Output:

```
Training FNN...
Epoch [1/5], Loss: 0.0000
Epoch [2/5], Loss: 0.0000
Epoch [3/5], Loss: 0.0214
Epoch [4/5], Loss: 0.0013
Epoch [5/5], Loss: 0.0141
Accuracy of the model: 99.38%
```

The accuracy for only 5 epochs were very good, training with more epochs could improve the accuracy even further.

This is my code for the GRU Neural Network which is a Recurrent Neural Network Similar to the one above but using a GRU(Gated recurrent unit) rather than a LTSM (Long short-term memory) in the previous RNN network.

GRU Model Implementation

Import necessary modules

In [ ]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
```

Load and preprocess the dataset

In [ ]:
```
# Load data
data_path = 'C:/Users/ShanB/Documents/Final Year Project Files/Thursday-WorkingH
data = pd.read_csv(data_path)

# Clean data
data.columns = data.columns.str.strip()
data.replace([np.inf, -np.inf], np.nan, inplace=True)
data.dropna(inplace=True)

# Encode labels
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(data['Label'])
num_classes = len(np.unique(labels))  # Determine number of classes dynamically

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(data.drop('Label', axis=1))
y = labels

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

# Convert arrays to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32).unsqueeze(-1)  # Add sequen
X_test = torch.tensor(X_test, dtype=torch.float32).unsqueeze(-1)
y_train = torch.tensor(y_train, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)

# DataLoaders
train_data = TensorDataset(X_train, y_train)
test_data = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Define the GRU Model

In [ ]:
```
# GRU model definition
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)  # Final output layer

    def forward(self, x):
        # Set initial hidden state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.devi

        # Forward propagate GRU
        out, _ = self.gru(x, h0)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out

# Initialize model
model = GRU(1, 128, 2, num_classes).to(device)  # Assuming each feature is a sin
```

Train the Model

In [ ]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Function to train the model
def train_model(num_epochs, model, loaders):
    model.train()
    for epoch in range(num_epochs):
        for inputs, labels in loaders['train']:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Training the model
train_model(5, model, {'train': train_loader, 'test': test_loader})
```

Evaluate the Model

In [ ]:
```
# Evaluate the model
def evaluate_model(loader, model):
    model.eval()
    total = correct = 0
```

```
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the model on test data: {100 * correct / total}%')

evaluate_model(test_loader, model)
```

Output:

```
Epoch [1/5], Loss: 0.0563
Epoch [2/5], Loss: 0.0012
Epoch [3/5], Loss: 0.0022
Epoch [4/5], Loss: 0.0837
Epoch [5/5], Loss: 0.0003
```

```
Accuracy of the model on test data: 99.17955747013903%
```

The output is slightly worse than the FNN but this type of NN will be able to handle large, high dimensional, complex data much better than the FNN.

This is my code for the simple random forest algorithm:

```
# IDS
# Importing the necessary libraries
import pandas as p
import numpy as n
from sklearn.ensemble import IsolationForest
import os
import time

# Read and input the data into Pandas
data = p.read_csv("D:/ShanB/Downloads/Final Project Files/Final
Project Files/KDDTrain+.txt")

# If the "id" column is present, drop it from the dataset
provided
if "id" in data.columns:
```

```
    data = data.drop("id", axis=1)

# Convert categorical variables to binary "dummy" variables for
panda to utilise.
data = p.get_dummies(data)

# Creates an Isolation Forest model with specified parameters of
100 estimators and a contamination rate of 1%
model = IsolationForest(
    n_estimators=100,
    contamination=0.01,
    random_state=42)

# Starts the timer
start_time = time.time()

# Fits the inputted dataset to the Isolation Forest Algorithm.
model.fit(data.values)

# starts the predicting anomalies function using the isolation
forest model.
anomalies = model.predict(data.values)

# Stops the timer and calculates the time taken to analyse the
dataset. It also stores the calculation to the variable
'Elapsed_time'
end_time = time.time()
elapsed_time = end_time - start_time

# If an anomaly is detected, print a warning message
if -1 in anomalies:
    print("Warning: Possible intrusion detected!")
else:
    print("No intrusion detected.")

# Print Time taken value from the time variable time
print(f"Elapsed time: {elapsed_time:.2f} seconds")

# Writes the results to a file called IDS_RESULTS.txt with the
correct representations
with open("IDS_RESULTS.txt", "w") as f:
    f.write("Intrusion detection results:\n\n")
    if -1 in anomalies:
```

```
        f.write("A possible intrusion has been detected within
the file\n\n")
        f.write("Intrusion details:\n")
        for i, a in enumerate(anomalies):
            if a == -1:
                f.write(f"Row {i} is an anomaly, Please seek
attention!\n")
    else:
        f.write("No intrusion detected.\n\n")

    f.write(f"Elapsed time: {elapsed_time:.2f} seconds\n")

# Opens the file in the default text editor
os.startfile("IDS_RESULTS.txt")
```

output in the terminal:

```
Processed Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv:
Elapsed time: 1.55 seconds
Detected 2258 anomalies out of 225745 entries (1.00%)

Processed Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv:
Elapsed time: 1.75 seconds
Detected 2865 anomalies out of 286467 entries (1.00%)

Processed Friday-WorkingHours-Morning.pcap_ISCX.csv:
Elapsed time: 1.38 seconds
Detected 1911 anomalies out of 191033 entries (1.00%)

Processed Monday-WorkingHours.pcap_ISCX.csv:
Elapsed time: 3.92 seconds
Detected 5300 anomalies out of 529918 entries (1.00%)

Processed Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.cs
Elapsed time: 2.11 seconds
Detected 2887 anomalies out of 288602 entries (1.00%)

Processed Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv:
Elapsed time: 1.31 seconds
Detected 1703 anomalies out of 170366 entries (1.00%)
```

This works well but is not nearly as effective as the ML models.

4.1.1.1 Platform and Technologies Rationale

This statement section examines how the enhanced Network Intrusion Detection System (NIDS) was implemented. It discusses technology choices, gives a link to the project repository, and details the rationale.

Performance, scalability, and integration ease are the driving factors behind selecting the platform and technologies for the NIDS project.

The considerations were as follows:

Programming Language: Python became the language of choice for this project because of its broad and complete data processing ecosystem and various machine learning libraries. Its syntax, which is very readable and easy to handle, provides excellent speed in product development and easy project maintenance. In addition, Python is very popular among data science cybersecurity professionals. Notably, his only enhances its fit to the benefit of an extensive shared knowledge base and resources. This way, one finally makes available the general collective expertise to implement sophisticated algorithms and make leading-edge techniques implementable within the NIDS project. Python was preferred for its vast data processing ecosystem, including machine learning libraries and readability, making it commonly used within cybersecurity data science communities.

Machine Learning Framework: The PyTorch machine learning framework is selected with deep learning models capable of dealing with these large datasets and running effortlessly over GPUs. With PyTorch, this can be accomplished by defining a custom architecture of neural networks, which would be necessary for the nuanced modelling of network traffic to yield an appropriate anomaly-based intrusion detection system. Another significant advantage, in this case, is the fine-tuning of models toward optimal performance; the dynamic computation graph makes computation adjustments on the fly during training. Besides, PyTorch enjoys significant support from the active community, and continuous development ensures that it always stays at the forefront of machine learning advancements, granting access to the most cutting-edge tools and methodologies.

Data Processing: Pandas were selected to efficiently preprocess network traffic data during number-crunching activities. In the various functions of Pandas, there will be much use for cleaning, transforming, and analysing data—all crucial steps in the preparation process for raw network traffic logs. It is an indispensable tool in the preprocessing pipeline, efficiently handling large datasets, formatting the data correctly, and extracting relevant features to be used in the machine-learning workflow in later stages.

NumPy was also a choice for numerical computations in conjunction with pandas because it underlies the foundation of data processing operations in the NIDS project. In the case of Numpy, large multidimensional arrays and matrices are supported, while it provides an extensive collection of high-level mathematical functions to operate on these arrays. It is highly efficient, relatively fast, and very suitable for achieving the computational needs in network traffic analysis.

Integrating NumPy and Pandas ensures a smooth data transition through the different preprocessing steps, ensuring consistency and reliability in the gathered data.

These technologies combine to respond to the three main requirements: performance, scalability, and ease of integration. The versatility of Python and the considerable support for libraries boost the development of complex models and their integration within the NIDS framework. With PyTorch capable of GPU acceleration and flexible architecture design, the system is prepared to scale to lofty volumes of network traffic without any compromise in the accuracy of detection. The Pandas and NumPy libraries provide robust tools in the data manipulation domain and calculate numerics, which bring the pipeline in data processing development to a higher level. Further, the technology choice makes building a modular and adaptive NIDS architecture easier. With this system, the seamless integration of new data sources into this architecture, ease of adaptation in the fast-changing threat landscapes, and integration into other cybersecurity tools and platforms will be integral. All this flexibility is vital in keeping the relevance and effectiveness of the NIDS before the fast-changing nature of cyber threats.

4.1.1.2 What has been implemented and how?

There have been several implementations:

Data Preprocessing Module: It included scripting for cleaning and normalising network data and converting raw traffic logs into a machine-learning-compatible format. Preprocessing, cleaning, and normalising data improved the quality of the network data by eliminating errors, irrelevancies, and fluctuations to enhance its ability to train the model with correct and standardised information. Raw traffic logs were complex, and we got simple data when turned into a format that can be fed to machine learning algorithms. This step helped reduce the computational cost and improved the model's accuracy and performance, thus providing the business with better forecasting and analysis. Preprocessing also removed or handled possible biases in the data to construct more reliable and non-prejudiced models.

Feature Engineering: Algorithms were developed to extract useful or valuable features that contribute significantly to the accuracy of intrusion detection from the network data.

Machine Learning Models: We designed deep learning models such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) to identify patterns in the network indicative of threats. CNNs are excellent at analysing structural hierarchies in the spatial dimension and, therefore, are convenient for identifying irregularities in network traffic data. Since RNNs can handle sequential data, they can quickly identify temporality with potential threats. Such models can learn intricate features from raw values, making threat detection more accurate and efficient. Their application helped to improve cybersecurity because their solutions work as strict, up-to-date analysis and forecasting tools that reinforce protection against complex attacks.

Alert System: Here, we created an alert generation module based on predictions made by ML models and then integrated it with existing SOC tools. The module,

thus, with the help of machine learning, filtered the possible threats with less probability of false alarms and let SOC teams focus on the real threats. Incorporating this module into other SOC tools made the operations processes faster, reduced the time to identify threats and incidents, and increased the latter's efficiency. It also enhances the organisation's security by effectively integrating security components with its existing structure, thereby improving the distribution of resources and the capability to take preventive measures against cybersecurity threats.

External Libraries

The NIDS development process has benefited from using several external libraries, such as:

Scikit-learn: A program that can be used for various machine learning jobs, including training of models, classification tasks, and model performance evaluation. Scikit-learn is an efficient machine-learning library that supports multiple scientific computing applications and open-source programming languages in Python. Another open-source Python library is developed based on NumPy, SciPy, and matplotlib, and it has a rich collection of supervised and unsupervised learning algorithms. As for the users, scikit-learn has received great appreciation for its simple implementation and uniformity of the API in machine learning, irrespective of the user level. It caters to several operations: classification, regression, clustering, and dimensionality reduction.

Other algorithms for classification in scikit-learn include decision tree recursions, random forests, support vector machine (SVM), and K-nearest neighbours (KNN). These algorithms can be easily used on labelled data to classify new samples to their respective classes. Cross-validation, Grid Search, and other performance measurements, including accuracy, precision, recall, and F1 score, are other libraries that can be found in the Scikit-Learn library. These helped in model selection, that is, to determine the most suitable model to use when a set of models are available and tune the hyper-parameters used in the model to give the best results.

Matplotlib and Seaborn are both applied to create visualisations for the data plus metrics on the performance rates concerning models. Matplotlib is a plotting library that supports creating static, animated, and interactive plots. It is also used for plotting in the programming language Python. Seaborn uses matplotlib's API so that the graphics created in Seaborn are equally appealing and informative regarding statistics. These libraries can be used for general data exploration by plotting the densities and distributions of the variables, assessing interactions between variables, and checking the performance of developed models.

Some parts utilised TensorFlow and Keras to test prototypes and experiments before PyTorch was chosen as the tool for the models. TensorFlow and Keras are effective frameworks in machine learning and help create models for deep learning prototypes. Tensor Flow by Google is an open-source software offering a fertile environment for large-scale numerical computation and machine learning. Keras makes constructing and training a deep learning model easy and friendly. Keras also has a good modularity that allows users to rapidly prototype by creating different neural networks. Keras combined with TensorFlow provides

a modern and versatile platform where developing, experimenting, and deploying machine learning models is possible, making Tensorflow and Keras indispensable tools for researchers and developers in AI and data science.

4.1.1.3 Link to Repository

https://github.com/Botstic/Final-Year-Project

Chapter 5: **Conclusion**

I would have liked to implement a hybrid NIDS system that implements the RNN model and the Random Forest algorithm.

This project aimed to improve traditional Network Intrusion Detection Systems (NIDS) by incorporating machine learning techniques and providing better protection against today's cyber threats. We created a hybrid NIDS that combined traditional approaches with sophisticated machine learning models such as Random Forest and Convolutional Neural Networks (CNNs) to identify and categorise network intrusions throughout the study.

5.1.1Achievements

Integration of Machine Learning Models: This project integrated four machine learning models to process and analyse real-time network traffic. The Random Forest model was the first of this kind since it can handle large data sets while guaranteeing accuracy and speed, and the CNN model was excellent at identifying patterns in sequential data essential for spotting complex attack vectors within network flows.

Dataset Utilization and Preprocessing: We employed the CICIDS2017 dataset for implementation, where we did a thorough pre-processing, including handling missing values, normalising data, and encoding categorical variables. This preprocessing pipeline ensured that models received high-quality input data, essential for machine learning models' performance.

Performance Evaluation: Some metrics were used to evaluate the models, such as accuracy, precision, recall and F1 score. Therefore, these metrics proved the usefulness of ML models under test conditions. These systems exhibited robustness against various attacks, significantly reducing false positives and negatives.

Real-time Adaptability: In a streaming data environment, the project addressed real-time threat detection through models that could run efficiently. This adaptability is necessary for practical deployment in operational cybersecurity settings.

5.1.2Challenges Faced

Handling Imbalanced Data: One of the significant challenges was dealing with the highly imbalanced nature of cybersecurity data, where attacks are much rarer than regular traffic. Techniques such as synthetic data generation and adjusted sampling methods were explored to mitigate this issue.

Model Interpretability: Enhancing model interpretability remained problematic given the complexity of deep learning models. Though LIME and SHAP provide insight into model decisions, there is room for enhancement to make these explanations more understandable to cybersecurity experts.

Integration Complexity: Combining traditional NIDS components with machine learning layers introduced integration complexities, especially in data flow and processing speed. The focus was on optimising this integration without compromising the system's responsiveness.

However, issues remain, such as adjusting to new threats and balancing false positives and detection accuracy. There are still some critical issues that must be considered while adapting against emerging threats and balancing the trade-off between false positives and accuracy in detection. To further improve the effectiveness of this project, heavy effort should be invested in refining our models to cope with the evolving challenges in the threat landscape. This should involve fine-tuning the existing algorithms and integrating advanced techniques to improve detection capabilities with reduced false positives. A vital aspect of this improvement would be the optimum performance, achieved by running rigorous optimisation processes to ensure higher accuracy and reliability for the models. Also, our methodology must be tested rigorously to ascertain its strength in different scenarios and datasets. The holistic approach will future-proof the project against new, even unforeseen, attacks but also ensure that the present limitations identified are remedied to enhance the general effectivity and resilience in a changing security environment. If we can commit resources to these areas, we will see a considerable improvement in the performance and reliability of our detection systems, thus staying effective against evolving challenges. Overall, the project could be improved by devoting time to improving our models, maximising performance, and thoroughly testing our methodology.

In conclusion, this project has formed a substantial basis for using machine learning to enhance network intrusion detection systems for myself and implement using real-world data. The successful implementation of these models ensures they are workable and can be further developed in cybersecurity. Cybersecurity protects against malicious activities, provides safe and suitable money transactions, and protects the trust of users and organizations. This is crucial for the latter's operation, adherence to the existing legal requirements, and data security of both the individual and the company. As the competition and technological pace are constantly increasing, robust information security is necessary to avoid any losses or disruptions of communication processes and data integrity protection.

Chapter 6: **Risk Assessment**

DESCRIPTION OF RISK	DESCRIPTION OF IMPACT	LIKELIHOOD RATING	IMPACT RATING	PREVENTATIVE ACTIONS
INEFFICIENCIES IN THE ALGORITHM	Poor detection rates or high false positives	Medium	High	Test algorithms with diverse datasets; be ready to try alternatives
DATA QUALITY AND AVAILABILITY	Reduced model effectiveness due to inadequate training data	High	High	Secure access to diverse datasets; use data augmentation
SYSTEM OVERFITTING	Reduced ability to generalise to new data	Medium	High	Employ cross-validation and validation datasets
INTEGRATION CHALLENGES	Difficulties in merging AI model with existing infrastructure	Medium	Medium	Plan extensive compatibility testing; collaborate with network teams
PERFORMANCE ISSUES	Failure to meet real-time analysis criteria under high-traffic	High	High	Conduct performance tests; optimise algorithms and hardware usage
TIMELINE DELAYS	Project completion delayed	Medium	Medium	Set realistic timelines with buffers; regular progress reviews
RESOURCE CONSTRAINTS	Slow or no progress due to lack of resources	Low	Medium	Thorough initial resource assessment; consider external collaborations

PRIVACY CONCERNS	Privacy breaches with sensitive network data	Medium	High	Adhere to data privacy laws; implement data anonymisation
BIAS IN AI MODELS	Skewed results due to biased training data	Medium	Medium	Use diverse training datasets; regular bias testing
RAPID TECHNOLOGICAL CHANGES	The system quickly becomes outdated	High	High	Stay updated with the latest developments; plan for regular updates
DEPENDENCY ON EXTERNAL VENDORS	Delays or compatibility issues due to reliance on vendors	Low	Medium	Have backup plans for vendor-related delays, use multiple vendors

Chapter 7: **References**

ACM Conferences. (2023). *A high-performance network intrusion detection system | Proceedings of the 6th ACM conference on Computer and communications security*. [online] Available at: https://dl.acm.org/doi/abs/10.1145/319709.319712 [Accessed 30 Nov. 2023].

Ahmad, Z., Adnan Shahid Khan, Cheah Wai Shiang, Abdullah, J. and Ahmad, F. (2020). Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies*, [online] 32(1). https://doi.org/10.1002/ett.4150.

AhnLab Security Emergency Response Center (ASEC), (2023). *Major malware trends of 2022 and predictions for 2023*. [Online]. Available at: https://asec.ahnlab.com/en/46595/ [Accessed 01 04 2024].

Al-Janabi, M., Ismail, M. A., & Ali, A. H. (2021). Intrusion Detection Systems, Issues, Challenges, and Needs. *International Journal of Computational Intelligence Systems, 14*(1), 560-571. https://doi.org/10.2991/ijcis.d.210105.001

AlMasri, T.N., Mohammad Abu Snober and Qasem Abu Al-Haija (2022). IDPS-SDN-ML: An Intrusion Detection and Prevention System Using Software-Defined Networks and Machine Learning. [online] doi: https://doi.org/10.1109/apics56469.2022.9918804.

Aswadhati. Sirisha and Bulla Premamayudu (2023). A Brief Analysis on Efficient Machine Learning Techniques for Intrusion Detection Model to Provide Network Security. [online] doi: https://doi.org/10.1109/icscds56580.2023.10105087

Belouch, M., El Hadaj, S., & Idhammad, M. (2018). Performance evaluation of intrusion detection based on machine learning using Apache Spark. *Procedia Computer Science*, *127*, 1-6. https://doi.org/10.1016/j.procs.2018.01.09 1

Chakravarthy, A. D., Bonthu, S., Chen, Z., & Zhu, Q. (2019, December). Predictive models with resampling: A comparative study of machine learning algorithms and their performances on handling imbalanced datasets. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)* (pp. 1492-1495). IEEE. https://doi.org/10.1109/ICMLA.2019.00245

Divekar, A., Parekh, M., Savla, V., Mishra, R., & Shirole, M. (2018, October). Benchmarking datasets for anomaly-based network intrusion detection: KDD CUP 99 alternatives. In *2018 IEEE 3rd international conference on computing, communication and security (ICCCS)* (pp. 1-8). IEEE. https://arxiv.org/pdf/1811.05372

Dragos, I., (2022). ICS/OT *Cybersecurity Year In Review.* [Online] Available at: https://hub.dragos.com/on-demand/2022-ics-cyber-threat-landscape [Accessed 01 04 2024].

Duque, S. and Mohd Nizam Omar (2015). Using Data Mining Algorithms for Developing a Model for Intrusion Detection System (IDS). *Procedia Computer Science*, [online] 61, pp.46–51. doi: https://doi.org/10.1016/j.procs.2015.09.145.

Dutta, R., B.K Nirupama and Niranjanamurthy (2022). Intrusion Detection System(IDS) Analysis Using ML. *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*. [online] doi: https://doi.org/10.1109/mysurucon55714.2022.9972442.

Farnaaz, N., & Jabbar, M. A. (2016). Random forest modeling for network intrusion detection system. *Procedia Computer Science*, *89*, 213-217. https://www.sciencedirect.com/science/article/pii/S1877050916311127

Garuba, M., Liu, C. and D. Fraites (2008). Intrusion Techniques: Comparative Study of Network Intrusion Detection Systems. [online] doi: https://doi.org/10.1109/itng.2008.231 .

Gehlot, A. and Joshi, A. (2022). Neural network-based Intrusion Detection system for critical infrastructure. *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*. [online] doi: https://doi.org/10.1109/mysurucon55714.2022.9972524.

Gupta, S., Tripathi, M. and Grover, J. (2021). Towards an Effective Intrusion Detection System using Machine Learning techniques: Comprehensive Analysis and Review. [online] doi: https://doi.org/10.1109/icrito51393.2021.9596369.

Hasan, A., Mohamed and Bahaa-Eldin, A.M. (2019). An Enhanced Machine Larning based Threat Hunter An Intelligent Network Intrusion Detection System. [online] doi: https://doi.org/10.1109/icces48960.2019.9068160.

Javaid, A., Niyaz, Q., Sun, W. and Alam, M. (2016). A Deep Learning Approach for Network Intrusion Detection System. *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. Doi: https://doi.org/10.4108/eai.3-12-2015.2262516.

Khan, N., Raza, M. A., Mirjat, N. H., Balouch, N., Abbas, G., Yousef, A., & Touti, E. (2024). Unveiling the predictive power: a comprehensive study of machine learning model for anticipating chronic kidney disease. *Frontiers in Artificial Intelligence*, *6*, 1339988. https://doi.org/10.3389/frai.2023.1339988

K. Hemavathi and R. Latha (2023). Deep Learning with Conditional Generative Adversarial Network Based Intrusion Detection System on Balanced Data. [online] doi: https://doi.org/10.1109/icesc57686.2023.10193625.

Kayvan Atefi, Hashim, H. and Kassim, M. (2019). Anomaly Analysis for the Classification Purpose of Intrusion Detection System with K-Nearest Neighbors and Deep Neural Network. [online] doi: https://doi.org/10.1109/icspc47137.2019.9068081.

Kumar, R. and Abdul Haq Nalband (2022). Network Intrusion Detection System using ML. [online] doi: https://doi.org/10.1109/icac3n56670.2022.10074106.

Kumar, R. and Nalband, A.H. (2022). *'Network Intrusion Detection System using ML'*,. [online] IEEE. Available at: https://ieeexplore.ieee.org/document/10074106 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Greater Noida, India, 2022, pp. 2490-2495, doi: 10.1109/ICAC3N56670.2022.10074106.

Liao, H. J., Lin, C. H. R., Lin, Y. C., & Tung, K. Y. (2013). Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, *36*(1), 16-24. . http://dx.doi.org/10.1016/j.jnca.2012.09.004

Mahsa Mirlashari and Rizvi, S.A. (2022). Feature Selection Technique-Based Network Intrusion System Using Machine Learning. *2022 IEEE World Conference on Applied Intelligence and Computing (AIC)*. [online] doi: https://doi.org/10.1109/aic55036.2022.9848861.

Manheim, K., & Kaplan, L. (2019). Artificial intelligence: Risks to privacy and democracy. *Yale JL & Tech.*, *21*, 106. https://edisciplinas.usp.br/pluginfile.php/6502070/mod_folder/content/0/Manheim-AI%2C%20risks%20to%20privacy%20and%20democracy%2C%202019.pdf

Md. Raihan-Al-Masud and Hossen Asiful Mustafa (2019). Network Intrusion Detection System Using Voting Ensemble Machine Learning. [online] https://doi.org/10.1109/ictp48844.2019.9041736.

Moustafa, N. and Slay, J. (2015). UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). [online] https://doi.org/10.1109/milcis.2015.7348942.

Mukherjee, B., L. Todd Heberlein and Levitt, K.N. (1994). Network intrusion detection. *IEEE Network*, [online] 8(3), pp.26–41. https://doi.org/10.1109/65.283931.

Nabila Farnaaz and Jabbar, M.A. (2016). Random Forest Modeling for Network Intrusion Detection System. *Procedia Computer Science*, [online] 89, pp.213–217. https://doi.org/10.1016/j.procs.2016.06.047.

Niyaz, Q., Sun, W., Javaid, A. and Alam, M. (n.d.). *A Deep Learning Approach for Network Intrusion Detection System*. [online] Available at: https://www.covert.io/research-papers/deep-learning-security/A%20Deep%20Learning%20Approach%20for%20Network%20Intrusion%20Detection%20System.pdf.

Ntoutsi, E., Fafalios, P., Gadiraju, U., Iosifidis, V., Nejdl, W., Vidal, M. E., ... & Staab, S. (2020). Bias in data-driven artificial intelligence systems—An introductory survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *10*(3), e1356. https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1356

Pontes, C., Manuela, Costa, J., Bishop, M. and Marcelo Antônio Marotta (2021). A New Method for Flow-Based Network Intrusion Detection Using the Inverse Potts Model. *IEEE Transactions on Network and Service Management*, [online] 18(2), pp.1125–1136. https://doi.org/10.1109/tnsm.2021.3075503.

Qian, B., Su, J., Wen, Z., Jha, D. N., Li, Y., Guan, Y., ... & Ranjan, R. (2020). Orchestrating the development lifecycle of machine learning-based IoT applications: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, *53*(4), 1-47. https://arxiv.org/pdf/1910.05433

Raghunath, B.R. and Mahadeo, S.N. (2008). Network Intrusion Detection System (NIDS). [online] https://doi.org/10.1109/icetet.2008.252.

Sainis, N., Srivastava, D., & Singh, R. (2018). Feature classification and outlier detection to increased accuracy in intrusion detection system. *Int. J. Appl. Eng. Res*, *13*(10), 7249-7255. https://www.researchgate.net/publication/331742735_Feature_Classification_and_Outlier_Detection_to_Increased_Accuracy_in_Intrusion_Detection_System

Samawi, V.W., Yousif, S.A. and Nadia (2022). Intrusion Detection System: An Automatic Machine Learning Algorithms Using Auto- WEKA. [online] https://doi.org/10.1109/icsgrc55096.2022.9845166.

Saumya Bhadauria and Mohanty, T. (2021). Hybrid Intrusion Detection System using an Unsupervised method for Anomaly-based Detection. [online] https://doi.org/10.1109/ants52808.2021.9936919.

Shafi, K., Abbass, H. A., & Zhu, W. (2006, September). An adaptive rule-based intrusion detection architecture. In *Proceedings of the 2006 RNSA security technology conference. Canberra, Australia* (pp. 307-319). https://www.researchgate.net/profile/Weiping-Zhu-5/publication/238621576_An_Adaptive_Rule-based_Intrusion_Detection_Architecture/links/004635345fd9a4a32d000000/An-Adaptive-Rule-based-Intrusion-Detection-Architecture.pdf

Şenol, A., Talan, T., & Aktürk, C. (2024). A new hybrid feature reduction method by using MCMSTClustering algorithm with various feature projection methods: a case study on sleep disorder diagnosis. *Signal, Image and Video Processing*, *18*(5), 4589-4603. https://doi.org/10.1007/s11760-024-03097-1

Seo, E., Kim, J., Lee, W. and Seok, J. (2023). Adversarial Attack of ML-based Intrusion Detection System on In-vehicle System using GAN. [online] https://doi.org/10.1109/icufn57995.2023.10200297.

Shanmugavadivu, R. (n.d.). *NETWORK INTRUSION DETECTION SYSTEM USING FUZZY LOGIC*. [online] Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=87614275882023d81b0c8814c4f7a13f6ba4cb7b.

Sharma, B., Sharma, L. and Lal, C. (2022). Anomaly Based Network Intrusion Detection for IoT Attacks using Convolution Neural Network. *2022 IEEE 7th International Conference for Convergence in Technology (I2CT)*. [online] https://doi.org/10.1109/i2ct54291.2022.9824229.

Shun, J.-Y. and Malki, H. (2008). Network Intrusion Detection System Using Neural Networks. [online] https://doi.org/10.1109/icnc.2008.900

Soltanzadeh, P., & Hashemzadeh, M. (2021). RCSMOTE: Range-Controlled synthetic minority over-sampling technique for handling the class imbalance problem. *Information Sciences*, *542*, 92-111. https://doi.org/10.1016/j.ins.2020.07.014

Sharafaldin, I., Lashkari, A. H., & Ghorbani, A. A. (2018). Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, *1*, 108-116. https://www.scitepress.org/Papers/2018/66398/66398.pdf

Verizon, (2023). 2023 Data Breach Investigations Report. [Online]. Available at: https://www.verizon.com/business/resources/reports/dbir/ [Accessed 01 04 2024].