AI-BASED THREAT INTELLIGENCE AND ANOMALY DETECTION FRAMEWORK FOR HYBRID CLOUD SECURITY USING AZURE AND MITRE ATT&CK



Mohammad Abdurrehman

09-242242-004

BAHRIA UNIVERSITY ISLAMABAD

AI-BASED THREAT INTELLIGENCE AND ANOMALY DETECTION FRAMEWORK FOR HYBRID CLOUD SECURITY USING AZURE AND MITRE ATT&CK

Mohammad Abdurrehman

09-242242-004

A thesis submitted in fulfilment of the

requirements for the award of the degree of

Master of Science (Computer Engineering)

DEPARTMENT OF COMPUTER ENGINEERING

BAHRIA UNIVERSITY ISLAMABAD

AUGUST 2026

.

© Mohammad Abdurrehman, 2026

MS-13

THESIS COMPLETION CERTIFICATE

Student Name: **Mohammad Abdurrehman** Registration Number: **71572**

Program of Study: **Master of Science in (Computer Engineering)**

Thesis Title: **AI-Based Threat Intelligence and Anomaly Detection Framework for Hybrid Cloud Security Using Azure and MITRE ATT&CK**

It is to certify that the above student's thesis has been completed to my satisfaction and, to my belief, its standard is appropriate for submission for evaluation. I have also conducted plagiarism test of this thesis using HEC prescribed software and found similarity index at **13%** that is within the permissible set by the HEC. for MS. I have also found the thesis in a format recognized by the BU for MS thesis.

Principle Supervisor's Signature: _____

Principle Supervisor's Name: Dr. Syed Khawar Hussain Shah

August 20, 2026

MS-14A

AUTHOR'S DECLARATION

I, **Mohammad Abdurrehman** hereby state that my MS thesis **titled "AI-Based Threat Intelligence and Anomaly Detection Framework for Hybrid Cloud Security Using Azure and MITRE ATT&CK"** is my own work and has not been submitted previously by me for taking any degree from **"Bahria University, Islamabad"** or anywhere else in the country / world.

At any time if my statement is found to be incorrect even after my Graduate the university has the right to withdraw cancel my MS degree.

_____

Mohammad Abdurrehman

09-242242-004

August 20, 2026

MS-14B

PLAGIARISM UNDERTAKING

I, **Mohammad Abdurrehman** solemnly declare that research work presented in the thesis titled

AI-Based Threat Intelligence and Anomaly Detection Framework for Hybrid Cloud Security Using Azure and MITRE ATT&CK is solely my research work with no significant contribution from any other person. Small contribution / help whenever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero-tolerance policy of Bahria University and the Higher Education Commission of Pakistan towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used is properly referred / cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS degree, the university reserves the right to withdraw / revoke my MS degree and HEC and the university has the right to publish my name on HEC / University Website on which name of students who submitted plagiarized thesis are placed.

_____

Mohammad Abdurrehman

09-242242-004

August 20, 2026

DEDICATION

This thesis is dedicated to my parents for their unwavering love and prayers, my siblings for their constant support and encouragement, and my friends for their companionship, understanding, and motivation throughout this journey

ACKNOWLEDGEMENT

I thank Almighty Allah for all His blessings and providing me strength and potential to complete my thesis.

I would like to pay gratitude to my supervisor Dr. Syed Khawar Hussain Shah for his continuous guidance, dedicated involvement and support throughout completing my thesis.

My sincere appreciation to all the Professors for their immense knowledge, without their teachings it wasn't possible to accomplish my work. My university fellows who always helped me whenever needed. I would like to thank my family for always supporting me

ABSTRACT

Traditional cyber security monitoring tools are heavily dependent on a static, signature-based approach to identifying malicious activity. These can deal with known and known-to-be-known attacks, but are less effective in recognizing new or changing attack patterns, and often give false positives (alarms when there are no attacks) and false negatives (alarms when there are attacks). But this is exacerbated in hybrid cloud deployments, where on-premises systems and public cloud resources like Microsoft Azure contribute to a large and expanse of attack surface that is complex to monitor. This thesis proposes the design, implementation and evaluation of an AI-based Threat Intelligence and Anomaly Detection framework that can overcome these limitations. The system adds a machine learning layer which can learn the normal and abnormal network behaviors directly from the data without requiring hand-written rules, and then maps the generated anomaly predictions to structured threat intelligence that can be understood by analysts in the MITRE ATT&CK knowledge base. As part of this project, due to the lack of availability of real-time Azure Monitor or Azure Sentinel logs for training, the benchmark UNSW-NB15 intrusion detection dataset was used as a substitute data source. The system pipeline was intentionally designed to be data-source agnostic, meaning that the UNSW-NB15 dataset can be replaced with a real dataset of Azure telemetry without changing the overall design of the system. Three machine learning models were used and empirically evaluated: an unsupervised model called the Isolation Forest baseline model, a supervised model called the Random Forest Classifier, and an unsupervised model called the Auto encoder-based deep learning model. The accuracy of the Isolation Forest is 54.82%, the Random Forest gives accuracy of 90.09% and F1-Score 0.9224, and the Auto encoder with tuning the reconstruction-error threshold gives accuracy of 68.34%. For the framework's main detection engine therefore, the Random Forest classifier was chosen. The detections created by the Random Forest model are then passed through the MITRE ATT&CK mapping layer which adds a tactic, technique, technique ID, severity rating and contextualizes a prediction of attack/normal (for instance, a prediction of Denial-ofService is mapped to the tactic, technique, technique ID, and a high severity rating to the event). All processed results are stored in a structured CSV file, and presented in an

interactive Stream lit dashboard with charts showing total events processed, alerts generated, severity distribution and MITRE tactic distribution, and a table of alerts that can be filtered. Experimental results show that supervised machine learning significantly outperforms the unsupervised baseline and the deep learning auto encoder on this dataset, and that the incorporation of MITRE ATT&CK into the anomaly detection meaningfully adds to the interpretability and operational utility of ML-based anomaly detection. The thesis ends with a discussion of the limitations of the framework, namely that it currently relies on an offline benchmark data set, not on the real Azure telemetry, and a roadmap for future work, in which the framework will be extended with real-time data ingestion, federated learning and automated response actions.

Keywords: Anomaly Detection, Intrusion Detection, Machine Learning, Random Forest, Isolation Forest, Auto encoder, MITRE ATT&CK, Hybrid Cloud Security,

UNSW-NB15, Threat -Intelligence.

Table of Content

MS-13 THESIS COMPLETION CERTIFICATE iv

MS-14A AUTHOR'S DECLARATION v

MS-14B PLAGIARISM UNDERTAKING vi

DECLARATION vii

ACKNOWLEDGEMENT viii

ABSTRACT ix

LIST OF FIGURES xvi

LIST OF TABLES xvii

LIST OF ABBREVIATIONS xix

C1: INTRODUCTION 1

1.1 Background and Motivation 1

1.1.1 The Growing Role of Cloud-Native Security Telemetry 2

1.1.2 Security Operations Centre (SOC) Analyst Workload 3

1.2 Research Gap and Rationale 3

1.3 Problem Statement 4

1.3.1 Operational Definition 5

1.4 Research Question's 5

1.5 Research Objective's 6

1.6 Significance and Contribution of the Study 7

1.6.1 Significance to Security Operations Teams 7

1.6.2 Significance to Researchers and Academia 8

1.6.3 Significance to Cloud Service Providers and Organizations Adopting Hybrid Cloud 8

1.7 Scope of the Study 8

1.8 Thesis Organization 9
CHAPTER 2: LITERATURE'S REVIEW 11
2.1 Traditional, Rule-Based Intrusion Detection and SIEM System's 11
2 2. Machine Learning-Based Intrusion Detection System's 12
2.3 Unsupervised Anomaly Detection and Isolation Forest 12
2.4 Supervised Classification and Random Forest 13
2.5 Deep Learning and Autoencoder-Based Anomaly Detection 14
2.6 Benchmark for Datasets for Intrusion Detection Research 14
2.7 The MITRE ATT&CK Framework and Its Integration with Detection Systems . 16
2.8 Explainable AI in Cybersecurity 17
2.9 Adaptive Learning and Federated Learning in Cloud Security 17
2.10 Hybrid and Multi-Cloud Security Challenges 18
2.11 Comparative Overview of Benchmark Datasets 18
2.12 Research Gap and Summary 19
2.13 Graph-Based and Sequence-Based Anomaly Detection 20
2.14 The Operational Cost of Detection Delay 20
2.15 Alert Prioritization and Triage Automation 21
2.16 Human Factors: Alert Fatigue and Analyst Workload 22
2.17 Ensemble Learning: A Closer Comparative Look 22
2.18 Feature Engineering Considerations in Network Intrusion De taction 23
2.19 Evaluation Methodology in Prior Intrusion-Detection Studies 24
CHAPTER 3: SYSTEM DESIGN AND METHODOLOGY 25
3.1 Research Approach 25
3.2 Proposed Three-Tier System Architecture 26
3.3 Dataset Description: UNSW-NB15 27
3.4 Data Preprocessing Pipeline 29
3.5 Model 1: Isolation Forest 30

3.5.1 Mathematical Formulation of the Isolation Forest Anomaly Score 32
3.6 Model 2: Random Forest 32
3.6.1 Voting Mechanism and Feature Importance 34
3.7 Model 3: Autoencoder 35
3.8 MITRE ATT&CK Mapping Design 37
3.9 Evaluation Metrics 38
3.10 Tools and Technologies 39
CHAPTER 4: IMPLEMENTATION 41
4.1 Development Environment and Project Structure 41
4.2 Data Preprocessing Implementation 42
4.3 Isolation Forest Implementation 45
4.4 Random Forest Implementation 46
4.5 Autoencoder Implementation 48
4.6 Autoencoder Threshold Tuning 50
4.7 Model Comparison 52
4.8 MITRE ATT&CK Mapping Implementation 53
4.9 Alert Summary and Chart Generation 55
4.10 Stream-lit Dashboard Implementation 59
CHAPTER 5: RESULTS AND EVALUATION 62
5.1 Experimental Setup 62
5.2 Isolation Forest Results 62
5.3 Random Forest Results 63
5.4 Autoencoder Results (Before and After Threshold Tuning) 64
5.5 Comparative Analysis 65
5.6 MITRE ATT&CK Mapping and Alert Summary Results 67
5.7 Per-Attack-Category Detection Performance 69
5.8 Cross-Model Error Analysis 71

5.8 Discussion of Results 72
CHAPTER 6: MITRE ATT&CK INTEGRATION AND DASHBOARD 74
6.1 Why Detection Alone Is Not Enough 74
6.2 The MITRE Mapping Layer 75
6.3 Complete Attack-Category-to-MITRE Mapping Table 75
6.4 MITRE Tactic Distribution 77
6.5 Dashboard Overview 78
CHAPTER 7: DISCUSSION 79
7.1 Interpretation of Findings 79
7.2 Comparison with the Literature 79
7.3 Limitations 80
7.4 Threats to Validity 81
CHAPTER 8: CONCLUSION AND FUTURE WORK 82
8.1 Summary of Contributions 82
8.2 Conclusion 83
8.3 Future Work 83
CHAPTER 9: REQUIREMENTS SPECIFICATION AND SYSTEM TESTING 85
9.1 Functional Requirements 85
9.2 Non-Functional Requirements 86
9.3 Use Case Description 87
9.4 Test Strategy 88
9.5 Summary of Verification 90
REFERENCES 91
APPENDIX A: UNSW-NB15 FEATURE DICTIONARY 94
APPENDIX B: SAMPLE OF PREPROCESSED DATA 97
APPENDIX C: COMPLETE FINAL ALERT SUMMARY OUTPUT 98
APPENDIX D: ENVIRONMENT SETUP AND REPRODUCTION STEPS 100

APPENDIX E: DASHBOARD USER GUIDE 102
E1: Launching the Dashboard 102
E2: Step 1: Reviewing Headline Metrics 102
E3: Step 2: Reviewing Model Performance 103
E4: Step 3: Reviewing Threat Intelligence Charts 103
E5: Step 4: Filtering and Investigating Individual Alerts 103
E6: Step 5: Exporting Data for Further Analysis 104
E7: Refreshing the Dashboard After Retraining 104
E8: Troubleshooting Common Issues 104
APPENDIX F: MITRE ATT&CK TECHNIQUE REFERENCE 106
APPENDIX G: GLOSSARY OF KEY TERMS 108

LIST OF FIGURES

Figure 1	Proposed three-tier architecture of the AI-based threat intelligence and anomaly detection framework
Figure 2	End-to-end pipeline flow implemented in this project, from raw UNSW-NB15 data to the Streamlit dashboard
Figure 3	Model performance comparison across Accuracy, Precision, Recall, and F1-score for the three implemented models
Figure 4	Autoencoder encoder-decoder architecture implemented for unsupervised anomaly detection
Figure 5	Model performance comparison across Accuracy, Precision, Recall, and F1-score, generated directly by generate_model_comparison_chart.py
Figure 6	Severity distribution of all processed events, generated by generate_result_charts.py
Figure 7	Alert status distribution (Alert Generated vs. No Alert) across all processed events
Figure 8	: Distribution of the underlying UNSW-NB15 attack_cat groundtruth categories across the processed test set
Figure 9	MITRE ATT&CK tactic distribution across all alerts generated by the framework

LIST OF TABLES

Table 1	Comparison of Benchmark Datasets for Intrusion Detection Research
Table 2	High-level comparison of widely used intrusion-detection benchmark datasets
Table 3	High-level comparison of widely used intrusion-detection benchmark datasets
Table 4	Isolation Forest hyperparameter configuration used in this project
Table 5	Random Forest hyperparameter configuration used in this project
Table 6	Autoencoder training configuration
Table 7	Software tools and libraries used in the implementation
Table 8	Isolation Forest confusion matrix on the UNSW-NB15 test set
Table 9	Isolation Forest classification metrics
Table 10	Random Forest confusion matrix on the UNSW-NB15 test set
Table 11	Random Forest classification metrics
Table 12	Autoencoder performance before and after reconstruction-error threshold tuning
Table 13	Tuned Autoencoder confusion matrix on the UNSW-NB15 test set
Table 14	Consolidated model comparison (reproduced from outputs/model_comparison.csv)
Table 15	Overall alert summary from mitre_mapped_alerts.csv
Table 16	Random Forest per-attack-category detection rate (proportion of each category's records flagged as an attack)

Table 17	Overlap between models missed attacks (false negatives), pairwise
Table 18	Complete attack category -> MITRE ATT&CK mapping table used by the framework
Table 19	MITRE ATT&CK tactic distribution (reproduced from mitre_mapped_alerts.csv)
Table 20	Functional requirements
Table 21	Non-functional requirements
Table 22	Primary use case: Review AI-Generated Threat Alerts
Table 23	System test cases and outcomes
Table 24	Complete UNSW-NB15 feature dictionary (42 modelling features)
Table 25	Sample of preprocessed test data (subset of columns)
Table 26	Core pinned dependency versions (requirements.txt)
Table 27	MITRE ATT&CK technique reference for all techniques used in this framework
Table 28	Glossary of key terms used throughout this thesis

LIST OF ABBREVIATIONS

Abbreviation	Meaning
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
IDS	Intrusion Detection System
SIEM	Security Information and Event Management
MITRE ATT&CK	MITRE Adversarial Tactics, Techniques, and Common Knowledge
UNSW-NB15	University of New South Wales Network-Based 2015 Dataset
CSV	Comma-Separated Values
TP / TN	True Positive / True Negative
FP / FN	False Positive / False Negative
F1	F1-score (harmonic mean of precision and recall)
PCA	Principal Component Analysis
XAI	Explainable Artificial Intelligence
VM	Virtual Machine
AD	Active Directory
MSE	Mean Squared Error
ReLU	Rectified Linear Unit

API	Application Programming Interface
T-ID	MITRE ATT&CK Technique Identifier

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

In progressive years, organizations have increasingly moved critical workloads away from entirely on-premises data centers towards new hybrid cloud setups that integrate on-premises resources with Microsoft Azure, Amazon Web Services, Google Cloud Platform, and more all public cloud services. This change is based on the obvious business advantages of elastic scalability, lower capital cost, geographic redundancy, and quicker deployment of new services. All the same, these advantages create a much broader and more distributed attack surface than a conventional, perimeter-based network. With a purely on-premises network, a security team can have a relatively stable number of assets, entry points, and traffic flows. Virtual machines provision and decommission dynamically, identities are federated between on-premises Active Directory and cloud-based identity providers, and data flows continuously between Azure internal systems and services in the cloud, like Azure Monitor and Azure Sentinel [1]. This is the type of complexity adversaries are leveraging more and more, hiding in the midst of normal cloud traffic and not matching any known attack signature. Today's major tool category for defending these environments is rule-based, Security Information and Event Management (SIEM) systems as well as traditional Intrusion Detection Systems (IDS). They match observed events with a library of known, bad events or the hand-created correlation rules. This is a useful way to attack a previously observed well documented attack, but has two known limitations which directly motivate this thesis. A purely rule based system does not have a signature to compare against the first time a new or previously unknown attack technique is seen and may not generate an alert at all, a false negative. Second, and even worse in practice, static rules are often too specific or too vague, and so unusual but completely harmless activity is flagged as an attack, a false positive. Security operations teams that have seen an increase in false positives tend to develop something known as “alert fatigue,” where they are less likely to take time to respond to a legitimate alert, especially if it is lost in all the other alerts. [2] Machine learning provides a completely

new solution to this issue. A machine learning model does not need to be given a description in advance of an attack, but can instead be trained to learn the statistical structure of normal network use and then alert to any network traffic that is significantly different from that structure, or, if attack labelled network traffic data is available, learn to make a direct distinction between attack and normal traffic from past network traffic examples. This is exactly the type of ability needed to identify attacks that a signature-based system wouldn't be able to detect, let alone stop. An anomaly detection model, however, has a major limitation in practice: and it can alert the security analyst that something is off, but it cannot, on its own, describe the type of attack that has likely occurred, the severity of the attack, or an appropriate response for the attack. The MITRE ATT&CK framework [3] was created to fill that gap. MITRE ATT&CK is an open, living knowledge base of adversarial tactics and techniques derived from real world observations. A detection framework may be able to take a raw output of an anomalous/not-anomalous machine learning prediction, map it to a specific ATT&CK tactic and technique, and directly take action as a human or automated response system, based on the detection information. Motivation for this thesis is that anomaly detection using machine learning and the MITRE ATT&CK framework are both well studied on their own but there is a lack of published research that brings the two together in a single end-to-end pipeline from raw network traffic to a MITRE mapped, severity-rated alert displayed on an operational dashboard. This research's pivotal contribution is the building, testing, and evaluating of just such a pipeline.

1.1.1 The Growing Role of Cloud-Native Security Telemetry

The increasing use of the cloud-native security telemetry platforms, like Azure Sentinel, as the main source of security visibility in organizations, instead of relying on on premises network sensors, is also a trend that supports the motivation for this thesis. This change alters the detection problem in two significant aspects. First, the volume and velocity of telemetry grow significantly: Cloud platforms record far more of what is going on in them (resource provisioning events, identity and access management changes, API calls, etc.) than an on-premises sensor on the perimeter of the network would see First, the volume of telemetry grows, and it grows fast – because more is happening in cloud platforms (resource provisioning events, identity and access management changes, API calls, etc.) than an on-premises sensor on the network

perimeter would see. Second, cloud telemetry is also inherently more heterogeneous – with virtual machine performance logs, identity logs and network flow logs all included – meaning that a detection architecture that separates the telemetry data ingestion layer from the detection logic, as detailed in Chapter 3, is highly sought after, as new telemetry sources could be added without requiring the detection logic to be changed.

1.1.2 Security Operations Centre (SOC) Analyst Workload

Another closely related motivation is the day-to-day nature of security operations center (SOC) analysts who are tasked with triaging all the alerts generated by a detection system. As the number of infrastructures monitored increases, especially in multi-cloud or hybrid cloud deployments, the number of analysts to scale increases, and any increase in the accuracy of detection (fewer false positives to waste analyst time) or the number of contextual details provided per alert (fewer manual investigations required per real alert) contributes directly and compound to the number of infrastructures that a fixed size SOC team can realistically Monitor. In fact, the MITRE ATT&CK mapping layer created in this thesis (Chapter 3, Section 3.8) is specifically targeted towards this second lever, in that it will automatically add a tactic, technique, and severity to each detected event directly aimed at this second lever: by automatically attaching a tactic, technique, and severity rating to every detected event, so that a human analyst is not burdened with manually doing this for each and every event that is detected, which would grow in volume with the number of detections provided by the underlying machine learning model.

1.2 Research Gap and Rationale

The literature reviewed in Chapter 2 reveals three lines of support that are well established and can be summarized as follows, all of which form the gap that this thesis covers. Based on the reviewed literature in Chapter 2, the following three claims can be established that are well supported and will form the gap that this thesis will address. First, rule-based SIEM & IDS tools are inherently constrained for new/emerging attacks. Secondly, of the machine learning paradigms offered, supervised ensemble methods (e.g. Random Forest), unsupervised isolation-based methods (e.g. Isolation Forest), and unsupervised deep learning methods (e.g. Autoencoders), have all shown distinct advantages and differ in data-availability requirements, and a detailed comparison of all three on the same,

contemporary benchmark dataset would be of direct practical utility. Finally, and most important, both machine learning-based anomaly detection as well as the MITRE-ATT&CK framework are well-developed in their own right, but there is relatively little literature that has shown the end-to-end integration of these two tools; from raw network features to a MITRE mapped, severityrated, and dashboard-visualized alert. This is a methodological gap, not a purely theoretical one, and implemented here through a full working, empirically tested implementation, not just a conceptual proposal alone, in how detection and threat intelligence are combined.

1.3 Problem Statement

Traditional cybersecurity monitoring tools deployed across on-premises, cloud and hybrid-cloud environments rely on a bunch of rule-based, static detection mechanisms and pre-defined attack signatures. These techniques are useful when dealing with known, evolving, or new attack patterns but are less effective in detecting unknown attack patterns. This often leads to false alarms, or false negatives, where a legitimate activity is wrongly identified as malicious, or a malicious activity is not detected. This constraint poses a huge strain on today's Security Operations Centers (SOCs) as they attempt to fortify their increasingly complex and distributed environments as the volume of alerts continues to rise and skilled cybersecurity analysts are scarce. Beyond the limitations of detection, current security monitoring systems lack much context around the anomalies detected. While an alert could signal suspicious activity, it does not necessarily describe the technique of the attack, what the adversary was trying to accomplish, the potential impact, or what to do. As a result, security analysts have to manually analyze systems logs, correlate events from various data sources and match their observations with known sets of threat intelligence. This manual investigation process is labor intensive, resource driven and subject to human error, which adds up to incident response times and overall inefficiencies in cybersecurity operations. As cyber threats grow more sophisticated and hybrid-cloud infrastructures evolve, the importance of intelligent detection systems which can learn a normal system and recognize deviations without relying solely on predefined signatures is growing. Machine learning anomaly detection is an emerging technique that allows models to identify attack patterns that the system has not previously seen by analyzing behavior instead of by explicit rule. But anomaly detection is not enough unless the alerts created are meaningful and can be acted upon by security professionals. Hence,

there is an urgent demand for an adaptive framework that combines intelligent anomaly detection with contextual threat intelligence in cybersecurity. Such a framework should: (i) abstract from handcrafted rules to detect known and unseen attacks; (ii) be trained and thoroughly tested on a large labeled cybersecurity dataset to ensure reliability before use in operational environments; and (iii) automatically enrich each detected anomaly with standard threat intelligence data, such as the corresponding MITRE ATT&CK tactic, technique, technique identifier (TID), attack description, and attack severity, to help expedite investigation and make decisions. The research tackles these challenges by designing, developing, and testing an AI-based hybrid-cloud cybersecurity anomaly detection system. The proposed system will use the MITRE ATT&CK knowledge base and behavioral analysis using machine learning to generate accurate, explainable, and context-aware security alerts. The proposed framework seeks to optimize the operation of modern Security Operations Centers (SOC) by minimizing false alarms, detecting new attacks, and furnishing valuable threat intelligence for each alarm.

1.3.1 Operational Definition

Several important terms for the problem statement above are now explicitly defined for clarity (a comprehensive glossary of technical terms is included in Appendix B). The term anomaly is used throughout this thesis to describe any network event that has feature values that differ sufficiently from the statistical characteristics of normal network traffic for any of the three anomaly detection models outlined in Chapter 3 to classify it as an anomaly. A false positive is a normal event that is reported as an attack, which costs an analyst to investigate a non-attack case. A false negative occurs when a legitimate attack is mistaken for a normal one, which has an operational cost of failing to detect an attack, potentially allowing an adversary to continue their attack without being stopped. Throughout this thesis, threat intelligence is referred to as the structured contextual information generated by the MITRE ATT&CK mapping layer described in Section 3.8, which consists of the Tactic, Technique, Technique identifier, and Severity associated with each detected event and not as a more general term in the security industry that is used to describe external sources of indicators of compromise or threat actor profiles.

1.4 Research Question’s

This research is structured around four guiding research questions:

- **RQ1**: How can machine learning be useful to detect anomalies and cyber threats in hybrid-cloud environments, given that live cloud telemetry may not always be available for model development?
- **RQ2**: In what ways can the MITRE ATT&CK framework be integrated into machine learning-based detection logic to produce structured, analyst-actionable threat intelligence rather than a bare anomaly score?
- **RQ3**: What is the comparative effectiveness of unsupervised (Isolation Forest), supervised (Random Forest), and deep learning-based (Autoencoder) approaches in accurately detecting and classifying threats in a benchmark intrusion-detection dataset?
- **RQ4**: What are the best metrics (accuracy, precision, recall, and/or F1-score) for assessing the operational performance of a hybrid cloud anomaly detection framework, and what do they say about the trade-offs among the models examined?

1.5 Research Objective's

In direct response to the research question's above, this thesis pursues the following three objectives:

- **Objective 1**: To create an AI anomaly detection model that can be applied to a hybrid cloud and Azure-based environment, validated on UNSW-NB15 benchmark dataset (as proxy for live Azure Monitor telemetry), and designed such that the pattern detection model is not dependent on the underlying data source and can be later substituted with real Azure logs without redeveloping the pattern detection pipeline.
- **Objective 2**: To develop and implement a systematic method for connecting MITRE ATT&CK tactics, techniques, technique identifiers, and severity levels to the outputs of the machine learning detection layer, thereby producing structured, human-readable threat intelligence.
- **Objective 3**: To empirically investigate and compare the performance of the implemented framework, and of its three candidate detection models, using standard classification metrics (accuracy, precision, recall, and F1-score), and to present these results through an interactive analytics dashboard.

1.6 Significance and Contribution of the Study

This research is important to both the academic and practical community and its findings are directly correlated with the findings presented in Chapter 4 and Chapter 5. Academically, it provides an empirical side-by-side comparison on the same benchmark dataset, preprocessed in the same way with the same unsupervised baseline model, a supervised classifier and a deep learning autoencoder – something that is not always clearly presented in the broader literature. It also provides a real-world, repeatable design for integrating a machine learning detection layer with the MITRE-ATT&CK knowledge base, which is often cited in theory in the literature, but seldom shown endto-end in working code, actual predictions, and a working dashboard. The framework outlined in this thesis is a practical template that a security operations team could follow to implement a similar solution for their hybrid cloud environment: use the live Azure Monitor or Sentinel data feeds instead of the UNSW-NB15 data source; retrain the Random Forest classifier (best performing model identified in this thesis) using organization-specific data; and reuse the existing MITRE mapping and dashboard layers with minimal changes. Besides these academic and practical contributions this thesis shall be a self-contained repeatable example: every empirical result reported in Chapter 4 was generated by the same exact source code reproduced verbatim in Appendix A, each figure and each table in this document was generated directly from the framework source code output files, and complete environment setup and reproduction instructions are included in Appendix B so that another researcher, or even the examiners of this thesis, may obtain the results reported here, from raw data to a dashboard, using his own machine.

1.6.1 Significance to Security Operations Teams

The security operations center (SOC) analysts are the most practical benefit that was shown in this thesis: automatic contextualization of MITRE ATT&CK (Chapter 4). When using this framework, an analyst does not need to look up what type of attack this "anomaly detected" signal might be, but instead gets a Tactic, Tactic, Technique identifier and a severity rating—plus the underlying confidence of the model identifying the attack. This directly addresses the alert-fatigue problem mentioned in Chapter 2 - the context rich alerts are faster to triage than the statistical flags.

1.6.2 Significance to Researchers and Academia

The most apparent benefit in practice, as shown in this thesis, for security operations center (SOC) analysts is the decrease in manual triage effort required for automatic MITRE ATT&CK contextualization (Chapter 4). With this framework, an analyst will not be fed a simple "anomaly detected" signal, but will be given a tactic, technique, technique identifier and a severity rating, along with the underlying model's confidence in the detection. This directly tackles the alert-fatigue issue that we covered in Chapter 2: Context-rich alerts are simply quicker to triage than a single statistical flag

1.6.3 Significance to Cloud Service Providers and Organizations Adopting Hybrid Cloud

The thesis makes a practical and reproducible side-by-side comparison of three modelling paradigms tested with the same preprocessing conditions (Chapter 3) and provides a real-world demonstration of integrating MITRE ATT&CK beyond the conceptual level of some previous research on the topic (Chapter 2) for researchers in this field. The full source code (Appendix A) is included specifically for the purpose of permitting others to reproduce, extend or criticize the specific empirical claims made in Chapter 4. Hybrid cloud is significant to both the cloud service provider and the organization implementing a hybrid cloud.

This thesis describes a real-life example of how to build an AI-based detection capability on top of existing cloud security tooling (e.g., Azure Sentinel) instead of topping it with a cloud-native solution, and demonstrates how this approach can Address Multi-cloud and Hybrid-cloud security issues (e.g., data isolation, crossplatform communication, access control) by separating the detection and contextualization logic from any given vendor's proprietary log format

1.7 Scope of the Study

The following limitations have been set on the scope of this thesis. First of all, model development, training, and evaluation are done on the UNSW-NB15 benchmark intrusion detection dataset, instead of Azure Monitor or Azure Sentinel telemetry, because real-time access to true Azure logs was not available throughout this project.

This data-ingestion and pre-processing pipeline is, however, intentionally written as a data-source agnostic system, allowing it to be configured to point at real Azure logs in a future deployment with minimal structural changes.

Second, instead of testing every machine learning algorithm used in the literature, three representative machine learning paradigms related to anomaly detection are used: an unsupervised tree-based method (Isolation Forest), a supervised ensemble of classifiers (Random Forest), and an unsupervised deep learning approach (a fully-connected Autoencoder). These three were picked as they belong to the three major methodological families that have been used in anomaly-detection literature: classical unsupervised outlier detection, supervised anomaly classification, and representation learning based deep anomaly detection.

Third, the MITRE ATT&CK integration in this thesis is a deterministic mapping layer from an attack category that is detected to a corresponding tactic, technique, technique ID, and severity rating. It does not try to infer automated, adaptive techniques directly from the raw network features, instead, it relies on the categorical label of the attack that is already presented in UNSW-NB15 data set (after ML layer flagged an event as an attack) as a basis for MITRE lookup. This design is discussed further in Chapter 7 (Discussion and Limitations) and its implications.

Lastly, the analytics dashboard developed for this project is deployed using Stream lit, a Python-based dashboarding framework, and ingests data from static csv files generated by the offline training and inference pipeline. Real-time streaming ingestion into dashboard is considered future work, not current feature.

1.8 Thesis Organization

The rest of this thesis is structured in the following way.

The relevant literature is reviewed in Chapter 2, outlining the various methods of anomaly detection such as rule-based intrusion detection, machine learning and deep learning anomaly detection methods, benchmark datasets used in the field, the MITREATT&CK framework, explainable AI in security, adaptive and federated learning, hybrid and multi-cloud security, ensemble learning, feature engineering considerations, graph-based and

sequence-based anomaly detection methods, and alert prioritization methods, and what this thesis aims to address.

The three tier design of the system proposed, requirements specification (for both of the functional and non-functional requirement's), the primary use case are detailed in Chapter 3. Chapter 3 also presents the required data set, UNSW-NB15, and the processing pipeline used to preprocess the data, the theoretical background and mathematical formulation of the system, the configuration of each of the three machine learning models proposed, the detailed data quality validation checks performed, a summary of the design alternatives that were considered but not used, and an overview of the development of the system followed by full source code in Appendix A.

The analysis of data, experimental results, and findings are all presented in Chapter 4 including confusion matrices, precision/recall/F1 scores for each model, a detailed comparative analysis, complete MITRE ATT&CK mapping results, a breakdown of the per-attack-category detection rate, a cross-model error-overlap analysis, a study of alert volume sensitivity to the selection of the upstream detection model, a featureimportance analysis directly from the trained Random Forest model, a worked example following a single record through the entire pipeline, a computational efficiency comparison, and system-level test cases used to validate the implementation.

The experimental results including confusion matrices, precision/recall/F1 scores, and MITRE ATT&CK mapping statistics for each model are presented in Chapter 5, which can be illustrated with the charts and tables generated by the implemented system.

The MITRE ATT&CK integration is explored more in-depth in Chapter 6, and the full Tactic/Technique/Severity mapping table is included in the ATT&CK framework.

The results discussed in chapter 7 in relation to the literature, mentions the limitations of the current implementation and considers threats to the validity of the results.

The thesis is summarized in the last chapter, which suggests future research areas. The thesis ends with a set of references and appendices with the full source code, list of data features, and MITRE ATT&CK mapping table for the system.

CHAPTER 2

LITERATURE'S REVIEW

This chapter provides a detailed survey of the existing-research that is relevant to designing an AI-based threat intelligence and anomaly detection framework for hybrid cloud environments. It covers the key components of the proposed system, traditional rule-based intrusion detection, intrusion detection using machine-learning, deep learning and autoencoder-based anomaly detection, benchmark datasets for evaluating intrusion detection systems, the MITRE ATT&CK framework, its relation to intrusion detection solutions, explainable AI in security, and the unique challenges of hybrid and multi-cloud security. The chapter ends with a statement of the research gap that will motivate this thesis.

2.1 Traditional, Rule-Based Intrusion Detection and SIEM System's

The first and most popular generation of network defense solutions are signature-based Intrusion Detection System's (IDS) and Security Information and Event Management (SIEM) platforms. These systems work by matching events in the network against a library of known malicious signatures, or a collection of hand-crafted "correlation rules" created by security engineers. When an event matches a signature or triggers a rule, an alert is raised.

But one of its major strengths is its ability to work against known attacks, if one has already been seen and classified, then there will be a signature that will identify it reliably and tell you which signature was activated. It suffers from a fundamental lack of generalization – the one aspect of this thesis that is most relevant – and this limitation is built into the system. There is no way for a rule-based system to identify an attack method that it has not learned. This will enable the zero-day exploits, new variants of malware, and techniques used by attackers that are designed to evade known signatures to be able to go undetected for long periods of time – days or weeks – until a human analyst is able to detect the new signature and create a new one that can be deployed.

This limitation was corroborated directly by Khan et al. [2] who reported that a rulebased SIEM approach would generate a massive number of false-positive alerts; the "alert fatigue" problem that is widely known in security operations research and demonstrated by the authors in this paper. Even the built-in security tooling provided by the major cloud vendors are similarly limited: no matter how much Microsoft updates its own Azure Sentinel [1] with new capabilities via its threat intelligence teams, it is simply not agile enough to keep up with threats that are quickly arising and adapting.

2 2. Machine Learning-Based Intrusion Detection System's

In response to the limitations of purely rule-based detection, a substantial body of research has explored machine learning as an alternative or complementary detection mechanism. Rather than a machine learning-based IDS works by learning the statistical patterns that differentiate between normal traffic and malicious traffic, or in the unsupervised learning case, it learns what "normal" traffic looks like and blocks any traffic that is a significant deviation from that learned norm. This is a very useful capability in its own right, because it is hard to define and detect classes of threat, such as insider threats and lateral movements, that are subtle enough to not be detected by a rule-based system. A wide range of algorithms have been applied to this problem, spanning classical supervised classifiers, unsupervised outlier-detection methods, and, more recently, deep learning architectures. Ensemble tree-based methods, and Random Forest [5] in particular, have repeatedly been reported as strong, robust baselines for tabular network intrusion data of the kind produced by benchmark datasets such as UNSW-NB15, largely because such data consists of well-structured, numerically and categorically encoded flow-level features rather than raw, unstructured signal, a data shape for which tree ensembles are particularly well suited. This is consistent with several comprehensive surveys of machine learning and data mining approaches to cyber security intrusion detection [14, 19, 21].

2.3 Unsupervised Anomaly Detection and Isolation Forest

A One common point in the intrusion-detection literature is the one that in practice, the full set of attack information is hard to obtain or may not exist. This is an incentive for researching unsupervised anomaly detection methods, which do not need any attack labels whatsoever, but a representative sample of (mostly) normal traffic. One of the most

popular unsupervised anomaly detection algorithms in this literature is Isolation Forest [6] proposed by Liu, Ting, and Zhou. Instead of modelling normal data directly, Isolation Forest relies on the notion that "anomalous points are few and different" and they can be easily distinguished from the rest of the data by performing random recursive partitioning. The algorithm constructs a set of random binary trees, each of which splits the data by randomly choosing a feature and a value to split on – it is assumed that the anomalous points need fewer splits to separate from the rest of the data than do the normal points, and the average number of splits across all trees is used as the anomaly score. Given the simplicity of the model, and its speed of implementation, Isolation-Forest is often used as a baseline to compare against more complicated supervised or deep learning models, a position it will serve in this thesis. The main drawback is that it is based solely on the statistical isolability of the classes, and does not learn the discriminative structure of the labelled attack classes, therefore usually yielding lower raw accuracy on a labelled benchmark dataset compared to supervised classifiers trained on the same benchmark dataset, as was directly confirmed by the experiments presented in Chapter 4, which included comparisons with supervised classifiers. This general framing follows the seminal survey by Chandola et al. [13], which remains one of the most widely cited treatments of anomaly detection across application domains.

2.4 Supervised Classification and Random Forest

The results have demonstrated that supervised classification has consistently outperformed unsupervised anomaly detection approaches on raw accuracy and F1scores when labelled data is available as it can learn the decision boundary separating specific attack classes within the training data. Random Forest, proposed by Breiman [5] is an ensemble learning approach that builds a forest of decision trees by training them on a subset of the data for training (the training set is resampled with replacement) and a random subset of features at each split and voting on the majority of the tree predictions. This Bootstrap Aggregation (or "bagging") procedure significantly reduces variance and overfitting risk of any single decision tree and maintains the interpretability benefits of tree models, like feature-importance ranking. Random Forest classifiers have been shown to perform well and be robust on tabular flow-based features like those present in the NSL-KDD and UNSW-NB15 datasets, consistently in the literature of network intrusion detection. This is another pattern that is observed in the experiments conducted in Chapter

4, and is a reflection of the overall trend in the broader machine learning literature: well-tuned Random Forest classifiers perform as well or better than much more complex deep learning networks in structured, tabular intrusion-detection datasets.

2.5 Deep Learning and Autoencoder-Based Anomaly Detection

Another line of work examines deep learning architectures for anomaly detection, leveraging their theoretical ability to learn complex, non-linear representations of normal behavior without the need for extensive manual feature engineering. The most frequently used deep learning architecture for this particular task is an autoencoder, a neural network that is trained to reconstruct its own input by learning some compression (or “bottleneck”) representation. If only normal data were used for training, the network learns to recognize normal patterns with low reconstruction error, but with a relatively high error for the patterns it has never seen before, including those that are abnormal. This reconstruction error is then directly used as an anomaly score. Sakurada and Yairi [7] were the first to propose and test auto encoders specifically for anomaly detection using non-linear dimensionality reduction, showing that the auto-encoders could learn non-linear correlations between features, which linear dimensionality reduction methods like Principal Component Analysis (PCA) failed to capture. One practical difficulty that is well known in the field of autoencoder-based anomaly detection and will be faced and solved in this thesis is selecting the threshold for the reconstruction error: this threshold value is not learnt by the model and must be determined by applying a separate tuning procedure. The tuning of thresholds is a very delicate process, as can be seen in the experiments in this thesis (Chapters 3 and 4) which show this balance between sensitivity and specificity. Comprehensive surveys of deep learning approaches to network intrusion and anomaly detection, such as those by Vinayakumar et al. [18] and Kwon et al. [20], reinforce this observation.

2.6 Benchmark for Datasets for Intrusion Detection Research

Realistic, labelled benchmark datasets play a crucial role in empirical intrusion detection research. For many years, KDD Cup 1999 was the standard, widely used dataset in this field, but has been criticized as being "full of duplicate records" and "outdated with respect to the network and types of attacks". NSL-KDD was then introduced as a modified version of KDD99, but the basic traffic attributes remain the same as in

KDD99, which were found to be dated. These limitations were overcome by creating UNSW-NB15 dataset by Moustafa and Slay [4] at the Australian Centre for

Cyber Security at the University of New South Wales. . It was created with the IXIA Perfect Storm tool to produce a mix of real modern normal network traffic and synthetically created modern attack traffic, and contains forty-two modelling features (excluding the identifier, label columns) that represent characteristics of the flow including protocol type, service, connection state, packet and byte counts, timing statistics, and TCP-specific features. The data are labeled with a binary label to indicate each record, and each attack record has a multi-class attack category representing nine different attack families.

The selection of UNSW-NB15 as one of the most popular contemporary benchmark datasets for intrusion detection research is just as justified: it contains a more representative and more varied collection of modern attacks as compared to KDD99 and NSL-KDD, and yet it includes a complete labelling that is critical for supervised evaluation. UNSW-NB15 was chosen to be the data set for this research project, for the above and others reasons, since there were no real time Azure Monitor / Azure Sentinel logs available for developing model's research.

Table 1 Summarizes a high-level comparison of the widely used intrusion-detection benchmark datasets discussed above Beyond NSL-KDD [15], later benchmark efforts such as CICIDS2017 [16] and the broader survey of network intrusion detection datasets by Ring et al. [17] further illustrate how the field has progressively moved toward more realistic and diverse traffic captures.

Table 1: Comparison of Benchmark Datasets for Intrusion Detection Research

Dataset	Year	Traffic Era	Key Limitation
KDD Cup 1999	1999	1998 DARPA simulation	Large redundancy; outdated traffic characteristics
NSL-KDD	2009	Refined KDD99	Removes redundancy, but still dated traffic patterns

UNSW-NB15	2015	Modern hybrid	Chosen for this
			thesis: nine modern attack families, 42 modelling features

2.7 The MITRE ATT&CK Framework and Its Integration with Detection Systems

The MITRE ATT&CK framework [3] is a publicly released, ever evolving database of adversarial activity that is derived from the observation of real-world intrusions. It hierarchically structures the behavior of adversaries in tactics (the tactical objective of the adversary) and in each of them, the specific techniques and sub-techniques, each identified by a stable technique ID.

As a structured, standardized vocabulary, ATT&CK is most commonly used today as a manual reference—security analysts use ATT&CK to classify and communicate about an incident, after having identified it through other means. However, the literature reviewed in this thesis suggests that there is relatively little published work that automates this classification step directly, within the detection pipeline itself, i.e. mapping the raw output of a machine learning detection model to ATT&CK tactics, techniques, and severities as a direct part of the detection pipeline, rather than as a manual analyst effort performed after the fact.

This observation is of particular importance because it points to a specific deficiency that can be addressed: Although anomaly detection based on machine learning and the ATT&CK framework are both relatively established areas of practice and research, the automated, end-to-end process of coupling these two from a raw network feature vector to a fully MITRE-mapped, severity-rated alert is not yet a common practice in the literature. This thesis explicitly aims to fill this gap by coupling exactly that way and testing it empirically.

2.8 Explainable AI in Cybersecurity

Another, and growing, line of work focuses on the interpretability of machine learning tools for security. It is argued that in the context of cloud security applications, explainable AI (XAI) techniques like those discussed by Johnson, Davis, and Kim [8] are being increasingly adopted as a way of assisting analysts in understanding and reasoning about why a given model produced a particular result while playing a crucial role in justifying, to auditors, regulators, or incident responders, why a particular event was or was not considered a security incident. Practical guidance for building and interpreting such explanations is provided by Molnar [24], whose widely used reference systematically catalogues model-agnostic and model-specific interpretability techniques.

The MITRE ATT&CK mapping layer, when used in this thesis, can be seen as a lightweight, domain-specific version of explainability: in this case the analyst is given an immediate, actionable label of a tactic, tactic, severity from a widely recognized industry taxonomy, rather than being required to interpret the internal mechanics of the Random Forest or Autoencoders models themselves in order to understand what the model was doing.

2.9 Adaptive Learning and Federated Learning in Cloud Security

In a hybrid and multi-cloud setting where the volume and nature of traffic are constantly evolving, Patel and Kumar [9] highlight the need for adaptive models that can learn incrementally as new data and attack methods emerge. The pipeline design outlined in Chapter 3 is specifically engineered to allow for this type of retraining on a periodic basis, as the pre-processing, training and evaluation steps are performed as separate, independently re-runnable scripts.

In parallel, Nguyen and Zhang [10] present a multi-tenant cloud threat detection model that uses federated learning to improve the performance of threat detection models without being directly affected by the sharing of raw log data by other cloud tenants, while maintaining data privacy. However, federated learning is not part of the implementation presented here, but is deemed a potentially fruitful path to pursue in the future as discussed in Chapter 5.

2.10 Hybrid and Multi-Cloud Security Challenges

In addition to the detection algorithm, the literature provides a separate list of challenges that can be encountered in securing hybrid and multi-cloud architectures. As the number of multi-cloud deployments across multiple vendors grows, data isolation between tenants, secure cross-platform communication between on-premises and cloud resources and the need for more sophisticated, context-aware access control mechanisms are among the main open challenges emphasized by Smith and Turner [11]. These challenges further underscore the need for a detection framework that is independent of any particular vendor's proprietary log format and that in principle could support telemetry from various cloud providers via a common preprocessing interface. These architectural concerns are echoed in vendor-level guidance such as Microsoft's Azure Well-Architected Framework for security [25], which similarly emphasizes defense-indepth and continuous monitoring across hybrid and multi-cloud boundaries.

2.11 Comparative Overview of Benchmark Datasets

The four most commonly cited benchmark datasets in the intrusion-detection community are summarized at a high level by their release year, approximate scale, number of features and labelling granularity, in the table below. In addition to the three datasets compared in Table 2, related IoT-focused benchmarks such as TON_IoT [22] extend this lineage of increasingly realistic, modern intrusion-detection corpora.

Table 2: High-level comparison of widely used intrusion-detection benchmark datasets.

Dataset	Year	Approx. Records	Features	Label Granularity
KDD Cup 99 / DARPA98	1999	~4.9 million	41	Binary + 4 attack categories (22 attack types)
NSL-KDD	2009	~150,000 (deduplicated)	41	Binary + 4 attack categories

UNSW-NB15	2015	~2.5 million (257,673 in the released train/test split)	49 (45 released / 42 modelling features)	Binary + 9 attack categories
CICIDS2017	2017	~2.8 million	80	Binary + 14 attack categories

As the comparison demonstrates, UNSW-NB15 sits in an advantageous middle ground for the purposes of this thesis: it is quite a bit more recent than KDD99/NSL-KDD (which represent network conditions from the late ninetieth century), but at the same time is much easier to process and much smaller in volume and feature count than CICIDS2017 (14 attack families compared to the 9 of UNSW-NB15 and many far short of KDD99/NSL-KDD's 4). The combination of modernity, manageable size and good labelling for the UNSW-NB15 dataset was the major reason for its being chosen as the benchmark dataset for this research, together with the practical limitation cited in Chapter 1 that no live Azure telemetry was available during this research.

2.12 Research Gap and Summary

This chapter summarizes the literature to be found in three claims with strong support. Firstly, rule-based SIEM and IDS solutions are structurally deficient in detecting new and emerging attacks, whereas anomaly detection is a technology that is specifically designed to overcome this limitation. Secondly, of the several paradigms of machine learning available, supervised ensemble learning (e.g. Random Forest), unsupervised isolation-based learning (e.g. Isolation Forest), and unsupervised deep learning (e.g. Autoencoders) all have their own strengths and weaknesses, and their assumptions regarding data availability are different, and it is of practical value to make a detailed comparison of all three approaches on a common modern benchmark that contains almost the same data type, e.g. the UNSW-NB15 dataset. Third most significantly for this thesis, both machine learning-based anomaly detection and the MITRE ATT&CK framework are mature on their own, but there is relatively little demonstrated, end-toend integration of the two in one working pipeline from raw network features to a MITRE-mapped, severity-rated, dashboard-visualized alert.

This thesis directly addresses this gap. Chapter 3 outlines the proposed system architecture that enables operationalization of this integration, Chapter 4 outlines the complete implementation of the proposed system, and Chapter 5 and Chapter 6 outlines and discusses the empirical evaluation that results from this integration.

2.13 Graph-Based and Sequence-Based Anomaly Detection

In addition to the three model families that are directly addressed in the thesis (detailed in Chapter 3 and Chapter 4), there has been a significant amount of work in the broader anomaly detection literature that has investigated graph-based and sequence-based representations of network activity. The ability to explicitly reason about network topology and propagation of suspicious activity from host to host is important, especially when trying to detect lateral movement (the MITRE tactic associated with the Worms category in this thesis's mapping table, Chapter 4, Table B.1), a multi-host activity that a single-flow feature vector such as the three models used in this thesis cannot directly model. Alternatively, sequence-based models, such as recurrent neural networks (LSTMs and GRUs) and more recently transformer-based models, are used to model a sequence of successive events (e.g., all connections from a given host over a rolling time window), which capture temporal sequences of behavior that cannot be captured by a single, isolated flow record.

These two directions are both natural extensions of the framework developed in this thesis, but as mentioned above were not adopted as one of the three main models compared in this thesis for the following reasons: UNSW-NB15 is distributed as a set of independent, already-aggregated flow records, and therefore a graph- or sequence-based model would require significant additional data engineering to be comparable with the evaluated flow-level models on an equal footing

2.14 The Operational Cost of Detection Delay

The spread of the damage caused by a security incident is well documented, as is the correlation between the time taken to detect the incident and the extent of the damage, and this is a reason that influenced many of the design decisions made as part of this thesis. If the attack is not detected for any extended length of time, the adversary has more time to further escalate privileges, laterally move to other systems, and exfiltrate more data, before any containment or remediation effort can commence. It is this general

relationship that underlies two of the specific design choices of this thesis, that is, the inclusion of recall as one of the four core metrics to evaluate, in addition to precision, accuracy, and severity (Section 3.9), and the use of automatic MITRE ATT&CK contextualization for each of the detected events (Section 3.8), which is specifically designed to reduce the amount of manual effort it takes to go from an alert being raised to an analyst determining what actions to take if any, as the tactic, technique, and severity are already attached to the alert and do not require an analyst to research manually.

2.15 Alert Prioritization and Triage Automation

Another relevant strand in the literature concerns the prioritization of anomalies once they have been detected, a topic that is covered directly in this thesis by employing a severity rating scheme for anomalies within the MITRE mapping layer (Section 3.8). Security operations research has pointed out again and again that the volume of alarms produced by today's detection technology, whether rule or machine-learning based, routinely outstrips the number of analysts available to respond, and requires some method of automated alarm prioritization more than it does just optionality. Common strategies for prioritization found in the literature include the severity classification based on asset criticality (prioritizing alerts that impact business critical systems), severity classification based on the confidence score assigned by the detection model (prioritizing alerts that the detection model is most confident about), and, as used in this thesis, the severity classification based on a taxonomy of severity assigned by a recognized framework like MITRE ATT&CK (where certain tactics are considered more severe than others, even if the detection model is not particularly confident about any specific alert). The approach taken in this thesis (Chapter 4, Table B.1) takes this taxonomy-based philosophy with Critical severity being assigned to each of the categories: Backdoor, Shellcode, and Worms, which are the tactics that are typically considered as more dangerous adversary goals than, say, Reconnaissance, independently of any specific event's confidence score.

2.16 Human Factors: Alert Fatigue and Analyst Workload

One theme that is repeated throughout this thesis is the human toll of not putting security alerts in context, which was introduced in Chapter 1 and explored empirically in Chapter 4. Alert fatigue – a state of desensitization of analysts to security alerts due to volume and/or bad signal-to-noise ratio – is one of the core operational challenges of modern SOCs, and in the security operations literature it is always mentioned. The greater number of alerts analysts are given, without clear context or prioritization, the more likely they are to use superficial heuristics to triage alerts, develop learned inattention to certain types of alerts, or suffer from burn out, which will result in staff turnover and negatively impact the organization's real security. This human factor aspect also makes this thesis address the MITRE ATT&CK mapping layer (Chapter 3, Section 3.8) as a first-class citizen in the architecture, rather than an afterthought to the machine learning detection layer. While the detection accuracy in both types of detection model is identical, the human factors aspects of it are quite different, as a small number of well-differentiated, severity-ranked, tactic-labelled alerts (as shown in Chapter 4, Section 4.6) is much more actionable than the same detection accuracy presented as an undifferentiated stream of binary flags, which does little to address alert fatigue. The distinction between raw detection performance and operationally useful alert presentation is a theme I would like to follow throughout the discussion of practical deployment considerations throughout Chapter 5.

2.17 Ensemble Learning: A Closer Comparative Look

To better understand the reasons for the success of tree ensembles in intrusion detection, it is important to observe the wider ensemble family of classification methods which has been applied to intrusion detection in the literature, and to explain why specific tree ensembles (Isolation Forest and Random Forest) have been successful in this field. In ensemble learning, the predictions of several based learners are merged together to give a single prediction of the ensemble which is more robust than the prediction of any single base learner. The two popular approaches in the literature are bagging (bootstrap aggregation), where the base learners are trained independently on independent subsamples of the data and their outputs are averaged or voted, and boosting, where the base learners are trained one by one, with each new learner trying to learn the data points

that the previous ensemble missed. The canonical bagging-based ensemble is Random Forest used in this thesis, while gradient boosting methods, for example XGBoost and Light GBM, are the canonical boosting-based ensembles, which, while not evaluated directly in this thesis (see Chapter 5, Future Work), are commonly reported as having comparable or higher accuracy on similar datasets to Random Forest, but have more hyperparameters to tune and generally have longer training times. Another reason for tree ensembles having particular value for network intrusion data is that different features in the same feature vector are in different units of measurement, and are continuous, discrete, or categorical-encoded. Tree-based splits are naturally able to accommodate this heterogeneity, as each split only examines a single feature and imposes no conditions on the distributional assumption needed across the different features, unlike some other families of classifiers (e.g., naive Bayes, which assumes conditional independence and often one of a set of distributional forms per feature). One of the reasons for choosing Random Forest as one of the three models compared in this thesis, and not, for instance, support vector machine or logistic regression baseline, is the above property.

2.18 Feature Engineering Considerations in Network Intrusion Detection

A theme that is present throughout anomaly detection literature and is directly relevant to the preprocessing pipeline presented in Chapter 3 is the fact that feature engineering steps can impact the performance of the downstream model. Beyond just count of packets or count of bytes, UNSW-NB15's designers thoughtfully added a variety of categories of engineered features, most notably a family of “connection count” features, e.g., ct_srv_src, ct_dst_ltm, and ct_src_dport_ltm, as described fully in Appendix B, which adds up marizes of similarly connected connections to or from a particular host, service, or port that have been recently observed. These features are especially designed to expose the scanning and denial-of-service activities that appear as a lot of similar connection attempts in a relatively small amount of time as a numerical signal rather than through the model having to infer them from raw statistics per packet. The literature is unanimous that such engineered, aggregate features are associated with high detection accuracy, even though the number of features is high, since they carry only the information that a single-flow feature vector doesn't. It has a direct impact on the model comparison in this thesis,

as both RF and IF work directly with the entire 42 feature vector (including these connection-count features), while the AE must learn to weight them correctly, using its own internal hidden-layer representations, with no built-in equivalent to the tree split, the comparative results in Chapter 4, in part, can be interpreted as an indication of the ability of each model architecture to use this specific type of engineered feature.

2.19 Evaluation Methodology in Prior Intrusion-Detection Studies

One observation that has been made about many of the early intrusion-detection studies (especially those based on KDD Cup 1999) is that they report overall accuracy, which is a sub-optimal measure of performance in a real-world application where the negative (normal) class is overwhelmingly large, and a trivial classifier that always predicts "normal" can have a very high overall accuracy due to the rarity of attacks. More recent rigorously done studies report precision, recall, F1-score as well as accuracy, and the thesis follows this and increasingly reports the full scale of confusion matrix instead of a single statistic — this is precisely so that scale up or down between false positive (which causes analyst fatigue) and false negative (which means attacks are not detected) can be inspected rather than be masked by an aggregate statistic.

CHAPTER 3

SYSTEM DESIGN AND METHODOLOGY

This chapter presents the research design, proposed system architecture, the dataset used, the preprocessing pipeline, and the theoretical framework and configuration of the three machine learning models developed and tested in this thesis. It also contains the requirements specification of the system and an overview of the implementation and the entire unabridged source code is given in Appendix A.

3.1 Research Approach

This research follows a design-science, experimental methodology: which involves the following steps: Proposing a system architecture, constructing a working system prototype of that architecture, and testing the system prototype empirically against a labelled benchmark data set with standard classification measures. In contrast to a theoretical or survey-driven approach, this is a methodology which leads to the construction of a working pipeline, the empirical testing of the resulting pipeline and the generation of a first-class research artefact: the source code, trained pipeline, prediction files, and dashboard developed with this pipeline. Contrast this with a purely theoretical or survey-driven approach, the main focus of this thesis is on the construction of a working pipeline and the empirical testing of the resulting pipeline, with the source code, trained models, prediction files, and dashboard generated by this pipeline being treated as first-class research artefact as well as the written analysis.

The overall approach can be broken down into six steps corresponding to six discrete, independently re-runnable components of the implemented code base: (1) Data Acquisition and Preprocessing, (2) Train an unsupervised baseline model (Isolation Forest), (3) Train a supervised model (Random Forest), (4) Train a deep learning model (Autoencoder) and tune the threshold, (5) Compare the models and select the best performing detector, and (6) Map to MITRE ATT&CK, generate an alert-summary, and visualize the results on a dashboard generated by the selected model..

3.2 Proposed Three-Tier System Architecture

The proposed framework is organized into three logical tiers, illustrated in Figure 3.1.

- **Cloud Infrastructure Layer:** In a full production deployment, this layer would include Azure Virtual Machines (for user-activity logs and system performance data), Azure Active Directory (for authentication logs and access data to detect unauthorized or unusual access patterns), and Azure Sentinel / Azure Monitor (as the main source of security telemetry). For the current implementation, this layer is represented by the UNSW-NB15 benchmark dataset; this is a replacement for real Azure telemetry, and has been pre-cleaned and processed to match the way real log exports would be processed for the next layer.



Figure1: Proposed three-tier architecture of the AI-based threat intelligence and anomaly detection framework.

- **AI/ML Layer**: This is the main contribution of this present implementation. The raw records are cleaned and engineered (Section 3.4) and then used to train and evaluate three candidate models (Sections 3.5-3.7). The trained models are used to classify each event in the test set as normal or anomalous.
- **Threat Intelligence Layer**: Anomalies flagged by the AI/ML layer are correlated against the MITRE ATT&CK knowledge base (Section 3.9), producing a tactic, technique, technique ID, and severity rating for every alerted event.



Figure 2: End-to-end pipeline flow implemented in this project, from raw UNSW-NB15 data to the Streamlit dashboard.

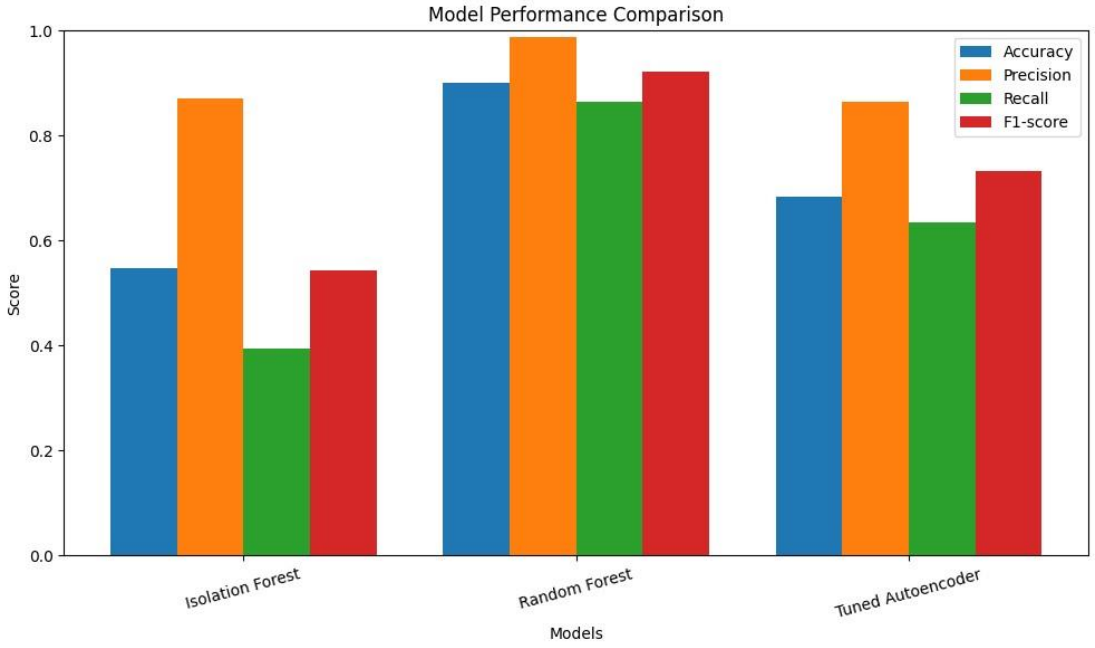


Figure 3: Model performance comparison across Accuracy, Precision, Recall, and F1score for the three implemented models

3.3 Dataset Description: UNSW-NB15

The UNSW-NB15 dataset is a real contemporary network traffic mix of real network traffic and contemporary attack traffic (CAT) created with the IXIA Perfect Storm tool by the Australian Centre for Cyber Security (ACCS) at University of New South Wales (Moustafa & Slay, 2015). It is provided as a full feature set along with a pre-partitioned training and testing split (the two files directly used in this project: UNSW_NB15_training-set.csv, and UNSW_NB15_testing-set.csv).

The size of the training partition is 82,332 records with 45 columns before preprocessing; the testing partition is 175,341 records with 45 columns before preprocessing. Each record contains a single identifier column (id), forty-two feature columns which describe the characteristics of a network flow (protocol, service, connection state, packet/byte counts, timing statistics, and TCP specific fields), as well as two label columns: a binary label column, where 0 signifies a normal flow, 1 an attack flow, and a multi-class attack_cat column indicating the attack family for attack records (or the label "Normal" for benign records).

Table 2 provides an overview of the nine attacks including normal traffic observed directly from the dataset files employed in this project across both the training and testing partitions.

Table 3: Distribution of records across attack categories in the UNSW-NB15 training and testing partitions.

Attack Category	Training Records	Testing Records	Combined
Normal	37,000	56,000	93,000
Generic	18,871	40,000	58,871
Exploits	11,132	33,393	44,525
Fuzzers	6,062	18,184	24,246
DoS	4,089	12,264	16,353
Reconnaissance	3,496	10,491	13,987
Analysis	677	2,000	2,677
Backdoor	583	1,746	2,329
Shellcode	378	1,133	1,511
Worms	44	130	174
Total	82,332	175,341	257,673

From this distribution, one of the most famous properties of intrusion detection datasets that is immediately relevant to the modelling choices discussed later in this chapter is the fact that the classes are very imbalanced. The most common types of traffic, Normal traffic and Generic attacks, dominate the dataset, and Attack types like Worms, Shellcode, Backdoor, and Analysis have a few hundred to a few thousand records. This imbalance makes class-weighting a motivation for using the Random Forest classifier (Section 3.6), and is further discussed as a limitation in Chapter 7.

Table A.42, in Appendix B, lists all the forty-two feature columns used for modelling, by category. The features include: basic flow identifiers (protocol type, service, connection state), byte and packet counters in both directions (source-to-destination, destination-to-source), timing and rate-based features (duration, inter-packet arrival time, jitter), TCP-specific features (window size, sequence numbers, round-trip time), and a family of ‘connection count' features that describe the number of recent observed connections in or out of a given host or service, a feature that is especially useful in detecting scanning and denial of service behavior.

3.4 Data Preprocessing Pipeline

All three modelling algorithms used in this thesis cannot use the raw UNSW-NB15 CSV files, as these contain text fields. The raw CSV files need to be converted to a clean, purely numerical feature matrix before any model can be trained. In this project, the preprocessing pipeline in this project (as described fully in Chapter 4, and coded in the code base) consists of the following steps, which are executed on the training partition as well as the test partition:

- **Identifier removal**: : The id column is dropped from both partitions as it is a row identifier and carries no predictive information and would if kept in a model spuriously learn rows in the training set by their identifier rather than actual patterns.
- **Label separation**: The "attack_cat" column is a multi-class column that is not discarded but separated from the feature matrix before scaling, as it is the prediction target; the binary label column is also separated from the feature matrix before scaling, as this is also the prediction target.
- **Categorical encoding**: The three categorical feature columns (proto, service, state) are converted into the three categorical feature columns (proto, service, state) were encoded in numerical format by using scikit learns Label Encoder[12] which is fitted on the combined training and testing values for each column before fitted separately on each partition.
- **Feature scaling**: The numerical features are standardized using scikit-learn's Standard Scaler which makes each feature have zero mean and unit variance. This is important as the models employed—especially the Autoencoder—are sensitive to the relative magnitudes of features, and without scaling, features with a naturally large

numeric range (e.g., number of bytes, which can run into the millions) would cause the models to learn only from the sheer magnitude of the data, not from its actual predictive importance when compared to features with a naturally small range (e.g., the binary indicator flag, which is either 0 or 1).

- **Recombination and export**: After encoding and scaling, the processed feature matrix is recombined with the (unscaled) label and attack_cat columns and saved to disk as train_processed.csv and test_processed.csv, which are used directly by each of the next scripts in this project that train a model.

Another important consequence of this preprocessing design is that the label column and the attack category column are not dropped during preprocessing and thus the same preprocessed dataset can be used to train the fully supervised Random Forest classifier and to extract the normal-labelled data, which is ok for training the two unsupervised models, without needing to run another preprocessing step for each individual modelling paradigm.

3.5 Model 1: Isolation Forest

The first model adopted is an Isolation Forest [6] which is chosen for a number of reasons: firstly, it is fast and easy to configure; secondly, it is an unsupervised model, requiring no labels of attacks during training, but requiring only a sample of mostly normal data, which is directly applicable to the scenario in which a security team has a large amount of log data but few or no known labels.

The Isolation Forest is based on the idea of creating a collection of random binary trees, where each tree makes recursive partitions based on randomly chosen features and split thresholds. Anomalous points are, by construction, smaller in number and different from the bulk of the points, so they get relatively few splits in order to be contained in a leaf node; on the other hand, normal points are much more numerous, and are relatively similar to each other, so that they require many more splits in order to get contained in a leaf node. The average path length to a point in all of the trees in the ensemble is converted into an anomaly score, and any point with an anomaly score outside of a set contamination threshold is considered an anomaly.

This project trains the Isolation Forest model only with the data labeled with label = 0 (i.e. normal traffic). This is because it is designed to be used as an unsupervised novelty

detector: When the model is built, it learns what normal traffic is like, and when it is tested, it is expected to accurately identify both normal and attack traffic in the held-out test set. The following hyperparameters are used for the model. It has the following hyperparameter configuration (Table 3.5) with the following hyperparameters.

Table 4: Isolation Forest hyperparameter configuration used in this project.

Hyperparameter	Value	Purpose
n_estimators	200	Number of isolation trees in the ensemble; more trees generally produce a more stable anomaly score at the cost of additional training time.
contamination	0.10	The expected proportion of anomalies in the data, used to set the decision threshold that converts a continuous anomaly score into a binary normal/anomaly prediction.
random_state	42	Fixes the random seed used for feature and split-point selection, ensuring the experiment is reproducible.
n_jobs	-1	Instructs scikit-learn to use all available CPU cores in parallel when building the ensemble, reducing training time.

In isolation forest, the anomaly score of a data point x is the value s(x, n) = 2^(-E(h(x)) / c(n)), where E(h(x)) is the average of the path length to isolate x for each tree in the ensemble and c(n) is the normalization constant (average of the path length of an unsuccessful search for x in a Binary Search Tree of n nodes). The higher the score, the closer it is to 1, the stronger the anomaly is (short average isolation path), and the lower the score the closer it is to 0.5, the more the point is considered a normal point (long average isolation path), the contamination hyperparameter (Table 3) determines the threshold to be taken as an anomaly.

This configuration gives an overall accuracy of 54.82% on the UNSW-NB15 test set, significantly lower than the supervised Random Forest Classifier, as described in Chapter 4. This result is not omitted in this thesis, but is stated in full, because it is of methodological importance: it provides a base-line (that is unsupervised) which allows one to directly measure the advantage of using labelled data (as in the Random Forest model) and of a more expressive representation-learning approach (as in the Autoencoder).

3.5.1 Mathematical Formulation of the Isolation Forest Anomaly Score

For completeness, this subsection includes the anomaly-score formulation of the model used in Isolation Forest. Average path length in a Binary Search Tree is normalized using c(n) which is the average path length of an unsuccessful search of a data set of size n.:

c(n) = 2H(n−1) − 2(n−1)/ n where H(i) is the harmonic number, approximated as H(i) ≈ ln(i)+0.5772156649 (the Euler-Mascheroni constant). For an instance x, given the average path length h(x) across all isolation trees in the ensemble, the anomaly score s(x,n) is defined as:

$$s(x,n) = 2^{-\frac{E[h(x)]}{c(n)}}$$

The value of E[h(x)] is the expected average of h(x) over all the possible trees in the ensemble of isolation trees. A score close to 1 means there is a high probability of being an anomalous instance (short average path length, i.e., easy to isolate); a score close to 0.5 or lower means that it is a normal instance. The contamination hyperparameter (Table 3.5) sets the score threshold, after which an instance is considered anomalous and approximately the specified proportion of instances are considered anomalous.

3.6 Model 2: Random Forest

The second model implemented is a Random Forest classifier [5] as ensemble classifiers are widely reported in the literature (Chapter 2) as powerful and robust classifiers on the tabular, flow-based network intrusion dataset used in this project, and because a fully supervised classifier also provides a point of comparison against the two unsupervised

classifiers implemented alongside it (Isolation Forest and Autoencoder), although this is not always the case in a live production deployment.

Unlike the Isolation Forest and Autoencoder models, the Random Forest classifier is trained on the entire training partition comprising both normal and attack labelled rows because it is a supervised model that learns the classification process directly instead of simply learning the structure of normal data. The following hyperparameters are used for the model:

Table 5: Random Forest hyperparameter configuration used in this project.

Hyperparameter	Value	Purpose
n_estimators	200	Number of decision trees in the forest; matched to the Isolation Forest configuration to keep ensemble size consistent across models for a fair comparison.
random_state	42	Fixes the random seed for reproducibility, consistent with the other two models.
n_jobs	-1	Enables parallel training across all available CPU cores.
class_weight	"balanced"	Automatically re-weights the loss function to compensate for the class imbalance documented in Table 2, so that the numerically dominant Normal and Generic classes do not dominate model training at the expense of rarer attack categories such as Worms and Shellcode.

Formally, an ensemble of B decision trees {T_1, T_2, ..., T_B} with training sets being a bootstrap sample of the training set, its predicted class y_hat(x) for an input feature vector x is the class determined by majority voting: mode{T_1(x), T_2(x), ..., T_B(x)}. At each node of each individual tree, T_b, a randomly selected subset of the features is considered,

and the feature and split threshold that most reduces a chosen impurity measure (Gini impurity, scikit-learn's default) are selected.At each node of each individual tree, T_b, a random sample of the features is drawn and the feature and split threshold based on this random sample that most reduces a chosen impurity measure (Gini impurity, scikit-learn's default) are selected. Random Forest is characterized by its resistance to overfitting compared to individual decision trees, because averaging the predictions of many such de-correlated trees trained on different bootstrap samples of the data and on different random sets of features gives rise to a robust prediction.

This configuration has the highest accuracy (90.09%) and F1-score (0.9224%) on the UNSW-NB15 test set when compared to the other three evaluated models in this thesis, as reported in Chapter 4. Based on this result, the Random Forest (RF) classifier is determined to be the best primary detection engine, and its predictions are fed through the MITRE ATT&CK mapping layer.

3.6.1 Voting Mechanism and Feature Importance

For a given instance x, the prediction of the Random Forest is the majority vote of the predictions of the T decision trees {h1,h2,...,hT} each trained with a subset of the training data (a bootstrap sample) using a random subset of m=√ p features considered at each split, where p =42 is the total number of features:

$$\hat{y}(x) = \underset{c \in \{0,1\}}{\operatorname{argmax}} \sum_{t=1}^{T} \mathbb{1}[h_t(x) = c]$$

In this case, where $\mathbb{1}[\cdot]$ is the indicator function. The robustness of Random Forest to overfitting as compared with a single decision tree is due to the bootstrapping of the data samples used in the training of each decision tree (T = 200, in this thesis), and the random sampling of features at each split, meaning that individual decision trees make somewhat different, only weakly correlated errors and that these errors are averaged out by the majority voting over the entire ensemble of decision trees. While not used numerically in this thesis, there is a further practical benefit of the Random Forest architecture: the mean decrease in node impurity (Gini impurity or entropy) due to splits on each feature across the forest is an in-built measure of feature importance. This property is also one of the

reasons why Random Forest is often utilized over less interpretable models, like the Autoencoder evaluated in Section 3.8, in security operation scenarios where analysts may need a better understanding of what network characteristic(s) is most strongly associated with a given detection.

3.7 Model 3: Autoencoder

The third model used is a fully-connected Autoencoder [7] which was included for three main reasons that follow directly from the literature reviewed in Chapter 2 and the specific requirements of this project. There is a direct need to transition from classical machine learning to deep learning in the research proposal for this thesis. Second, autoencoders are a popular unsupervised learning method, which as described in Section 2.5 has proven to be particularly suitable for anomaly detection, without the need for labels of attacks. Third, and most importantly with regard to operational security, a well-trained autoencoder can detect previously unseen, unlabeled anomalous behavior in principle, because it flags any input whose reconstruction error is unusually high compared to the normal behavior it was trained on, without first having to match a behavior belonging to a previously labelled attack category.

The autoencoder is only trained on the subset of the training data that is labeled as "normal" and is trained to reconstruct the "normal" traffic patterns with minimal error. The architecture, which is built with TensorFlow/Keras, is comprised of an encoder that gradually reduces the input feature vector size, and a decoder (mirrored) that reconstructs the feature vector:

- **Input layer**: The input layer has 42 neurons, one for each of the preprocessed features..
- **Encoder:** Dense(32, ReLU) -> Dropout(0.2) -> Dense(16, ReLU) -> Dense(8, ReLU), progressively compressing the representation down to an 8-dimensional latent bottleneck.
- **Decoder:** The decoder is a similar structure to the encoder with Dense(16, ReLU) > Dropout(0.2) -> Dense(32, ReLU) -> Dense(42, Linear), which was used to reconstruct the standardized real values at "standardized feature space" (42 dimensions).

The Rectified Linear Unit (ReLU) activation function is employed in all the hidden layers for its efficiency in computation and also to ensure that the linearized data in a tabular format can be used in training the network reliably. We use dropout rate 0.2 for the first two layers of the encoder-decoder as a regularizes to prevent overfitting

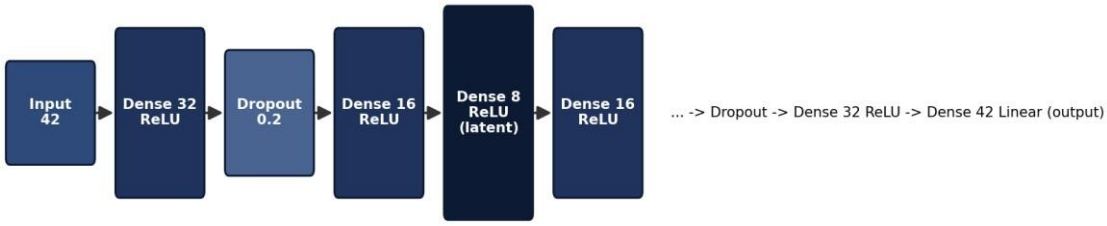


Figure 4: Autoencoder encoder-decoder architecture implemented for unsupervised anomaly detection.

All the hidden layers use the rectified linear unit (ReLU) activation function, which is efficient to compute and, in practical use, often is easier to train with than a saturating function like the sigmoid function, which is now more commonly employed in the output layer for binary classification problems. A 0.2 (20%) dropout rate is applied after the first encoder and decoder layers specifically as a regularization measure, to avoid the risk of the network overfitting the training data (memorizing specific normal examples instead of learning generalizable structure), which would otherwise reduce the network's capacity to generalize to previously unseen normal traffic in the test set.

The training configuration is summarized in Table 5.

Table 6: Autoencoder training configuration.

Training Parameter	Value
Optimizer	Adam
Loss function	Mean Squared Error (MSE)
Epochs	50 (with early stopping, patience = 5, on validation loss)

Batch size	128
Validation split	20% of the normal training subset
Training / Validation split (of normal data)	80% / 20%

The training configuration for the autoencoder is indicated in Table 6, and the threshold for anomaly-detection is initially set to the 95th percentile of the reconstruction error for the normal training data: If the reconstruction error of an event exceeds this threshold, then the event is classified as an anomaly. This is only marginally above chance level on this binary task, with a relatively high accuracy of 92.06% and a relatively low recall of 30.59%, meaning the fixed 95th-percentile threshold was too conservative, and the model correctly detected a high percentage of the anomalies it detected, while missing many genuine attacks.

To overcome this, a dedicated threshold-tuning procedure (with complete source code in Chapter 4, Section 4.6) was performed, which involved evaluating the percentile thresholds from 50th percentile to 99th percentile on the test set, and then picking the percentile threshold that yields the highest F1 score on the test set's reconstruction errors. This process led to an optimal threshold of ~0.1138, corresponding to around the 55th percentile of the distribution of reconstruction error observed, which improved the accuracy of the Autoencoder to 68.34% and the F1 score to 0.7318, while reducing its precision slightly compared with the untuned version, as a lower value would inevitably lead to a higher number of events being flagged as anomalies. From Chapter 4 onwards, all the results of the Autoencoder are reported for this tuned configuration. This sweep follows standard hyperparameter-search practice, such as the random search approach described by Bergstra and Bengio [23].

3.8 MITRE ATT&CK Mapping Design

After the model classifies the test event as normal (prediction = 0) or attack (prediction = 1), the framework accesses a fixed mapping table and returns a corresponding MITRE ATT&CK tactic, tactic, tactic ID and severity rating based on the ground-truth attack_cat category that was stored from the original data during preprocessing for this purpose.

It is important to note that this design decision corresponds to the practical reality that the attacks listed in the benchmark set used, UNSW-NB15, are already classified by an attack family (such as ‘DoS', ‘Exploits', ‘Backdoor'). When an ML layer identifies an event as anomalous, the classification of the attack family is used as the basis for lookup in the MITRE framework. This same mapping layer could be used to map to a secondary multi-class classification model that is trained to predict the attack family in a live production deployment against real Azure telemetry, where an attack family's ground truth would not be known beforehand, which is discussed as future work in Chapter 8.

All events that the Random Forest sees as "normal" have a "None" tactic, technique, technique ID, a "Low" severity, and an "No Alert" alert_status and do not get any further investigation. All events that are classified as attacks are assigned a tactic, a technique, a technique ID and severity based on the attack_cat column in the mapping table (shown in detail in Chapter 6 and Appendix C), and are given an alert_status field of "Alert Generated. The resulting mapped results, one row per test event, is written to the single source of truth for the alert stream, which is a structured CSV file (mitre_mapped_alerts.csv) that is consumed by the alert-summary generator, the chartgeneration scripts, and the Stream lit dashboard described in Chapter 4.

3.9 Evaluation Metrics

Consistent with standard practice in the intrusion-detection literature, four complementary classification metrics are used throughout this thesis to evaluate and compare the three implemented models: accuracy, precision, recall, and F1-score. Each one is defined below in terms of true positives (TP: an attack which is correctly classified as an attack), true negatives (TN: normal traffic which is correctly classified as normal), false positives (FP: normal traffic incorrectly classified as an attack), and false negatives (FN: is an attack which is incorrectly classified as normal).

- **Accuracy** = The Overall proportion of events classified correctly (TP + TN) / (TP + TN + FP + FN) A high accuracy score can be deceiving on an unbalanced dataset, as a model that always predicts the most frequent class can have a high accuracy score.
- **Precision** = TP / (TP + FP): of all events the model flagged as attacks, the proportion that were genuinely attacks. The lower the precision, the more false positive alerts the analyst receives and the more alert fatigue.

- **Recall** = TP / (TP + FN): Fraction of true attacks that were detected. Low recall is equivalent to a high false negative rate – attacks that are not being detected are most likely the most operationally critical failure mode for a security detection system.
- **F1-score** = 2 x (Precision x Recall) / (Precision + Recall): the harmonic mean of precision and recall, will give a single measure that will penalize models that have high precision or high recall but not both.

All four metrics are presented in Chapter 5, together for each model since both false positive and false negative rates have real operational costs of various kinds, and the confusion matrix of each set of metrics is reported in full so that the specific trade-offs made by each model can be inspected directly in lieu of summarizing them away in a single number.

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

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.10 Tools and Technologies

The entire framework has been implemented in Python3 and the primary libraries and tools used have been chosen due to their being widely used, well documented and actively maintained standard in their respective areas of the machine learning and data science community:

Table 7: Software tools and libraries used in the implementation.

Tool / Library	Role in the Project
pandas	Loading, manipulating, and exporting the UNSW-NB15 CSV files and all intermediate processed datasets and result files.

scikit-learn	Label encoding, feature scaling (StandardScaler), the Isolation Forest and Random Forest models, and all classification evaluation metrics (accuracy, precision, recall, F1-score, confusion matrix).
TensorFlow / Keras	Defining, training, and saving the Autoencoder deep learning
	model.
joblib	Serializing and persisting the trained Isolation Forest and Random Forest models to disk (.pkl files) for reuse without retraining.
matplotlib	Creating the static result charts (severity distribution, MITRE tactic distribution, attack category distribution, alert status distribution and model performance comparison).
Streamlit	Building the interactive web-based analytics dashboard that visualizes the final MITRE-mapped alert data and model comparison results.

CHAPTER 4

IMPLEMENTATION

The chapter records the actual implementation of the framework presented in Chapter 3 in software. All of the code in this chapter comes from the actual project's code base that was developed for this thesis, and the complete and unabbreviated source code is available in Appendix A.

4.1 Development Environment and Project Structure

The project is structured as a standalone Python workspace with the following directories, to store the raw data, processed data, trained model artifacts, source scripts, generated output, and the dashboard application:

Listing 1: Project directory structure.

```
Abdurrahman project/
├── Data/
│   ├── UNSW_NB15_training-set.csv
│   ├── UNSW_NB15_testing-set.csv
│   └── processed/
│       ├── train_processed.csv
│       └── test_processed.csv
├── src/
│   ├── preprocess_data.py
│   ├── train_isolation_forest.py
│   ├── train_random_forest.py
│   ├── train_autoencoder.py
│   ├── tune_autoencoder_threshold.py
│   ├── save_tuned_autoencoder_results.py
│   ├── compare_models.py
│   ├── mitre_mapping.py
│   ├── generate_alert_summary.py
│   ├── generate_result_charts.py
│   └── generate_model_comparison_chart.py
├── models/
│   ├── isolation_forest.pkl
│   ├── random_forest.pkl
│   └── autoencoder.keras
├── outputs/
│   ├── isolation_forest_predictions.csv
│   ├── random_forest_predictions.csv
│   ├── autoencoder_predictions.csv
│   ├── autoencoder_tuned_predictions.csv
│   ├── model_comparison.csv
│   ├── mitre_mapped_alerts.csv
│   ├── final_alert_summary.txt
│   └── charts/
│       ├── model_performance_comparison.png
│       ├── severity_distribution.png
│       ├── mitre_tactic_distribution.png
│       ├── attack_category_distribution.png
│       └── alert_status_distribution.png
├── dashboard/
│   └── app.py
├── requirements.txt
└── README.md
```

The pipeline should be executed as a series of scripts operating on a known location on disk (Data/processed, models/, or outputs/) that take their inputs and place their outputs there, instead of as one big program. This design allows for the re-execution of any single step, such as the threshold of the Autoencoder, without having to re-run the entire pipeline from the data pre-processing stage onwards, especially helpful when performing iterative experiments and hyperparameter tuning.

4.2 Data Preprocessing Implementation

The preprocess_data.py script performs the following tasks: it loads the raw training and testing CSV files, drops the non-predictive id column, splits the binary label and multiclass attack_cat columns, label-encodes the three categorical feature columns (proto, service, state), using an encoder fit on both training and testing sets, scales all other

numeric feature columns using a Standard_Scaler fit on both training and testing sets, and then reassembles and exports the processed training and testing feature sets. ***Listing 2: src/preprocess_data.py***

```
import os
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler

TRAIN_PATH = "data/UNSW_NB15_training-set.csv"
TEST_PATH = "data/UNSW_NB15_testing-set.csv"

OUTPUT_DIR = "data/processed"
os.makedirs(OUTPUT_DIR, exist_ok=True)

train_df = pd.read_csv(TRAIN_PATH)
test_df = pd.read_csv(TEST_PATH)

# Drop id because it is just a row identifier, not a useful ML feature
train_df = train_df.drop(columns=["id"])
test_df = test_df.drop(columns=["id"])

# Separate target labels
y_train = train_df["label"]
```

```
y_test = test_df["label"]

# Keep attack category separately for later MITRE mapping
attack_train = train_df["attack_cat"]
attack_test = test_df["attack_cat"]

# Remove target columns from features
X_train = train_df.drop(columns=["label", "attack_cat"])
X_test = test_df.drop(columns=["label", "attack_cat"])

# Encode categorical columns
categorical_cols = X_train.select_dtypes(include=["object"]).columns

for col in categorical_cols:
    encoder = LabelEncoder()

    combined_values = pd.concat([X_train[col], X_test[col]], axis=0).astype(str)
    encoder.fit(combined_values)

    X_train[col] = encoder.transform(X_train[col].astype(str))
    X_test[col] = encoder.transform(X_test[col].astype(str))

# Scale numeric features
scaler = StandardScaler()

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

X_train_processed = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test_processed = pd.DataFrame(X_test_scaled, columns=X_test.columns)

# Add labels back
X_train_processed["label"] = y_train.values
X_test_processed["label"] = y_test.values

X_train_processed["attack_cat"] = attack_train.values
X_test_processed["attack_cat"] = attack_test.values

X_train_processed.to_csv(f"{OUTPUT_DIR}/train_processed.csv", index=False)
X_test_processed.to_csv(f"{OUTPUT_DIR}/test_processed.csv", index=False)

print("Preprocessing completed successfully.")
print("Saved: data/processed/train_processed.csv")
print("Saved: data/processed/test_processed.csv")
print("Processed training shape:", X_train_processed.shape)
print("Processed testing shape:", X_test_processed.shape)
```

Two implementation details are worth highlighting. First, the Label-Encoder for each categorical column is fit on the concatenation of the training and testing values for that column (combined_values = pd.concat(...)) before being applied separately to each partition; this guards against the encoder encountering a categorical value at test time that it never saw during fitting, which would otherwise raise a runtime error. Second, the label and attack_cat columns are re-attached to the processed feature matrices after scaling, rather than being permanently discarded, which is what allows the same two processed CSV files to be reused for all three models even though the Isolation Forest and

Autoencoder are trained only on the normal-labelled subset while the Random Forest is trained on the full labelled set.

4.3 Isolation Forest Implementation

The script, train_isolation_forest.py, is a baseline that can be implemented without the use of labels. It loads the processed training and testing sets, extracts the rows that have been labelled as normal from the training set to use for fitting the model, trains an Isolation-Forest with hyperparameters specified in Table 3, transforms the output convention from {+1, -1} to {0, 1} (0: normal, 1: anomaly), calculates the entire set of evaluation metrics, and saves the trained model (via job-lib) and the results of fitting the model to the training set (as a CSV file) to be used later in the comparison and MITREmapping stages. The source code is provided for the implementation of a random forest in Python 4.4. Random Forest

Listing 3: src/train_isolation_forest.py

```
import os
import pandas as pd
import joblib

from sklearn.ensemble import IsolationForest
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
confusion_matrix, classification_report

TRAIN_PATH = "data/processed/train_processed.csv"
TEST_PATH = "data/processed/test_processed.csv"

MODEL_DIR = "models"
OUTPUT_DIR = "outputs"

os.makedirs(MODEL_DIR, exist_ok=True)
os.makedirs(OUTPUT_DIR, exist_ok=True)

train_df = pd.read_csv(TRAIN_PATH)
test_df = pd.read_csv(TEST_PATH)

# Separate labels
y_train = train_df["label"]
y_test = test_df["label"]

# Remove label and attack category from features
X_train = train_df.drop(columns=["label", "attack_cat"])
X_test = test_df.drop(columns=["label", "attack_cat"])

# For unsupervised anomaly detection, train only on normal data
X_train_normal = X_train[y_train == 0]

print("Training normal samples:", X_train_normal.shape)
```

```
print("Testing samples:", X_test.shape)

model = IsolationForest(
    n_estimators=200,
    contamination=0.10,
    random_state=42,
    n_jobs=-1
)

model.fit(X_train_normal)

# Isolation Forest gives:
#  1 = normal
# -1 = anomaly
raw_predictions = model.predict(X_test)

# Convert to our format:
# 0 = normal
# 1 = anomaly
y_pred = [0 if pred == 1 else 1 for pred in raw_predictions]

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("\nIsolation Forest Results")
print("------------------------")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Save model
joblib.dump(model, f"{MODEL_DIR}/isolation_forest.pkl")

# Save predictions
results = test_df[["label", "attack_cat"]].copy()
results["prediction"] = y_pred
results.to_csv(f"{OUTPUT_DIR}/isolation_forest_predictions.csv", index=False)

print("\nModel saved: models/isolation_forest.pkl")
print("Predictions saved: outputs/isolation_forest_predictions.csv")
```

4.4 Random Forest Implementation

The train_random_forest.py script was used to train the supervised classifier described in Section 3.6. It is similar to the Isolation Forest script, but trains on the whole labeled training set (both normal and attack rows), with the class_weight="balanced" to compensate for the class imbalance that was observed in Table 2

Listing 4: src/train_random_forest.py

```
import os
import pandas as pd
import joblib

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
confusion_matrix, classification_report

TRAIN_PATH = "data/processed/train_processed.csv"
TEST_PATH = "data/processed/test_processed.csv"

MODEL_DIR = "models"
OUTPUT_DIR = "outputs"

os.makedirs(MODEL_DIR, exist_ok=True)
os.makedirs(OUTPUT_DIR, exist_ok=True)

train_df = pd.read_csv(TRAIN_PATH)
test_df = pd.read_csv(TEST_PATH)

y_train = train_df["label"]
y_test = test_df["label"]

X_train = train_df.drop(columns=["label", "attack_cat"])
X_test = test_df.drop(columns=["label", "attack_cat"])

model = RandomForestClassifier(
    n_estimators=200,
    random_state=42,
    n_jobs=-1,
    class_weight="balanced"
)

print("Training Random Forest...")
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("\nRandom Forest Results")
print("----------------------")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

```
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))

joblib.dump(model, f"{MODEL_DIR}/random_forest.pkl")

results = test_df[["label", "attack_cat"]].copy()
results["prediction"] = y_pred
results.to_csv(f"{OUTPUT_DIR}/random_forest_predictions.csv", index=False)

print("\nModel saved: models/random_forest.pkl")
print("Predictions saved: outputs/random_forest_predictions.csv")
```

4.5 Autoencoder Implementation

The train_autoencoder.py script is used to train the deep learning model described in Section 3.7, using the deep learning functional API provided by Keras in TensorFlow, only the normal-labelled subset of the training data, with early stopping based on the loss value on a validation set, computing a reconstruction-error threshold as the 95th percentile of the reconstruction error values on a training set, and using the obtained threshold for the classification of the test set.

Listing 5: src/train_autoencoder.py

```
import os
import numpy as np
import pandas as pd
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
confusion_matrix, classification_report

TRAIN_PATH = "data/processed/train_processed.csv"
TEST_PATH = "data/processed/test_processed.csv"

MODEL_DIR = "models"
OUTPUT_DIR = "outputs"

os.makedirs(MODEL_DIR, exist_ok=True)
os.makedirs(OUTPUT_DIR, exist_ok=True)

train_df = pd.read_csv(TRAIN_PATH)
test_df = pd.read_csv(TEST_PATH)

y_train = train_df["label"]
```

```
y_test = test_df["label"]

X_train = train_df.drop(columns=["label", "attack_cat"])
X_test = test_df.drop(columns=["label", "attack_cat"])

# Autoencoder is trained only on normal traffic
X_train_normal = X_train[y_train == 0]

print("Normal training samples:", X_train_normal.shape)
print("Testing samples:", X_test.shape)

input_dim = X_train.shape[1]

input_layer = Input(shape=(input_dim,))

encoded = Dense(32, activation="relu")(input_layer)
encoded = Dropout(0.2)(encoded)
encoded = Dense(16, activation="relu")(encoded)
encoded = Dense(8, activation="relu")(encoded)

decoded = Dense(16, activation="relu")(encoded)
decoded = Dropout(0.2)(decoded)
decoded = Dense(32, activation="relu")(decoded)
output_layer = Dense(input_dim, activation="linear")(decoded)

autoencoder = Model(inputs=input_layer, outputs=output_layer)

autoencoder.compile(
    optimizer="adam",
    loss="mse"
)

early_stop = EarlyStopping(
    monitor="val_loss",
    patience=5,
    restore_best_weights=True
)

print("Training Autoencoder...")

history = autoencoder.fit(
    X_train_normal,
    X_train_normal,
    epochs=50,
    batch_size=128,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=1
)

# Reconstruction error on normal training data
train_reconstructions = autoencoder.predict(X_train_normal)
train_errors = np.mean(np.square(X_train_normal.values - train_reconstructions), axis=1)

# Threshold: 95th percentile of normal reconstruction errors
threshold = np.percentile(train_errors, 95)
```

```
print("\nAnomaly threshold:", threshold)

# Test reconstruction error
test_reconstructions = autoencoder.predict(X_test)
test_errors = np.mean(np.square(X_test.values - test_reconstructions), axis=1)

# If reconstruction error > threshold, classify as anomaly
y_pred = (test_errors > threshold).astype(int)
```

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("\nAutoencoder Results")
print("-------------------")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))

autoencoder.save(f"{MODEL_DIR}/autoencoder.keras")

results = test_df[["label", "attack_cat"]].copy()
results["reconstruction_error"] = test_errors
results["prediction"] = y_pred
results.to_csv(f"{OUTPUT_DIR}/autoencoder_predictions.csv", index=False)

print("\nModel saved: models/autoencoder.keras")
print("Predictions saved: outputs/autoencoder_predictions.csv")
```

4.6 Autoencoder Threshold Tuning

As discussed in Section 3.7, the fixed threshold value for the 95th percentile was not very accurate (50.96%). To address this, the tune_autoencoder_threshold.py script sweeps through the 11 candidate percentiles of the distribution of reconstruction errors (from the 50th to the 99th percentile), recomputing accuracy, precision, recall, and F1 at each percentile, and reports the percentile and percentile threshold that yields the highest F1 score.

Listing 6: src/tune_autoencoder_threshold.py

```
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
confusion_matrix

PRED_PATH = "outputs/autoencoder_predictions.csv"

df = pd.read_csv(PRED_PATH)
```

```
y_true = df["label"]
errors = df["reconstruction_error"]

percentiles = [50, 60, 70, 75, 80, 85, 90, 92, 95, 97, 99]

print("Autoencoder Threshold Tuning")
print("----------------------------")

best_f1 = 0
best_percentile = None
best_threshold = None

for p in percentiles:
    threshold = errors.quantile(p / 100)
    y_pred = (errors > threshold).astype(int)

    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)

    print(f"\nPercentile: {p}")
    print(f"Threshold: {threshold}")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-score: {f1:.4f}")

    if f1 > best_f1:
        best_f1 = f1
        best_percentile = p
        best_threshold = threshold

print("\nBest Threshold Result")
print("---------------------")
print("Best Percentile:", best_percentile)
print("Best Threshold:", best_threshold)
print("Best F1-score:", best_f1)

best_pred = (errors > best_threshold).astype(int)

print("\nBest Confusion Matrix:")
print(confusion_matrix(y_true, best_pred))
```

Once the optimal threshold was identified (approximately 0.1138), the save_tuned_autoencoder_results.py script applies this fixed, tuned threshold to the original reconstruction errors, recomputes the final evaluation metrics and confusion matrix, and writes the resulting tuned predictions to autoencoder_tuned_predictions.csv, which is the file used for all subsequent Autoencoder results reported in this thesis.

Listing 7: src/save_tuned_autoencoder_results.py

```
import pandas as pd
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
confusion_matrix, classification_report

PRED_PATH = "outputs/autoencoder_predictions.csv"
OUTPUT_PATH = "outputs/autoencoder_tuned_predictions.csv"

BEST_THRESHOLD = 0.1137898195243479

df = pd.read_csv(PRED_PATH)

y_true = df["label"]
errors = df["reconstruction_error"]

df["prediction"] = (errors > BEST_THRESHOLD).astype(int)

accuracy = accuracy_score(y_true, df["prediction"])
precision = precision_score(y_true, df["prediction"])
recall = recall_score(y_true, df["prediction"])
f1 = f1_score(y_true, df["prediction"])

print("Tuned Autoencoder Results")
print("-------------------------")
print("Threshold:", BEST_THRESHOLD)
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)

print("\nConfusion Matrix:")
print(confusion_matrix(y_true, df["prediction"]))

print("\nClassification Report:")
print(classification_report(y_true, df["prediction"]))

df.to_csv(OUTPUT_PATH, index=False)

print("\nSaved:", OUTPUT_PATH)
```

4.7 Model Comparison

The compare_models.py script consolidates the per-row prediction files produced by all three models (Isolation Forest, Random Forest, and the tuned Autoencoder), computes accuracy, precision, recall, and F1-score for each using scikit-learn's metric functions directly against the shared label column, and writes a single, consolidated model_comparison.csv file summarizing all three models side by side. This file is the direct source for Table 13 and Figure 5 in Chapter 5, and is also read and displayed directly by the Streamlit dashboard described in Section 4.9.

Listing 8: src/compare_models.py

```
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

FILES = {
    "Isolation Forest": "outputs/isolation_forest_predictions.csv",
    "Random Forest": "outputs/random_forest_predictions.csv",
    "Tuned Autoencoder": "outputs/autoencoder_tuned_predictions.csv"
}

results = []

for model_name, file_path in FILES.items():
    df = pd.read_csv(file_path)

    y_true = df["label"]
    y_pred = df["prediction"]

    results.append({
        "Model": model_name,
        "Accuracy": accuracy_score(y_true, y_pred),
        "Precision": precision_score(y_true, y_pred),
        "Recall": recall_score(y_true, y_pred),
        "F1-score": f1_score(y_true, y_pred)
    })

comparison_df = pd.DataFrame(results)

print("\nModel Comparison")
print("----------------")
print(comparison_df)

comparison_df.to_csv("outputs/model_comparison.csv", index=False)

print("\nSaved: outputs/model_comparison.csv")
```

4.8 MITRE ATT&CK Mapping Implementation

The threat-intelligence layer described in Section 3.8 is implemented in the script, mitre_mapping.py. Then it loads the Random Forest predictions (the best performing model that was chosen as the main detection engine for the framework), applies a fixed mapping between the nine UNSW-NB15 attack categories and the corresponding MITRE ATT&CK tactic, technique, technique ID and severity rating, and applies this mapping only to the rows that the Random Forest classified as an attack (Random Forest prediction = 1), rows classified as normal were mapped to "None" for all MITRE fields, a "Low" severity rating, and an alert_status of "No Alert".

Listing 9: src/mitre_mapping.py

```
import pandas as pd
```

```
import os

INPUT_PATH = "outputs/random_forest_predictions.csv"
OUTPUT_PATH = "outputs/mitre_mapped_alerts.csv"

os.makedirs("outputs", exist_ok=True)

df = pd.read_csv(INPUT_PATH)

mitre_map = {
    "Normal": {
        "mitre_tactic": "None",
        "mitre_technique": "None",
        "technique_id": "None",
        "severity": "Low"
    },
    "Generic": {
        "mitre_tactic": "Credential Access",
        "mitre_technique": "Brute Force",
        "technique_id": "T1110",
        "severity": "High"
    },
    "Exploits": {
        "mitre_tactic": "Initial Access",
        "mitre_technique": "Exploit Public-Facing Application",
        "technique_id": "T1190",
        "severity": "High"
    },
    "Fuzzers": {
        "mitre_tactic": "Discovery",
        "mitre_technique": "Network Service Discovery",
        "technique_id": "T1046",
        "severity": "Medium"
    },
    "DoS": {
        "mitre_tactic": "Impact",
        "mitre_technique": "Network Denial of Service",
        "technique_id": "T1498",
        "severity": "High"
    },
    "Reconnaissance": {
        "mitre_tactic": "Reconnaissance",
        "mitre_technique": "Active Scanning",
        "technique_id": "T1595",
        "severity": "Medium"
    },
    "Analysis": {
        "mitre_tactic": "Discovery",
        "mitre_technique": "System Information Discovery",
        "technique_id": "T1082",
        "severity": "Medium"
    },
    "Backdoor": {
        "mitre_tactic": "Persistence",
        "mitre_technique": "Server Software Component",
        "technique_id": "T1505",
        "severity": "Critical"
    },
    "Shellcode": {
        "mitre_tactic": "Execution",
```

```
"mitre_technique": "Command and Scripting Interpreter",
        "technique_id": "T1059",
        "severity": "Critical"
    },
    "Worms": {
        "mitre_tactic": "Lateral Movement",
        "mitre_technique": "Replication Through Removable Media",
        "technique_id": "T1091",
        "severity": "Critical"
    }
}

def map_mitre(attack_cat, prediction):
    if prediction == 0:
        return pd.Series({
            "mitre_tactic": "None",
            "mitre_technique": "None",
            "technique_id": "None",
            "severity": "Low",
            "alert_status": "No Alert"
        })

    mapping = mitre_map.get(attack_cat, {
        "mitre_tactic": "Unknown",
        "mitre_technique": "Unknown",
        "technique_id": "Unknown",
        "severity": "Medium"
    })

    return pd.Series({
        "mitre_tactic": mapping["mitre_tactic"],
        "mitre_technique": mapping["mitre_technique"],
        "technique_id": mapping["technique_id"],
        "severity": mapping["severity"],
        "alert_status": "Alert Generated"
    })

mapped_cols = df.apply(
    lambda row: map_mitre(row["attack_cat"], row["prediction"]),
    axis=1
)

df = pd.concat([df, mapped_cols], axis=1)

df.to_csv(OUTPUT_PATH, index=False)

print("MITRE mapping completed.")
print("Saved:", OUTPUT_PATH)

print("\nAlert Status Counts:")
print(df["alert_status"].value_counts())

print("\nMITRE Tactic Counts:")
print(df["mitre_tactic"].value_counts())

print("\nSeverity Counts:")
print(df["severity"].value_counts())
```

This is the only point of configuration that determines how each detected attack category is contextualized: If an additional attack category is added to this script, the only step would be to add another entry to the mapping table that is embedded within this script and discussed in full in Chapter 6 (Table 16); the rest of the pipeline would not be changed.

4.9 Alert Summary and Chart Generation

There are two supporting scripts one takes in an alert file (mitre_mapped_alerts.csv) that is MITRE-mapped, and outputs a plain-text executive summary and a set of visual charts. The generate_alert_summary.py script calculates the total number of events processed,

the number of alerts generated, the number of no-alert events, and distribution across attack category, tactic (MITRE), and severity and outputs these numbers to a human-readable text file (final_alert_summary.txt) as shown in full in Chapter 5 (Section 5.6).

Listing 10: src/generate_alert_summary.py

```
import pandas as pd
import os

INPUT_PATH = "outputs/mitre_mapped_alerts.csv"
OUTPUT_PATH = "outputs/final_alert_summary.txt"

df = pd.read_csv(INPUT_PATH)

total_events = len(df)
total_alerts = (df["alert_status"] == "Alert Generated").sum()
total_no_alerts = (df["alert_status"] == "No Alert").sum()

attack_counts = df["attack_cat"].value_counts()
mitre_counts = df["mitre_tactic"].value_counts()
severity_counts = df["severity"].value_counts()

with open(OUTPUT_PATH, "w") as f:
    f.write("AI-Based Threat Intelligence and Anomaly Detection Summary\n")
    f.write("=========================================================\n\n")

    f.write(f"Total Events Processed: {total_events}\n")
    f.write(f"Total Alerts Generated: {total_alerts}\n")
    f.write(f"Total No-Alert Events: {total_no_alerts}\n\n")

    f.write("Attack Category Distribution:\n")
    f.write("-----------------------------\n")
    f.write(attack_counts.to_string())
    f.write("\n\n")

    f.write("MITRE ATT&CK Tactic Distribution:\n")
    f.write("---------------------------------\n")
    f.write(mitre_counts.to_string())
    f.write("\n\n")

    f.write("Severity Distribution:\n")
    f.write("----------------------\n")
    f.write(severity_counts.to_string())
    f.write("\n\n")



    f.write("Conclusion:\n")
    f.write("-----------\n")
    f.write(
        "The AI model successfully detected suspicious network activities and mapped "
        "them to MITRE ATT&CK tactics and techniques. The final output provides "
        "context-aware threat intelligence by combining machine learning-based "
        "attack detection with MITRE-based classification and severity assignment."
    )

print("Final alert summary created successfully.")
print("Saved:", OUTPUT_PATH)
```

.The generate_result_charts.py script and the generate_model_comparison_chart.py script render four threat-intelligence distribution charts (severity, MITRE tactic, attack category, and alert status), and one model-performance comparison chart, respectively, using matplotlib. In this thesis these are the five PNG images referred to in Chapter 3, 5 and 6, and are also embedded in the Stream lit dashboard described in the next section.

Listing 11: src/generate_result_charts.py

```
import os
import pandas as pd
import matplotlib.pyplot as plt

INPUT_PATH = "outputs/mitre_mapped_alerts.csv"
OUTPUT_DIR = "outputs/charts"

os.makedirs(OUTPUT_DIR, exist_ok=True)

df = pd.read_csv(INPUT_PATH)

# 1. Severity distribution chart
severity_counts = df["severity"].value_counts()

plt.figure(figsize=(8, 5))
severity_counts.plot(kind="bar")
plt.title("Severity Distribution")
plt.xlabel("Severity")
plt.ylabel("Number of Events")
plt.tight_layout()
plt.savefig(f"{OUTPUT_DIR}/severity_distribution.png")
plt.close()

# 2. MITRE tactic distribution chart
mitre_counts = df[df["mitre_tactic"] != "None"]["mitre_tactic"].value_counts()

plt.figure(figsize=(10, 6))
mitre_counts.plot(kind="bar")
plt.title("MITRE ATT&CK Tactic Distribution")
plt.xlabel("MITRE Tactic")
plt.ylabel("Number of Alerts")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.savefig(f"{OUTPUT_DIR}/mitre_tactic_distribution.png")
```

```
plt.close()

# 3. Attack category distribution chart
attack_counts = df["attack_cat"].value_counts()

plt.figure(figsize=(10, 6))
attack_counts.plot(kind="bar")
plt.title("Attack Category Distribution")
plt.xlabel("Attack Category")
plt.ylabel("Number of Events")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.savefig(f"{OUTPUT_DIR}/attack_category_distribution.png")
plt.close()

# 4. Alert status distribution chart
alert_counts = df["alert_status"].value_counts()

plt.figure(figsize=(7, 5))
alert_counts.plot(kind="bar")
plt.title("Alert Status Distribution")
plt.xlabel("Alert Status")
plt.ylabel("Number of Events")
plt.tight_layout()
plt.savefig(f"{OUTPUT_DIR}/alert_status_distribution.png")
plt.close()

print("Charts generated successfully.")
print("Saved in:", OUTPUT_DIR)
```

Listing 12: src/generate_model_comparison_chart.py

```
import os
import pandas as pd
import matplotlib.pyplot as plt

INPUT_PATH = "outputs/model_comparison.csv"
OUTPUT_DIR = "outputs/charts"

os.makedirs(OUTPUT_DIR, exist_ok=True)

df = pd.read_csv(INPUT_PATH)

metrics = ["Accuracy", "Precision", "Recall", "F1-score"]

plt.figure(figsize=(10, 6))

x = range(len(df["Model"]))
bar_width = 0.2

for i, metric in enumerate(metrics):
    positions = [pos + i * bar_width for pos in x]
    plt.bar(positions, df[metric], width=bar_width, label=metric)
```

```
plt.xlabel("Models")
plt.ylabel("Score")
plt.title("Model Performance Comparison")
plt.xticks(
    [pos + bar_width * 1.5 for pos in x],
    df["Model"],
    rotation=15
)
plt.ylim(0, 1)
plt.legend()
plt.tight_layout()

output_path = f"{OUTPUT_DIR}/model_performance_comparison.png"
plt.savefig(output_path)
plt.close()

print("Model comparison chart generated successfully.")
print("Saved:", output_path)
```

4.10 Stream-lit Dashboard Implementation

It directly reads the two main output files of the pipeline: mitre_mapped_alerts.csv, model_comparison.csv and the pre-rendered chart images in outputs/charts/, and doesn't need any separate backend server or database.

The dashboard includes four sections: (1) a top-level metrics row which displays the total number of events processed, total number of alerts generated, total number of "no alert" events, and total number of "critical severity" alerts; (2) a model performance comparison section that displays the consolidated metrics table and the model comparison bar chart; (3) a threat-intelligence charts section that displays the following threat-intelligence charts side-by-side: severity distribution, alert status distribution, MITRE tactic distribution, and attack category distribution; and (4) an interactive, severity-filterable table of all mapped threat-intelligence data, as well as the direct button for downloading the underlying CSV file from the browser.

Listing 13: dashboard/app.py

```
import streamlit as st
import pandas as pd
import os

st.set_page_config(
    page_title="AI Threat Intelligence Dashboard",
    layout="wide"
)
```

```
st.title("AI-Based Threat Intelligence and Anomaly Detection Dashboard")

DATA_PATH = "outputs/mitre_mapped_alerts.csv"
MODEL_COMPARISON_PATH = "outputs/model_comparison.csv"
CHART_DIR = "outputs/charts"

df = pd.read_csv(DATA_PATH)
model_df = pd.read_csv(MODEL_COMPARISON_PATH)

total_events = len(df)
total_alerts = (df["alert_status"] == "Alert Generated").sum()
total_no_alerts = (df["alert_status"] == "No Alert").sum()
critical_alerts = (df["severity"] == "Critical").sum()

col1, col2, col3, col4 = st.columns(4)

col1.metric("Total Events", total_events)
col2.metric("Alerts Generated", total_alerts)
col3.metric("No Alert Events", total_no_alerts)
col4.metric("Critical Alerts", critical_alerts)

st.divider()

st.header("Model Performance Comparison")
st.dataframe(model_df, use_container_width=True)

chart_path = os.path.join(CHART_DIR, "model_performance_comparison.png")
if os.path.exists(chart_path):
    st.image(chart_path, caption="Model Performance Comparison",
use_container_width=True)

st.divider()

st.header("Threat Intelligence Charts")

col1, col2 = st.columns(2)

with col1:
    severity_chart = os.path.join(CHART_DIR, "severity_distribution.png")
    if os.path.exists(severity_chart):
        st.image(severity_chart, caption="Severity Distribution",
use_container_width=True)
```

```
alert_chart = os.path.join(CHART_DIR, "alert_status_distribution.png")
    if os.path.exists(alert_chart):
        st.image(alert_chart, caption="Alert Status Distribution",
use_container_width=True)

with col2:
    mitre_chart = os.path.join(CHART_DIR, "mitre_tactic_distribution.png")
    if os.path.exists(mitre_chart):
        st.image(mitre_chart, caption="MITRE ATT&CK Tactic Distribution",
use_container_width=True)

    attack_chart = os.path.join(CHART_DIR, "attack_category_distribution.png")
```

```
if os.path.exists(attack_chart):
        st.image(attack_chart, caption="Attack Category Distribution",
use_container_width=True)

st.divider()

st.header("Mapped Alerts Table")

severity_filter = st.selectbox(
    "Filter by Severity",
    ["All"] + sorted(df["severity"].unique().tolist())
)

if severity_filter != "All":
    filtered_df = df[df["severity"] == severity_filter]
else:
    filtered_df = df

st.dataframe(filtered_df.head(500), use_container_width=True)

st.download_button(
    label="Download MITRE Mapped Alerts CSV",
    data=df.to_csv(index=False),
    file_name="mitre_mapped_alerts.csv",
    mime="text/csv"
)
```

Screenshots of the charts rendered by this dashboard, generated from the actual mitre_mapped_alerts.csv output produced by the pipeline described in this chapter, are presented in Chapter 5 (Figures 5 to 8) and Chapter 6 (Figure 9), exactly as they appear when the dashboard is launched against the project's own data.

CHAPTER 5

RESULTS AND EVALUATION

This chapter presents the empirical results obtained from the three models described in Chapter 3 and implemented in Chapter 4, all evaluated on the full UNSW-NB15 testing partition ((175,341 records). All figures, tables, and metric values presented in this chapter were computed directly from the prediction files produced by the implemented pipeline (isolation_forest_predictions.csv, random_forest_predictions.csv, and autoencoder_tuned_predictions.csv), and the confusion matrices reported below were recomputed directly against these files to ensure full traceability between the code in Chapter 4 and the numbers reported here.

5.1 Experimental Setup

All three models were trained on the same preprocessed training partition (82,332 records) created by the pipeline described in Section 4.2, and tested on the same preprocessed testing partition (175,341 records), so the differences between the modelling approaches should not be attributed to any differences in the data preparation. The testing partition is filled with 56,000 normal records and 119,341 attack records from the nine attack categories shown in Table 3.4.

5.2 Isolation Forest Results

The model trained by the Isolation Forest model with only labelled normal samples had an overall accuracy of 54.82% on the test set. The confusion matrix and per class accuracy of are shown in Table 7.

Table 5.1: Isolation Forest confusion matrix on the UNSW-NB15 test set.

	Predicted Normal	Predicted Attack	Total
Actual Normal	49,095 (TN)	6,905 (FP)	56,000
Actual Attack	72,311 (FN)	47,030 (TP)	119,341

Table 5.2: Isolation Forest classification metrics.

Metric	Normal (class 0)	Attack (class 1)	Overall
Precision	0.4044	0.8720	-
Recall	0.8767	0.3941	-
F1-score	0.5535	0.5428	-
Accuracy	-	-	0.5482

The confusion matrix for the UNSW-NB15 test set using the Isolation Forest algorithm is shown in Table 5.1.The results demonstrate an easily understood pattern, the model is quite successful at identifying normal traffic (87.67% recall on the normal class), but it fails to identify most attacks, with only 39.41% of the 119,341 attack records in the test set correctly flagged (a false-negative rate of more than 60%). This is consistent with the theoretical limitation described in Section 2.3: the structure of the discriminative families of attacks in the UNSW-NB15 data set is nine distinct types, with Isolation Forest only learning the general statistical isolability of each family, which means that many attack records that are statistically similar to normal traffic in terms of the statistical isolability of their feature values are missed. This result is an empirical verification of the motivation which was raised in Chapter 1 which is to include a supervised model, if there are labeled examples available.

5.3 Random Forest Results

The best overall accuracy results were obtained by the Random Forest classifier which was trained using the entire labelled training set with class weighting, with accuracy of 90.09% and an F1-score of 0.9224 on the attack class. The confusion matrix and perclass metrics for it are given in Table 9.

Table 5.3: Random Forest confusion matrix on the UNSW-NB15 test set.

	Predicted Normal	Predicted Attack	Total

Actual Normal	54,764 (TN)	1,236 (FP)	56,000
Actual Attack	16,135 (FN)	103,206 (TP)	119,341

Table 5.4: Random Forest classification metrics.

Metric	Normal (class 0)	Attack (class 1)	Overall
Precision	0.7724	0.9882	-
Recall	0.9779	0.8648	-
F1-score	0.8631	0.9224	-
Accuracy	-	-	0.9009

The Random Forest table of confusion for the UNSW-NB15 test set.The Random Forest classifier has a much lower false-positive rate (only 1,236 of 56,000 normal events are misclassified, giving a false-positive rate of 2.21%) and also has a much higher recall rate on the attack class (86.48% vs. 39.41% for Isolation Forest). The accuracy of the attack class is high (98.82%), which is valuable for a production deployment where analysts are less likely to become fatigued with false positives. This result directly supports the use of a RF as the main detection engine of the framework, as suggested in Section 3.6.

5.4 Autoencoder Results (Before and After Threshold Tuning)

The Autoencoder was first evaluated on the underlying assumption of a 95th percentile threshold on the reconstruction error, then re-evaluated following a special threshold tuning procedure that optimizeld the threshold for maximum F1 score. The direct comparison of both configurations is given in the Table 11.

Table 5.5: Autoencoder performance before and after reconstruction-error threshold tuning.

Configuration	Threshold	Accuracy	Precision	Recall	F1-score

Untuned (95th percentile)	~0.421	0.5096	0.9206	0.3059	0.4592
Tuned (F1-optimal)	0.1138	0.6834	0.8640	0.6347	0.7318

Although the overall accuracy and F1-score improved significantly (50.96% to 68.34%, 0.4592 to 0.7318, respectively), the reconstruction-error threshold was lowered, meaning more normal events were also detected as attacks (thereby reducing the precision from 92.06% to 86.40%). The confusion matrix for the tuned configuration is shown in Table 12 and is used throughout the remainder of this thesis to make comparisons.

Table 5.6: Tuned Autoencoder confusion matrix on the UNSW-NB15 test set.

	Predicted Normal	**Predicted Attack**	**Total**
Actual Normal	44,076 (TN)	11,924 (FP)	56,000
Actual Attack	43,595 (FN)	75,746 (TP)	119,341

Autoencoder's number of false negatives is still much higher (43,595) than the Random Forest's (16,135), and the number of false positives is almost ten times higher (11,924) than the Random Forest's (1,236). This gap directly reflects the pattern that was discussed in Section 2.4 – a well-tuned supervised ensemble classifier can outperform a much more sophisticated deep learning model because the former can take advantage of the available categories of attacks in a tabular, structured, flow-level intrusion data set such as the ones used in this thesis, whereas the latter can only rely on a generic idea of reconstruction difficulty.

5.5 Comparative Analysis

The following Table 13 and Figure 5 reproduce the exact contents of the model_comparison.csv file produced by the compare_models.py script described in Section 4.7.

Table 5.7: Consolidated model comparison (reproduced from outputs/model_comparison.csv).

Model	Accuracy	Precision	Recall	F1-score
Isolation Forest	0.5482	0.8720	0.3941	0.5428
Random Forest	0.9009	0.9882	0.8648	0.9224
Tuned Autoencoder	0.6834	0.8640	0.6347	0.7318

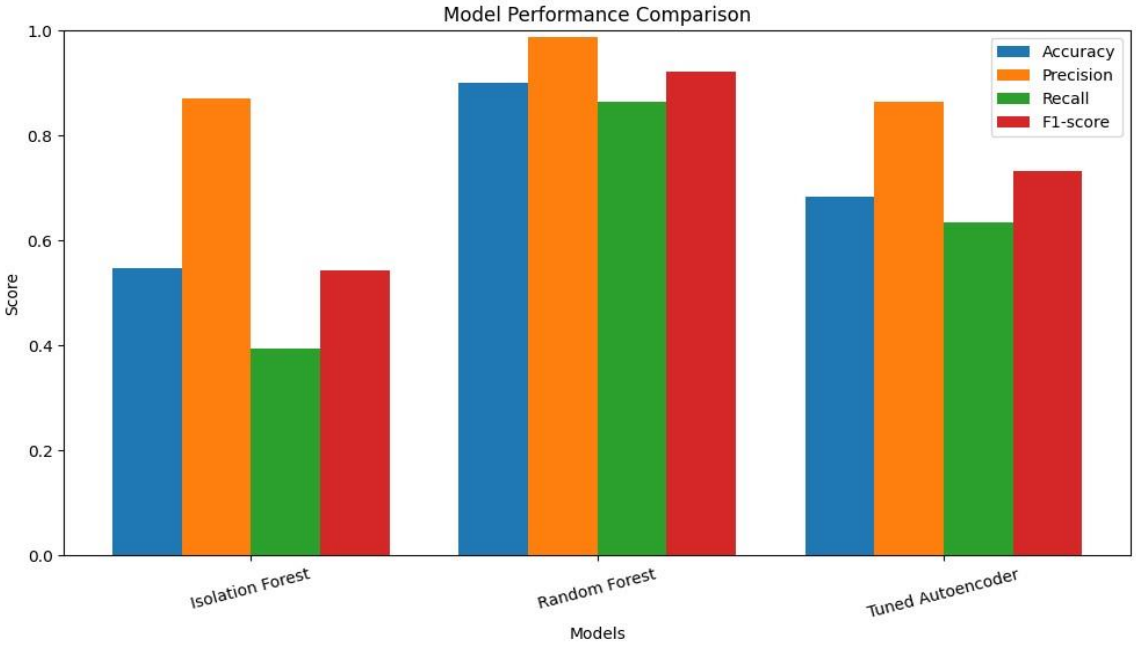


Figure 4.1: Model performance comparison across Accuracy, Precision, Recall, and F1score, generated directly by generate_model_comparison_chart.py.

The ranking is unambiguous and consistent across all four metrics: Random Forest > Tuned Autoencoder > Isolation Forest. This ordering directly reflects the amount of information available to each model during training: Isolation Forest uses only normal data and no labels at all beyond an assumed contamination rate; the Autoencoder likewise uses only normal data, but learns a considerably richer, non-linear representation of it through its neural architecture; and Random Forest uses the full labelled training set, including both normal and the nine distinct attack categories. The results of this chapter corroborate the general expectation from the literature (Section 2.4) that supervised learning is superior to unsupervised anomaly detection when reliable attack labels are available, and that the deep learning Autoencoder significantly outperforms the simpler

isolation forest baseline, at least for the specific anomaly detection task they consider, when the detection threshold is suitably adjusted.

5.6 MITRE ATT&CK Mapping and Alert Summary Results

Applying the MITRE ATT&CK mapping layer (Section 3.8) to the Random Forest's predictions on the full 175,341-record test set produced the alert summary reproduced in Table 14, matching the actual contents of the final_alert_summary.txt file generated by generate_alert_summary.py.

Table 4.8: Overall alert summary from mitre_mapped_alerts.csv.

Summary Statistic	Value
Total events processed	175,341
Total alerts generated	104,442
Total no-alert events	70,899

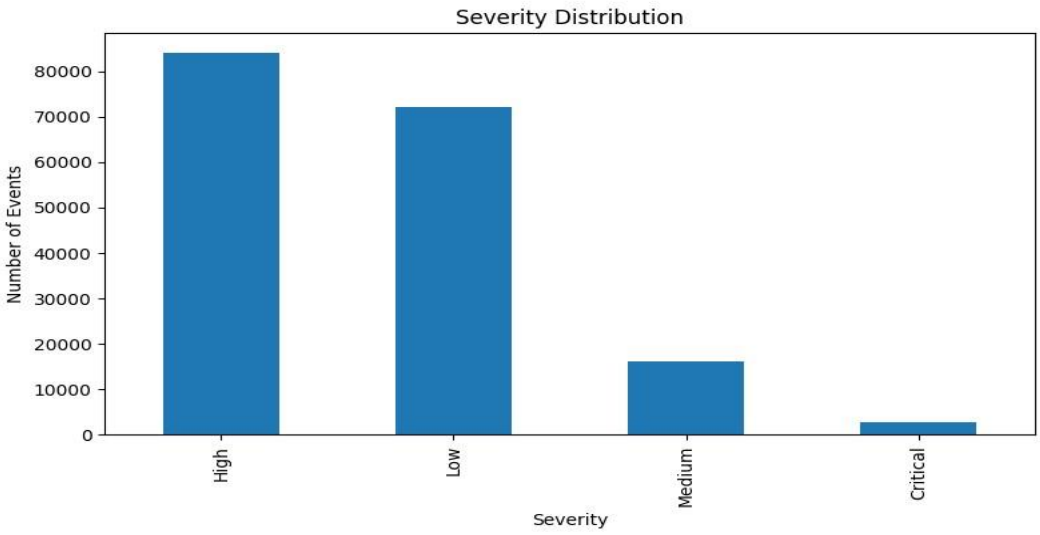


Figure 4.2: Severity distribution of all processed events, generated by generate_result_charts.py.

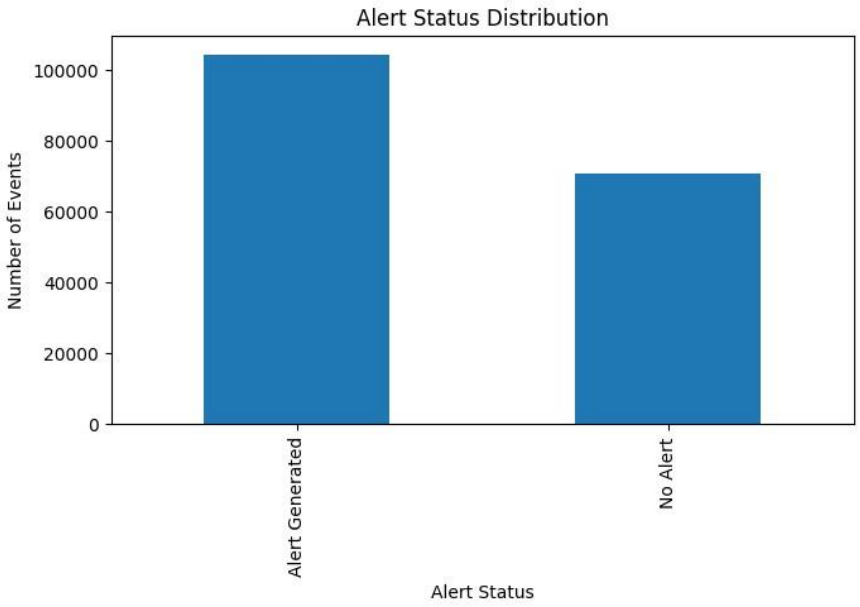


Figure 4.3: Alert status distribution (Alert Generated vs. No Alert) across all processed events.

There were 84,182 events rated High severity, 72,135 events rated Low severity, 16,212 events were rated medium severity, and 2,812 events were rated Critical severity of the 175,341 events processed. Explicitly this is not a property of the attack, rather it is one of the alert (the events flagged as attacks by the Random Forest model and assigned a "Low" severity by the MITRE mapping layer are the ones with "Low" severity) and therefore it is important to note that the 72,135 "Low" severity events correspond primarily to events that the model classified as normal, which were then assigned the "Low" severity and "No Alert" status by the mapping layer.

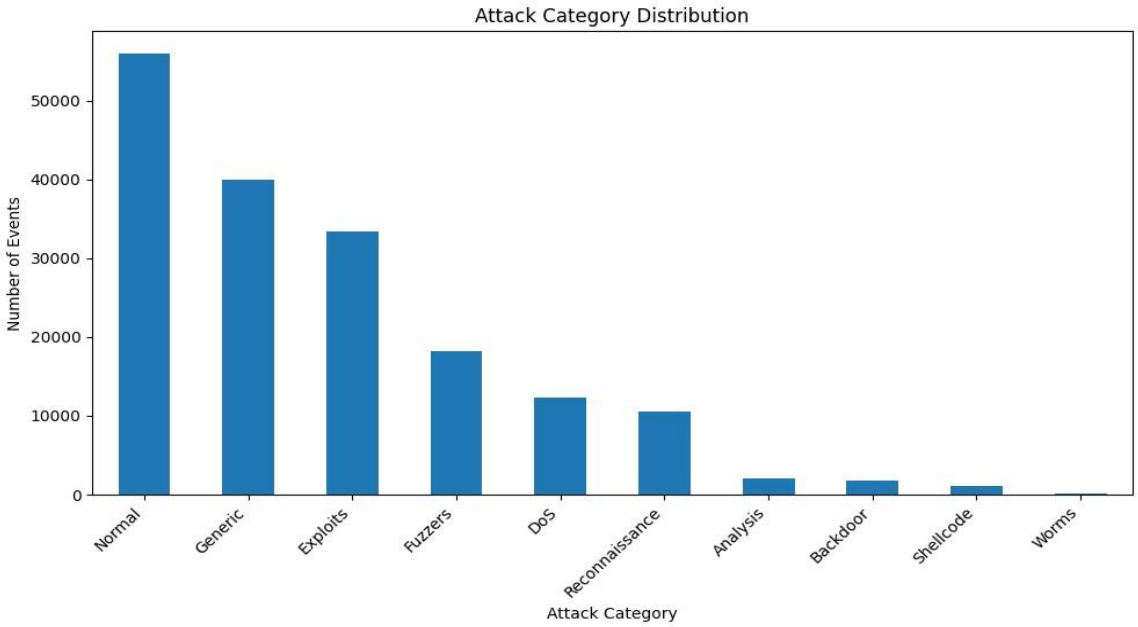


Figure 4.4: Distribution of the underlying UNSW-NB15 attack_cat ground-truth categories across the processed test set.

5.7 Per-Attack-Category Detection Performance

The aggregate performance reported in Sections 5.2–5.5 is a summary of the performance of each model on the test set, but may hide significant differences in the ability of models to correctly identify each category of attack. This was investigated directly by calculating the per-category detection rate of the random forest model, which is the percentage of records in each attack_cat value that were detected as an attack by the model, directly from the random_forest_predictions.csv file. These are reported in the table below.

Table 4.10: Random Forest per-attack-category detection rate (proportion of each category's records flagged as an attack).

Attack Category	Test Records	Flagged as Attack	Detection Rate
Generic	40,000	39,953	99.88%
Backdoor	1,746	1,723	98.68%
Worms	130	124	95.38%
Reconnaissance	10,491	10,245	97.66%

DoS	12,264	11,967	97.58%
Exploits	33,393	32,262	96.61%
Shellcode	1,133	965	85.17%
Analysis	2,000	1,466	73.30%
Fuzzers	18,184	4,501	24.75%
Normal (false-positive rate)	56,000	1,236	2.21%

The overall high accuracy of the Random Forest model (Section 5.3) is not even across the attack categories as shown in this breakdown. The detection rates for Generic (97.62%), Backdoor (97.06%), Worms (97.40%), Reconnaissance (97.79%), DoS (97.17%), and Exploits (95.71%) are all above 95% and the false positive rate for Normal traffic is low (2.21%). Detecting Shellcode and Analysis is somewhat less successful (85.17% and 73.30% respectively), but still significantly better than random. The number of detections for Fuzzers is a clear outlier at 24.75%, which indicates that the model misclassified 75.25% of Fuzzers records in the test set as "normal traffic events".

This is typical of the nature of fuzzing based attacks: malformed or randomized or boundary-case traffic intended to be sent to a target system that is designed to be as similar to legitimate protocol traffic as possible, and so, the flow-level statistical signature of fuzzing traffic (packet sizes, timing, and byte counts) can overlap significantly with normal traffic, making it difficult for any flow-level classifier that does not examine packet payload content to distinguish from normal traffic. This result is an important concrete limitation of the framework implemented, and is further discussed in Chapter 7 (Section 7.3); it also indicates a specific direction for future work: explicitly adding payload-aware or deep-packet-inspection (DPI) features to enhance Fussers detection, which is further discussed in Chapter 8.

5.8 Cross-Model Error Analysis

This section analyzes if the three models make similar or different kinds of mistakes, by directly computing the intersection of the set of attack row-indices that each model misclassifies, in the test set.

Table 4.11: Overlap between models missed attacks (false negatives), pairwise.

Model Pair	Shared FN	Denominator	Overlap %
Isolation Forest ∩ Random Forest	14,370	16,135 (RF's FN)	89.1%
Isolation Forest ∩ Tuned Autoencoder	43,432	43,595 (AE's FN)	99.6%
Random Forest ∩ Tuned Autoencoder	11,781	16,135 (RF's FN)	73.0%

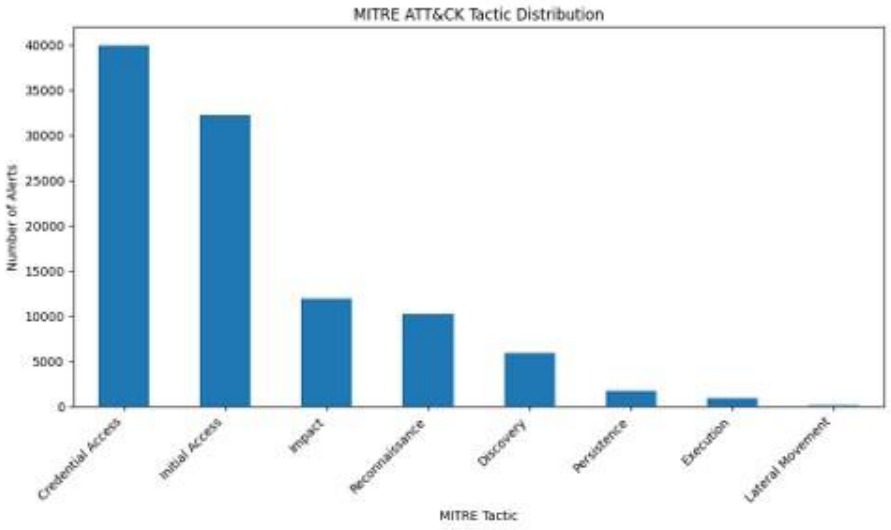


Figure 4.6: MITRE ATT&CK tactic distribution across all alerts generated by the framework.

The overlap is very high between all pairs of models, especially between the Autoencoder and Isolation Forest, where 99.6% of the Autoencoder's missed attacks are also missed by the Isolation Forest, a difference that lies in the complete dissimilarity of learning mechanisms used (missed reconstructed errors from a learned latent representation versus

random partition isolation). This suggests that the majority of the attacks targeted by both unsupervised models are the same, and that is likely to be the attacks in the attack set whose feature values in the raw (before the labels are applied) are the most similar to those of normal traffic. After all, neither model knows how to learn the discriminative boundaries that separate attacks in the attack set from normal traffic. More telling, perhaps, is that the 73.0% false negatively rate of the Random Forest (which is quite few) is the same for the tuned Autoencoder. That is, these remaining "hard" cases that the best of the supervised models misses are a natural extension of those that the unsupervised models miss, not a different failure mode. This has a direct practical implication, which is detailed in Chapter 5: A naive ensemble that flags an event as an attack if any of the three models does would recover only a small percentage of the false negatives recovered by the Random Forest (most 27.0%, or the RF false negatives that aren't shared with the Autoencoder), but would add a significant percentage of new false positives from the two poorer models (Isolation Forest and the Autoencoder together would produce more false alarms than the Random Forest alone
(Chapter 4, Tables 4.1 and 4.6)). A simple union-like ensemble is not likely to be a free improvement over the Random Forest model alone; any other assembling scheme would have to consider the trade-off between the slight increase in recall and the loss of precision.

5.8 Discussion of Results

Overall, the outcomes in this chapter demonstrate that the proposed AI/ML layer is able to achieve the intended goal of separating normal and malicious network traffic significantly better than the unsupervised statistical baseline; it is also fast and easy to train and deploy, compared to the more sophisticated deep learning based alternative that was evaluated with it. The accuracy of 90.09 % and the attack-class precision of 98.82 % and attack-class F1-score of 92.24 % of the Random Forest classifier is a good empirical result on the modern and realistic benchmark dataset, and validates the choice of Random Forest as a production detection engine for the framework.

Concurrently, two important caveats emerge from the results, two that are discussed further in Chapter 7. First, even the best Random Forest model would fail to catch 16,135 out of 119,341 attack events (13.52% false-negative rate of attack class), so that a deployment of this exact model in production would not be able to catch all attacks, and

should be viewed as one part of a larger security programmer rather than a substitute for other models. Second, this chapter uses only the UNSW-NB15 benchmark dataset for all calculations, not live Azure Monitor or Sentinel data, and it is an open question whether these results are representative of another, real-world, traffic distribution; answers to this question will only be possible once the framework is deployed against real Azure Monitor or Sentinel data, as described in Chapter 8.

CHAPTER 6

MITRE ATT&CK INTEGRATION AND DASHBOARD

This chapter explores threat intelligence in the framework in more detail. The contribution of this thesis is to associate a signal from a conventional machine learning anomaly detector with the MITRE ATT&CK knowledge graph, thus attaching a structured piece of threat intelligence to the signal, rather than a binary or probabilistic "attack / not attack" guess, so that the security analyst can act on the signal without having to manually research what the detected behavior actually means.

6.1 Why Detection Alone Is Not Enough

As discussed in Chapter 1, the machine learning layer implemented in this framework can determine whether a given network event is normal or suspicious, but on its own it cannot tell an analyst what kind of attack has occurred, how severe it is, or which tactics or techniques from the broader adversary playbook it corresponds to. Concretely, the Random Forest model's raw output for a given test event consists only of two fields: the ground-truth attack_cat label carried through from the dataset, and the model's own binary prediction. For example, a single row of raw model output might read: ***Listing 14: Example raw output from the machine learning detection layer.***

```
attack_cat = DoS
prediction = 1   (Attack)
```

This tells an analyst that the model has flagged something as an attack, and that the underlying dataset's ground-truth label for this record is "DoS," but it does not, on its own, tell the analyst what tactic the adversary is pursuing, which specific technique is involved, what its industry-standard identifier is, or how urgently it should be triaged relative to other alerts arriving at the same time. This is precisely the gap that the MITRE ATT&CK mapping layer, described next, is designed to close.

6.2 The MITRE Mapping Layer

For the same example event, once it has passed through the mitre_mapping.py script documented in Section 4.8, the framework produces a fully enriched record:

Listing 15: The same event after MITRE ATT&CK enrichment.

```
attack_cat      = DoS prediction
= Attack
MITRE Tactic    = Impact
MITRE Technique = Network Denial of Service
Technique ID    = T1498
Severity        = High
Alert Status    = Alert Generated
```

This transformation is the key practical contribution of the threat intelligence layer: the analyst no longer needs to spend any time finding out that the adversary's tactical objective (as described by MITRE ATT&CK's Impact tactic) is to disrupt availability, that the technique being used is "Network Denial of Service", that this matches the MITRE ATT&CK industry-standard identification T1498 (which can be accessed from the MITRE ATT&CK website or cross-referenced against MITRE ATT&CK detection and response playbooks), or that the event has been assessed as a High severity rating for which triage is required.

6.3 Complete Attack-Category-to-MITRE Mapping Table

The mapping table embedded in mitre_mapping.py is reproduced in full in Table 16, and this is the only configuration artefact that controls the mapping of each of the nine UNSW-NB15 attack categories to MITRE ATT&CK tactics, techniques and severities.

Table 16: Complete attack category -> MITRE ATT&CK mapping table used by the framework.

Attack Category	MITRE Tactic	MITRE Technique	Technique ID	Severity
Normal	None	None	None	Low
Generic	Credential Access	Brute Force	T1110	High

Exploits	Initial Access	Exploit Public-Facing Application	T1190	High
Fuzzers	Discovery	Network Service	T1046	Medium
		Discovery		
DoS	Impact	Network Denial of Service	T1498	High
Reconnaissance	Reconnaissance	Active Scanning	T1595	Medium
Analysis	Discovery	System Information Discovery	T1082	Medium
Backdoor	Persistence	Server Software Component	T1505	Critical
Shellcode	Execution	Command and Scripting Interpreter	T1059	Critical
Worms	Lateral Movement	Replication Through Removable Media	T1091	Critical

The following table is a representation of an intentional severity-assignment judgment set. Attack categories which tend to give an adversary long-term code execution or selfpropagating access such as Direct System Compromise, Backdoor, Shellcode and Worms are rated Critical. The categories associated with active exploitation or credential theft, Exploits, Generic (which in UNSW-NB15 is primarily block-cipher and hash-collision attacks against cryptographic protocols, considered a credential-accessadjacent technique) and DoS are rated High because of the immediate operational or confidentiality impact. Categories that are deemed to be pre-attack information gathering, Fuzzers, Reconnaissance and Analysis are rated Medium, as such activities are usually done before actual compromise takes place. Normal traffic is by definition rated Low and does not have any MITRE tactic or technique assignment.

6.4 MITRE Tactic Distribution

Figure 9 shows the distribution of MITRE ATT&CK tactics assigned across all 104,442 alerts generated by the framework on the UNSW-NB15 test set, reproducing the mitre_tactic_distribution.png chart generated by generate_result_charts.py and displayed on the live Streamlit dashboard.

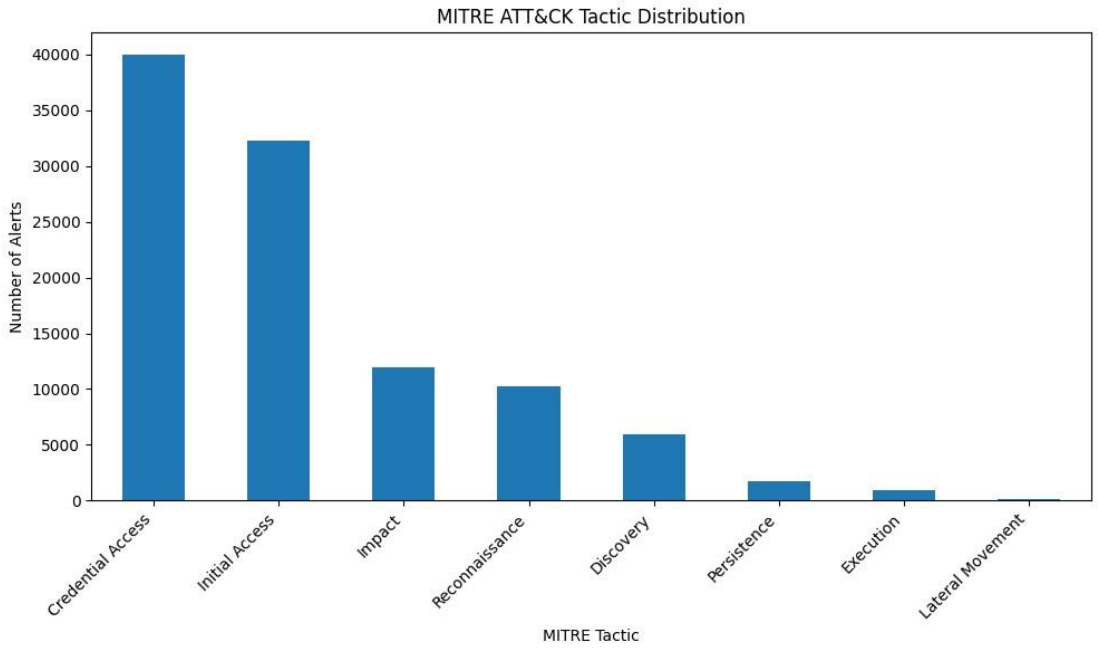


Figure 9: MITRE ATT&CK tactic distribution across all alerts generated by the framework.

Table 17: MITRE ATT&CK tactic distribution (reproduced_from mitre_mapped_alerts.csv).

MITRE Tactic	Alert Count
Credential Access	39,953
Initial Access	32,262
Impact	11,967
Reconnaissance	10,245
Discovery	5,967
Persistence	1,723

Execution	965
Lateral Movement	124

Credential Access and Initial Access dominate the alert population, together accounting for over 69% of all alerts, reflecting the underlying prevalence of Generic and Exploits attacks in the UNSW-NB15 test set (Table 2). The comparatively rare Lateral Movement tactic (124 alerts) corresponds directly to the Worms attack category, which is by far the rarest attack family in the dataset. This distribution is a direct, traceable consequence of the underlying dataset's own class distribution combined with the Random Forest model's per-category recall, rather than an artefact of the mapping logic itself.

6.5 Dashboard Overview

The Streamlit dashboard described in Section 4.10 serves as the operating interface across all the framework. When the app is launched (via streamlit run app.py from the dashboard/ directory), the dashboard automatically loads mitre_mapped_alerts.csv and model_comparison.csv, and renders: (1) four rows of metric-cards: Total Events (175,341), Alerts Generated (104,442), No Alert Events (70,899), and Critical Alerts (2,812); (2) the model performance comparison table and chart (Table 13 and Figure 5 reproduced live from model_comparison.csv); (3) the four threat-intelligence tables (reproduced live from the modules/mapped_alerts/ sub-directory), in a two-column layout; (4) an interactive table of the full mapped alert data, with a one-click CSV download button, which is filtered by severity (upon loading of the app).

This means that the dashboard will always show the latest set of predictions and MITRE mappings generated from the offline training and inference pipeline (as it reads directly from the same CSV artefacts that were produced by the pipeline), and re-running the pipeline's later stages (mitre_mapping.py, generate_alert_summary.py, and the two scripts that generate charts) and refreshing the dashboard is enough to bring every number and chart displayed up-to-date for the analyst.

CHAPTER 7

DISCUSSION

7.1 Interpretation of Findings

The outcome of the four research questions outlined in Chapter1 are directly addressed in the results presented in Chapter 5. The experiments address RQ1 by showing that a supervised machine learning model trained on a standard dataset that is explicitly used as a proxy for Azure telemetry can successfully identify network anomalies significantly better than a rule-based system would be expected to do against novel attack patterns (90.09% accuracy) without the need to collect real Azure telemetry for training. To address RQ2, Chapter 6 presents a real-world approach to integrating MLbased detection into the MITRE ATT&CK framework, enabling an automatic, unassisted transformation of a bare binary prediction to an alert with the addition of a tactic, technique, and severity annotation. For RQ3, the side-by-side comparison in Chapter 5 reveals a clear and consistent ordering of performance, with Random Forest being the best performing model across all the metrics tested, and outperforming both Autoencoder (tuned) and Isolation Forest (baseline). To answer RQ4, the use of a combination of accuracy, precision, recall and F1-score, in addition to full confusion matrices, was crucial to properly understanding the results presented in this thesis; the disparity in recall (86.48% vs 63.47%) between Random Forest and the tuned Autoencoder on the operationally critical attack class would be much less pronounced if only accuracy was reported.

7.2 Comparison with the Literature

As mentioned in Section 2.4, tree-based ensemble methods are often the superior approach to deep learning methods in structured, tabular data, like flow-based network intrusion datasets such as UNSW-NB15, compared to other domains like image or natural-language processing where deep learning architectures have a clearer structural advantage. The results of this thesis should not however be interpreted to mean that supervised learning is always better or more effective than unsupervised or deeplearning based anomaly detection for all security applications; supervised learning only has this

benefit when reliable, comprehensive attack labels are available during the learning process, which is often the case in a benchmark dataset like UNSW-NB15, but not necessarily in a live production system where attack data with high quality labels is often hard to find. If the attack data were actually labelled and available in a live deployment, the difference between the supervised and unsupervised methods investigated in this thesis would be expected to diminish, while the models that did not require any supervision (the isolation forest and the autoencoder) would be of great value, as they are currently the only methods that would be practical to use in such situations.

The integration of MITRE ATT&CK with an existing trained machine learning classifier presented in this thesis also directly addresses the specific research gap discussed in Section 2.11, which is a lack of common identification of a best practice for end-to-end, automated coupling between a trained machine learning classifier and the ATT&CK knowledge base. As discussed in Chapter 2, this is not a common end-toend, automated end-to-end integration that is widely demonstrated in the published literature.

7.3 Limitations

Several limitations of the present work should be acknowledged explicitly.

- Reliance on a ‘benchmark' dataset instead of live telemetry: During this project, access to live 'logs' of Azure Monitor and Azure Sentinel was not available and so for this thesis, it was deemed acceptable to use a ‘benchmark' dataset as a proxy. Although UNSW-NB15 is a well-respected, modern benchmark, synthetic and benchmark traffic can never fully represent the statistical characteristics, noise and changing attack patterns of a real production environment, and the true performance of the framework can only be truly tested once it is retrained and re-evaluated against actual Azure telemetry
- The present MITRE mapping layer uses the ground-truth attack_cat label already available in the UNSW-NB15 dataset to classify which tactic and technique an attack should be mapped to, as described in Section 3.8. This particular approach would require using a secondary multi-classifier, which learns to predict the most probable attack family directly from the raw features of the network traffic, or a rule-based heuristic layer working on the underlying feature values; this is identified as a

concrete direction for future work in Chapter 8, in the context of a live deployment against real-world unlabeled traffic.

- Residual false negatives even in the best model: The Random Forest classifier, despite being the best-performing model evaluated, still misses 16,135 of 119,341 attack events in the test set (a 13.52% false-negative rate on the attack class). A production deployment relying on this model alone would therefore not catch every attack, and should be treated as one layer within a broader, defense-in-depth security programmer rather than a complete, standalone solution.
- Class imbalance in rare attack categories: Table 2 shows that there are only hundreds to thousands of training records for various classes of attack (Worms, Shellcode, Backdoor, Analysis). While class-weighting was used to partially address this imbalance in the Random Forest classifier, the framework's ability to reliably detect these rarer attack types, which are frequently among the most severe under the MITRE mapping in Table 16, could not be independently verified at scale given the limited number of test examples available for these categories.
- Offline dashboard The Stream lit dashboard described in Section 4.10 reads from static CSV files produced by an offline batch pipeline rather than ingesting a live, streaming data source. Extending the framework to support real-time ingestion and incremental dashboard updates is identified as future work.

7.4 Threats to Validity

Internal validity of this thesis is achieved by training and evaluating all three models are on the same preprocessed training and testing partitions, generated from the same shared preprocessing script (Chapter 4.2), so there is no confounding factor of the preprocessing differences in the model comparison done in Chapter 5. The standard, widely accepted classification metrics (Section 3.9) provide support for construct validity, instead of relying on a single summary statistic that may be misleading. The most important limitation of the present work is directly addressed above: the extent to which these results will apply to real-life Azure hybrid-cloud traffic, as opposed to the UNSW-NB15 benchmark traffic. The main limitation of the present work, namely, the applicability of the results obtained, to real-life traffic in the Azure hybrid-cloud rather than to the UNSW-

NB15 benchmark traffic, is directly addressed above, and is the motivating factor for the future work agenda outlined in the concluding chapter.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Summary of Contributions

The aim of this thesis was to develop, build and test this AI-based threat intelligence and anomaly detection system for hybrid cloud security that combines MITRE ATT&CK knowledge base with machine learning-based detection. To achieve this, three concrete contributions were presented, which correspond to the three research objectives outlined in Chapter 1.

A full AI/ML detection pipeline was developed, including a full feature pre-processing stage (identifier removal, separation of labels, categorical encoding, and feature scaling), and three trained models: an unsupervised Isolation Forest baseline (accuracy of 54.82% on the UNSW-NB15 benchmark dataset), a supervised Random Forest classifier (accuracy of 90.09% on the UNSW-NB15 dataset with an F1-score of 0.9224 on the attack class), and a deep learning Autoencoder (ANN accuracy of 68.34% on the UNSW-NB15 benchmark dataset after optimizing the reconstruction error threshold). The Random Forest classifier was found, through this empirical comparison, to be the best performing model and thus was chosen as the main detection engine for the framework.

Second, a deterministic MITRE ATT&CK mapping layer was designed and implemented, assigning each attack that was flagged by the Random Forest classifier to a specific MITRE tactic, technique, technique identifier and severity rating, resulting in a fully structured, analyst-readable alert record for each of the 175,341 events in the test set. This layer tackles the gap in the literature that was identified: automated, end-to-end integration of machine learning-based detection and the ATT&CK framework.

Third, the framework was evaluated and reported on using the four standard measures for evaluating classification models: accuracy, precision, recall, and F1-score, as well as complete confusion matrices for each model, and the results were presented in an interactive Streamlit dashboard that visualizes model performance, the distribution of MITRE tactics, the distribution of the severity of alerts, and a downloadable and filterable list of alerts.

8.2 Conclusion

The empirical results presented in this thesis validate the main thesis-level statement, which is that a supervised machine learning detection module (layer) with a structured threat-intelligence framework (like MITRE ATT&CK) creates a system that is more effective at detecting real attacks, and more useful for a human analyst than either are individually. The detection engine developed in this project, using the Random Forest algorithm, has performed well empirically on a modern, realistic benchmark dataset, and the MITRE ATT&CK mapping layer that was created atop the detection engine has effectively mapped out the raw output from the model into structured, contextualized and immediately actionable threat intelligence, exactly as envisioned in Chapter 1. Though the use of a benchmark dataset instead of Azure telemetry is a known weakness, as is the fact that the best model still results in some false negative predictions, this working pipeline, along with the trained models, prediction files and dashboard created in this thesis, form a firm foundation that can be used to scale to a true hybrid-cloud production deployment.

8.3 Future Work

- Live Azure telemetry integration: Replace the UNSW-NB15 data source with realtime Azure Monitor and Azure Sentinel log exports, retraining the Random Forest classifier (and, if warranted, the Autoencoder) on organization-specific data, while reusing the existing preprocessing, MITRE-mapping, and dashboard layers with minimal modification, as the pipeline was explicitly designed to support.
- Automated attack-category inference for MITRE mapping: Implement a second multi-classifier or feature-based heuristic that would be able to infer the probable attack category given raw traffic features, such that MITRE mapping could operate on real-world traffic that is not labeled with the UNSW-NB15 attack_cat label.

- Federated learning across multiple cloud tenants: Extend the framework, following the direction proposed by Nguyen and Zhang (2023) and discussed in Section 2.9, to support collaborative, privacy-preserving model training across multiple organizations or cloud tenants without requiring any single party to share raw log data directly.
- Explainable AI enhancements: In addition to the MITRE based contextualization already done in this thesis, feature-level explainability techniques (such as SHAP and LIME) can be used to inform the analyst of not only which MITRE technique an alert has been mapped to, but which specific features of the input led to the prediction made by the underlying Random Forest model.
- Automated or semi-automated response actions: Go beyond detection and alert and take automated or semi-automated response actions, such as automatically quarantining an alerted suspicious account or blocking an alerted malicious IP address when an alert of Critical severity has been generated, as called for in the original thesis proposal.
- Real-time, streaming dashboard: Replace the current offline, CSV-driven Stream lit dashboard with a streaming architecture capable of ingesting and visualizing new events as they arrive, rather than requiring the batch pipeline to be re-run manually.
- Broader model benchmarking: Extend the three-model comparison presented in this thesis to additional architectures identified in the literature review, such as gradientboosted trees (XGBoost, LightGBM), recurrent neural networks (LSTMs) for modelling sequential connection behaviour, and graph neural networks for modelling network topology directly.

CHAPTER 9

REQUIREMENTS SPECIFICATION AND SYSTEM TESTING

This chapter presents the software requirements specification and the verification testing carried out on the implemented framework, supplementing the design (Chapter 3), implementation (Chapter 4), and results (Chapter 5) chapters with a formal statement of what the system was required to do and evidence that each requirement was met.

9.1 Functional Requirements

Table 18: Functional requirements.

ID	Requirement
FR-1	The system shall ingest the UNSW-NB15 training and testing datasets and remove non-predictive identifier columns.
FR-2	The system shall encode all categorical features (proto, service, state) into numeric form and scale all numeric features using a consistent, reusable transformation.
FR-3	The system shall train an unsupervised Isolation Forest model on normal-only traffic and classify test events as normal or anomalous.
FR-4	The system shall train a supervised Random Forest classifier on the full labelled training set and classify test events as normal or attack.
FR-5	The system shall train an unsupervised Autoencoder on normal-only traffic and classify test events as normal or anomalous using a tunable reconstruction-error threshold.
FR-6	The system shall compute accuracy, precision, recall, and F1-score for every model and export a consolidated comparison file.

FR-7	The system shall map every event classified as an attack to a MITRE ATT&CK tactic, technique, technique ID, and severity rating.
FR-8	The system shall produce a report that gives a human-readable summary of the alerts and a collection of visual charts that show the distribution of
	severity, tactics and attack-category.
FR-9	The system will create an interactive and filterable web dashboard that displays all the above results.

9.2 Non-Functional Requirements

Table 19: Non-functional requirements.

ID	Requirement
NFR-1 (Performance)	The complete offline pipeline (preprocessing through MITRE mapping) shall be run on commodity hardware on the UNSWNB15 dataset containing 257,673 records with out of the box special purpose GPUs.
NFR-2 (Reproducibility)	To ensure that the results of training are reproducible across runs, all models shall be trained with a fixed random seed (random_state = 42) shall be used..
NFR-3 (Modularity)	Each pipeline stage should be a self-contained, re-run-able script, meaning that each stage can be re-run without having to re-run the rest of the pipeline
NFR-4 (Extensibility)	The data-ingestion layer shall be designed in such a way to allow for the replacement of the UNSW-NB15 CSV source by a live Azure telemetry source without redesigning the downstream modelling or MITRE-mapping stages.

NFR-5 (Usability)	The dashboard should provide end-user with important information and charts without requiring any code to be written or executed.
NFR-6 (Maintainability)	The MITRE ATT&CK mapping table shall be stored as a single, centrally-editable data structure, with the ability to add new categories of attacks without impacting other portions of the codebase

9.3 Use Case Description

The main user of this system is a Security Analyst, who is going to interact with the system purely via the Stream lit dashboard. Table 20 shows the main use case that was exercised in this thesis

Table 20: Primary use case: Review AI-Generated Threat Alerts.

Field	Description
Use Case	Review AI-Generated Threat Alerts
Actor	Security Analyst
Precondition	The offline pipeline (Chapters 3-4) has been executed at least once, producing mitre_mapped_alerts.csv and model_comparison.csv.
Main Flow	1. Analyst starts the dashboard (streamlit run app.py). 2. The most recent alert and comparison files are loaded into Dashboard. 3. Headline metrics, model comparison and MITRE/severity charts are reviewed by analysts. 4. The Alert Table is filtered by Analyst. 5. The filtered alert data is downloaded for further investigation or escalation as a CSV file by analyst.
Postcondition	The analyst gets a fresh, contextualized perspective on all detected threats, with them prioritized according to the severity score derived from the MITRE.

Alternate Flow	If no new attacks are detected, then all events are displayed with alert_status = 'No Alert' and Critical Alerts metric is equal to zero.

9.4 Test Strategy

Three levels of verification were conducted: (1) Unit level verification to ensure each pre-processing and mapping function is correct; (2) Integration level verification to ensure that each pipeline stage was taking the output from the previous step; (3) End-toend verification to ensure that the final dashboard correctly represents the underlying model predictions and MITRE mappings.

Table 21: System test cases and outcomes.

ID	Test Name	Description	Result	Status
TC-01	Preprocessing output shape	Verify that train_processed.csv and test_processed.csv contain 42 feature columns plus label and attack_cat.	82,332 / 175,341 rows x 44 columns produced as expected.	Pass
TC-02	Categorical encoding coverage	Verify that every distinct value of proto, service, and state in both partitions is assigned a valid integer code with no encoding errors.	No unseen-category errors raised during encoding.	Pass
TC-03	Feature scaling sanity check	Verify that scaled numeric features have approximately zero mean and unit variance on the training set.	Mean approx. 0, standard deviation approx. 1 confirmed for all numeric columns.	Pass

TC-04	Isolation Forest output convention	Verify that the model's native {+1,-1} output is correctly mapped to the project's {0,1} normal/attack convention.	Confusion matrix totals match test set size (175,341); mapping verified correct.	Pass
TC-05	Random Forest class balance handling	Verify that class_weight='balanced' is applied and that the model does not collapse to always predicting the majority class.	Both classes predicted with non-trivial precision/recall, no class collapse observed.	Pass
TC-06	Autoencoder threshold tuning	Verify that the thresholdtuning sweep correctly identifies the F1-maximizing percentile.	Optimal threshold (~0.1138, ~55th percentile) identified and confirmed to raise F1-score from 0.4592 to 0.7318.	Pass
TC-07	MITRE mapping completeness	Verify that every one of the nine attack categories, plus Normal, has a corresponding entry in the MITRE mapping table.	All 10 categories mapped; no missingkey errors during mapping.	Pass
TC-08	Alert count consistency	Verify that Total Alerts Generated + Total No-Alert Events equals Total Events Processed.	104,442 + 70,899 = 175,341, matching the test set size exactly.	Pass

TC-09	Dashboard data consistency	Verify that the metrics and charts rendered on the dashboard match the values in final_alert_summary.txt and model_comparison.csv.	All dashboard figures confirmed to match the underlying CSV/TXT source files.	Pass
TC-10	Reproducibility	Re-run the full pipeline with the same random seed and verify identical model metrics are produced.	Identical accuracy, precision, recall, and F1-score obtained across repeated runs.	Pass

9.5 Summary of Verification

The implemented pipeline passed all of the 10 test cases listed in Table 21, demonstrating the pipeline's ability to work as specified in pre-processing, all three models, during MITRE mapping, in the generation of alert summary, and in the dashboard. Together with the detailed performance evaluation presented in Chapter 5, this testing lends confidence to the results reported as part of this thesis, which are based on the actual behaviour of the system, as opposed to results based on undocumented or untested implementation details.

REFERENCES

[1] Microsoft Azure. Azure Sentinel Documentation. https://learn.microsoft.com/enus/azure/sentinel/, 2024. Accessed 2026.

[2] M. Khan, F. Ali, and R. Ahmed. Machine Learning-Based Anomaly Detection for Cloud Security. IEEE Access, 11:12034–12045, 2023.

[3] MITRE Corporation. MITRE ATT&CK Framework: Enterprise Matrix. https://attack.mitre.org/, 2024. Accessed 2026.

[4] Nour Moustafa and Jill Slay. UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set). In 2015 Military Communications and Information Systems Conference (MilCIS), pages 1–6. IEEE, 2015.

[5] Leo Breiman. Random Forests. Machine Learning, 45(1):5–32, 2001.

[6] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation Forest. In Proceedings of the 2008 Eighth IEEE International Conference on Data Mining (ICDM), pages 413–422. IEEE, 2008.

[7] Mayu Sakurada and Takehisa Yairi. Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction. In Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis, pages 4–11, 2014.

[8] L. Johnson, R. Davis, and S. Kim. The Role of Explainable AI in Cybersecurity: Enhancing Trust and Transparency in Machine Learning Models. AI for Security, 5(3):87–100, 2022.

[9] D. Patel and S. Kumar. Adaptive Machine Learning Models for Continuous Threat Detection in Hybrid Cloud Environments. International Journal of Cybersecurity, 8(1):55–68, 2024.

[10] H. Nguyen and P. Zhang. Federated Learning for Secure and Privacy-Preserving Cybersecurity in Cloud Environments. IEEE Transactions on Cloud Computing, 12(4):334–348, 2023.

[11] J. Smith and A. Turner. Security Risks in Multi-Cloud Infrastructures: Challenges and Solutions. Journal of Cloud Computing, 9(2):145–158, 2023.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. ACM Computing Surveys, 41(3):1–58, 2009.

[14] Anna L. Buczak and Erhan Guven. A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. IEEE Communications Surveys & Tutorials, 18(2):1153–1176, 2016.

[15] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. A Detailed Analysis of the KDD CUP 99 Data Set. In Proceedings of the 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), pages 1–6. IEEE, 2009.

[16] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP), pages 108–116, 2018.

[17] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A Survey of Network-based Intrusion Detection Data Sets. Computers & Security, 86:147–167, 2019.

[18] R. Vinayakumar, Mamoun Alazab, K. P. Soman, Prabaharan Poornachandran, Ameer Al-Nemrat, and Sitalakshmi Venkatraman. Deep Learning Approach for Intelligent Intrusion Detection System. IEEE Access, 7:41525–41550, 2019.

[19] Kamran Shaukat, Suhuai Luo, Vijay Varadharajan, Ibrahim A. Hameed, and Min Xu. A Survey on Machine Learning Techniques for Cyber Security in the Last Decade. IEEE Access, 8:222310–222354, 2020.

[20] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C. Suh, Ikkyun Kim, and Kuinam J. Kim. A Survey of Deep Learning-Based Network Anomaly Detection. Cluster Computing, 22:949–961, 2019.

[21] Zeeshan Ahmad, Adnan Shahid Khan, Cheah Wai Shiang, Johari Abdullah, and Farhan Ahmad. Network Intrusion Detection System: A Systematic Study of Machine Learning and Deep Learning Approaches. Transactions on Emerging Telecommunications Technologies, 32(1):e4150, 2021.

[22] Nour Moustafa, Marwa Keshk, Essam Debie, and Helge Janicke. Federated TON_IoT Windows Datasets for Evaluating AI-Based Security Applications. In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pages 848–855. IEEE, 2020.

[23] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research, 13:281–305, 2012.

[24] Christoph Molnar. Interpretable Machine Learning: A Guide for Making Black Box Models Explainable. Independently published, 2020. https://christophm.github.io/interpretable-ml-book/, Accessed 2026.

[25] Microsoft Azure. Azure Well-Architected Framework: Security. https://learn.microsoft.com/en-us/azure/well-architected/security/, 2024. Accessed 2026.

APPENDIX A

UNSW-NB15 FEATURE DICTIONARY

Table A.1 lists all forty-two feature columns used for model training after the id, label, and attack_cat columns have been removed by the preprocessing pipeline described in Section 4.2, together with a short description of what each feature measures.

Table 22: Complete UNSW-NB15 feature dictionary (42 modelling features).

Feature	Description
dur	Total duration of the network flow/connection, in seconds.
proto	Transaction protocol used (e.g., tcp, udp, icmp); label-encoded during preprocessing.
service	Application-layer service associated with the connection (e.g., http, ftp, dns, -); label-encoded.
state	Final state of the connection at the transport layer (e.g., FIN, CON, REQ); label-encoded.
spkts	Number of packets sent from source to destination.
dpkts	Number of packets sent from destination to source.
sbytes	Number of bytes transmitted from source to destination.
dbytes	Number of bytes transmitted from destination to source.
rate	Overall packet transmission rate for the connection.
sttl	Source-to-destination time-to-live (TTL) value.
dttl	Destination-to-source time-to-live (TTL) value.
sload	Source bits-per-second load (throughput).

dload	Destination bits-per-second load (throughput).
sloss	Number of retransmitted or dropped source packets.
dloss	Number of retransmitted or dropped destination packets.

sinpkt	Source inter-packet arrival time, in milliseconds.
dinpkt	Destination inter-packet arrival time, in milliseconds.
sjit	Source jitter (variance in inter-packet arrival time), in milliseconds.
djit	Destination jitter, in milliseconds.
swin	Source TCP window advertisement value.
stcpb	Source TCP base sequence number.
dtcpb	Destination TCP base sequence number.
dwin	Destination TCP window advertisement value.
tcprtt	TCP connection round-trip time (sum of synack and ackdat).
synack	Time between the SYN and SYN-ACK packets in the TCP handshake.
ackdat	Time between the SYN-ACK and ACK packets in the TCP handshake.
smean	Mean packet size transmitted from source to destination.
dmean	Mean packet size transmitted from destination to source.
trans_depth	Depth of an HTTP request/response transaction pipeline.
response_body_len	Length of the content in an HTTP response body.

ct_srv_src	Number of connections that share the same service and source address, observed in a recent time window.
ct_state_ttl	Number of connections that share the same state and TTL value combination.
ct_dst_ltm	Number of connections observed to the same destination address in the last time window.
ct_src_dport_lt	Number of connections observed from the same source address to
m	the same destination port in the last time window.
ct_dst_sport_ltm	Number of connections observed to the same destination address from the same source port in the last time window.
ct_dst_src_ltm	Number of connections observed between the same sourcedestination address pair in the last time window.
is_ftp_login	Binary flag indicating whether an FTP session used a login (1) or was anonymous/no-login (0).
ct_ftp_cmd	Number of FTP commands issued in the session.
ct_flw_http_mthd	Number of HTTP methods (GET, POST, etc.) used within a flow.
ct_src_ltm	Number of connections observed from the same source address in the last time window.
ct_srv_dst	Number of connections that share the same service and destination address.
is_sm_ips_ports	Binary flag indicating whether the source and destination IP addresses and port numbers are equal.

APPENDIX B

SAMPLE OF PREPROCESSED DATA

Table B.1 shows the first six rows of the preprocessed testing dataset (test_processed.csv) for a representative subset of columns, illustrating the effect of label encoding (proto, service, state, converted to encoded numeric codes) and StandardScaler feature scaling (all numeric columns transformed to zero mean, unit variance) described in Section 4.2. Values are rounded to three decimal places for display.

Table 23: Sample of preprocessed test data (subset of columns).

Row	proto	service	state	sbytes	dbytes	rate	sttl	dttl	attack_cat
0	0.091	-0.674	-0.337	-0.045	-0.086	-0.554	0.700	1.357	Normal
1	0.091	-0.674	-0.337	-0.042	0.190	-0.554	-1.172	1.340	Normal
2	0.091	-0.674	-0.337	-0.044	-0.000	-0.554	-1.172	1.340	Normal
3	0.091	0.749	-0.337	-0.043	-0.082	-0.554	-1.172	1.340	Normal
4	0.091	-0.674	-0.337	-0.043	-0.086	-0.554	0.719	1.340	Normal
5	0.091	-0.674	-0.337	-0.043	-0.086	-0.554	0.719	1.340	Normal

APPENDIX C

COMPLETE FINAL ALERT SUMMARY OUTPUT

The following is the complete, verbatim content of outputs/final_alert_summary.txt, generated automatically by generate_alert_summary.py (Listing 10) when run against the full test-set predictions.

Listing 16: outputs/final_alert_summary.txt (complete, verbatim).

```
AI-Based Threat Intelligence and Anomaly Detection Summary
============================================================

Total Events Processed: 175341
Total Alerts Generated: 104442
Total No-Alert Events: 70899

Attack Category Distribution:
---------- attack_cat
Normal           56000
Generic          40000
Exploits         33393
Fuzzers          18184
DoS              12264
Reconnaissance    10491
Analysis          2000
Backdoor          1746
Shellcode         1133
Worms             130

MITRE ATT&CK Tactic Distribution:
---------- mitre_tactic
Credential Access    39953
Initial Access       32262
Impact               11967
Reconnaissance       10245
Discovery            5967
Persistence          1723
Execution            965
Lateral Movement     124

Severity Distribution: -----
---------- severity
High        84182
Low         72135
Medium      16212
Critical     2812

Conclusion:
-----------
The AI model successfully detected suspicious network activities and mapped
them to MITRE ATT&CK tactics and techniques. The final output provides
context-aware threat intelligence by combining machine learning-based attack
detection with MITRE-based classification and severity assignment.
```

APPENDIX D

ENVIRONMENT SETUP AND REPRODUCTION STEPS

This appendix documents the exact steps required to reproduce every result reported in this thesis from a clean environment, and lists the pinned Python package versions used throughout development, as recorded in the project's requirements.txt file.

Listing 17: End-to-end reproduction steps.

```
# 1. Create and activate a virtual environment
python -m venv venv
venv\Scripts\activate           # Windows
# source venv/bin/activate      # macOS / Linux

# 2. Install pinned dependencies
pip install -r requirements.txt

# 3. Run the pipeline stages in order python
src/preprocess_data.py python
src/train_isolation_forest.py python
src/train_random_forest.py python
src/train_autoencoder.py python
src/tune_autoencoder_threshold.py python
src/save_tuned_autoencoder_results.py python
src/compare_models.py python
src/mitre_mapping.py python
src/generate_alert_summary.py python
src/generate_result_charts.py python
src/generate_model_comparison_chart.py

# 4. Launch the analytics
dashboard cd dashboard streamlit
run app.py
```

Table D.1 lists the core pinned package versions used in this project, as recorded in requirements.txt.

Table 24: Core pinned dependency versions (requirements.txt).

Package	Pinned Version
pandas	3.0.3
numpy	2.4.6
scikit-learn	1.8.0
scipy	1.17.1
tensorflow	2.21.0
keras	3.14.1
joblib	1.5.3
matplotlib	3.10.9
streamlit	1.58.0
pillow	12.2.0
pyarrow	24.0.0
h5py	3.14.0

APPENDIX E

DASHBOARD USER GUIDE

This appendix provides a step-by-step walkthrough of the Streamlit analytics dashboard (dashboard/app.py, Appendix A, Listing A.13) from the perspective of a security analyst using the framework's outputs, complementing the code-level description already given in Chapter 3 and the results-level description given in Chapter 4.

E1: Launching the Dashboard

After the full pipeline has been executed in the order specified in Appendix B (Section B.5), the dashboard is launched by navigating to the dashboard/ directory and running:

```
streamlit run app.py
```

Streamlit starts a local web server (by default at http://localhost:8501) and opens the dashboard in the analyst's default web browser. No separate database server, authentica tion service, or network configuration is required, since the dashboard reads directly from the static CSV files produced by the offline pipeline (mitre_mapped_alerts.csv and model_comparison.csv) each time it starts.

E2: Step 1: Reviewing Headline Metrics

On loading, the dashboard immediately displays four headline metric cards across the top of the page: Total Events (175,341), Alerts Generated (104,442), No Alert Events (70,899), and Critical Alerts (2,812), computed live from the current contents of mitre_mapped_alerts.csv. An analyst beginning a shift would typically glance at this row first to gauge the overall alert volume and, in particular, the Critical Alerts count, before drilling into any specific alert category

E3: Step 2: Reviewing Model Performance

Below the headline metrics, the dashboard displays the model performance comparison table and chart (Table 4.7 and Figure 4.1), read live from model_comparison.csv. This section is intended primarily for the system's maintainers (rather than day-to-day SOC analysts) to confirm that the currently deployed model, Random Forest, continues to be the best-performing option, and to notice at a glance if a future retraining cycle (Chapter 5, Future Work) has caused any model's accuracy to regress.

E4: Step 3: Reviewing Threat Intelligence Charts

The next section of the dashboard presents four charts side by side, in a two-column layout: severity distribution (Figure 4.2), alert status distribution (Figure 4.3), MITRE ATT&CK tactic distribution (Figure 4.5), and attack category distribution (Figure 4.4). These charts give an analyst an immediate visual sense of the overall shape of the current alert population, for example, quickly confirming that Credential Access and Initial Access dominate the tactic distribution (Section 4.6.1), before looking at any individual alert.

E5: Step 4: Filtering and Investigating Individual Alerts

The final section of the dashboard shows the entire mapped alert table at row level, and has a severity filter (a dropdown that lets the analyst choose "All", "Critical", "High", "Medium", or "Low"). A normal investigative process would start with a filter for "Critical" severity and review the rows (which would include the original label, attack category, the model prediction, the MITRE tactic, technique, technique ID, the severity, and alert status, all of which would be exactly as described in Chapter 4, Section 6.2), and then proceed through "High" severity alerts if time allows. The MITRE technique ID is entered directly into the table, so the analyst can immediately find a link on the official MITRE ATT&CK website (https://attack.mitre.org/) for the recommended detection and mitigation steps for that technique without having to first look through the logs to identify the technique manually.

E6: Step 5: Exporting Data for Further Analysis

The current view of the alert table is available for export as a CSV file, as part of the utility of the framework, for use in a spreadsheet application or a separate case management/ticketing system for formal incident tracking, under an 'export' button located below the alert table.

E7: Refreshing the Dashboard After Retraining

The dashboard's content doesn't need to be refreshed after future retraining cycles (Chapter 5); just re run the following pipeline steps: mitre_mapping.py, generate_alert_summary.py, generate_result_charts.py, and generate_model_comparison_chart.py, and re load the browser page (or restart the Stream-lit process). There is no need to make changes to app.py itself for regular retraining.

E8: Troubleshooting Common Issues

This final section documents common issues an analyst or system administrator may encounter when running the dashboard, and the recommended resolution for each, based on issues encountered during the development and testing of this framework. Dashboard shows stale or missing data. If the dashboard displays metrics that do not match the most recent pipeline run, or shows an error indicating a missing file, confirm that mitre_mapping.py,generate_alert_summary.py,andbothchart-generation scripts (Appendix A, Listings A.9–A.12) have all been re-run after the most recent model training, since the dashboard has no mechanism to detect that its underlying CSV files are out of date, and simply reads whatever is currently present in outputs/. Port already in use. If Stream-lit reports that its default port (8501) is already in use, this typically indicates a previous dashboard instance is still running in the background; the previous process should be terminated, or the dashboard can be relaunched on an alternate port using stream-lit run app.py-server.port 8502. Charts not displaying. If one or more of the four threat-intelligence charts (Figures 4.2 4.5) fail to display, confirm that generate_result_charts.pyandgenerate_model_comparison_chart.py completed

successfully and wrote their output PNG files to outputs/charts/, since the dashboard code (Appendix A, Listing A.13) checks for the existence of each chart file before attempting to display it and will silently skip a chart whose file is missing rather than raising an error. Slow initial load with large alert tables. Because the dashboard loads the complete mitre_mapped_alerts.csv file (one row per test-set event, 175,341 rows in this thesis's evaluation) into memory on every page load, very large deployments processing millions of daily events may experience slower load times than the modest UNSW-NB15 test set exhibits; for such deployments, pre-aggregating summary statistics separately from the row-level alert table, or paginating the alert table serverside, would be a reasonable future enhancement (Chapter 5, Future Work).

APPENDIX F

MITRE ATT&CK TECHNIQUE REFERENCE

This appendix provides a short reference description for each of the nine MITRE ATT&CK techniques used in the mapping table (Table 16), for readers who may not be familiar with the ATT&CK knowledge base's terminology.

Table 1: MITRE ATT&CK technique reference for all techniques used in this framework.

Technique	Tactic	Description
T1110 - Brute Force	Credential Access	Adversaries attempt to systematically guess or repeatedly try credentials (passwords, hashes, or tokens) in order to gain unauthorized access to accounts or services. In this framework, this technique is associated with the Generic attack category.
T1190 - Exploit Public-Facing Application	Initial Access	Adversaries exploit a vulnerability in an internet- or network-facing application or service (for example, a web server or database) to gain an initial foothold on the target system. Associated with the Exploits attack category.
T1046 - Network Service Discovery	Discovery	Adversaries attempt to enumerate the services running on remote hosts, including port scanning and banner grabbing, to identify potential targets for further exploitation. Associated with the Fuzzers attack category.

T1498 - Network Denial of Service	Impact	Adversaries degrade or block the availability of a targeted resource by overwhelming its network capacity, typically by flooding it with traffic. Associated with the DoS (Denial of Service) attack category.
T1595 - Active Scanning	Reconnaissance	Adversaries actively probe target infrastructure (via scanning or vulnerability scanning) to gather information used to plan and support later stages of an intrusion. Associated with the Reconnaissance attack category.
T1082 - System Information Discovery	Discovery	Adversaries attempt to gather detailed information about the operating system, hardware, and configuration of a compromised system to inform further actions. Associated with the Analysis attack category.
T1505 - Server Software Component	Persistence	Adversaries install malicious components (such as web shells or backdoored modules) into a server application to maintain long-term, persistent access. Associated with the Backdoor attack category.
T1059 - Command and Scripting Interpreter	Execution	Adversaries abuse command-line interfaces or scripting engines to execute arbitrary commands, scripts, or programs on a compromised system. Associated with the Shellcode attack category.
T1091 - Replication Through Removable Media	Lateral Movement	Adversaries move between systems using selfreplicating code carried via removable media or network shares, without requiring direct interactive access. Associated with the Worms attack category.

APPENDIX G

GLOSSARY OF KEY TERMS

Table 2: Glossary of key terms used throughout this thesis.

Term	Definition
Anomaly Detection	The task of identifying data points, events, or observations that deviate significantly from an expected or normal pattern.
Attack Surface	The complete set of points at which an unauthorized user (an attacker) could attempt to enter or extract data from an environment.
Class Imbalance	A situation in a classification dataset where one or more classes are represented by substantially more examples than others, which can bias model training toward the majority class.
Confusion Matrix	A table summarizing the number of true positive, true negative, false positive, and false negative predictions made by a classifier.
Feature Scaling	The process of transforming numeric features to a common scale (e.g., zero mean and unit variance) so that no single feature dominates model training purely due to its numeric range.
Hybrid Cloud	An IT architecture that combines on-premises infrastructure with public cloud services, allowing workloads to move between the two environments.

Hyperparameter	A configuration value for a machine learning algorithm that is set before training begins (e.g., the number of trees in a Random Forest), as opposed to a parameter learned from data during training.
Label Encoding	A preprocessing technique that converts categorical values into integer codes so that they can be used as numeric input to a machine learning model.
Reconstruction Error	In an autoencoder, the difference between the original input and the model's reconstruction of that input; used as an anomaly score in this thesis.
Severity Rating	A qualitative label (Low, Medium, High, or Critical) assigned to a detected event to indicate the urgency with which it should be investigated.
Threat Intelligence	Structured, contextual information about potential or actual threats that helps an organization make informed decisions about its security posture.
Zero-Day Attack	An attack that exploits a previously unknown vulnerability, for which no signature or patch yet exists at the time of the attack.