

Ελληνικό Μεσογειακό Πανεπιστήμιο

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Πρόγραμμα Σπουδών Μηχανικών Πληροφορικής ΤΕ

Τίτλος:

ΑΝΙΧΝΕΥΣΗ ΠΑΡΕΙΣΦΡΗΣΗΣ ΜΕ ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ ΜΕΣΩ ΒΑΘΕΩΝ ΑΥΤΟΚΩΔΙΚΟΠΟΙΗΤΩΝ ΣΕ ΔΙΚΤΥΑ ΕΠΟΜΕΝΗΣ ΓΕΝΙΑΣ

Title:

AI-POWERED INTRUSION DETECTION WITH DEEP AUTOENCODERS IN NEXT-GENERATION NETWORKS

ΔΗΜΗΤΡΙΟΣ ΞΟΛΙΑΣ

Επιβλέπων εκπαιδευτικός :

ΕΥΑΓΓΕΛΟΣ ΜΑΡΚΑΚΗΣ

Επιτροπή Αξιολόγησης :

- **ΕΥΑΓΓΕΛΟΣ ΜΑΡΚΑΚΗΣ**
- **ΔΗΜΗΤΡΙΟΣ ΣΤΡΑΤΑΚΗΣ**
- **ΗΛΙΑΣ ΠΟΛΙΤΗΣ**

Ημερομηνία παρουσίασης: 03/20/2026

Ευχαριστίες

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες προς τον επιβλέποντα καθηγητή κ. Ευάγγελο Μαρκακή για την καθοδήγηση, τις ουσιαστικές παρατηρήσεις και τη συνεχή επιστημονική υποστήριξη που παρείχε καθ' όλη τη διάρκεια εκπόνησης της παρούσας διπλωματικής εργασίας. Ιδιαίτερες ευχαριστίες απευθύνονται επίσης στον βοηθό εργαστηρίου κ. Δημήτρη Συγκλέτο για τις πολύτιμες επισημάνσεις και τη συμβολή του σε σημαντικά ζητήματα που αναδείχθηκαν κατά την ανάπτυξη της εργασίας. Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου για την αδιάλειπτη στήριξη και εμπιστοσύνη που μου παρείχε σε όλη τη διάρκεια των σπουδών μου, καθώς και τους φίλους μου, οι οποίοι γνωρίζουν πόσο σημαντική υπήρξε η παρουσία και η υποστήριξή τους.

Abstract

The contemporary technological landscape is undergoing a profound paradigm shift, characterized by the convergence of massively interconnected, software-defined network infrastructures such as 5G, the Internet of Things (IoT), and Software-Defined Networking (SDN). This architectural transformation has concurrently introduced a significant expansion and complexification of the cyber-attack surface, presenting formidable challenges to established security monitoring paradigms. The high volume, velocity, and dimensionality of the resulting data traffic render traditional, signature-based security systems increasingly inadequate for real-time threat identification. Furthermore, while classical machine learning models offer greater adaptability, their efficacy is often encumbered by the "curse of dimensionality" and a dependency on extensive manual feature engineering. This thesis directly addresses these identified gaps through the systematic design, implementation, and rigorous empirical validation of a novel, AI-powered Intrusion Detection System (IDS). The proposed framework was engineered as a lightweight, containerized Virtual Network Function (VNF), ensuring architectural compliance with European Telecommunications Standards Institute - Network Function Virtualisation (ETSI-NFV) standards for scalable deployment in next-generation networks. The core of the system's analytical engine is a hybrid, two-stage deep learning architecture, specifically designed to balance high classification accuracy with computational efficiency. The initial stage employs a Deep Autoencoder (DAE) for unsupervised dimensionality reduction. By being trained to reconstruct the feature vectors of normal traffic, this component learns a compact and information-rich latent representation of high-dimensional network flows, effectively capturing the underlying manifold of benign network behavior. In the second stage, a lightweight one-dimensional Convolutional Neural Network (1D-CNN) is utilized for the efficient and accurate classification of this compressed data, leveraging its inherent capability to detect localized, sequential patterns often indicative of malicious activity. The performance of this framework was subjected to exhaustive experimental validation using the comprehensive and modern RT-IoT2022 dataset. Conducted within a precisely controlled, emulated SDN environment on standard CPU hardware, the proposed model achieved a high overall accuracy of 97.8% and a robust macro-averaged F1-Score of 92.0%, thereby outperforming powerful classical machine learning baselines. Significantly, a mean inference latency of approximately 79 milliseconds was recorded, confirming the system's capacity for near-real-time operation without reliance on specialized hardware. The collective findings of this research validate that the proposed architecture successfully bridges the gap between the high performance of deep learning models and the practical requirements for low-latency, scalable, and deployable security solutions.

Keywords: Intrusion Detection System (IDS), Deep Learning, Autoencoder, Convolutional Neural Network (CNN), Network Function Virtualisation (NFV), Software-Defined Networking (SDN), 5G, Internet of Things (IoT), Network Security, Anomaly Detection.

Περίληψη

Η παρούσα διπλωματική εργασία εξετάζει την ιδιαίτερα σημαντική πρόκληση της ασφάλειας στα Δίκτυα Επόμενης Γενιάς, όπου η σύγκλιση τεχνολογιών όπως το 5G, το Διαδίκτυο των Πραγμάτων (IoT) και η Προγραμματιζόμενη Δικτύωση (Software Defined Networking – SDN) έχει οδηγήσει στη δημιουργία ενός περιβάλλοντος αυξημένης πολυπλοκότητας. Ο μεγάλος όγκος και η υψηλή ετερογένεια των παραγόμενων δεδομένων δικτυακής κίνησης καθιστούν τις παραδοσιακές και στατικές προσεγγίσεις ανίχνευσης εισβολών ανεπαρκείς για την αποτελεσματική αντιμετώπιση σύγχρονων και εξελιγμένων κυβερνοεπιθέσεων. Στο πλαίσιο αυτό, επιχειρείται η αντιμετώπιση του συγκεκριμένου ερευνητικού κενού μέσω του συστηματικού σχεδιασμού, της υλοποίησης και της εμπειρικής αξιολόγησης ενός προηγμένου Συστήματος Ανίχνευσης Εισβολών (Intrusion Detection System – IDS) βασισμένου σε τεχνικές Τεχνητής Νοημοσύνης. Ο πυρήνας του προτεινόμενου πλαισίου βασίζεται σε μια υβριδική αρχιτεκτονική βαθιάς μάθησης δύο σταδίων, η οποία διαμορφώθηκε με στόχο την επίτευξη ισορροπίας μεταξύ υψηλής ακρίβειας ανίχνευσης και υπολογιστικής αποδοτικότητας. Στο πρώτο στάδιο αξιοποιείται ένας Βαθύς Αυτοκωδικοποιητής (Deep Autoencoder), ο οποίος χρησιμοποιείται για τη μη επιβλεπόμενη μείωση της διαστατικότητας των δεδομένων, μαθαίνοντας μια συμπιεσμένη αλλά πληροφοριακά πλούσια λανθάνουσα αναπαράσταση της κανονικής δικτυακής συμπεριφοράς. Στη συνέχεια, στο δεύτερο στάδιο, εφαρμόζεται ένα ελαφρύ Μονοδιάστατο Συνελικτικό Νευρωνικό Δίκτυο (1D Convolutional Neural Network – 1D-CNN), το οποίο αναλαμβάνει την ταχεία και αξιόπιστη ταξινόμηση των παραγόμενων διανυσμάτων χαρακτηριστικών. Το προτεινόμενο σύστημα υλοποιήθηκε ως Εικονική Δικτυακή Λειτουργία (Virtual Network Function – VNF), διασφαλίζοντας τη συμβατότητά του με τις σύγχρονες αρχιτεκτονικές εικονικοποίησης δικτύου και τα πρότυπα ETSI-NFV. Η απόδοση του προτεινόμενου πλαισίου αξιολογήθηκε εκτενώς μέσω πειραματικής διαδικασίας χρησιμοποιώντας το σύγχρονο σύνολο δεδομένων RT-IoT2022 σε ένα εξομοιωμένο περιβάλλον SDN. Τα αποτελέσματα έδειξαν ότι το προτεινόμενο μοντέλο επιτυγχάνει υψηλότερη απόδοση σε σύγκριση με καθιερωμένες μεθόδους μηχανικής μάθησης, όπως οι αλγόριθμοι Support Vector Machines (SVM) και Random Forest (RF). Συγκεκριμένα, επιτεύχθηκε συνολική ακρίβεια 97.8% και μακρο-μέσος δείκτης F1-Score 92.0%. Ιδιαίτερη σημασία παρουσιάζει επίσης το γεγονός ότι ο μέσος χρόνος απόκρισης του συστήματος καταγράφηκε στα 79 χιλιοστά του δευτερολέπτου σε συμβατικό υπολογιστικό υλικό (CPU), γεγονός που επιβεβαιώνει την καταλληλότητά του για εφαρμογές ανίχνευσης εισβολών σε πραγματικό χρόνο. Συνολικά, τα ευρήματα της μελέτης καταδεικνύουν ότι η προτεινόμενη αρχιτεκτονική συμβάλλει ουσιαστικά στη γεφύρωση του χάσματος μεταξύ της υψηλής απόδοσης που προσφέρουν τα μοντέλα βαθιάς μάθησης και της πρακτικής ανάγκης για λύσεις ασφάλειας με χαμηλή υπολογιστική καθυστέρηση.

Table of Contents

ABSTRACT 2

ΠΕΡΙΛΗΨΗ **3**

LIST OF FIGURES 6
LIST OF TABLES 7

INTRODUCTION **8**

STATE OF THE ART **11**

TECHNOLOGIES ENABLERS **19**

ENVIRONMENT & SOFTWARE 19
UBUNTU *19*
PYTHON PROGRAMMING LANGUAGE *19*
RYU CONTROLLER *19*
DATA CLEANING & DATASET 20
DATASET: RT-IoT2022 ..... *20*
TENSORFLOW *20*
KERAS *20*
PANDAS *20*
NUMPY *21*
REQUEST *21*
JOBLIB *21*
PYSHRAK *21*
DATETIME *21*
SUBPROCESS *22*
TIME *22*
OS *22*
CLASSICAL MACHINE LEARNING: scikit-learn *22*
VISUALIZATION *22*
REAL-TIME INFERENCE & FLASK API 23
FLASK *23*
API *23*
FUTURE CONTAINERIZATION *23*
DOCKERFILE *23*
MININET *23*
MINIEDIT *24*
PACKET CAPTURE: tcpdump *24*
NETWORK TRAFFIC 24
WIRESHARK *24*
CICFLOWMETER *24*

IMPLEMENTATION **25**

3.1 CHAPTER SCOPE AND BOUNDARIES 25
3.2 SYSTEM REQUIREMENTS, ARCHITECTURE AND ENVIRONMENT SETUP 27
3.2.1 Python Environment and Dependency Management *29*
3.2.3 Repository and Artifact Layout *29*
3.3 OVERALL SYSTEM ARCHITECTURE 30
3.3.1 Architectural Layers and Data Flow *32*
3.3.1.1 Layer 1: Traffic Capture and Flow Aggregation *32*
3.3.1.2 Layer 2: Feature Normalization and Schema Enforcement *32*
3.3.1.3 Layer 3: Model Inference Service *33*

3.3.1.4 Layer 4: Real-Time Client and Orchestration ... *33*
3.4 Data and Feature Engineering Pipeline ... 34
3.5 Model Development and Artifact Management ... 37
3.6 Real-Time Inference & Deployment ... 38
3.6.1 The Inference Service Interface ... *39*
3.6.2 API Contract and Data Flow ... *40*
3.6.3 Artifact Management and Server Initialization ... *43*
3.7 SDN Testbed for Traffic Generation and Capture ... 45
3.7.1 Topology and Control Plane Integration ... *45*
3.7.2 Traffic Generation Scenarios ... *47*
3.7.3 Packet Capture Configuration and Time Discipline ... *48*
3.7.4 PCAP-to-Features Bridge and Schema Normalization ... *49*
3.8 Diagnostics, Testing Hooks, and Observability ... 51
3.9 Operational Runbooks ... 53
3.9.1 Bare-Metal Execution Runbook ... *54*
3.9.2 Containerized Execution Runbook ... *55*
3.9.3 Operational Hygiene and Failure Remediation ... *56*
3.10 Security and Hardening ... 56
3.11 Limitations and Technical Debt ... 58

EVALUATION ... 61

4.1 Aim and Scope of the Empirical Analysis ... 61
4.2 Evaluative Methodology ... 62
4.2.1 Experimental Environment and Dataset Partitioning ... *62*
4.2.2 Theoretical and Practical Significance of Performance Metrics ... *63*
4.2.3 Justification for Baseline Model Selection ... *63*
4.2.4 Protocol for Real-Time Performance Measurement ... *64*
4.3 Experimental Variables ... 64
4.4 Formulation of Hypotheses ... 65
4.5 Presentation of Empirical Results ... 66
4.5.1 Analysis of Model Training and Convergence Dynamics ... *66*
4.5.2 Granular Classification Performance of the Proposed 1D-CNN Model ... *68*
4.5.3 Comparative Performance Against Baseline Models ... *69*
4.5.4 Real-Time Inference Latency Measurements ... *71*
4.6 Discussion and Interpretation of Results ... 72
4.6.1 Interpretation of the 1D-CNN Model's Classification Efficacy ... *72*
4.6.2 Analysis of Comparative Performance with Classical Models ... *73*
4.6.3 Implications of the Measured System Latency ... *74*
4.6.4 Examination of Key Misclassifications and Anomalous Results ... *75*
4.7 Concluding Evaluation and Synthesis ... 76
4.7.1 Assessment of Findings Against Stated Aims ... *77*
4.7.2 Positioning of Results Within the State of the Art ... *77*
4.7.3 Acknowledgment of Experimental Limitations ... *77*

CONCLUSION ... 79

REFERENCES ... 81

List of figures

FIGURE 1 SIMULATION ENVIRONMENT....27
FIGURE 2TENSORFLOW INITIALIZATION LOG SHOWING THE ABSENCE OF CUDA DRIVERS, RESULTING IN THE SYSTEM DEFAULTING TO CPU-BASED EXECUTION FOR ALL TRAINING....28
FIGURE 3 HIGH-LEVEL SYSTEM ARCHITECTURE AND DATA FLOW....31
FIGURE 4 THE DATA PROCESSING AND MACHINE LEARNING PIPELINE FLOWCHART. 35
FIGURE 5 END TO END POINT SCHEMA....36
FIGURE 6 A SUCCESSFUL CONNECTION TO THE BASE ENDPOINT OF THE RUNNING FLASK API, CONFIRMING THAT THE SERVICE IS ACTIVE AND REACHABLE OVER THE NETWORK....40
FIGURE 7 A SUCCESSFUL PREDICTION CYCLE AS OBSERVED FROM THE FLASK API SERVER LOG, SHOWING AN INCOMING FEATURES. 41
FIGURE 8: CONSOLE OUTPUT SHOWING THE CREATION OF THE MININET NETWORK TOPOLOGY 46
FIGURE 9 EXECUTION OF A TCP SYN FLOOD ATTACK FROM WITHIN THE MININET COMMAND-LINE INTERFACE USING THE HPING3 TOOL. 47
FIGURE 10 REAL-TIME PACKET CAPTURE USING THE TCPDUMP UTILITY ON THE CORE SWITCH INTERFACE, SHOWING THE NUMBER OF PACKETS CAPTURED. 49
FIGURE 11 PACKET-LEVEL ANALYSIS OF A CAPTURED SYN FLOOD ATTACK IN WIRESHARK, CONFIRMING THE HIGH VOLUME OF SYN REQUESTS. 49
FIGURE 12 PACKET-LEVEL ANALYSIS OF A CAPTURED UDP SCECARIO IN WIRESHARK, CONFIRMING THE HIGH VOLUME OF UDP REQUESTS. 50
FIGURE 13: A SECTION OF THE JSON VALIDATION REPORT GENERATED BY THE SELF-TEST HARNESS PART_1 52
FIGURE 14 A SECTION OF THE JSON VALIDATION REPORT GENERATED BY THE SELF-TEST HARNESS PART_2 53
FIGURE 15 CONSOLE OUTPUT FROM THE REAL-TIME PREDICTOR CLIENT, DEMONSTRATING THE STRUCTURED LOGGING OF EACH SUCCESSFUL INFERENCE CYCLE....55
FIGURE 16 THE ARCHITECTURE OF THE REAL-TIME INFERENCE PIPELINE, ILLUSTRATING THE FLOW OF DATA FROM THE NETWORK INTERFACE THROUGH FEATURE EXTRACTION. 58
FIGURE 17 PACKET-LEVEL ANALYSIS OF THE BENIGN BACKGROUND TRAFFIC GENERATED WITHIN THE SDN TESTBED. 60
FIGURE 188 AUTOENCODER TRAINING AND VALIDATION LOSS OVER EPOCHS 67
FIGURE 19 1D-CNN CLASSIFIER TRAINING AND VALIDATION ACCURACY. 68
FIGURE 20 CONFUSION MATRIX FOR THE PROPOSED 1D-CNN MODEL ON THE TEST DATASET. 69
FIGURE 21 CONFUSION MATRIX FOR THE RANDOM FOREST BASELINE MODEL ON THE TEST DATASET. 70
FIGURE 22 CONFUSION MATRIX FOR THE SVM BASELINE MODEL ON THE TEST DATASET. 71
FIGURE 23 CONSOLE OUTPUT OF THE REAL-TIME CLIENT DEMONSTRATING CONTINUOUS INFERENCE. 76

List of tables

TABLE 1: REPOSITORY DIRECTORY STRUCTURE AND ARTIFACT LOCATIONS. .. **ERROR! BOOKMARK NOT DEFINED.**
TABLE 2: KEY SOFTWARE COMPONENTS AND THEIR ARCHITECTURAL ROLES. **ERROR! BOOKMARK NOT DEFINED.**
TABLE 3: DISTRIBUTION OF TRAFFIC PATTERNS IN THE RT-IOT2022 DATASET. ..... **ERROR! BOOKMARK NOT DEFINED.**
TABLE 4: ARCHITECTURE OF THE 1D-CNN MODEL. ..... **ERROR! BOOKMARK NOT DEFINED.**
TABLE 5 SUMMARY OF REAL-TIME INFERENCE LATENCY STATISTICS ..... **ERROR! BOOKMARK NOT DEFINED.**
TABLE 6 COMPARATIVE OVERALL PERFORMANCE OF EVALUATED MODELS 72
TABLE *7 COMPARATIVE AVERAGE PREDICTION TIME PER SAMPLE* 74

Introduction

The contemporary technological landscape is undergoing a profound and accelerating paradigm shift, characterized by the convergence of massively interconnected, software-defined network infrastructures. The evolution towards Next-Generation Networks, a domain encompassing 5G telecommunications, the ubiquitous Internet of Things (IoT), and the architectural principles of Software-Defined Networking (SDN) has transitioned from theoretical constructs to become the operational backbone of modern digital services. This transition from static, hardware-centric systems to dynamic, highly virtualized environments has been a catalyst for innovation, enabling unprecedented levels of automation and massive-scale connectivity. However, this architectural transformation has concurrently introduced a significant expansion and complexification of the cyber-attack surface, presenting formidable challenges to established security monitoring paradigms.

The security implications of this new landscape are multi-faceted. The immense data throughput and high speed generation of data generated by billions of heterogeneous IoT devices render traditional, centralized logging and manual monitoring computationally untenable [1]. Furthermore, the necessary adoption of encryption to protect data in transit presents a dual challenge for network security; while essential for privacy, it effectively blinds conventional deep-packet inspection tools, making traffic analysis significantly more difficult. The architectural innovations of SDN, such as the programmatic decoupling of the control and data planes, also introduce new, centralized points of potential failure and unique vectors for sophisticated attacks targeting the control logic itself. It has been well-documented that traditional Intrusion Detection Systems (IDS) are architecturally and methodologically ill-equipped to operate effectively within this new context. Security mechanisms that rely on predefined, static signatures are inherently incapable of identifying threats that do not have a known signature. This includes novel threats, zero-day attacks that exploit previously unknown vulnerabilities, and polymorphic threats which dynamically alter their characteristics to evade detection. Moreover, early anomaly detection systems, while conceptually more adaptive, have often been hindered by the "curse of dimensionality" and have historically struggled with high false positive rates, leading to "alert fatigue" and a consequential lack of trust in their operational utility. A critical gap was therefore identified for a security solution that is not only highly accurate but is also computationally efficient and architecturally congruent with the virtualized, dynamic nature of modern networks.

This thesis was undertaken to directly address this identified gap through the systematic design, implementation, and rigorous empirical validation of an advanced, AI-powered IDS. The research presented herein is positioned at the confluence of network engineering and artificial intelligence, leveraging the proven capabilities of Deep Learning (DL) to overcome the inherent limitations of classical machine learning paradigms. While traditional ML models, such as Support Vector Machines (SVM) and Random Forest classifiers (RF), have demonstrated considerable power, their efficacy often depends on laborious and expert-driven manual feature engineering. In contrast, the multi-layered architectures of DL models are capable of automatic hierarchical

feature extraction, enabling them to learn intricate, non-linear patterns and subtle correlations directly from raw, high-dimensional data. This capability is particularly well-suited to the analysis of network traffic, where complex, sequential, and multi-dimensional feature interactions can signify sophisticated malicious activity.

The core technical contribution of this work is a novel, hybrid DL architecture, which was specifically engineered for the dual imperatives of high accuracy and low-latency network intrusion detection. A carefully designed two-stage process was developed. The first stage employs a Deep Autoencoder (DAE) for the task of unsupervised feature compression. This component directly addresses the critical challenge of data dimensionality by learning a compact, yet information-rich, latent representation of the network flow features. By being trained to reconstruct normal traffic patterns, the autoencoder implicitly learns the underlying manifold of benign activity. The second stage utilizes a lightweight one-dimensional Convolutional Neural Network (CNN) for the efficient classification of traffic. This architectural choice was motivated by the hypothesis that the 1D-CNN is adept at identifying localized and sequential patterns within the compressed feature vector produced by the autoencoder patterns that may correspond to specific attack signatures. This synergistic combination was hypothesized to balance the powerful representational learning of deep learning with the stringent computational efficiency required for real-time inference.

While the novelty of the proposed model is a primary contribution, of this work is the implementation of the entire IDS as a lightweight, stateless, and containerized Virtual Network Function (VNF). This approach was deliberately chosen to ensure that the proposed solution is not merely a theoretical construction but a practical and deployable artifact. Its design is fully compliant with the European Telecommunications Standards Institute - Network Function Virtualization (ETSI-NFV)[1]framework, the foundational standard for modern 5G and cloud network architecture. This architectural alignment ensures that the system is prepared for scalable, automated lifecycle management and can be dynamically instantiated and managed within discrete network slices, a key feature of 5G. The entire system was constructed and validated within a reproducible, emulated SDN environment using a suite of carefully selected open-source technologies, including Mininet for network emulation, the Ryu SDN controller for control plane management, TensorFlow for model development, and Flask for the creation of a RESTful inference API. This technological stack was chosen to create a complete and verifiable end-to-end pipeline, from packet capture to final classification.

This final chapter serves to synthesize the extensive research conducted throughout this thesis. It will begin by presenting a consolidated and detailed view of the problem domain and the proposed technical and architectural solution. The performance of the developed framework, as empirically validated on the comprehensive and modern RT-IoT2022 dataset, will be presented in detail. This includes a granular analysis of the model's classification accuracy across twelve distinct traffic classes and a direct, quantitative comparison against the well-established SVM and RF benchmarks. A pivotal focus of this system's real-time performance, quantified through meticulous measurements of its inference latency, will be analyzed to substantiate its practical

[1] Etsi, "Etsi," [Online]. Available: https://www.etsi.org/technologies/nfv

deploy ability. The implications of these findings are then discussed, positioning the contribution of this work within the broader landscape of network security research. Finally, the inherent limitations of the current study are transparently acknowledged, and substantive, actionable directions for future research are proposed, thereby concluding the scientific narrative of this thesis.

The remainder of this thesis is structured as follows:

The scientific narrative of this thesis is structured to guide the reader logically from the foundational context to the final conclusions. The research journey begins in **Chapter 2**, which provides a comprehensive review of the state of the art in network intrusion detection. This chapter charts the historical progression from classical machine learning paradigms to advanced deep learning architectures, thereby establishing the theoretical context and identifying the critical research gaps that motivate the present work. Building upon this foundation, **Chapter 3** delineates the specific technologies and the software environment employed, offering a detailed exposition of the tools, libraries, and frameworks that were selected to construct the proposed system.

The core technical contribution is detailed in **Chapter 4**, which is dedicated to the implementation of the proposed Intrusion Detection System. This chapter serves as a complete technical blueprint, providing a systematic account of the system's architecture, its data processing pipeline, the model development procedures, and the construction of the real-time inference service and SDN testbed. Subsequently, **Chapter 5** presents the empirical evaluation of the system's performance. It details the evaluative methodology, presents the quantitative results of the experiments, and provides a thorough interpretation of the findings in a comparative context. Finally, **Chapter 6** concludes the thesis by synthesizing the key contributions of the research, transparently acknowledging the study's limitations, and proposing substantive, actionable directions for future investigation in this domain.

State of the Art

Over the past decade, the increasing complexity and data volume in modern network infrastructures, such as the IoT and SDN, have rendered traditional security methodologies insufficient. In response, a significant body of research has been directed towards the application of Artificial Intelligence (AI) for creating adaptive and intelligent IDS. These systems have evolved from relying on predefined signatures to employing data-driven models capable of identifying novel and sophisticated cyber threats in real time. This chapter presents a critical review of the state of the art in this domain, charting the progression from classical machine learning paradigms to advanced deep learning architectures for network anomaly detection. The review is structured to highlight the key technical contributions, identify their inherent limitations, and thereby establish the research gaps that necessitate the development of new, more effective solutions.

In the work by Ujjan [3] a transitional approach was proposed that bridges traditional security tools with modern AI techniques. A hybrid IDS was developed specifically for the dynamic environment of SDN, combining a signature-based component, Snort, with a flow-based anomaly detection mechanism powered by a Deep Neural Network (DNN). This collaborative architecture was designed to leverage the distinct strengths of both paradigms: Snort was utilized for its high accuracy in detecting known threats based on a predefined signature database, while the DNN, implemented with a Stacked Autoencoder (SAE) algorithm, was employed to analyze statistical features extracted from network flows, enabling the identification of unknown or zero-day anomalies that lack a signature. . The efficacy of this collaborative detection model was assessed by simulating both malicious and benign traffic. The empirical results demonstrated that the integration of deep learning yielded a marked improvement in the true positive detection of TCP, UDP, and ICMP attacks while simultaneously suppressing false alarm rates. This validated the hybrid model as a more robust and comprehensive solution than either of its constituent parts operating in isolation.

While research in SDN explored hybrid models, parallel efforts were directed towards securing Wireless Sensor Networks (WSNs), which are widely considered precursors to modern IoT environments due to authors [4] shared characteristics of resource constraints and distributed topologies. In this context, a trust-based intrusion detection system for the network layer, termed NL-IDS. This system was uniquely designed to detect blackhole attacks not by inspecting packet content, but by evaluating the trustworthiness of the sensor nodes themselves. The trust value for each node was dynamically calculated based on the deviation of key network-layer factors, with a primary focus on hop count, under the assumption that malicious nodes would advertise artificially low hop counts to illegitimately attract network traffic. The performance of the NL-IDS was simulated in MATLAB and showed superior detection accuracy and a lower false alarm rate compared to previous trust-based methods, demonstrating that security in resource-constrained networks could be approached through behavioral evaluation of network participants.

This shift from classical to more advanced methods was formally contextualized in a general survey conducted by Dol and Geetha [5], which was presented to formally outline the transition from classical Machine Learning (ML) to Deep Learning (DL). The study served to differentiate the two paradigms, explaining that while ML models often rely on manual feature engineering, DL models, characterized by their multi-layered neural network architectures, are capable of automatic hierarchical feature extraction. This work contextualized the broader shift in AI, arguing that DL's ability to learn intricate patterns from raw, high-dimensional data makes it particularly well-suited for complex tasks like computer vision, natural language processing, and, by extension, network traffic analysis. This survey provides the theoretical justification for the field's move away from traditional ML models toward the more sophisticated deep learning architectures discussed in the subsequent literature.

Narrowing this focus to the domain of network security, a study by Hussain and Hnamte provided a foundational overview of this evolution [6]. Their work outlined the theoretical progression from classical machine learning to more sophisticated deep learning models such as Recurrent Neural Networks (RNNs) and Autoencoders (AEs). It was argued that while classical methods have inherent shortcomings in handling high-dimensional and evolving data patterns, deep learning techniques possess the intrinsic potential for higher efficiency and accuracy in malicious traffic identification due to their ability to learn hierarchical feature representations automatically. The work served to codify the advantages of DL for intrusion detection, particularly within the architectural flexibility offered by SDN, and established the theoretical groundwork for the practical implementations that followed.

A significant and persistent challenge addressed by these deep learning models is the high dimensionality of traffic data, which can impede the computational efficiency required for real-time detection. To address this, a novel framework centered on a Deep Nested Clustering Auto-Encoder (DNCAE) was introduced by Nguyen et al. [7]. Autoencoders are a type of unsupervised neural network employed to learn compressed representations (encodings) of input data by training the network to reconstruct its own input. In the context of IDS, this is valuable for both dimensionality reduction and anomaly detection, as the model will exhibit higher reconstruction errors for data that deviates from the normal patterns it was trained on. The DNCAE model was specifically designed to learn a more compact and informative latent representation of normal network data in a semi-supervised manner. The architecture consisted of two nested Deep Auto-Encoders (DAEs), where a k-means clustering algorithm was uniquely integrated into the latent layer of the outer DAE. The purpose of this integration was to force the model to not only minimize the reconstruction error but also to organize the latent space into distinct, tight clusters of normal behavior. This approach was shown to create a more expressive data representation space, which significantly improved the performance of anomaly detection by making the boundary between normal and anomalous data clearer and easier for a classifier to learn, as demonstrated on benchmark datasets like NSL-KDD, UNSW-NB15, and CIC-IDS2017.

Concurrent with the push for higher accuracy was a growing need for model transparency and interpretability, as the "black-box" nature of deep learning models can

be a barrier to adoption in critical security applications. An explainable AI (XAI) based solution was implemented by Zebin et al. [8]to detect the modern and evasive threat of DNS over HTTPS (DoH) attacks. For this task, a Balanced Stacked Random Forest classifier was proposed, demonstrating the continued power of ensemble methods. Crucially, its high accuracy was supplemented with explainability methods such as SHAP (Shapley Additive Explanations), a game-theoretic approach that assigns a contribution value to each feature for a given prediction. This work highlighted the importance of not only detecting threats with high precision but also providing transparent and interpretable results, allowing security analysts to validate and understand the model's decision-making process.

To handle the complex and inherently sequential nature of network traffic, where the order and timing of packets are critical for identifying certain attacks, hybrid deep learning models were investigated by Altunay and Albayrak [9]. A system combining a Convolutional Neural Network (CNN) with a Long Short-Term Memory (LSTM) network was developed . This hybrid architecture was proposed to detect intrusions in Industrial IoT (IIoT) networks, a particularly sensitive environment. In this model, the CNN was utilized to extract local, spatial features from the network data (viewed as a 1D sequence or 2D matrix), while the LSTM was used to model the long-range temporal dependencies and patterns across these extracted features. The performance of this hybrid CNN+LSTM model was evaluated on the UNSW-NB15 and X-IIoTID datasets, where it was demonstrated to achieve superior detection accuracy in both binary and multi-class classification tasks compared to standalone CNN or LSTM models.

The application of deep learning was also specifically tailored for IIoT environments in other research, with a focus on creating practical and efficient systems for these critical infrastructures. A design for an anomaly detection system using a Deep Auto-Encoder (DAE) and a Deep Feed-Forward Neural Network (DFFNN) was presented by RajBalaji et al. [9]. The system was trained to recognize normal network behaviors from TCP/IP packet data and subsequently identify deviations indicative of malicious activity. The research emphasized a two-stage training approach: first, unsupervised learning with the DAE was used to establish baseline parameters and learn a compressed representation of the data. In the second stage, this DFFNN was then employed as a supervised classifier, which utilized the compressed features learned by the DAE to perform the final classification of traffic as either benign or malicious. This approach was shown to yield a low false positive rate and high detection accuracy on the NSL-KDD and UNSW-NB15 datasets.

Alternative learning paradigms have also been explored for intrusion detection in SDN, focusing on novel ways to process and classify feature sets to improve robustness presented by Sharma et al. [10]. The use of a Multiple Kernel Learning (MKL) algorithm was proposed for the detection of DDoS attacks. This approach utilized an efficient unsupervised method involving both shallow and deep autoencoders to generate multiple, diverse feature sets from the raw training data. The MKL algorithm was then employed to learn an optimal combination of these feature sets into a weighted kernel space for a final, more robust classification. Experimental results on the UNSW-NB15 dataset indicated that this MKL-based method provided better accuracy and

detection rates compared to models like SVM and LSTM, showcasing the benefit of fusing multiple feature representations.

Further advancing the state of the art for modern consumer networks, an intelligent IDS was developed by Javeed et al. [11] for smart Consumer Electronics (CE) environments, which are a significant component of the broader IoT landscape. The proposed solution was architected around an SDN-orchestrated framework to handle the distributed and heterogeneous nature of CE networks. The core of the detection mechanism was a Cuda-enabled Bidirectional Long Short-Term Memory (Cu-BLSTM) model. A BLSTM processes sequence data in both forward and reverse directions, allowing it to capture contextual information that a standard unidirectional LSTM might miss. The use of a Cuda-enabled implementation underscored the focus on high-performance, real-time processing by leveraging GPU acceleration. The model's performance was validated on the comprehensive CICIDS-2018 dataset, where it was shown to outperform baseline models like Cu-DNN and Cu-GRU in accurately identifying a wide range of attack types.

The applicability of DL models has been strongly validated in adjacent fields involving sequential and high-dimensional data, providing further justification for their use in network traffic analysis. For instance, in the work of Rani and Chauhan [12], sentiment analysis of textual data was performed using both traditional machine learning and deep learning models. The study employed an LSTM network, which is specifically designed to handle long-range dependencies in sequential data by using a series of gates to control information flow. The success of LSTMs in natural language processing, a domain characterized by sequential and contextual data, provides a strong analogue for their effectiveness in analyzing network traffic, which similarly consists of ordered sequences of packets and flows that unfold over time.

The efficacy of deep learning for automated feature extraction is further substantiated by its successful application in adjacent fields, particularly image processing, which similarly requires the analysis of complex, high-dimensional data without manual intervention. A system was developed by Patil and Mudholkar [13] that combined classical image processing with a CNN algorithm for the purpose of leaf pest detection. In a similar vein, a CNN-based method for image reconstruction in ultrafast ultrasound imaging was introduced by Zhang et al. [14]. In these contexts, CNNs were shown to be exceptionally effective at autonomously learning and extracting relevant spatial features from complex, high-dimensional data. This capability is directly transferable to network security, where network traffic can be structured into image-like representations to allow CNNs to identify spatial patterns indicative of malicious activity, such as those present in port scanning or certain malware communications.

The challenge of handling high within-class variability, a major issue in network traffic where "normal" behavior can vary widely, has been addressed in the field of image classification using highly specialized deep learning architectures. The work by Wang et al. [15] introduced SymNet, a deep learning method operating on Symmetric Positive Definite (SPD) manifolds. This approach was designed to learn invariant representations from image sets that exhibit significant variations. While the mathematical underpinnings are specific to image analysis, the core problem it solves

learning to distinguish classes despite high internal variability is directly analogous to the primary challenge in anomaly-based network intrusion detection. This research demonstrates that sophisticated deep learning architectures can be engineered to explicitly handle data variance, a principle that is highly relevant to the development of robust IDS.

A Novel by Zhang and Jiang [16] presented frameworks for the construction and visualization of deep learning models have also been researched, pointing toward future directions in the field of AI-assisted engineering. The elements of a Virtual Reality (VR) based deep learning model and its construction methods were explored . While not directly an IDS, this work investigated how immersive technologies could be used to enhance a learner's experience and understanding of abstract knowledge by materializing it in a virtual environment. This line of research suggests future possibilities in designing more intuitive tools for interpreting the complex, multi-layered architectures used in deep learning-based security systems, potentially complementing XAI methods and making them more accessible to security operations teams.

The paradigm of using ML and DL to automate complex, sequential, and time-consuming tasks has been successfully applied in fields entirely separate from network security, further validating the power of these techniques. A survey by Pandey et al. [17] examined the use of advanced ML and DL to solve the "phase-ordering problem" in compiler optimization. This problem involves finding the optimal sequence of compiler transformations to apply to a piece of code to maximize its performance, a task that is computationally immense. The success of deep learning models in this domain demonstrates their capability to automate complex, sequential decision-making processes that were traditionally the domain of human experts. This parallels the task of an IDS, which must analyze a sequence of network events to make a complex security decision, reinforcing the argument that DL is a suitable tool for automating network security.

With the explosive proliferation of IoT devices, securing these heterogeneous and often resource-constrained environments has become a primary focus of the research community, where the study of their work was presented by Sharma and Chauhan [18]. A general review of AI and IoT-based intrusion detection for cybersecurity. The study surveyed various AI models, including Random Forest, Naive Bayes, and XGBoost, and their application in detecting dangerous URLs and other threats within IoT networks. It was noted that while AI provides powerful tools for detection, significant challenges related to computational overhead, model adaptability, and the sheer diversity of IoT devices and protocols remain, particularly for real-time applications where latency is a critical factor.

To address these challenges in a practical setting, a hybrid AI-based IDS for IoT-enabled smart industries was proposed by Sathyaseelan et al. [19]. This system integrated a CNN for automated feature extraction with a k-Nearest Neighbors (KNN) algorithm for classification. A key contribution of this work was its direct confrontation of the data imbalance problem, which is endemic to network security datasets where malicious traffic is rare compared to benign traffic. The authors employed the Synthetic

Minority Over-sampling Technique (SMOTE) combined with Tomek-links under-sampling to create a more balanced training dataset. This approach was shown to significantly enhance detection accuracy for minority-class intrusion patterns when evaluated on the CIDDS-001 dataset, demonstrating that data preprocessing is as critical as model architecture.

The performance of different deep learning models on specific, modern datasets has also been a subject of intense study to establish reliable benchmarks for the community was conducted by Manan et al. [20] on the Bot-IoT dataset. An analysis of various DL approaches, including Feed-Forward Deep Neural Networks (FDNN) and different types of autoencoders, . The study evaluated the effectiveness of these models in both binary and multi-class classification scenarios, measuring performance based on correctness, false alarm rate, and detection rate. This research provided valuable benchmarks for how different DL architectures perform on a dataset specifically designed to represent modern IoT-based botnet attacks, which are a prevalent and growing threat.

Within the specific context of SDN, comparative analyses of different IDS technologies have been conducted to establish performance baselines for this new networking paradigm. A study by Shams et al. [21] integrated and compared two traditional open-source IDS, Snort and Suricata, within an SDN environment built with Mininet and OpenDaylight. By injecting malicious DoS/DDoS traffic, the true positive and false negative rates of both systems were evaluated. The results indicated that Snort demonstrated superior performance in this emulated SDN topology. This work is important as it provides a benchmark for traditional IDS capabilities in a programmable network, against which more advanced AI-based solutions can be measured to quantify their improvement.

More extensive comparative studies have been performed. A detailed analysis was carried out by Rakha et al. [22] on AI-based IDS for smart city environments presented to evaluate a wider range of AI-based classifiers in the context of smart cities, which represent a large-scale integration of IoT and modern networking. The study utilized the complex NF_BoT_IoT and IoTID20 datasets to evaluate the performance of numerous classifiers, including machine learning models like Random Forest and Gradient Boosting, as well as various neural network architectures. The findings showed that the Random Forest classifier generally performed better than conventional methods in terms of accuracy, precision, and recall, highlighting the robustness of ensemble methods on modern, complex datasets.

A critical component of developing effective IDS was presented by Hindy et al. [23] where the quality and relevance of the datasets used for training and evaluation, as the performance of any data-driven model is fundamentally limited by the data it is trained on. A comprehensive survey and taxonomy of network threats and datasets was publiced. The study highlighted a major gap in the field: current benchmark datasets often lack representations of modern, real-world network threats and include a large number of deprecated or unrealistic attack vectors. It was argued that this discrepancy severely limits the detection accuracy and real-world applicability of machine learning-

based IDS, as models trained on outdated data are ineffective against emerging zero-day attacks.

The unique security challenges inherent to IoT were further detailed in a survey by Conti et al. [24]. The paper discussed the major security and forensic challenges within the IoT domain, which stem from the wide distribution of nodes, the private nature of the collected data, and the resource constraints of many devices. The authors emphasized the need for specialized tools and techniques for securing IoT networks, as conventional security architectures are often ill-suited for these object-driven and heterogeneous environments, reinforcing the need for lightweight and adaptable AI-based solutions that can operate at the network edge.

Beyond the application of AI for detecting external threats, a broad review by Rani et al [25]. explored the critical field of securing the AI systems themselves. The study provided a comprehensive overview of safe AI system design, emphasizing the foundational principles of confidentiality, integrity, and availability. It was detailed how these principles are implemented through techniques such as robust data encryption (e.g., AES-256), role-based access control (RBAC), and cryptographic hashing (e.g., SHA-256) to ensure data integrity. A significant portion of the work was dedicated to discussing the vulnerabilities of AI models to adversarial attacks maliciously crafted inputs designed to cause misclassification categorizing them into evasion attacks (at inference time) and poisoning attacks (at training time). Furthermore, the integration of AI with Blockchain technology was examined as a novel paradigm for enhancing privacy and trust. It was argued that Blockchain's decentralized and immutable ledger offers a powerful mechanism to create tamper-resistant storage for training data and auditable logs for model predictions, thereby increasing the overall resilience of AI-driven security systems.

A practical demonstration of the principles of trust and data integrity using Distributed Ledger Technology (DLT) was presented by Yadav et al. [26] in a different domain. The authors proposed a scalable and novel framework for land property transactions, a system that requires a high degree of security against fraud. The architecture was built upon the InterPlanetary File System (IPFS), a peer-to-peer network for data storage and sharing, which provided the decentralized foundation. A key innovation of this work was the proposal of a "Trusted Node Consensus Algorithm" (TNCA), which was designed to minimize the computational overhead typically associated with traditional consensus mechanisms like Proof-of-Work (PoW). By using a trust value-based system to select miners, the TNCA was shown to reduce message exchange by over 58% compared to PoW. Although the application was not in network security, the architectural principles achieving a secure, tamper-proof, and decentralized system with an efficient consensus mechanism are directly analogous to the requirements for managing trust and data integrity in modern, distributed network security frameworks.

Looking toward the future of telecommunications, the foundational role of AI in 5G and, particularly, 6G wireless systems was reviewed by Gote et al. [27]. The paper surveyed the integration of AI, machine learning, and deep learning for managing the entire lifecycle of next-generation networks. It was highlighted that these technologies

are not merely add-ons but are essential for enabling the core functionalities that define 6G, such as ultra-reliable low-latency communications (URLLC), massive machine-type communications (mMTC), and dynamic network slicing. Each of these capabilities introduces unique and complex security challenges: URLLC requires instantaneous threat response, mMTC creates a massive attack surface of billions of IoT devices, and network slicing demands automated, per-slice security policies. The authors argued that AI-driven mechanisms for intelligent network orchestration, autonomous decision-making, and real-time threat detection are the only feasible way to manage this complexity. This forward-looking perspective underscores the critical need for intelligent, automated, and highly efficient security solutions to be natively integrated into the very architecture of future networks from their inception.

This comprehensive literature review reveals a clear trajectory toward intelligent, data-driven intrusion detection systems capable of operating in complex, high-speed network environments. However, several critical gaps persist that motivate the present work. While many advanced deep learning models demonstrate high detection accuracy, their computational complexity often renders them unsuitable for real-time deployment, particularly in the latency-sensitive 5G and IoT scenarios that are central to next-generation networks. Furthermore, much of the existing research focuses on model performance in isolated laboratory settings, with insufficient attention paid to practical deployment within standardized, scalable network architectures such as the ETSI-NFV framework, which is the foundation of modern telecommunication systems. There is, therefore, a distinct need for a solution that not only achieves high accuracy but also prioritizes computational efficiency through effective dimensionality reduction. This thesis directly addresses these gaps by proposing a sophisticated IDS that integrates Deep Autoencoders for efficient feature compression with a 1D-CNN for rapid and accurate anomaly classification. The entire system is designed as a lightweight, containerized Virtual Network Function (VNF), ensuring its compatibility with ETSI-NFV standards and enabling scalable, real-time deployment in modern SDN and 5G network slices.

Technologies Enablers

ENVIRONMENT & SOFTWARE

This section delineates the comprehensive array of technologies, tools, and software components that facilitated the development and implementation of the proposed Intrusion Detection System. It encompasses aspects such as data preprocessing, machine learning frameworks, SDN simulation, real-time packet capture, and virtualization. Each component is examined in relation to its role, functionality, and integration within the overall system architecture. The chapter highlights how these technologies were carefully selected and orchestrated to ensure performance, scalability, and real-time operation in a 5G-enabled network environment.

UBUNTU

The (IDS) was developed and deployed on a Linux platform to ensure compatibility with open-source networking and machine learning tools. The operating system that was selected for this purpose was Ubuntu 22.04 LTS[2] , the system is equipped with 32 GB of RAM, which facilitates the execution of memory-intensive processes such as deep learning model training and real-time packet processing without encountering resource limitations.

PYTHON PROGRAMMING LANGUAGE

The development environment was established using Python 3.12 [3] to facilitate the development and training of deep learning models, alongside deploying a Flask-based inference API. This version was selected for its enhanced performance and compatibility. In contrast, a separate environment was configured with Python 3.7 [29] to operate the Ryu SDN controller, as later versions presented compatibility challenges. This environment includes all necessary dependencies for network emulation and flow management while avoiding potential conflicts with current machine learning frameworks. This strategic separation of environments ensures a modular, reproducible, and easily addressable system.

RYU CONTROLLER

Ryu was selected for its modular design, ease of customization, and native implementation in Python, which facilitates seamless integration with other Python-based Intrusion Detection System (IDS) components. Unlike alternatives such as Open Daylight [4], Ryu [5]exhibits minimal overhead and possesses no significant external dependencies, making it particularly suitable for simulations and lightweight deployments. The controller logic was developed in a script named ryu_controller.py, which establishes flow rules for packet routing between testbed hosts and enables passive monitoring and analysis. It is important to note that Ryu is compatible only with

[2] "Ubuntu," [Online]. Available: https://ubuntu.com/download/desktop.
[3] "Python," [Online]. Available: https://www.python.org/downloads/.
[4]"Open Daylight," [Online]. Available: https://www.opendaylight.org/
[5]"TensorFlow-Keras," Google, [Online]. Available: https://www.tensorflow.org/guide/keras.

Python 3.7, necessitating the establishment of a dedicated virtual environment to avoid conflicts with other components utilizing newer versions of Python. This separation of concerns enabled stability and version isolation between the SDN control logic and the AI-based inference pipeline.

DATA CLEANING & DATASET

DATASET: RT-IoT2022

The RT-IoT2022 dataset[6] serves as a comprehensive benchmark for evaluating intrusion detection systems within Industrial Internet of Things (IIoT) environments. This dataset comprises both benign and malicious network flows that replicate communications typical of industrial control systems. Each flow is characterized by a combination of numerical and categorical features. For this study, we have identified 83 pertinent statistical features to be utilized in the classifiers for intrusion detection systems. These features encompass flow-based metrics, including duration, byte count, and packet count, as well as flag-based attributes such as counts of FIN, SYN, and RST. Additionally, we have included temporal characteristics, including inter-arrival intervals and durations of active and idle states.

TENSORFLOW

The foundational deep learning models were implemented utilizing **TensorFlow**[7], TensorFlow, Google's open-source machine learning framework, is known for its scalability and modular architecture. Version 2.19 was chosen for its compatibility with Python 3.12 and support for eager execution, simplifying debugging and experimentation.

KERAS

The **Keras**[8]**API**, built on TensorFlow, simplifies the creation and training of neural networks through a modular and user-friendly approach. By encapsulating low-level processes, Keras allows for quick development of deep architectures with straightforward Python code. Its integration with TensorFlow ensures easy access to model training, loss functions, optimizers, and serialization utilities, combining performance and flexibility for deep model creation.

PANDAS

Pandas[9] library is a robust Python toolkit designed for data manipulation and analysis. It offers flexible data structures, such as DataFrames and Series, which

[6] "UC Irvine Machine Learning Repository," [Online]. Available: https://archive.ics.uci.edu/dataset/942/rt-iot2022.
[7] "TensorFlow," Google, [Online]. Available: https://www.tensorflow.org/.
[8] "TensorFlow-Keras," Google, [Online]. Available: https://www.tensorflow.org/guide/keras.
[9] "Pandas," [Online]. Available: https://pandas.pydata.org/.

facilitate the efficient handling of structured tabular data. In the realm of intrusion detection systems, pandas is widely utilized for various preprocessing tasks, including filtering, merging, encoding, and cleaning of datasets.

NUMPY

NumPy[10]is a fundamental scientific computing library that provides high-performance support for multi-dimensional arrays and matrices. It is engineered for efficient numerical operations, offering a comprehensive suite of mathematical functions that operate at the array level. Within IDS workflows, NumPy plays a vital role in data transformation, feature normalization, and vectorized computations throughout both the training and inference phases.

REQUEST

The **Requests**[11]is a library is a widely utilized HTTP client library in Python, recognized for its simplicity and clear, human-readable syntax. It enables effective communication with web services through standard HTTP methods, including GET and POST. In the current implementation of the Intrusion Detection System (IDS), the `requests` library is employed to transmit JSON-formatted feature vectors from the client script to the Flask API's /predict endpoint.

JOBLIB

Joblib[12] is a Python library that is specifically designed for efficient serialization and persistence of Python objects, with a particular focus on those that require significant computational resources to generate, such as machine learning models and data preprocessing artifacts. It is optimized for handling large NumPy arrays, ensuring high performance and reliability in data management tasks.

PYSHRAK

PyShark[13] is a Python wrapper for the popular Wireshark packet analyzer, built on top of the tshark command-line tool. It provides programmatic access to network packets, allowing for live or offline capture and parsing of network traffic.

DATETIME

Datetime[14] module is a standard library in Python that offers a range of classes for the manipulation of dates and times. It facilitates various operations, including timestamp formatting, duration calculation, and time-based filtering, thereby enhancing the efficiency of time-related programming tasks.

[10] "Numpy," [Online]. Available: https://numpy.org/.
[11] "Pypi," [Online]. Available: https://pypi.org/project/requests/.
[12] "JobLib," [Online]. Available: https://joblib.readthedocs.io/en/stable/.
[13] "Pypi," [Online]. Available: https://pypi.org/project/pyspark/.
[14] "Pypi," [Online]. Available: https://pypi.org/project/DateTime/.

SUBPROCESS

Subprocess[15] module is an integral component of Python's standard library, designed to facilitate the execution of shell commands and external programs directly from within a Python script. It is effectively utilized to launch various tools such as tcpdump, tshark, or custom scripts within distinct system-level processes.

TIME

The **Time**[16] module is an integral part of the Python standard library, offering essential functions for system-level timekeeping. It facilitates the generation of timestamps, management of sleep intervals, and precise measurement of time, thereby supporting various time-related operations in programming.

OS

The **OS**[17] module offers a portable solution for accessing operating system-dependent functionalities, encompassing file path manipulation, environment variable access, and process control.

CLASSICAL MACHINE LEARNING: scikit-learn

Traditional machine learning models were implemented in conjunction with deep learning techniques utilizing scikit-learn, a widely recognized Python library that offers efficient tools for data mining and analysis. The (SVM) and Random Forest Classifier (RFC) were selected as benchmarks for comparison against the (CNN) model. Furthermore, to ensure that all input characteristics contributed equally during model training, feature normalization was conducted using the StandardScaler tool from the scikit-learn library. Also, the target variable (Label) was recorded in a separate column and was encoded as an integer ranging from 0 to 11, corresponding to one of the twelve predefined classes, such as Normal, Port Scan, DoS, and Replay. This encoding process was conducted using scikit-learn's [18]Label Encoder, thereby ensuring compatibility with all classifiers, regardless of their implementation.

VISUALIZATION

The visualizations were created using the Matplotlib[19] and Seaborn[20] tools. Matplotlib, a low-level plotting toolkit, enables the production of detailed figures, while Seaborn provides high-level APIs for statistical visualizations. Together, these tools facilitated the generation of clear, publication-quality graphics that contributed to the interpretation of evaluation data.

---

[15] "Python," [Online]. Available: https://docs.python.org/3/library/subprocess.html.
[16] "Python," [Online]. Available: https://docs.python.org/3/library/time.html.
[17] "Python," [Online]. Available: https://docs.python.org/3/library/os.html.
[18] "SciKit-Learn," [Online]. Available: https://scikit-learn.org/stable/.
[19] "Matplotlib," [Online]. Available: https://matplotlib.org/.
[20] "Seaborn," Pydata, [Online]. Available: https://seaborn.pydata.org/installing.html.

REAL-TIME INFERENCE & Flask API

FLASK

Flask[21]**,** a Python-based open-source microframework, is utilized for the creation of RESTful web applications and APIs with minimal overhead. Its selection for this project can be attributed to its simplicity, modularity, and extensive community support. The fast-paced employment of Python-based inference services is facilitated, thereby eliminating the need for complex application servers or external dependencies. Its asynchronous features and lightweight architecture, offer ideally suited for the deployment of the IDS prediction endpoint on local or containerized systems.

API

Within the Flask application, a dedicated route (/predict) was developed to receive HTTP POST requests containing 83 normalized input features formatted in **JSON**. This **API** endpoint was implemented in the .py file, which hosts the primary server logic.

FUTURE CONTAINERIZATION

Initially, the Flask API was operated as a standalone Python program on localhost, utilizing Python 3.12. This approach provided sufficient performance for development and local testing. However, to facilitate preparation for deployment in modular or cloud environments, the API was designed with containerization considerations in mind.

DOCKERFILE

A Dockerfile has been created to package the Flask server into a Docker container. This container exposes port 5000, which is designated as Flask's default port, and encompasses all dependencies and scripts required for model loading and inference. The inference module is enabled for integration into larger Network Function

MININET

It should be noted that compatibility was established solely with Python 3.7 for Ryu. As a result, a dedicated virtual environment was created to operate the controller, thereby preventing any interference with other system components that required newer versions of Python. This separation of concerns was implemented to ensure stability and version isolation between the SDN control logic and the AI-based inference pipeline.

[21] "Flask," [Online]. Available: https://flask.palletsprojects.com/en/stable/installation/.

MINIEDIT

The Mininet emulator was utilized to replicate the network structure and traffic characteristics of a realistic 5G/IoT scenario. Mininet, a widely recognized network emulation tool, was employed to generate virtual network topologies using lightweight Linux containers and virtual Ethernet connections. The deployment of entire (SDN) topologies, along with hosts, switches, and controllers, was facilitated in real time on a single machine.

PACKET CAPTURE: tcpdump

Tcpdump[22]is regarded as a sophisticated, low-level packet analyzer that intercepts and logs network packets in .pcap format sent across the specified interfaces. The captured .pcap files were subsequently utilized for offline or semi-real-time feature extraction, enabling the validation of the IDS model against recorded traffic. This approach afforded a controlled testing loop wherein known traffic patterns could be replayed and analyzed under various inference settings.

NETWORK TRAFFIC

WIRESHARK

Wireshark[23] is an open-source network protocol analyzer widely recognized for its robust capabilities in capturing, decoding, and visualizing packet-level traffic across a wide range of network protocols.

CICFLOWMETER

While Wireshark provides packet-level visibility, machine learning models typically require flow-level statistical features such as average packet size, duration, inter-arrival time, and flag counts. For this purpose, CICFlowMeter[24] was adopted as the core tool for automated extraction of flow-based features from .pcap files or live network interfaces.

[22] "Medium," [Online]. Available: https://medium.com/@prateekdalal01/use-of-tcpdump-command-in-ubuntu-18-x-x-to-22-x-x-to-inspect-and-capture-network-packet-data-96c8453c54a3.

[23] "Wireshark," [Online]. Available: https://www.cherryservers.com/blog/install-wireshark-ubuntu.

[24] "Cicflowmeter," Pypi, [Online]. Available: https://pypi.org/project/cicflowmeter/.

IMPLEMENTATION

3.1 Chapter Scope and Boundaries

This chapter provides a systematic and detailed explanation of the implementation of the proposed IDS. Its fundamental purpose is to serve as a comprehensive technical blueprint, detailing the translation of the theoretical constructs and architectural paradigms discussed in previous chapters into a functional, working system. The scope of the discussion herein is intentionally and strictly confined to the engineering processes, design choices, and structural components that constitute the system. This includes a detailed account of the development environment, the layered system architecture, the data and feature engineering pipeline, the procedures for model development and artifact management, and the architecture of the real-time inference service. Furthermore, the construction of the SDN testbed, the implementation of diagnostic and observability hooks, the formalization of operational runbooks, and the incorporation of security hardening measures are all documented within these boundaries.

In adherence with the principles of scientific discipline and reproducibility, the documentation of the system's implementation has been strictly separated from the analysis of its empirical performance. In experimental research, it is imperative to first establish a clear description of the experimental setup and procedure before presenting and interpreting the results obtained from that procedure. In this context, the implemented system itself represents the experimental setup. By dedicating this chapter exclusively to its construction, a stable and well-defined foundation is established, upon which the performance claims of the subsequent Evaluation chapter can be reliably tested and validated. This approach ensures that the work is both reproducible and falsifiable; an independent researcher, in principle, should be able to reconstruct the system as described herein and verify the empirical results presented later.

Consequently, this chapter will not contain any discussion of quantitative outcomes. All performance-related metrics including but not limited to model accuracy, precision, recall, F1-scores, the analysis of confusion matrices, measurements of inference latency, and system throughput benchmarks are explicitly deferred to the Evaluation chapter. This methodological purity prevents the conflation of implementation details with performance artifacts, a common issue that can obscure the root causes of both successes and failures in system behavior. For example, a discussion of model latency is meaningless without the foundational context of the inference API's architecture and the hardware environment, which are established here. Similarly, an analysis of a confusion matrix is only valuable after the processes of data preparation, feature engineering, and model training have been fully elucidated. The exposition that follows is therefore structured to build a bottom-up understanding of the system. It will

commence with the foundational setup of the software and hardware environment, followed by an examination of the high-level, four-layer architecture and the contractual interfaces that govern their interaction. The core of the data handling logic is then detailed in the section on the feature engineering pipeline, which defines the canonical data schema that underpins the entire system. Subsequent sections will document the creation of the machine learning artifacts, the design of the real-time inference API, and the construction of the SDN testbed for traffic generation, and the operational frameworks for testing, deployment, and security. By adhering to these clearly defined boundaries, this chapter provides the necessary and sufficient context upon which the empirical findings of the subsequent chapter are built and must be interpreted.

The architectural framework of the proposed (IDS) was deliberately engineered using a layered and modular paradigm. This design philosophy, which emphasizes the principle of "separation of concerns," was deemed essential for managing the system's complexity and ensuring its long-term viability. The system is logically decomposed into distinct functional blocks that handle the end-to-end process, from traffic generation to final prediction. The operational flow of data through this architecture is a sequential and automated process, as illustrated in Figure 1.

The process is initiated within the Simulation Environment, which was constructed using Mininet. This environment contains three virtual hosts, each with a distinct role: an Attacker host to generate malicious activity, a Victim host to serve as the target, and a Collector host for generating benign traffic and storing data. All traffic generated by these hosts is directed to a virtual L2 Switch. This switch serves two critical functions. Moreover, it acts as the primary data source for the IDS pipeline by forwarding packets between the hosts according to the rules established by the control plane. All network traffic passing through the switch is mirrored to a capture process. This is handled by `tcpdump, which ingests the raw network traffic and serializes it into `.pcap` files. The raw packet data is then passed to the Feature Extraction module. This component processes the `.pcap` files, aggregates packets into bidirectional flows, and computes a high-dimensional vector of statistical features for each flow. The output, a set of structured Features, is then sent to the Flask API.

The Flask API acts as the gateway to the analytical core of the system. It exposes an endpoint that receives the feature vectors and forwards them to the Trained Model for inference. The model, which has been previously trained and loaded into memory, processes the features and outputs a prediction. This final stage of the process, termed Model Label Prediction, represents the system's classification of the network flow as either benign or a specific type of attack.

Operating in parallel to this data plane activity is the Control Plane, managed by the Ryu framework. The L2 Switch communicates with the Ryu application, which acts as the Controller. The controller is responsible for installing the necessary flow rules in the switch to manage packet forwarding, thereby governing the overall behavior of the network. This separation of the control and data planes is a fundamental characteristic of the SDN architecture employed in this research. This strict separation of the control and data planes is a fundamental characteristic of the SDN architecture employed in

this research. The aforementioned architectural components and their sequential operational data flow are visually synthesized in Figure 1.

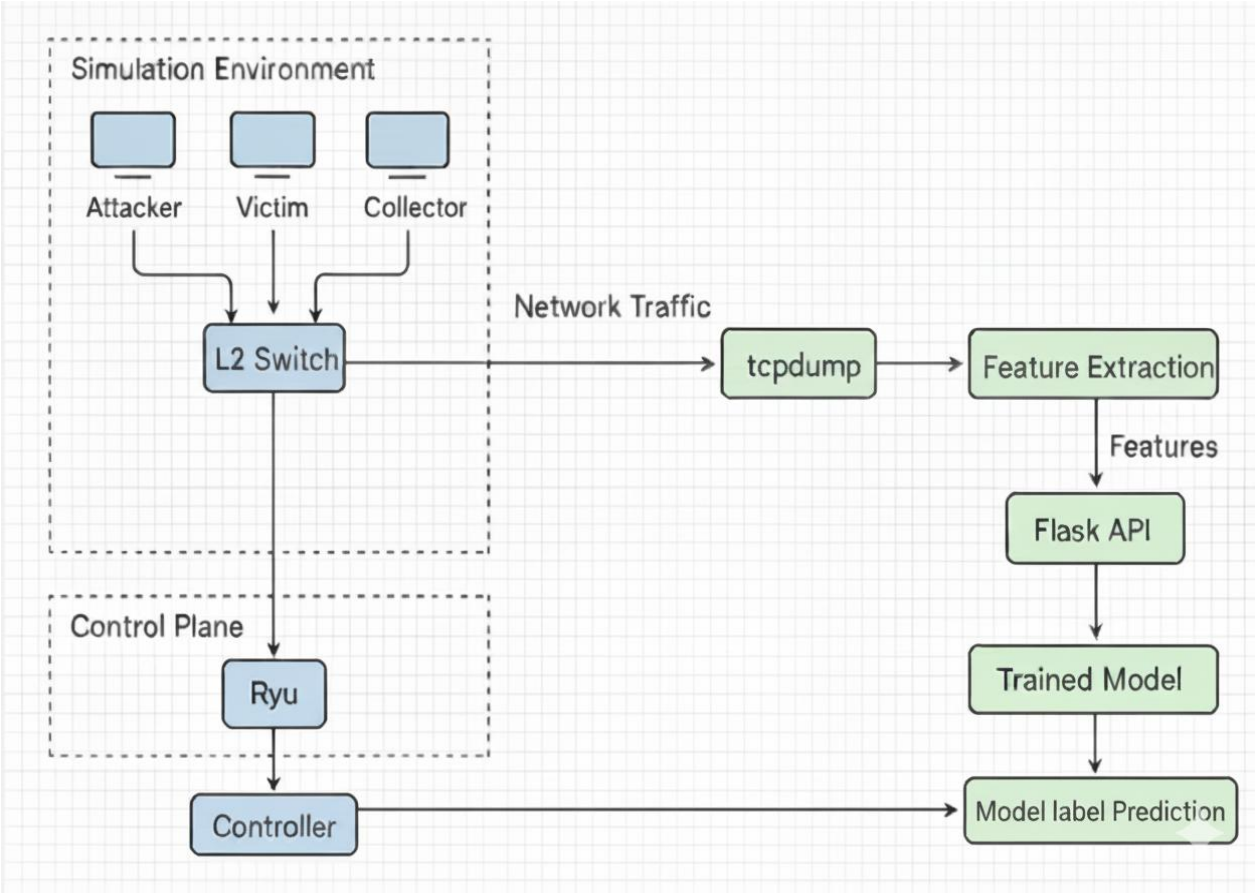


FIGURE 1 SIMULATION ENVIRONMENT

3.2 System Requirements, Architecture and Environment Setup

The development, training, and operational execution of the proposed IDS were conducted within a precisely defined and controlled hardware and software environment. Establishing this stable baseline was a foundational requirement to ensure the consistency, verifiability, and reproducibility of all experimental procedures and results. This section provides a detailed account of the specific hardware platform utilized, the operating system and its configuration, the strategic management of disparate Python environments, and the logical organization of the project repository and its critical artifacts. The specific hardware configuration, notably the absence of GPU acceleration which forced the system to default to CPU-based execution, is explicitly verified by the system's initialization logs, as depicted in Figure 2.

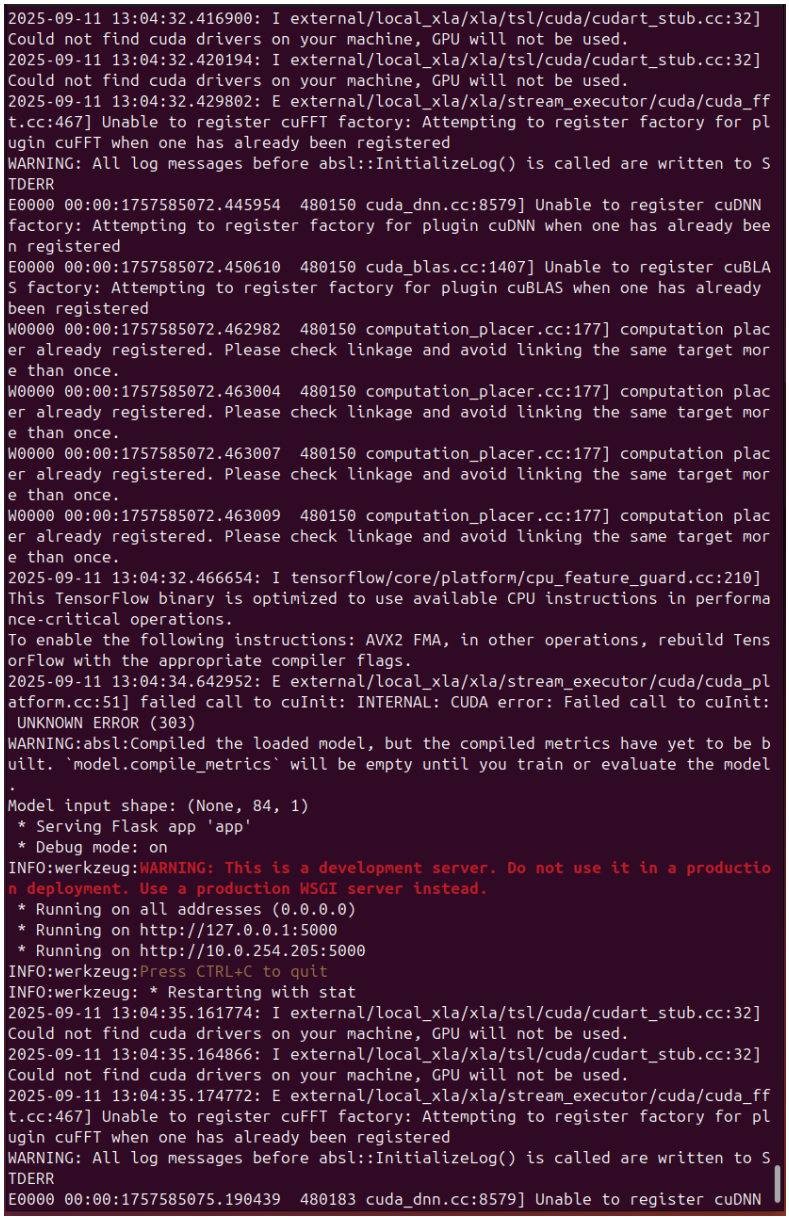


FIGURE 2TENSORFLOW INITIALIZATION LOG SHOWING THE ABSENCE OF CUDA DRIVERS, RESULTING IN THE SYSTEM DEFAULTING TO CPU-BASED EXECUTION FOR ALL TRAINING

This configuration establishes a crucial performance baseline for the system, demonstrating its viability on standard, non-specialized hardware. The TensorFlow initialization logs confirmed the absence of CUDA drivers, resulting in the system defaulting to CPU-based execution for all training and inference tasks. While the architecture is fully compatible with GPU acceleration, which would be recommended for a high-throughput production deployment, its functionality and effectiveness were first validated in this more common and accessible hardware environment.

The operating system chosen for all development and deployment was Ubuntu 22.04 LTS. This Long-Term Support release of the Linux distribution was selected for its well-established stability, extensive community support, and robust Advanced Package

Tool (APT) package management system. These characteristics were vital for simplifying the installation and management of the diverse array of open-source tools and libraries required for this research, including networking utilities, SDN frameworks, and machine learning dependencies.

3.2.1 Python Environment and Dependency Management

The Python programming language was employed as the primary development language for the entire project. A critical technical challenge that emerged during the initial setup phase was a significant version incompatibility between the modern machine learning (ML) software ecosystem and the legacy SDN control plane framework. To resolve this conflict while maintaining the stability of both systems, a strategic decision was made to maintain two separate and fully isolated Python environments using virtual environments.

1. Machine Learning Environment (Python 3.12): The core ML pipeline which includes all data preprocessing scripts, model training and validation routines, and the Flask-based inference API was developed using Python 3.12. This modern version of the language was selected to leverage its significant performance enhancements and to ensure full compatibility with the latest releases of the TensorFlow, scikit-learn, and Pandas libraries. This approach resulted in the ability to benefit from the most up-to-date features and optimizations in the data science ecosystem.
2. SDN Controller Environment (Python 3.7): The Ryu SDN controller, a key component of the network testbed, was found to have a strict dependency on legacy library versions that were not compatible with Python 3.12. To accommodate this, a separate and dedicated virtual environment was configured with Python 3.7. This environment was used exclusively for operating the Ryu controller and its associated network emulation scripts. This strategic separation of environments, while introducing a minor layer of operational complexity, was essential for the overall success of the implementation. It definitively prevented dependency conflicts that would have otherwise rendered the system inoperable and ensured a stable, version-isolated runtime for both the networking and machine learning subsystems.

3.2.3 Repository and Artifact Layout

To ensure clarity, maintainability, and ease of navigation, all project artifacts were organized in a standardized directory layout within the main repository. This convention-based approach simplified the configuration of scripts and services, as critical file paths could be predicted and standardized rather than hardcoded. The primary directories and their contents are described in Table 3.1. This structured layout was fundamental to the operational runbooks and the management of model artifacts, as will be detailed in subsequent sections.

TABLE 1 REPOSITORY DIRECTORY STRUCTURE AND ARTIFACT LOCATIONS

/models	Contains all serialized machine learning artifacts. This includes the trained model files (`.h5`), the fitted scaler object (`.pkl`), and the human-readable label map (`.Json`)
`/data	Serves as the central repository for all datasets, captured network traffic (`.pcap` files), intermediate CSV files, and all run-time logs (e.g., `results.log`)
/scripts	Contains all executable Python scripts responsible for model training, validation, real-time prediction, and data simulation
/utils	Contains supplementary utility scripts and configuration files, including the Python scripts for the Mininet topology and the Ryu controller application

3.3 Overall System Architecture

The architectural framework for the proposed IDS was deliberately engineered using a layered and modular paradigm. This design philosophy, which emphasizes the principle of "separation of concerns," was deemed essential for managing the system's complexity and ensuring its long-term viability. The system is logically decomposed into four distinct layers, each tasked with a specific and self-contained stage of the end-to-end intrusion detection process. This process begins with the controlled generation of network traffic and culminates in the real-time classification of that traffic for potential threats. The high-level architecture and the sequential progression of data through these distinct layers are visually summarized in Figure 3.

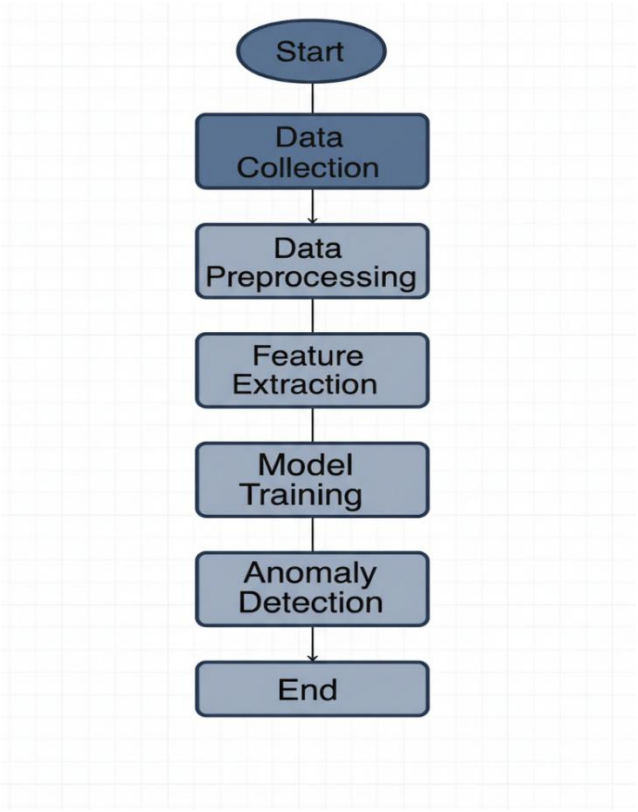


FIGURE 3 HIGH-LEVEL SYSTEM ARCHITECTURE AND DATA FLOW

A "contract-first" design methodology was adopted to govern the interactions between these layers. For each boundary, an explicit and versioned contract was defined, specifying (a) the precise data payload format, (b) the mandatory ordering and physical units of features, (c) the permissible error codes for invalid inputs, and (d) the expected side effects, such as structured logging. Constraining the interfaces with this level of detail, rather than the internal implementations of each layer, preserved system evolvability while guaranteeing compatibility across components. This approach further enabled the creation of lightweight test doubles and mocks for continuous integration, where the behaviour of an upstream or downstream component could be emulated without requiring the deployment of the full stack. This proved invaluable for isolated testing and straightforward fault localization.

The primary rationale for this layered approach was to create a highly decoupled system, where in each component operates with minimal dependency on the internal workings of the others. In a monolithic architecture, a change in one part of the system, such as an update to the ML model, could necessitate significant modifications across the entire codebase, leading to a brittle and difficult-to-maintain system. By contrast, the modular design adopted here ensures that each component can be developed, tested, and maintained independently. For instance, the SDN testbed (Layer 1) can be reconfigured with different network topologies or traffic generation tools without affecting the data ingestion or inference services. Similarly, the ML model within the inference service (Layer 3) can be retrained or even replaced with an entirely different

type of model (e.g., a Recurrent Neural Network (RNN)) without requiring further changes to the network infrastructure or the data capture mechanisms.

Consequently, this modularity indicates substantial practical benefits for both the development and operational phases of the system's lifecycle. During development, it allowed for parallelized work on different components and simplified the process of unit testing, as each layer could be evaluated in isolation. For example, the inference API could be thoroughly tested using pre-recorded data files, independent of the live network emulation environment. From an operational perspective, this decoupling is crucial for the extensibility and future-proofing of the system. It allows for seamless future upgrades to be implemented with minimal disruption. Updated or entirely different ML models can be integrated, the feature extraction process can be adapted for different datasets or network protocols, and the entire system can be migrated to different network infrastructures, such as a physical testbed or a cloud environment, by modifying or replacing individual layers without necessitating a complete system overhaul. This architectural foresight ensures the long-term relevance and adaptability of the IDS in the face of rapidly evolving network technologies and an ever-changing cybersecurity threat landscape.

3.3.1 Architectural Layers and Data Flow

The operational flow of data through the system's four layers is a sequential and automated process. Each layer performs a specific transformation on the data before passing it to the next, ensuring that the information is in the correct format and state for each stage of processing.

3.3.1.1 Layer 1: Traffic Capture and Flow Aggregation

This layer was responsible for transforming raw packet streams into structured flow records. In the simulated environment, packets were mirrored at the Open vSwitch data plane and serialized to rotate capture files in the .pcap format. Flow keys were defined over the canonical five-tuple (source and destination IP addresses, source and destination ports, and the transport protocol). Bidirectional aggregation was then applied with deterministic timeouts to group related packets into a single flow context. The output of this layer consisted of tabular flow features computed per direction and over the joint flow context, providing a rich, comprehensive representation of the network conversation.

3.3.1.2 Layer 2: Feature Normalization and Schema Enforcement

This layer imposed a canonical view over the potentially heterogeneous flow features produced by the upstream tools. A strict, versioned schema, referred to as EXPECTED_84, was enforced. This schema defines: **(a)** unique column names, **(b)** data types and physical units, **(c)** synthesis rules for derived quantities, **(d)** deterministic defaults for sparse indicators, and **(e)** a mandatory column order. Any records that failed this schema validation were rejected with explanatory diagnostic messages to prevent silent data corruption or model drift. The final output of

this layer was a stream of 84-dimensional, strictly ordered feature vectors, precisely conforming to the contract expected by the downstream scaler and model.

3.3.1.3 Layer 3: Model Inference Service

The Model Inference Service was implemented as a stateless HTTP API, backed by a persisted scaler object and a selected classifier model. The service was designed to accept a JSON payload containing exactly 84 raw (unscaled) features. Upon receiving a request, the service applied the feature scaling transformation on the server side and then executed the model inference. The response payload included the predicted class identifier, its mapped human-readable label, and a calibrated confidence score. To ensure robustness, the service implemented strict enforcement of input size, numeric data types, and guards against non-finite (NaN/Inf) values prior to executing the inference step.

3.3.1.4 Layer 4: Real-Time Client and Orchestration

This layer served as the orchestrator of the real-time pipeline. It continuously monitored a single live CSV stream, which acted as a unifying data bus. As new lines were appended to this stream by the flow layer, they were parsed, validated, and converted to the EXPECTED_84 order before being posted to the Model Inference Service API. A back-pressure-aware processing loop with bounded retries was implemented to ensure that transient API disruptions did not corrupt the input stream. All interactions were logged with high-precision timestamps to support subsequent correlation with controller events or packet capture times. The specific software components utilized across these four architectural layers, along with their primary roles and justifications for selection, are systematically detailed in Table 2.

TABLE 2 KEY SOFTWARE COMPONENTS AND THEIR ARCHITECTURAL ROLES

Layer Component	Primary Role and Justification
Traffic Capture & Flowization	Mininet:** Emulates network topology, chosen for its lightweight and rapid prototyping capabilities
Ryu Controller	Manages SDN flows, selected for its Python-native implementation and modularity
hping3	Generates malicious traffic, chosen for its versatility in crafting custom packets
Feature Normalization & Schema Enforcement	tcpdump: Captures raw network packets, selected for its high performance and low overhead
CICFlowMeter	Extracts flow-based features from .pcap files, chosen for its comprehensive feature set

Pandas	Manages data in DataFrames; used for data cleaning, alignment, and manipulation
Model Inference Service	TensorFlow/Keras:** The core framework for building and training the 1D-CNN and Autoencoder models
Flask	Hosts the real-time prediction API, chosen for its lightweight and simple deployment
Scikit-learn	Provides the StandardScaler for feature normalization and baseline models (SVM, RF)
Real-Time Client & Orchestration	Docker:** Containerizes the entire application, ensures reproducibility and portability
Python Scripts	Custom scripts for real-time listening, prediction, and orchestration of the pipeline

3.4 Data and Feature Engineering Pipeline

The data and feature engineering pipeline was designed to be a deterministic and robust system for transforming heterogeneous, raw traffic observations into a single, authorised representation suitable for reliable consumption by the inference service. The central design decision in this pipeline was the adoption of a fixed, 84-dimensional feature schema, referred to as EXPECTED_84. This schema serves as a strict contract between all upstream data capture processes and the downstream model-serving layer. By enforcing a consistent feature order, data type system, and set of derivation rules, the pipeline was engineered to prevent the subtle but significant data mismatches that can occur when different datasets or tools are used over time. The entire data path, from packet capture to server-side scaling, was implemented to be deterministic and idempotent. This ensures that a given set of raw packets will consistently produce the exact same feature vector, a property that is fundamental to the reproducibility of the research.

The overall workflow of the data pipeline, encompassing both the offline model training phase and the online live detection phase, is illustrated in Figure 4.

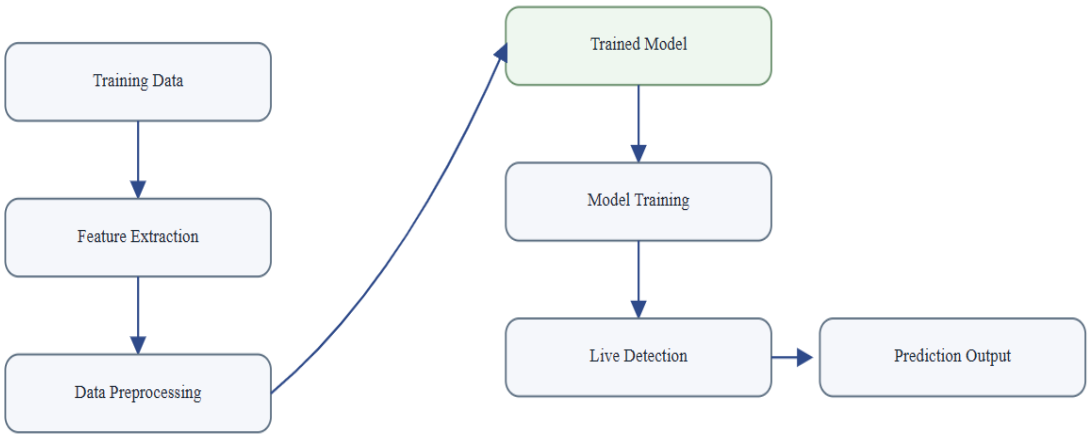


FIGURE 4 THE DATA PROCESSING AND MACHINE LEARNING PIPELINE FLOWCHART.

The engineering of the pipeline was guided by five core objectives: determinism, completeness, traceability, isolation of concerns, and backward compatibility. Determinism was prioritized to ensure that the same set of input measurements would always yield the identical authorised vector when processed under the same schema version. Completeness was achieved by guaranteeing that all 84 features required by the trained model were present at inference time, whether as directly captured values, deterministically derived quantities, or policy-based defaults for sparse indicators. Traceability was provided through a clear provenance for each feature, which explicitly states whether its value was observed, derived, or defaulted. Isolation of concerns was maintained by decoupling the packet and flow capture stages from the canonicalization and model-serving stages, which in practice means that different feature extraction tools can be used interchangeably. Finally, backward compatibility was preserved by designing the system to accept legacy data extracts and elevate them to the EXPECTED_84 schema without altering the trained model, provided that the semantic meaning of each feature was respected. These objectives were selected to ensure the system remains robust and survivable under real-world operational conditions.

The EXPECTED_84 schema itself was established as a normative list of 84 features with a strict, versioned order and a stable semantic meaning. While the implementation treats the feature vector as a flat list for performance reasons, the schema is conceptually organized into several logical groups. The first group encompasses the fundamental metadata of the flow, including flow start and end timestamps and the corresponding duration. A second group covers the core directional traffic statistics, such as forward and backward packet and byte counts, along with summary statistics for TCP segment sizes. A third group captures key transport-level protocol indicators, primarily consisting of counts of specific TCP flags like SYN, ACK, and PSH. A fourth group accommodates size and rate metrics that are computed per flow, typically derived from the directional counts and observed duration. The final set includes derived aggregate

statistics that provide averages and variances when only first-order aggregates are available from an upstream source. An authoritative, multi-page table that specifies the index, authorised name, data type, physical unit, and provenance for each of the 84 features is a critical piece of documentation for this system and is considered the single source of truth for the schema. The sequential stages of this canonicalization process, transforming raw network traffic into the final inference output, are visually delineated in Figure 5.

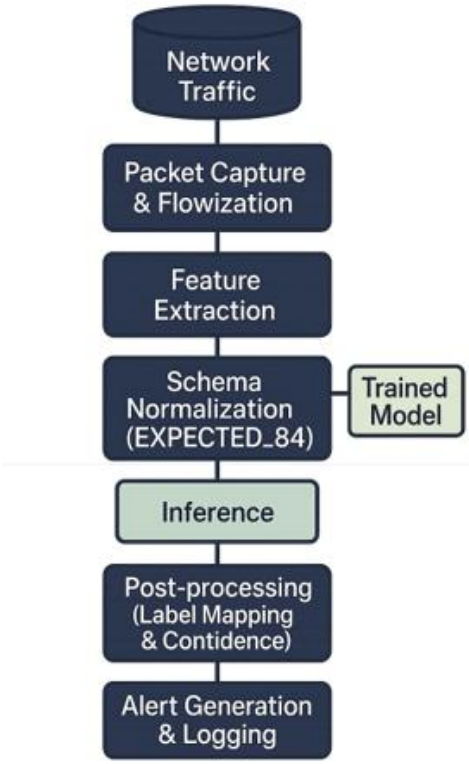


FIGURE 5 END TO END POINT SCHEMA

The process of converting raw, potentially heterogeneous feature data into the strict EXPECTED_84 format is referred to as canonicalization. This procedure is performed by a single, idempotent function, meaning that it can be safely applied multiple times with the same outcome as the first application. This function accepts either a batch of rows for offline processing or an incremental row from the live feed. The procedure proceeds through a series of ordered stages. In the first stage, the column headers of the input data are normalized by converting them to lowercase, trimming whitespace, and replacing known aliases with their authorised counterparts. In the second stage, data types are harmonized, with numeric fields that arrive as strings being parsed to their correct numeric types. A strict policy is enforced where the presence of non-numeric tokens in a numeric column causes the entire row to be quarantined for later inspection. In the third stage, derived metrics, such as averages, are synthesized from the available primitives. In the fourth stage, any columns from the `EXPECTED_84` schema that are missing in the input data are inserted with their defined policy defaults. In the fifth stage, the DataFrame is reindexed to enforce the

strict, authorised order of the `EXPECTED_84` schema, and the data types of all columns are cast to their final, agreed-upon types. In the final stage, a series of integrity checks are executed to confirm the vector length, reject non-finite values, and enforce non-negative constraints for counters and durations. This multi-stage, deterministic procedure ensures that only clean, valid, and consistently formatted data is ever passed to the model inference layer.

3.5 Model Development and Artifact Management

The development of the machine learning components was conducted with two primary objectives in mind. The first was to establish a dependable and reproducible pathway from raw, pre-processed flows to an inference-ready feature representation. The second was to guarantee that the artifacts produced, namely the trained models and their associated preprocessors could be deployed, audited, and reproduced without ambiguity. The present section therefore confines itself to the design choices, training and serialization procedures, file layouts, and compatibility rules that govern the creation and management of these artifacts. The empirical performance of these models is deferred to the Evaluation chapter.

A reproducible data-splitting protocol was implemented at the outset of the development process to prevent accidental data leakage and to stabilize all subsequent experiments. The fully pre-processed dataset was partitioned into training, validation, and test subsets in fixed proportions of 80%, 10%, and 10%, respectively. The indices for these splits were generated once, written to disk as CSV files, and thereafter treated as immutable references for the remainder of the development lifecycle. This approach ensured that the different model families, both neural and classical, would be exposed to the identical set of evidence during the training and optimization phases. This practice was critical for ensuring that any observed differences in performance were attributable to the models themselves and not to artifacts of fluctuating data boundaries. The class distribution across these splits was inspected before the indices were frozen, and modest adjustments were applied to avoid a pathological scarcity of specific attack types in any one subset. The indices and the statistical signature of each split were stored under version control in the project's data directory to facilitate later audits and ensure long-term reproducibility.

The preprocessing contract was organized into two distinct layers that were executed in a fixed, sequential order. The first layer enforced strict conformance to the authorised `EXPECTED_84` schema, as detailed in the preceding section. At this layer, feature names were normalized, derived attributes were computed deterministically from their primitives, default values were injected for any sparse counters, and the final column order was applied. The insistence on a strict column order may appear pedantic; however, it proved essential in practice, as even small deviations in feature indices or data types were observed to generate subtle but significant errors once the data was assembled into tensors for inference. The second layer of the contract applied statistical standardization using a StandardScaler object from the Scikit-learn library. This scaler was fitted exclusively on the training subset to prevent data leakage. The resulting scaler binary was then serialized to disk and treated as a first-class artifact, equal in status to

the trained models themselves. This status was assigned because its internal parameters the means and variances of the features were intrinsically entangled with the model's learned weights. At inference time, it was mandated that raw flows first pass through the schema enforcement layer and then through this exact scaler before the resulting vector could be forwarded to the predictor. Any bypass of this two-stage protocol was considered a contract violation and was programmatically rejected by the system.

Three distinct model families were developed to provide complementary inductive biases and operational characteristics: a one-dimensional convolutional neural network (1D-CNN), a support vector machine with a radial basis function kernel (SVM-RBF), and a random forest (RF) classifier. The 1D-CNN offered a deployable DL baseline that was capable of capturing local interactions among contiguous features when the canonical ordering of the EXPECTED_84 schema was respected. Its implementation in TensorFlow/Keras permitted disciplined control over model capacity via adjustments to convolutional width, depth, and regularization, while remaining compatible with the CPU-only servers used in this research. The SVM provided a strong classical baseline that was well-behaved on standardized tabular inputs; this was particularly valuable during the early stages of development when the integrity of the preprocessing pipeline needed to be validated independently of the representation learning model. The random forest, finally, contributed a high degree of resilience against mild scaling anomalies and delivered interpretable feature splits that greatly facilitated failure analysis when intermediate runs presented unexpected behavior. It was found during the development process that this diversity of modeling styles accelerated debugging by allowing for cross-checks and verifications under identical input contracts.

The resulting artifacts were organized under an explicit and version-controlled directory layout to remove guesswork during deployment. The trained weights for the 1D-CNN were saved as a portable HDF5 (.h5) file, while the SVM and RF models were persisted as joblib pickle files. The scaler and the label map were stored alongside the models in the same directory. Each training run was designed to produce a small metadata package that included the hyperparameters used, the software environment (including library versions and the operating system), the fixed random seeds, and SHA-256 checksums for every binary artifact produced. These checksums served two critical purposes. First, they allowed an operator to programmatically verify that the files loaded by a containerized service matched the files that had been reviewed and approved for deployment. Second, they enabled a robust audit trail in case subsequent outcomes appeared inconsistent with earlier deployments, allowing for a definitive check against artifact drift. This rigorous approach to artifact management ensures that the foundation upon which the subsequent evaluation rests is stable, transparent, and reproducible.

3.6 Real-Time Inference & Deployment

The operational deployment of a trained machine learning model requires a robust, scalable, and efficient architecture for serving real-time predictions. For this thesis, a service-oriented architecture was adopted, wherein the trained IDS model was encapsulated and exposed as a network service. This approach was chosen to decouple

the analytical core of the system (the model) from the various client applications that might consume its predictions, such as the real-time traffic listener. This section provides a detailed examination of the design and implementation of the Flask-based inference API, the precise structure of the data payloads that form its contract, and the server initialization process.

3.6.1 The Inference Service Interface

The interface to the Model Inference Service (Layer 3) was defined in a deliberately conservative and explicit manner to ensure stability and to minimize ambiguity during the integration of different system components. The selection of the Flask micro-framework was predicated on its inherent simplicity and minimal operational overhead, which are defining characteristics of a micro-framework as opposed to a more comprehensive, full-stack framework. Its seamless integration with the Python-based machine learning ecosystem made it an ideal choice. A single primary endpoint, exposed at the route /predict, was designed to accept HTTP POST requests.

The service was engineered to be completely stateless, meaning each incoming request is processed independently without any reliance on the memory or context of past requests. This statelessness is a critical architectural choice, as it is a fundamental prerequisite for straightforward horizontal scaling. While more comprehensive frameworks such as Django or Java Spring offer extensive features, their complexity and resource footprint were deemed unnecessary for creating a dedicated, single-purpose inference service. With no shared state to manage, multiple instances of the API can be run in parallel behind a standard reverse proxy or load balancer to handle increased request volumes. This design avoids the need for complex state management mechanisms like session affinity or distributed state coordination, thereby simplifying the deployment and operational management of the service.

The use of a micro-framework like Flask facilitates the rapid development of architectural choice, as it is a fundamental prerequisite for straightforward horizontal scaling. With no shared state to manage, multiple instances of the API can be run in parallel behind a standard reverse proxy or load balancer to handle increased request volumes. This design avoids the need for complex and state such a service without the need for complex application servers or external dependencies. The asynchronous capabilities and lightweight nature of Flask make it an ideal mechanism like session affinity or distributed state coordination, thereby simplifying the deployment and operational management of the service. In addition to the primary choice for deployment in either local development environments or orchestrated, containerized production systems. The successful initialization and network reachability of this Flask-based API are visually confirmed by the console output presented in Figure 6

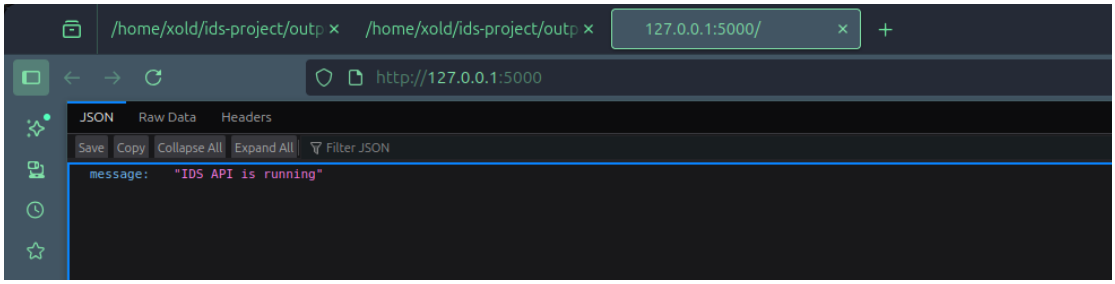


FIGURE 6 A SUCCESSFUL CONNECTION TO THE BASE ENDPOINT OF THE RUNNING FLASK API, CONFIRMING THAT THE SERVICE IS ACTIVE AND REACHABLE OVER THE NETWORK

A core architectural principle of the API prediction endpoint, auxiliary endpoints for integration tests and health checks were also provided, though these do not alter the primary data contract of the system is its statelessness. The service was engineered so that each prediction request is processed independently, without any reliance on the memory or context of past requests. This stateless design is a critical choice for ensuring scalability and fault tolerance. It allows for straightforward horizontal scaling, where multiple instances.

The contract for the `/predict` endpoint was constrained by design to enforce data "features," whose value is a real-valued array with an exact length of eighty of the API can be run in parallel behind a reverse proxy or load balancer to handle increased request volumes. In such a configuration-four elements. The server does not perform any implicit reordering of features or any best-effort imputation for missing elements. Before any scaling or inference is attempted, a strict validation to check performance on the incoming payload. The service first verifies, any available API instance can serve any request, eliminating the need for complex session affinity or shared state coordination between instances. This makes the system more resilient to failures, as the failure of a single instance does not impact the ability of others to continue processing requests.

3.6.2 API Contract and Data Flow

To ensure reliable and unambiguous communication between the client and the inference service, a strict and versioned contract was defined for the API's primary endpoint. A single endpoint, /predict, was implemented to handle all inference requests. Before any processing, a thorough validation of the incoming payload is performed. The service first confirms the array length, then rejects any non-numeric data types, and finally checks for the presence of non-finite (NaN or infinite) values. A defensive, "fail-fast" approach was adopted, wherein the server immediately returns a descriptive error for any malformed request. This approach of providing immediate and unambiguous feedback proved to be substantially more helpful in practice than quiet coercion (e.g., attempting to guess a missing value), as it encourages upstream client processes to correct their data generation and formatting logic at the source.

Once an incoming request has been successfully validated against the API contract, it is passed through a deterministic, server-side processing pipeline before the final inference is executed. This server-side logic ensures key named `"features"`. The value

associated with this key must be a real-valued array with an exact length of eighty-four elements, corresponding to the raw, unscaled features of a single network flow.

The first step in this process successful inference, the service returns a JSON object with a `200 OK` HTTP status code. This response contains the predicted class identifier as an integer, the corresponding human-readable label resolved from a persisted mapping file, and a scalar confidence score is the application of the normalization transform. This is handled by the `StandardScaler` object that was previously fitted on the training data and persisted as the `scaler_raw83.pkl` artifact. This scaler is loaded into memory once at the startup of the API representing the maximum posterior probability among the candidate classes.



FIGURE 7 A SUCCESSFUL PREDICTION CYCLE AS OBSERVED FROM THE FLASK API SERVER LOG, SHOWING AN INCOMING FEATURES.

The Inference Path :

1. Request Validation service to amortize the cost of disk I/O and deserialization. By centralizing the scaling logic on the server, the system guarantees that the exact same scaling parameters (i.e., the means and variances from the training set) are applied to every single prediction. Before any processing is attempted,

a validation of the incoming payload is performed. The service first confirms that the request contains a valid JSON object and that the `"features"` array is present. Moreover, it verifies that the array contains exactly eighty-four elements. This design prevents a common class of errors found in distributed systems where different clients might carry different or outdated versions of the scaler.

2. Data Integrity Checks: The elements of the feature array are then checked to ensure they are all numeric types and are finite (i.e., not NaN or infinite values). If any of these checks fail, the process is aborted, which would lead to inconsistent and unreliable predictions. The scaling step is performed within the request handler but does not acquire any global locks; it utilizes read-only state and pure functional operations so that multiple requests can be processed concurrently by different threads without contention.

3. Data Transformation: The validated list of features is converted into a NumPy array and reshaped to. The resulting scaled vector is then submitted to the active classifier. In the default configuration, the 1D-CNN model, which persisted as the `1dcnn_model.h5` artifact, is used for inference. This model the (1, 84, 1) tensor shape expected by the 1D-CNN model.

4. Into memory once during the service startup. The inference path itself consists of a single forward pass through the neuralServer-Side Scaling: The reshaped vector is then transformed using the `StandardScaler` object that was loaded into memory at the server's startup. This critical step ensures that the data fed to the model is normalized in exactly the same way as the training data.

5. Model Inference: The scaled tensor is passed to the `model.predict()` method of the loaded TensorFlow/ network, which produces a vector of probabilities for each of the possible output classes. An `argmax` operation is then performed on this vector to Keras model. This executes a forward pass through the neural network, which outputs a vector of twelve probability scores (one identify the class with the highest probability. The final human-readable label is then retrieved from a persisted `label_map.json for each class).

6. Post-processing and Response Generation: An `argmax` function is applied to the output vector to identify the index of the class with the highest probability. This integer index is then used to look up the human` file by looking up the predicted class index. This final step of using a mapping file, rather than embedding the labels within-readable class name from the loaded label map. The final JSON response is then constructed with the predicted label and the confidence the model itself, was a deliberate choice to ensure that the textual semantics of the class labels remain under explicit version control and can be updated without requiring the model to be retrained.

7. Logging: Finally, a structured log entry is created that records the timestamp, the outcome of the validation, and the final prediction, ensuring that all inference activities are auditable. The real-time client, also known as the predictor, serves as the primary orchestrator of the live inference pipeline. Its main responsibility is to monitor a continuous stream of network flow observations, transform them into the required format, and submit them to the inference service. This defensive, "fail-fast" approach, where malformed requests are rejected early with descriptive error messages, proved to be substantially more robust in practice than attempting quiet coercion or imputation of invalid data.

3.6.3 Artifact Management and Server Initialization

To ensure low-latency performance, all required machine learning artifacts are loaded into memory once when the Flask application is first started. This increases the cost of file I/O and deserialization, which would otherwise introduce significant latency if performed on a per-request basis. The handoff between the upstream feature extractor and the predictor is conducted through a single, append-only CSV file, which acts as a simple but effective unifying data bus.

To ensure low-latency performance, all required machine learning artifacts are loaded into memory once when the Flask application is first started. This amortizes the cost of file I/O and deserialization, which would otherwise introduce significant latency if performed on a per-request basis. The initialization process involves loading three key artifacts from the /models directory:

1. The trained 1D-CNN model from its .h5 file.
2. The fitted StandardScaler object from its .pkl file.
3. The integer-to-string label mapping from the label_map.json file.

This pre-loading strategy ensures that when a prediction request arrives, the server can immediately proceed with the validation and inference steps without incurring any disk access delays, which is critical for maintaining the system's real-time responsiveness.An analysis of the class distribution is particularly important for intrusion detection research because imbalanced datasets may influence model training and bias classification performance toward dominant traffic classes. Table 3 summarizes the number of instances associated with each traffic category used during the training and evaluation phases of the proposed intrusion detection system.

TABLE 3 DISTRIBUTION OF TRAFFIC PATTERNS IN THE RT-IOT2022 DATASET.

Category	Class Label	Number of Samples
	Attack Patterns	
	DOS_SYN_Hping	94,659
	ARP_poisioning	7,750
	NMAP_UDP_SCAN	2,590
	NMAP_XMAS_TREE_SCAN	2,010
	NMAP_OS_DETECTION	2,000
	NMAP_TCP_scan	1,002
	DDOS_Slowloris	534
	Metasploit_Brute_Force_SSH	37
	NMAP_FIN_SCAN	28
	Normal Patterns	
	Amazon-Alexa	86,842
	MQTT	8,108
	Thing_speak	4,146
	Wipro_bulb_Dataset	253

The implemented model is based on a one-dimensional Convolutional Neural Network (1D-CNN), which is particularly suitable for analyzing sequential feature vectors extracted from network traffic flows. The architecture consists of multiple convolutional layers responsible for identifying local feature patterns, followed by regularization mechanisms that reduce overfitting and fully connected layers that perform the final classification. Table 4 presents a detailed overview of the layers included in the implemented network architecture.

TABLE 4 ARCHITECTURE OF THE 1D-CNN MODEL.

Layer Type	Parameters	Output Shape
InputLayer	(84, 1)	(None, 84, 1)
Conv1D	64 filters, kernel_size=3, activation='relu'	(None, 82, 64)
MaxPooling1D	pool_size=2	(None, 41, 64)
Dropout	rate=0.25	(None, 41, 64)
Conv1D	128 filters, kernel_size=3, activation='relu'	(None, 39, 128)
MaxPooling1D	pool_size=2	(None, 19, 128)
Dropout	rate=0.25	(None, 19, 128)
Flatten	-	(None, 2432)
Dense	128 units, activation='relu'	(None, 128)
Dropout	rate=0.5	(None, 128)
Dense	12 units, activation='softmax'	(None, 12)

3.7 SDN Testbed for Traffic Generation and Capture

The SDN testbed was designed and implemented to achieve three primary objectives: first, to generate controllable and reproducible mixtures of benign and malicious network traffic; second, to capture the resulting raw packets at deterministic vantage points within the emulated data plane; and third, to transform these raw packet traces into the structured, flow-level features that conform precisely to the canonical EXPECTED_84 schema. A particular effort was made throughout the implementation of this testbed to ensure that all decisions were reproducible and that each resulting data artifact could be traced back to its originating process. This section documents, in implementation detail, how the network topology was constructed, how the Ryu controller was integrated, the mechanisms employed for stimulus generation and packet capture, and the design of the bridge that converts raw packet captures into normalized feature vectors.

3.7.1 Topology and Control Plane Integration

The emulated network environment was instantiated using Mininet, a network emulator chosen for its ability to create realistic virtual networks on a single host. By utilizing lightweight virtualization to run a real Linux kernel stack, Mininet ensures high-fidelity network behavior, which was a critical requirement for generating representative traffic data. The data plane of the testbed was implemented using Open vSwitch (OVS), a production-quality, multilayer virtual switch that provides the necessary programmatic interfaces for SDN control.

A simple, yet effective, network topology was constructed, consisting of a single core OVS switch that was used to aggregate all network traffic and provide a central interception point for packet capture. Three functional host roles were developed and connected to this switch: a set of "victim" hosts to serve as targets for malicious activity, a set of "attacker" hosts used to generate parameterized attack traffic, and a "collector" host designated for stable storage and the processing of captured data. To provide a

clearer illustration of the implemented experimental environment, the initialization process of the Mininet topology is shown in Figure 8.

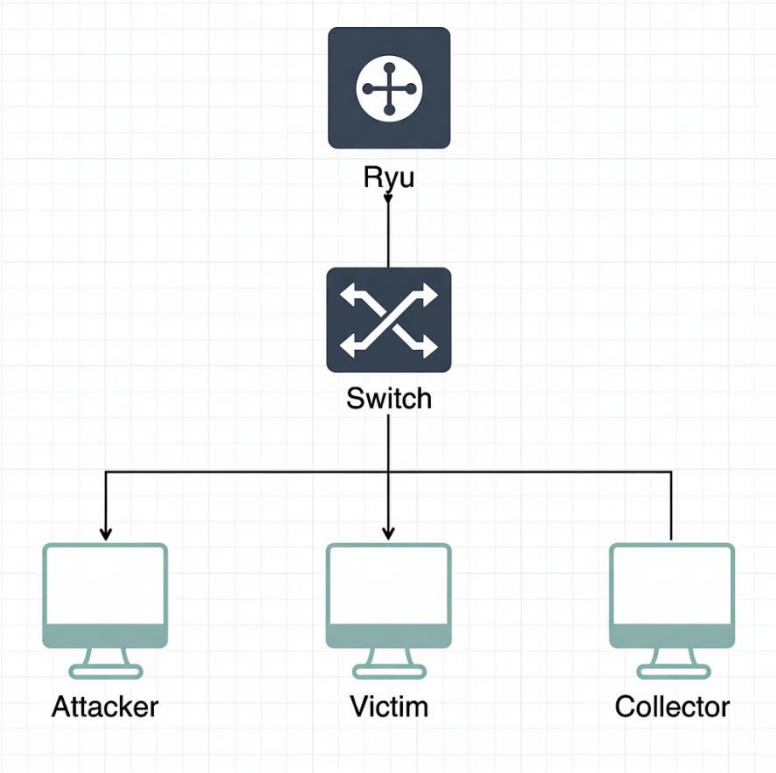


FIGURE 8: CONSOLE OUTPUT SHOWING THE CREATION OF THE MININET NETWORK TOPOLOGY

The control plane was managed by an external SDN controller, for which Ryu, a component-based SDN framework written in Python, was selected. The connection between the Ryu controller and the OVS data plane was established using a remote controller configuration with an explicit IP address and port. This approach was deliberately selected to eliminate the non-determinism that can often appear when in-process controllers are used within Mininet deployments. As detailed in Section 3.2.2, the Ryu controller was executed inside its dedicated Python 3.7 virtual environment to resolve its specific legacy dependencies.

The controller's configuration was intentionally kept minimal to ensure that the behavior of a realistically managed SDN network was faithfully replicated. A more detailed or "fully configured" controller might proactively install all possible flow paths at startup. In contrast, by limiting the initial bootstrapping rules to only what is necessary for basic network connectivity and Address Resolution Protocol (ARP), the controller is forced to reactively install specific flow paths as hosts communicate. This reactive behavior is characteristic of many real-world SDN deployments, and it was retained to ensure the traffic flows produced were representative of such an environment. No mitigation logic was embedded in the controller at this stage, as the primary objective was to produce and observe traffic flows, rather than to preemptively alter them.

3.7.2 Traffic Generation Scenarios

Network traffic for the experiments was produced by combining a baseline of benign background flows with a series of strictly parameterized malicious attacks. Against this benign baseline, a series of scripted attack scenarios were executed. These were launched from the "attacker" host and were targeted at the "victim" host, using well-established network testing tools such as hping3 and nmap. Each attack scenario was described by a compact run descriptor, which included the tool used, the transport protocol, the destination port, any key flags or options (e.g., the TCP SYN flag), and the desired rate or timing behaviour. This descriptor identifier was then carried through the entire pipeline, embedded in filenames and log entries, to maintain end-to-end traceability from the initial stimulus to the final inference result. To illustrate the execution of one of the malicious traffic scenarios within the experimental environment, Figure 9 presents the command-line execution of a **TCP SYN flood attack** initiated from the attacker host inside the Mininet environment.

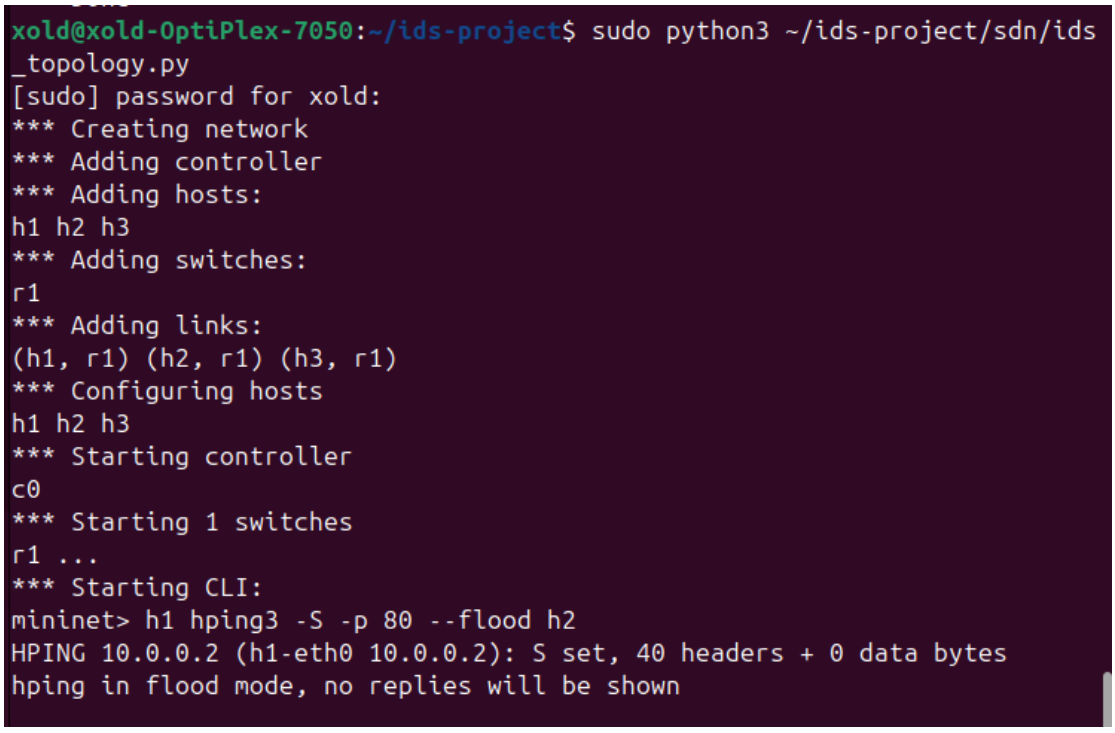


FIGURE 9 EXECUTION OF A TCP SYN FLOOD ATTACK FROM WITHIN THE MININET COMMAND-LINE INTERFACE USING THE HPING3 TOOL.

Against this benign baseline, a series of scripted attack scenarios were executed. These were launched from the "attacker" host and were targeted at the "victim" host, using well-established network testing tools such as hping3 and nmap. Each attack scenario was described by a compact run descriptor, which included the tool used, the transport protocol, the destination port, any key flags or options (e.g., the TCP SYN flag), and the desired rate or timing behaviour. This descriptor identifier was then carried through the entire pipeline, embedded in filenames and log entries, to maintain end-to-end traceability from the initial stimulus to the final inference result.

3.7.3 Packet Capture Configuration and Time Discipline

Packet capture was performed at two distinct points to ensure both comprehensive coverage and resilience. The primary vantage point was placed on the core OVS switch interface that aggregated all traffic toward the victim segment. A secondary vantage point was placed on the collector host to provide resilience during troubleshooting and to permit the cross-checking of timestamps when diagnosing clock skew.

All captures were written in the standard .pcap format with bounded file rotation to avoid the saturation of the storage device during longer-running scenarios. Berkeley Packet Filter (BPF) expressions were applied to limit the capture to only the protocols of interest. All filenames were timestamped in Coordinated Universal Time (UTC) and included the scenario identifier and the capture point to enable mechanical sorting and later reconciliation. Maintaining accurate and synchronized time across all components was treated as a critical requirement ("a first-class concern"). To achieve this, the Mininet hosts and the Ryu controller host were synchronized using the system's Network Time Protocol (NTP) service, and all filenames and logs embedded timestamps at second-level resolution to facilitate accurate event correlation. An example of the packet capture process within the experimental environment is illustrated in Figure 10, which shows the execution of the tcpdump command used to monitor and record traffic on the selected interface.

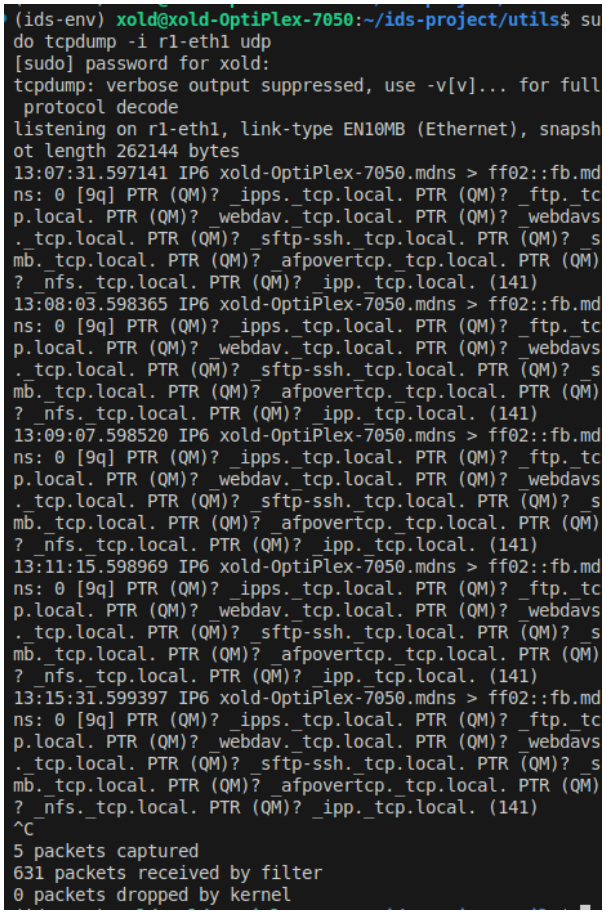
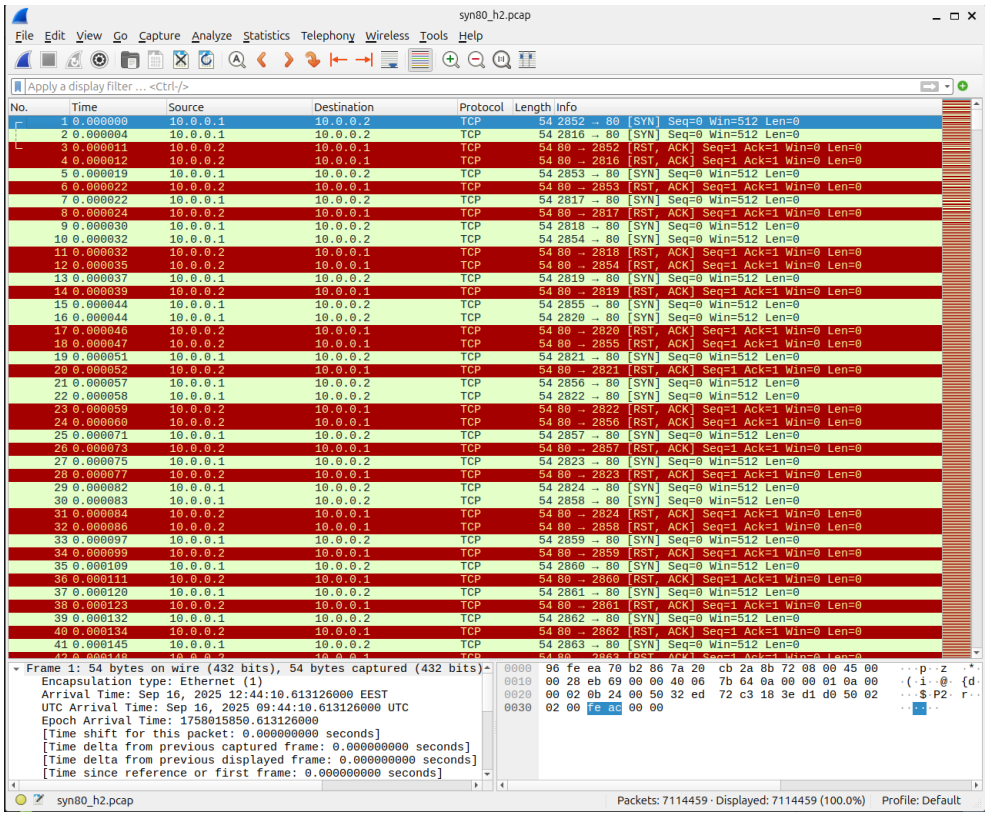


FIGURE 10 REAL-TIME PACKET CAPTURE USING THE TCPDUMP UTILITY ON THE CORE SWITCH INTERFACE, SHOWING THE NUMBER OF PACKETS CAPTURED.

3.7.4 PCAP-to-Features Bridge and Schema Normalization

The raw .pcap files were transformed into flow-level features by a custom-built software bridge that supported two interchangeable back-ends. The first back-end was a CICFlowMeter-style extractor, which provided a broad set of statistical descriptors familiar from the academic literature. The second was a Python-native extractor based on the PyShark and Scapy libraries. Regardless of the chosen extractor, the bridge emitted an intermediate CSV file, whose column names were then mapped into the authorised EXPECTED_84 schema by the normalization and canonicalization procedure described in **Section 3.4** *(Mentor's Note: This should be a cross-reference)*. This two-step process, which decouples the raw feature extraction from the final schema enforcement, was a key design choice that allowed for flexibility in the choice of upstream tools while maintaining a stable contract with the downstream inference service.



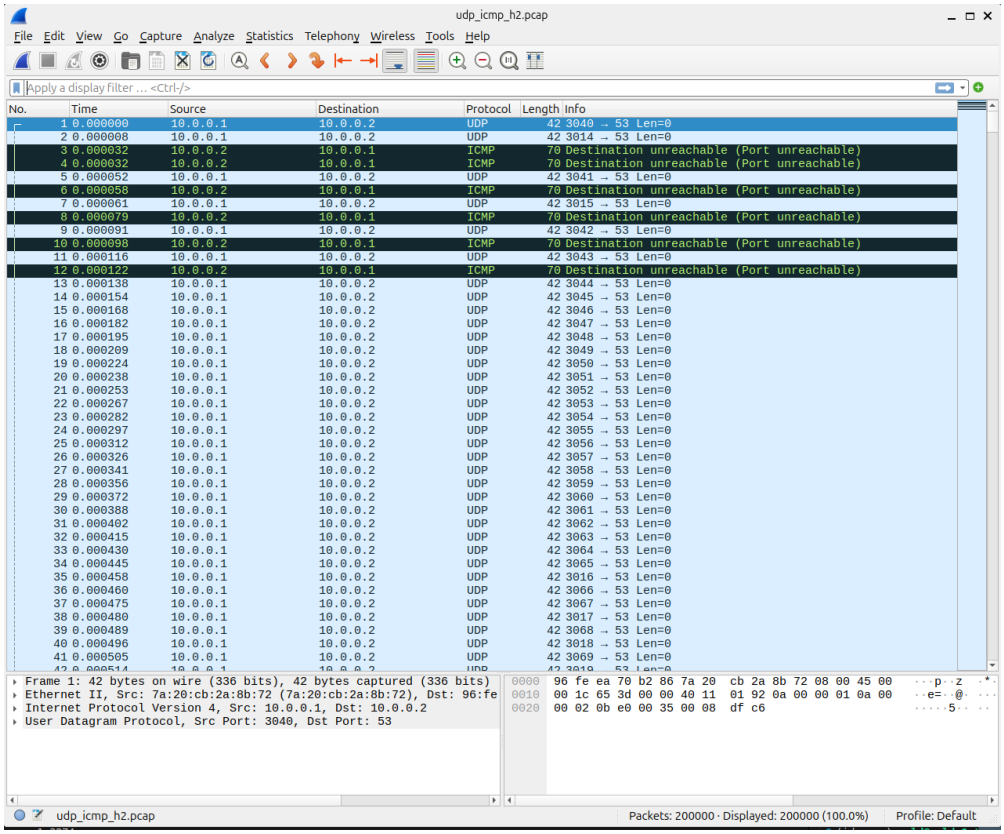


FIGURE 12 PACKET-LEVEL ANALYSIS OF A CAPTURED UDP SCECARIO IN WIRESHARK, CONFIRMING THE HIGH VOLUME OF UDP REQUESTS.

3.8 Diagnostics, Testing Hooks, and Observability

A robust and observable system is one that can be easily monitored, debugged, and verified during operation. To this end, a suite of diagnostics, testing hooks, and observability mechanisms was incorporated as a first-class design element throughout the implementation. These features were designed to provide clear insights into the internal state of the system, facilitate rapid fault isolation, and enable automated health monitoring without compromising the integrity of the core inference pipeline.

A concise test harness was embedded directly into the inference server to verify, in isolation, that the server could be started, that the model and scaler artifacts could be loaded from disk, and that a well-formed 84-dimensional input would produce a well-formed response. This harness exposes three distinct self-test endpoints. The first endpoint exercises a narrow path using a minimal, canonical feature vector, confirming that the persisted scaler artifact is applied correctly and that the model's post-processing stage correctly maps a class identifier to a human-readable label. The second endpoint extends this test to a small, stored batch of curated samples retained for deterministic validation. A stride parameter is accepted by this endpoint to allow a user to probe the path at periodic offsets, which avoids overfitting the test to a single, static case. The third endpoint performs a class-balanced probe by selecting a handful of samples from each known class identifier, ensuring that the label mapping and any downstream formatting logic are exercised uniformly across all possible output classes.

Robust, structured logging was used as the backbone of the system's observability. All components were designed to emit structured logs in JSON format that include a high-precision timestamp, a monotonic latency measurement for the specific operation, a correlation identifier propagated across the client and server, and a concise status field. This status field takes on predefined values such as "accepted," "schema_error," or "predicted," which allows for easy filtering and aggregation of log data. The inference server was also configured with a verbose mode, which, when enabled, records the minimum, maximum, and mean of the raw feature vector for each request. This feature was invaluable for the immediate detection of degenerate inputs, such as all-zero rows, which can arise when an upstream capture pipeline is misconfigured.

Health monitoring was implemented through an external watchdog process that performs two orthogonal checks: an HTTP-level probe of the server's self-test endpoint and a check for outdated or "stale" data in the predictor's output log. The first check confirms that the server process is responsive and that the full inference path can be successfully exercised on demand. A threshold on the response time can be configured to flag degraded behavior even when the server is still answering requests. The second check confirms that predictions are being produced continuously at or above a configured minimum rate. If the log file has not been appended to within a defined interval, the watchdog records a liveness alarm. The interaction of these two checks is particularly useful in containerized deployments, where a process can remain in an "up"

state from the orchestrator's perspective, even while a dependent volume or named pipe has stalled.

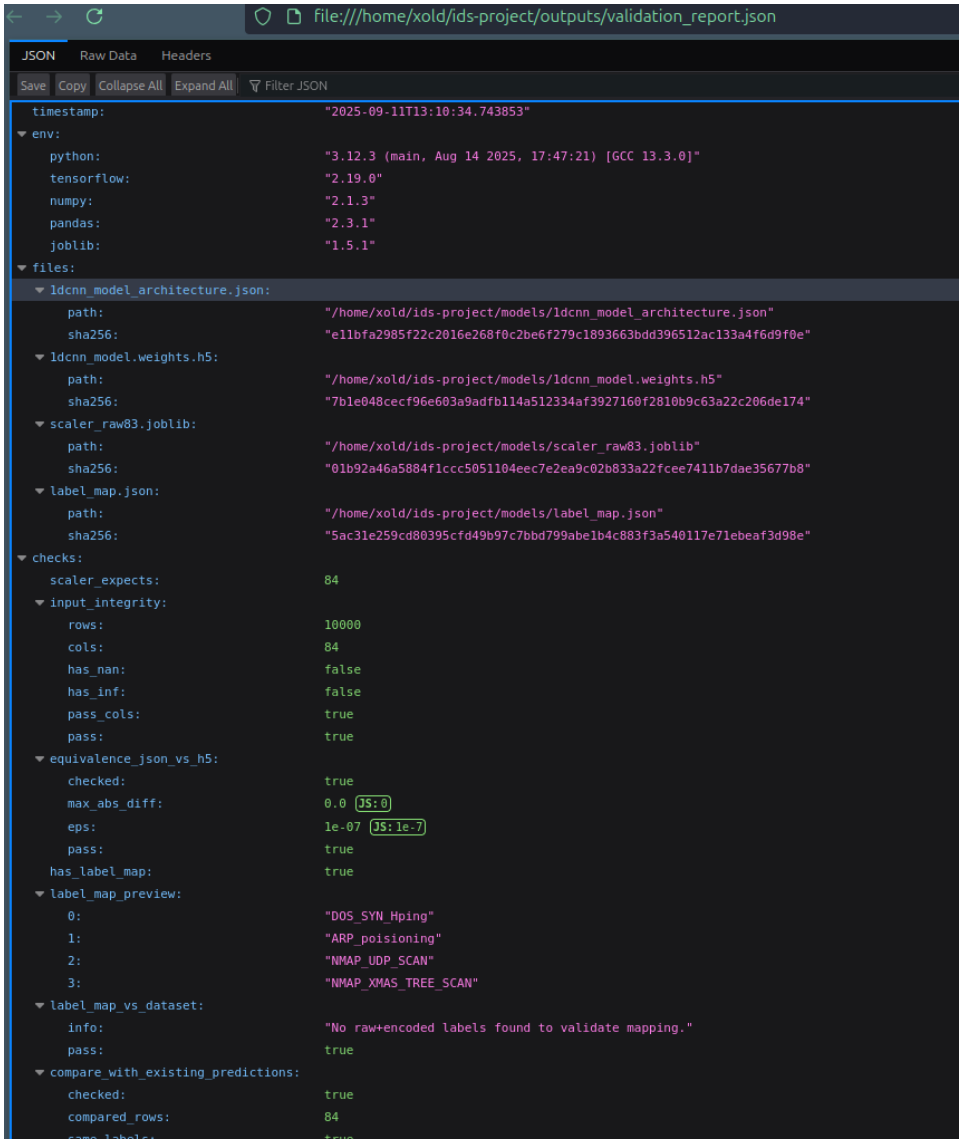


FIGURE 13: A SECTION OF THE JSON VALIDATION REPORT GENERATED BY THE SELF-TEST HARNESS PART_1

The diagnostic surface was also designed to cover the preprocessing and data-ingestion path. The client-side normalizer, which enforces the authorised schema, records the set of columns present in an incoming row and the set of columns required by the authorised order. When a mismatch is detected, the difference is logged explicitly, and the row is either rejected or repaired using deterministic derivations, depending on the configuration. This mechanism proved decisive in eliminating a class of errors where an upstream tool produced consistent but incomplete rows, which then led to predictions being dominated by a single class. By logging the number of zero-valued fields and the first few values of every submitted vector in verbose mode, degenerate input files could be identified within seconds rather than hours.

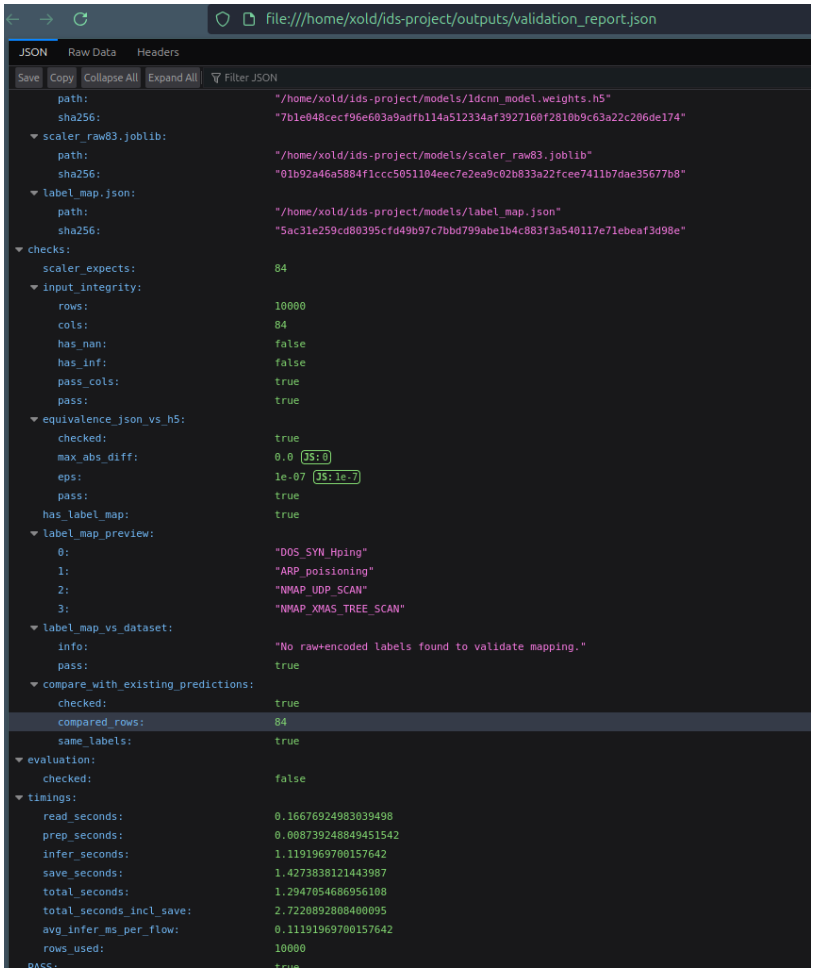


FIGURE 14 A SECTION OF THE JSON VALIDATION REPORT GENERATED BY THE SELF-TEST HARNESS PART_2

3.9 Operational Runbooks

To ensure the reproducibility and correct operation of the system, a set of formal operational runbooks was developed. A runbook is a detailed compilation of procedures and operations required to run the system. This section documents the step-by-step procedures for executing the entire IDS pipeline in two distinct modes: a bare-metal mode, intended for interactive development and debugging, and a containerized mode, designed for simplified, reproducible deployments. These runbooks also include procedures for post-run cleanup and the remediation of common failure modes.

3.9.1 Bare-Metal Execution Runbook

The bare-metal mode was designed primarily for interactive development, short demonstrations, and debugging. This mode involves launching the three cooperating processes of the pipeline in separate terminals on the same host. The execution order is critical to prevent transient connection errors and ensure a clean startup sequence.

1. Start the Inference API: The first process to be launched is the Flask application. This is done from within the Python 3.12 environment. The server loads the trained 1D-CNN artifact, the authorised label map, and the previously fitted `StandardScaler` into memory. It then exposes the `/predict` HTTP endpoint on the loopback interface. The application's startup log explicitly confirms that the model and scaler have been loaded successfully and reports their respective checksums, providing an immediate point of verification for the operator.
2. Start the Real-Time Predictor: Once the API has announced its readiness, the real-time predictor is executed from the same Python 3.12 environment. Upon initialization, the predictor resolves the absolute path to the live CSV feature stream, verifies the presence of a single-line header that matches the `EXPECTED_84` schema, and then positions its file handle at the end of the file so that only newly appended rows are processed. It then enters a continuous loop, tailing the file and submitting each new row as a request to the local API.
3. Activate the Data Source: Only after the first two services are confirmed to be running is the data source activated. This can be either a simulator, which replays tabular flows at a controlled rate from an evaluation file, or a capture-to-features bridge, which ingests `.pcap` traces from the SDN testbed. This final process writes the feature rows to the live CSV append-only file, which are then immediately picked up by the predictor.

This trio of long-lived processes forms a pipeline that is robust to transient delays. If the upstream simulator or bridge slows down, the predictor idles while the API remains resident in memory. When new rows appear, the entire path resumes without requiring any manual intervention.

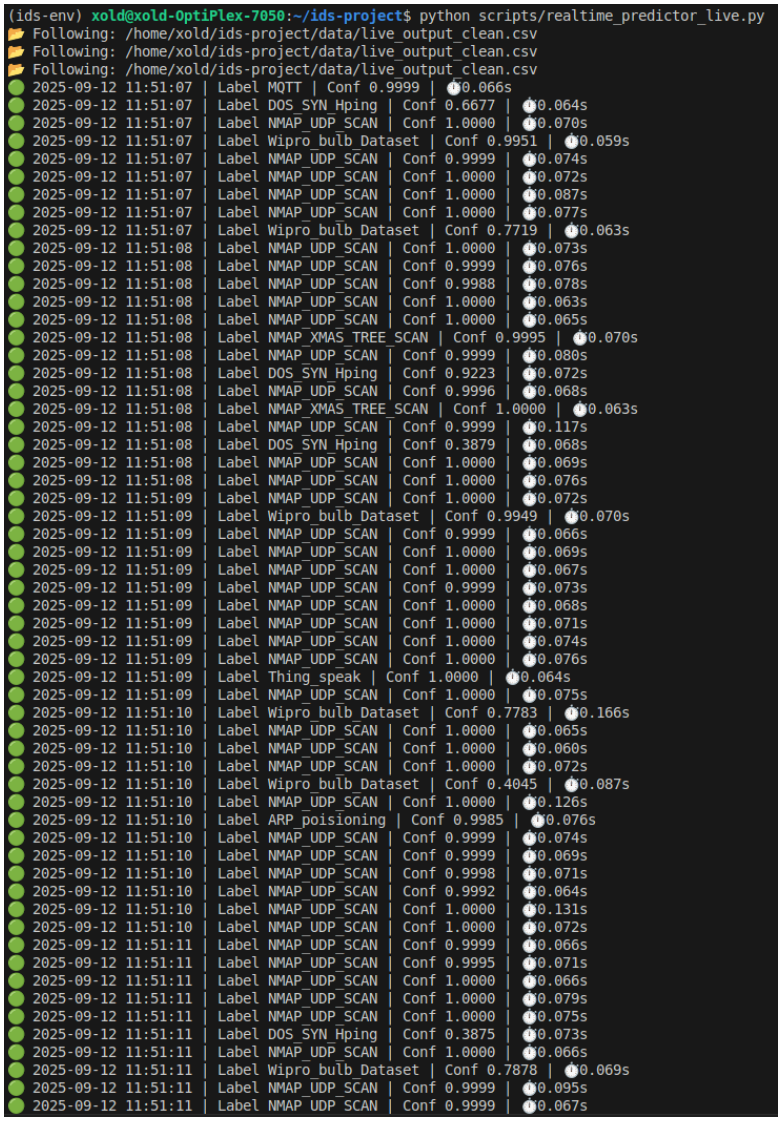


FIGURE 15 CONSOLE OUTPUT FROM THE REAL-TIME PREDICTOR CLIENT, DEMONSTRATING THE STRUCTURED LOGGING OF EACH SUCCESSFUL INFERENCE CYCLE.

3.9.2 Containerized Execution Runbook

The containerized runbook mirrors the functional steps of the bare-metal mode but replaces the per-terminal commands with a declarative orchestration using a docker-compose specification. This mode is designed for ensuring maximum reproducibility, isolation, and simplified orchestration.

A minimal docker-compose.yml file is used to describe four services: the API, the predictor, the simulator (or capture bridge), and an optional watchdog. The API container is built from a base image that matches the training environment to avoid numerical discrepancies. At startup, the container mounts a read-only volume that contains the model artifact, the scaler, and the label map, and a read-write volume that holds the application logs. The predictor container is configured with an explicit dependency on the API service, ensuring it does not start until the API is healthy. It

mounts the live CSV path as a volume, thereby preserving the append-only behavior across restarts. The simulator or bridge container is started last and is permitted to write on the same mounted path. The health check of the API, which is defined in the Docker Compose file, prevents the predictor from sending requests prematurely. Logs from all services are written to a persistent, shared directory under predictable names, and log rotation is performed at the level of the container runtime to prevent the exhaustion of local storage.

3.9.3 Operational Hygiene and Failure Remediation

The runbooks include explicit procedures for post-run cleanup and for remediating common failure modes. Once a run is concluded, the operator is instructed to archive the live CSV file, along with its corresponding logs, to a timestamped directory. The pipeline is then reset by truncating or deleting the current live file, which guarantees that a subsequent run will not accidentally inherit tail content from a previous experiment.

Common failure modes are also anticipated and are resolved with deterministic procedures. If the predictor reports that fewer than eighty-four columns are present, the operator is instructed to verify that the simulator or bridge wrote the header correctly and that the column names precisely match the canonical schema. If the API returns an HTTP 400 response, the remediating step is to inspect the predictor's log for the offending row and to confirm that non-numeric tokens or NaN placeholders were not introduced during capture. These documented actions are designed to maintain forward progress without altering the scientific content of the subsequent evaluation.

3.10 Security and Hardening

Although the system was developed and operated within a controlled laboratory environment, security controls were incorporated as first-class design elements throughout the pipeline. This was done to ensure that the inference service could be delivered reliably, even under adverse conditions, and with minimal operational risk. The implementation of these controls covered the inference API, the data ingestion path, the containerized runtime, the SDN control plane, and the supporting observability components. A defense-in-depth strategy was adopted, with an emphasis on the principle of least privilege, strict input validation, and the creation of tamper-evident operational records. To aid in reproducibility and auditability, these control choices were documented as explicit, testable policies rather than as ad-hoc scripts.

The public-facing inference surface was deliberately reduced to a single HTTP endpoint. In a production-oriented deployment, this endpoint is intended to be terminated behind a reverse proxy, where critical security functions such as Transport Layer Security (TLS) offloading, request normalization, and rate limiting would be performed. The Flask application itself was configured to be bound to either the loopback interface for local-only access or a container-internal interface, with external exposure being delegated entirely to the proxy. All requests were expected to be made

over HTTPS, and strict content length limits were applied at the proxy layer to prevent oversized payloads from consuming system resources. Timeouts were enforced on both sides of the proxy boundary to bound resource consumption in the presence of slow client behaviors, such as those seen in attacks.

Strict input validation was enforced at the API boundary prior to any model invocation. As detailed in Section 3.6, each request was required to provide exactly eighty-four numeric features in a predetermined order, with explicit checks for missing values, out-of-range magnitudes, and non-finite numbers. Any request that violated this schema was rejected with a precise diagnostic message that did not reveal internal implementation details. The integrity of the machine learning artifacts was also verified. The scaler object and the label map were loaded at process start, and their integrity was confirmed by computing and checking cryptographic digests (SHA-256) that were stored alongside the artifacts. The inference path was designed to be minimal in its output; it emitted only the predicted label identifier and a calibrated confidence value. High-resolution probability vectors were intentionally not exposed to reduce the risk of model extraction and iterative probing attacks.

The use of container hardening techniques was employed to strengthen operational containment. The API, the real-time predictor, the simulator, and auxiliary services were each executed in separate containers with distinct runtime profiles. The principle of least privilege was applied rigorously: Linux capabilities were dropped to the minimal set required, the root user was not employed within the containers, and read-only root filesystems were used wherever possible. Network exposure was restricted to only the ports strictly required by each process. Resource ceilings were defined using groups to ensure that CPU and memory exhaustion attacks could not propagate beyond a single container.

Supply chain risk was mitigated by pinning package versions, using cryptographic hashes for critical software wheels, and performing vulnerability scans as part of the build pipeline. The base container images were selected from maintained distributions, and updates were applied on a regular cadence. The dependency graphs were frozen so that unreviewed transitive upgrades could not silently alter the runtime environment. On startup, the application verified the checksums of the model, the scaler, and the label map; any discrepancy led to a fail-closed behavior, where the service would refuse to start, and a clear diagnostic was written to the audit log to signal potential tampering.

Finally, the SDN control plane was logically and physically isolated from the data-plane components. The Ryu controller was placed on a management network segment that was not reachable by the untrusted hosts of the data plane. Access to the controller was restricted to administrative users, and logging was enabled at the framework level to capture all flow-modification events and topology changes. The Open vSwitch channels were configured to communicate only with the designated controller endpoint, and where possible, the operating system firewall was used to augment this by accepting connections exclusively from trusted addresses.

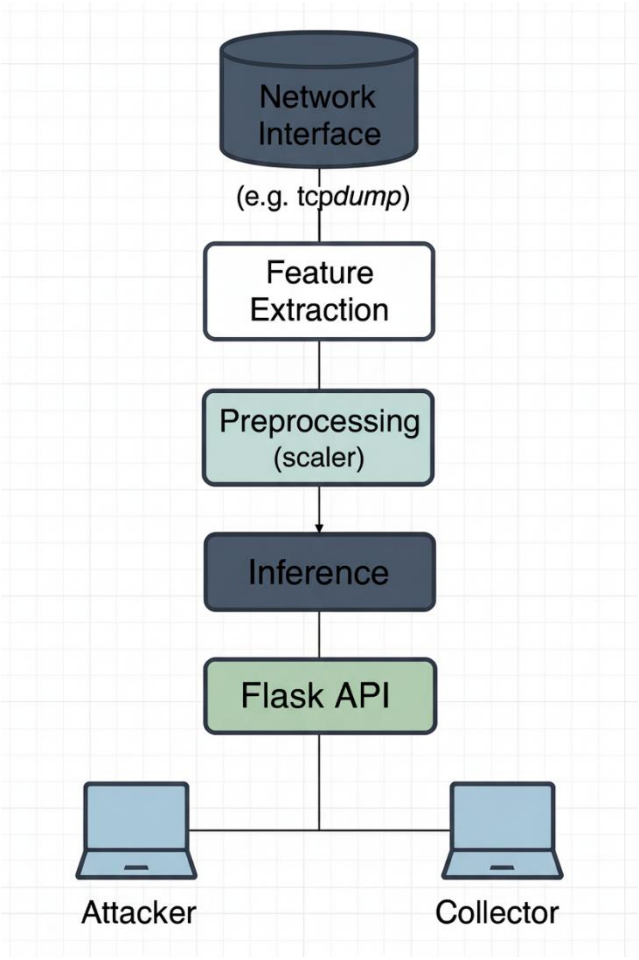


FIGURE 16 THE ARCHITECTURE OF THE REAL-TIME INFERENCE PIPELINE, ILLUSTRATING THE FLOW OF DATA FROM THE NETWORK INTERFACE THROUGH FEATURE EXTRACTION.

3.11 Limitations and Technical Debt

This implementation has been engineered to provide a dependable and reproducible end-to-end pipeline for intrusion detection over a software-defined networking testbed. However, in the course of this engineering effort, several design choices were made that, while necessary for `achieving the project's primary objectives, also introduced a set of limitations and areas of technical debt. This section provides a transparent acknowledgment of these constraints. The discussion that follows has been structured to elucidate the limitations that stem from data handling, real-time capture, the inference architecture, orchestration choices, and integration with the SDN control plane. The emphasis is placed on these implementation realities rather than on empirical outcomes.

A primary set of limitations arises from the strict data standardization that was enforced. The real-time inference path is fundamentally dependent on a rigid, canonical schema of eighty-four features, which must be ordered and typed in a predetermined manner. This design was intentionally adopted to ensure a stable and verifiable contract between the preprocessing pipeline and the loaded model artifacts; yet it introduces a degree of

fragility. When upstream tools, such as flow exporters or packet-to-flow bridges, vary in their header naming conventions or omit certain fields under specific edge conditions, the ingestion layer requires corrective logic to synthesize defaults, derive aggregates, or drop corrupted rows. Although a schema-normalization routine has been implemented to handle this, the system remains sensitive to silent drift in upstream CSV layouts and to heterogeneous units produced by third-party utilities.

A related constraint concerns the system's ability to generalize under conditions of dataset shift. The inference service was designed around an offline-trained model and a frozen scaler, both of which are loaded at runtime. While this approach ensures deterministic and reproducible behavior, it presupposes that the statistical distribution of live traffic will remain compatible with the distribution that was observed during the model's training. In realistic, long-running deployments, the distribution of flow-level features can be expected to drift due to seasonal effects, changes in network workload, or the adaptation of attacker methodologies. The current implementation does not include mechanisms for active drift detection or for the online recalibration of the scaler or model. Therefore, the system retains a "fixed scaler, fixed model" stance, which would require periodic, manual retraining to remain effective over long periods.

Further constraints are associated with the mechanisms for observability and diagnostics. While rich, structured logging has been implemented at each stage of the pipeline—ingestion, schema normalization, API entry, and model inference—a centralized correlation of these logs across different containers has not been completed within the scope of this chapter. In consequence, cross-component incident analysis is not instantaneous. Timestamps are recorded, but clock synchronization relies on the host's time service, and no distributed tracing headers are propagated along the predictor's API path, which can complicate the diagnosis of latency issues in a distributed environment.

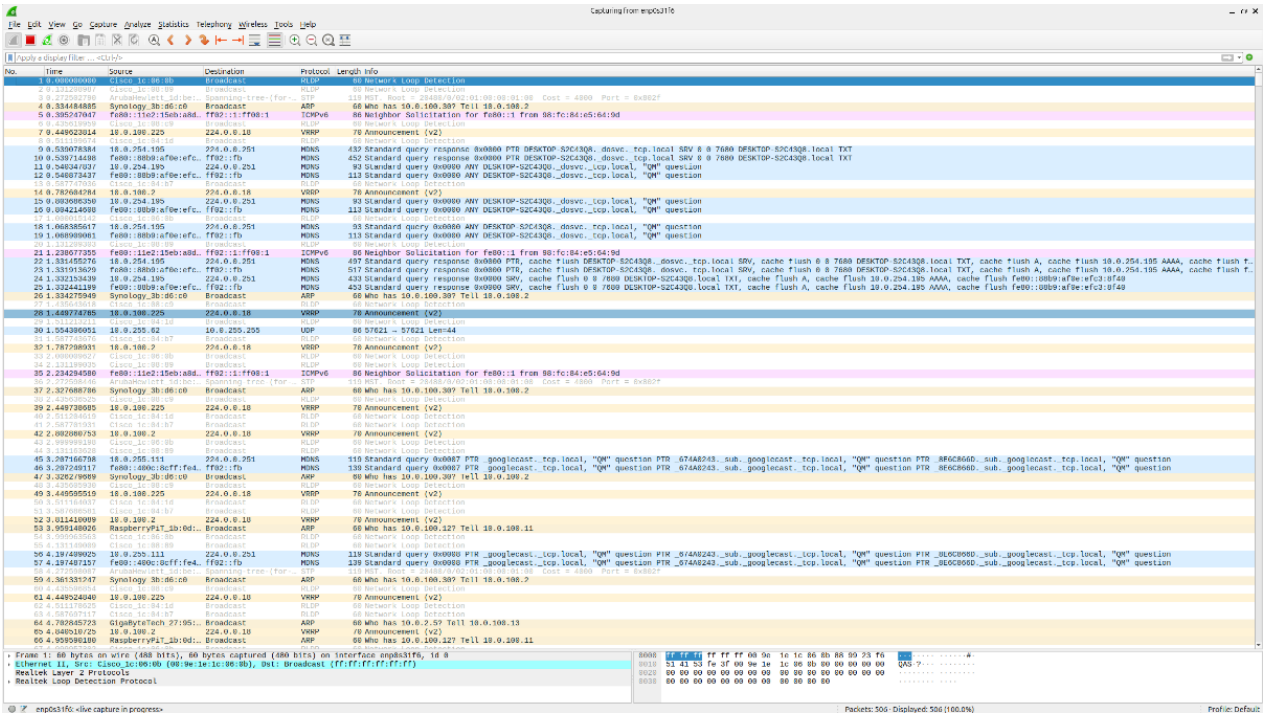


FIGURE 17 PACKET-LEVEL ANALYSIS OF THE BENIGN BACKGROUND TRAFFIC GENERATED WITHIN THE SDN TESTBED.

Finally, the last limitation to be noted relates to model lifecycle management. The current implementation loads a single, static model at the start of the service, and hot-swapping of models has not been implemented. In production environments, it is generally preferable to support versioned API endpoints or a model registry, which would allow for new model artifacts to be validated in a shadow mode before being promoted. In the present implementation, a full service restart is required to affect any model updates. In summary, the limitations described above do not undermine the core objective of this chapter, which is to document a reproducible, real-time IDS pipeline. However, they do delineate the envelope within which the implementation should be interpreted. The architecture has been intentionally kept transparent and modular so that each of these limitations is addressable through targeted future engineering.

EVALUATION

4.1 Aim and Scope of the Empirical Analysis

Following the systematic exposition of the system's architecture and construction in the preceding chapter, this chapter is dedicated to the empirical and quantitative validation of the proposed IDS. The narrative now transitions from the engineering processes and the architectural blueprint to a comprehensive analysis of the system's operational performance. The research presented in the "State of the Art" chapter identified a critical need for security solutions that not only achieve high detection accuracy but also maintain the computational efficiency required for deployment in modern, high-throughput network environments such as Software-Defined Networking and 5G. This evaluation was therefore designed to directly address this identified gap by providing a multi-faceted and quantitative analysis of the implemented system's capabilities, thereby bridging the divide between theoretical design and practical efficacy.

The primary aim of this evaluation was to quantitatively validate the effectiveness, accuracy, and efficiency of the proposed two-stage deep learning model, which integrates an Autoencoder (AE) for feature compression with a 1D-Convolutional Neural Network for classification. To achieve this, a series of comprehensive experiments were designed and executed. A central objective was to conduct a granular assessment of the model's classification accuracy . This was undertaken to determine its absolute effectiveness in the challenging task of multi-class classification, where its ability to accurately distinguish between benign network traffic and a diverse set of twelve predefined malicious activity classes from the RT-IoT2022 dataset was scrutinized. This analysis was considered fundamental to establishing the baseline efficacy of the proposed architecture.

Furthermore, the contextualization of this performance through a direct comparative analysis was deemed essential. Consequently, the efficacy of the proposed Autoencoder-CNN architecture was benchmarked against two widely recognized baseline classifiers: a SVM and a RF model. To ensure a fair and scientifically valid comparison, all three models were trained and tested under identical conditions, using the same data partitions. This comparison was designed to investigate a core hypothesis of this thesis: that a tangible and quantifiable performance advantage is offered by the representational learning capabilities inherent to the proposed deep learning approach over traditional machine learning paradigms for this specific task.

Beyond the dimension of classification accuracy, another critical aim was to analyze the practical viability of the system for real-world deployment. High detection accuracy is of limited utility if a system's response time is insufficient for near-real-time intervention. Consequently, a specific objective was established to measure and statistically analyze the inference latency of the deployed 1D-CNN model and to compare its raw prediction speed against that of the baseline models. This evaluation of computational efficiency was considered fundamental to validating the system's

suitability for deployment in the stringent, low-latency environments characteristic of next-generation networks.

Finally, the reliability of all empirical findings rests upon the robustness of the model development process itself. Therefore, a foundational objective of this evaluation was to validate the stability of the deep learning training procedure. The learning curves for model loss and accuracy were meticulously analyzed to confirm that a state of stable convergence was achieved and that the final trained model was well-generalized. The scope of this entire empirical analysis is strictly confined to the performance of the models on the test partition of the RT-IoT2022 dataset, ensuring that all conclusions are drawn from evidence that remained entirely unseen during the model training and hyperparameter tuning phases, a cornerstone of methodologically sound machine learning evaluation.

4.2 Evaluative Methodology

To ensure the scientific rigor and reproducibility of the empirical findings presented in this chapter, a controlled and consistent evaluative methodology was employed. This section provides a comprehensive account of the complete methodological framework that was established for the quantitative assessment of the proposed IDS and the baseline models. The discussion herein details the experimental environment, the protocol for data handling and partitioning, the theoretical and practical significance of the selected performance metrics, the justification for the selection of baseline models for comparative analysis, and the precise protocol used for the measurement of system performance. Each component of this methodology was deliberately chosen to align with established best practices in machine learning evaluation and to directly address the specific challenges inherent to the domain of network security.

4.2.1 Experimental Environment and Dataset Partitioning

All experiments were conducted within the precisely defined and controlled environment that was detailed in Chapter 3. To reiterate, the hardware platform consisted of a system equipped with 32 GB of RAM, operating on an Ubuntu 22.04 LTS distribution. A critical aspect of this evaluation is that all model training and inference procedures were executed exclusively on the central processing unit (CPU). The absence of GPU acceleration, as confirmed by the TensorFlow initialization logs, establishes a crucial performance baseline on common, non-specialized hardware, which is representative of a wide range of potential deployment environments. The RT-IoT2022 dataset served as the exclusive source of data for all training, validation, and testing procedures. To prevent any form of data leakage and to ensure an unbiased evaluation, the dataset was subjected to a strict and immutable partitioning protocol. The complete dataset was divided into training, validation, and test subsets with fixed proportions of 80%, 10%, and 10%, respectively. The indices for this partition were generated once and were subsequently treated as an immutable reference for all experiments. Consequently, the final performance of all classifiers was measured on the held-out test subset, which remained entirely unseen during all phases of model training, hyperparameter tuning, and validation.

4.2.2 Theoretical and Practical Significance of Performance Metrics

The quantification of classifier performance was accomplished through a standard suite of metrics derived from the confusion matrix generated for each model on the test dataset. While these metrics have universal definitions, their practical interpretation is highly context-dependent. In the domain of intrusion detection, they represent the critical trade-offs between system security and operational overhead.

Accuracy, defined as the proportion of total predictions that were correct, was calculated to provide a general, high-level indication of a model's overall performance. However, it is recognized that accuracy can be a misleading metric, particularly in the context of imbalanced datasets, which are endemic to network security. Given the significant class imbalance present in the RT-IoT2022 dataset, where certain normal and attack classes are far more prevalent than others, accuracy was not relied upon as the primary measure of efficacy. Instead, it was supplemented by more nuanced metrics that provide a clearer picture of a model's behavior.

Precision was utilized as a key metric and is interpreted in this context as a measure of the "cost of false alarms." In an operational security environment, a high volume of false positives can lead to "alert fatigue," where security analysts become desensitized to warnings, potentially causing them to overlook genuine threats. Therefore, high precision is of paramount importance, as it ensures that when an alert is raised, it is highly likely to correspond to a genuine malicious event, thereby maximizing the efficiency of human-in-the-loop security operations.

Conversely, Recall, also known as sensitivity or the true positive rate, was interpreted as a measure of the "cost of missed detections." This metric quantifies the model's ability to identify all actual threats present in the network traffic. For an IDS, high recall is a non-negotiable requirement. A system with low recall, regardless of its precision, would be operationally inadequate, as it would allow a significant fraction of malicious activities to go undetected, thereby failing in its primary security function.

To provide a single, consolidated measure that effectively balances the competing concerns of precision and recall, the F1-Score was employed. As the harmonic mean of precision and recall, the F1-Score is inherently sensitive to imbalanced values and penalizes models that excel in one metric at the severe expense of the other. For this reason, the F1-Score was selected as the primary metric for the comparative assessment of the different classifiers, as it offers the most robust and holistic measure of a model's practical effectiveness in the challenging and imbalanced domain of intrusion detection.

4.2.3 Justification for Baseline Model Selection

To properly contextualize the performance of the proposed 1D-CNN architecture, its results were benchmarked against two well-established classical machine learning models: the SVM and the RF. The selection of these specific models was deliberate and was based on their theoretical diversity and their proven efficacy in a wide range of classification tasks. The SVM is theoretically grounded in the principles of statistical learning theory, specifically structural risk minimization and margin maximization. Its effectiveness in high-dimensional feature spaces and its general robustness to overfitting make it a powerful and relevant baseline. The RF, as a premier ensemble

method, represents a different paradigm of classification. Its resilience to overfitting is derived from the use of bootstrap aggregating (bagging), and its inherent ability to capture complex, non-linear interactions between features makes it a highly competitive benchmark. By comparing the proposed DL model against these two distinct , and well known benchmark paradigms, a more meaningful and scientifically sound conclusion regarding its relative advantages can be drawn.

4.2.4 Protocol for Real-Time Performance Measurement

A core aim of this research was to develop a system suitable for real-time or near-real-time deployment. Therefore, the evaluation of computational efficiency, specifically inference latency, was considered a critical component of the methodology. Inference latency was formally defined as the temporal delta between the reception of a feature vector by the deployed inference service and the subsequent transmission of a classification response. The measurement of this latency was conducted within the operational context of the end-to-end pipeline constructed in IMPLEMENTATION. Specifically, latency was measured at the client-side orchestrator, which was responsible for submitting prediction requests to the Flask API. For each HTTP POST request sent to the /predict endpoint, a high-precision timestamp was recorded immediately before transmission and immediately upon reception of the response. The difference between these timestamps was calculated and logged as the total inference time for that transaction. The collected measurements were then aggregated to derive key statistical indicators, including the mean, median, and standard deviation of the latency, providing a quantitative basis for assessing the system's real-world deploy ability.

4.3 Experimental Variables

In accordance with established principles of experimental design, a clear definition of the experimental variables was established to ensure the validity and interpretability of the empirical results. The experiments conducted in this evaluation were designed to investigate the causal relationship between the choice of classification model and its resulting performance in the context of network intrusion detection. To this end, the variables of the study were categorized into independent and dependent variables..

The independent variable in this investigation was the classification model architecture itself. This variable was manipulated by systematically evaluating three distinct model types: the proposed 1D-Convolutional Neural Network, the SVM baseline, and the RF baseline. Each of these models represents a discrete level of this variable, and the systematic comparison of their performance was the central focus of the experiment. A secondary independent variable was the class of network traffic presented to the system, as defined by the twelve distinct labels within the RT-IoT2022 dataset. The performance of each model was assessed across all these classes to determine if its efficacy was uniform or if it exhibited a differential response to specific types of malicious or benign traffic.

The dependent variables were the quantitative metrics that were measured to gauge the performance of each model under evaluation. These variables were selected to provide a comprehensive and multi-faceted assessment of both the accuracy and the efficiency of the classifiers. The primary set of dependent variables consisted of the classification performance metrics, which included Accuracy, Precision, Recall, and the F1-Score. These metrics were calculated for each of the twelve traffic classes individually and were also aggregated to provide a macro-level view of overall model performance. The second key dependent variable was the inference latency, measured in seconds. This variable quantified the computational efficiency of the deployed system and was recorded for each prediction made during the real-time evaluation phase. The systematic measurement of these dependent variables, as a function of the manipulated independent variables, formed the empirical basis upon which the conclusions of this study were drawn.

4.4 Formulation of Hypotheses

Based on the comprehensive literature review conducted in the "State of the Art" chapter and the specific architectural design of the proposed system, a set of formal hypotheses was formulated prior to the execution of the experimental procedures. These hypotheses represent the anticipated outcomes of the evaluation and provide a clear framework for the subsequent analysis and interpretation of the empirical results. The formulation of these predictions is a critical step in the scientific method, ensuring that the investigation is guided by specific, testable propositions rather than being a purely exploratory exercise.

The primary hypothesis of this research posits that the proposed 1D-Convolutional Neural Network model is expected to demonstrate a superior classification performance when compared to the selected classical ML baselines. This prediction is practical in the theoretical advantages of the CNN architecture for the specific task of network traffic analysis. It is anticipated that the convolutional layers, by design, will be capable of autonomously learning localized patterns and hierarchical feature interactions within the EXPECTED_84 feature vector. This 84-feature schema, as detailed in the Implementation chapter, represents the canonical set of statistical and flow-based metrics extracted from the network traffic by the data processing pipeline. It was engineered to provide a comprehensive and standardized representation of each network flow. This capability is particularly relevant for identifying complex attack patterns that may be characterized by subtle, contiguous relationships between features, such as those present in various network scan or flooding attacks. It is therefore hypothesized that the 1D-CNN will yield a significantly higher macro-averaged F1-Score than both the SVM and the RF models, which, despite their power, do not possess an intrinsic architectural bias for learning such spatial hierarchies.

A second, related hypothesis pertains to the model's performance on a per-class basis. It is predicted that the performance advantage of the 1D-CNN will be most pronounced

for attack classes whose signatures are defined by sequential or localized patterns within the feature space. Conversely, for traffic classes that are more readily separable by global, non-sequential feature statistics, the performance difference between the 1D-CNN and the baseline models, particularly the RF, is expected to be less significant. This hypothesis seeks to move beyond a simple declaration of superiority and to investigate the specific conditions under which the proposed architecture is most effective.

A final hypothesis was formulated concerning the computational efficiency and practical deployability of the system. While DL models are often associated with high computational overhead, the proposed 1D-CNN architecture was deliberately designed to be lightweight. Furthermore, the stateless nature of the Flask-based inference service was engineered to minimize per-request processing time. It is therefore hypothesized that the system will exhibit a low and consistent inference latency, with a mean response time that is well within the acceptable thresholds for near-real-time security monitoring in modern network environments. This prediction asserts that the proposed system successfully balances the competing demands of high classification accuracy and low operational latency, thereby validating its suitability for practical deployment.

4.5 Presentation of Empirical Results

This section is dedicated to the objective presentation of the empirical data obtained from the execution of the experimental methodology detailed previously. The results are presented without interpretation, thereby ensuring a clear separation between the evidence itself and the subsequent analysis. The data is organized to logically follow the experimental process, commencing with an examination of the model training dynamics, followed by a granular analysis of the proposed model's classification performance, a direct comparison against the baseline models, and concluding with an assessment of the system's real-time inference latency.

4.5.1 Analysis of Model Training and Convergence Dynamics

A foundational step in the evaluation was the validation of the DL training process to ensure that the models achieved a state of stable convergence without exhibiting signs of significant overfitting. The training history of both the Autoencoder component and the primary 1D-CNN classifier were recorded across all epochs. The learning curves for the AE, which was responsible for the initial feature compression, are presented in Figure 18 Autoencoder Training and Validation Loss Over Epochs . The plot of Huber Loss against the training epochs shows a consistent and rapid decrease in loss for both the training and validation subsets, with the curves stabilizing after approximately 60 epochs. The close and persistent correspondence between the training and validation loss curves throughout the process indicates that the model learned generalizable representations from the training data without diverging into a state of overfitting.

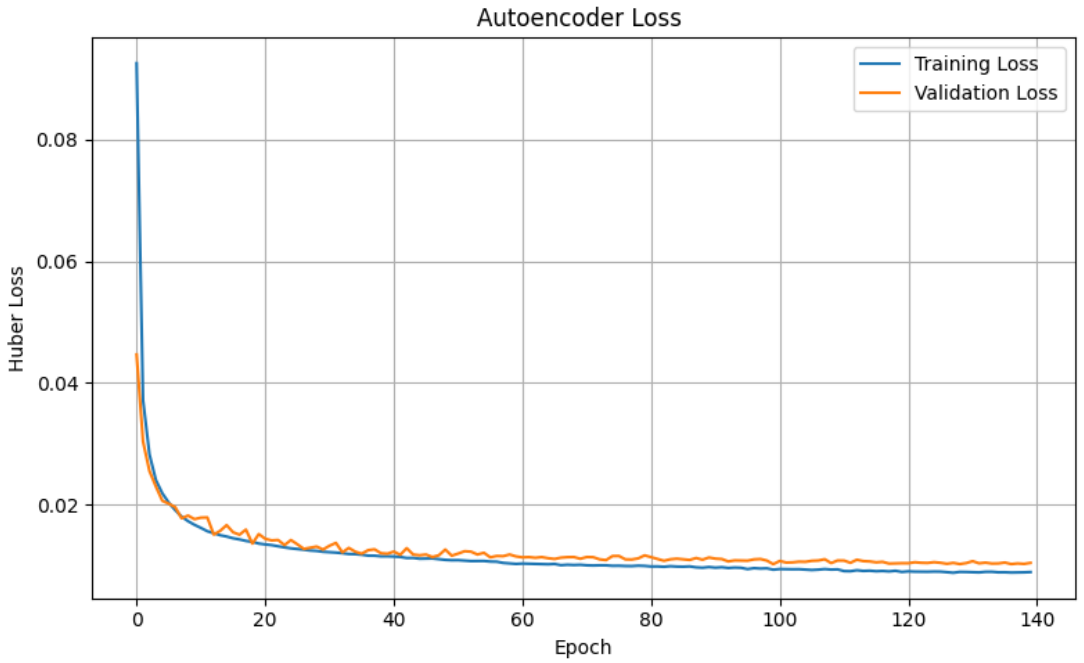


FIGURE 18 AUTOENCODER TRAINING AND VALIDATION LOSS OVER EPOCHS

Following the validation of the feature extraction component, the training dynamics of the primary 1D-CNN classifier were examined, as shown in **Error! Reference source not found.**. A rapid increase in accuracy was observed during the initial epochs for both the training and validation sets, which is characteristic of a successful learning process. This initial phase was followed by a period of more gradual improvement until a performance plateau was reached at a high level of accuracy. The validation accuracy curve is observed to closely track the training accuracy curve, which provides strong

evidence that a well-generalized classification model was successfully trained and that the final performance is reliable.

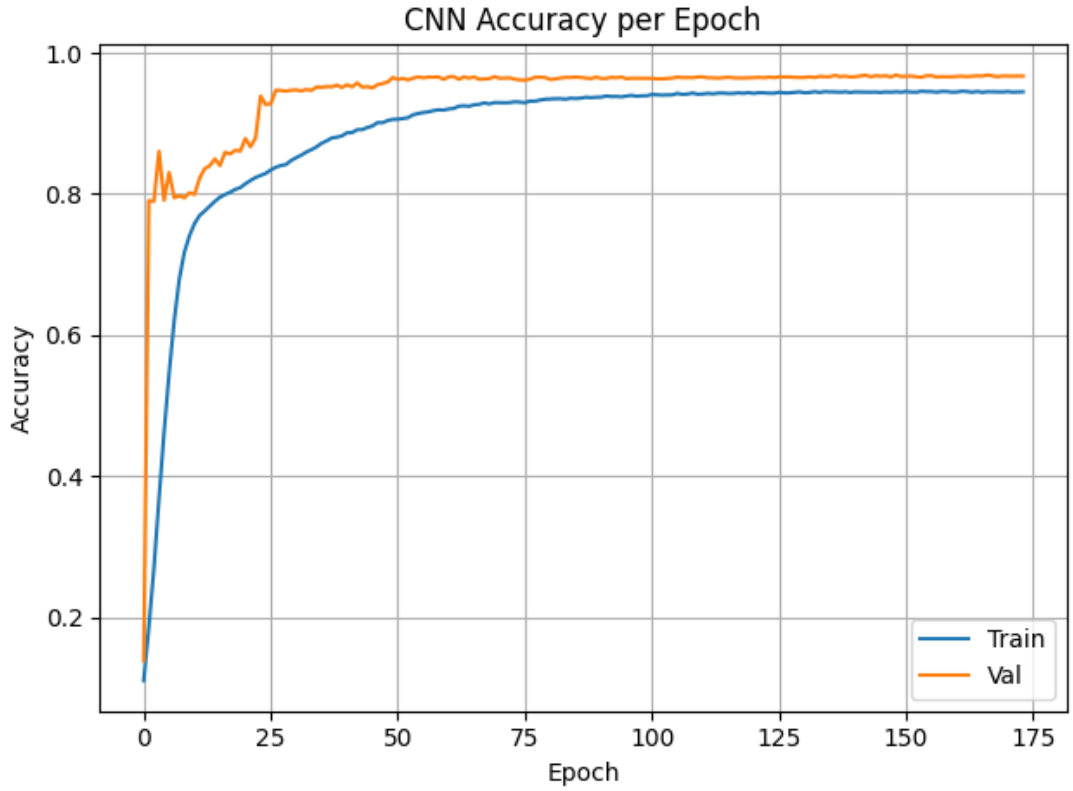


FIGURE 19 1D-CNN CLASSIFIER TRAINING AND VALIDATION ACCURACY.

4.5.2 Granular Classification Performance of the Proposed 1D-CNN Model

The definitive performance of the proposed 1D-CNN model was assessed on the held-out test dataset. The detailed results of this multi-class classification task are presented in the confusion matrix in **Error! Reference source not found.**. In this matrix, the rows correspond to the true labels of the traffic flows, and the columns correspond to the labels predicted by the model. A strong concentration of values along the main diagonal

is visually apparent, indicating a high number of correct predictions across all classes and a high overall level of accuracy for the model.

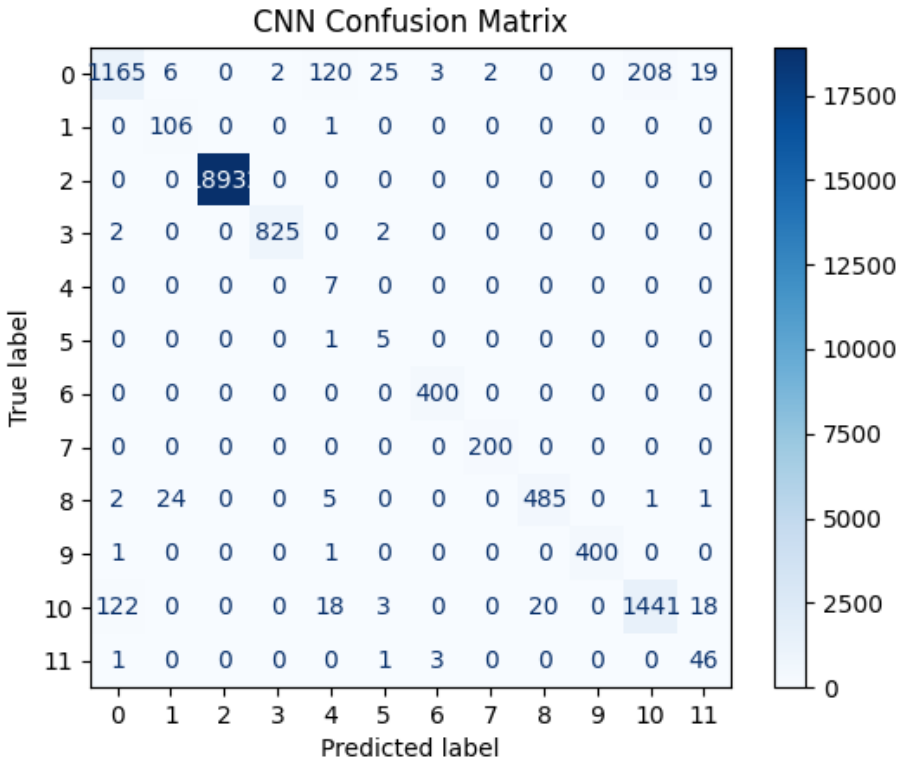


FIGURE 20 CONFUSION MATRIX FOR THE PROPOSED 1D-CNN MODEL ON THE TEST DATASET.

To provide a more precise quantitative assessment derived from the confusion matrix, the per-class performance metrics of the 1D-CNN classifier are summarized in **Table 1**, which presents the precision, recall, and F1-score for each of the twelve traffic classes. The results indicate that several attack categories achieved perfect or near-perfect classification performance. In particular, the classes **NMAP_UDP_SCAN**, **NMAP_XMAS_TREE_SCAN**, and **DDOS_Slowloris** exhibit F1-scores close to 1.0, indicating that the model was able to correctly identify these traffic patterns with very high reliability. In contrast, other traffic categories, including **NMAP_OS_DETECTION** and **NMAP_TCP_scan**, demonstrate comparatively lower performance, suggesting that these attack patterns present greater classification challenges for the model.

4.5.3 Comparative Performance Against Baseline Models

To contextualize the performance of the proposed architecture, it was benchmarked against the SVM and RF classifiers. The confusion matrices for the RF and SVM models are presented in Figure 21 Confusion Matrix for the Random Forest Baseline Model on the Test Dataset. Also **Error! Reference source not found.** , respectively. A visual inspection of these matrices also reveals a high concentration of values along the main diagonal, indicating strong overall performance for both baseline models. However, distinct patterns of off-diagonal

misclassifications are observable when compared to the 1D-CNN matrix.

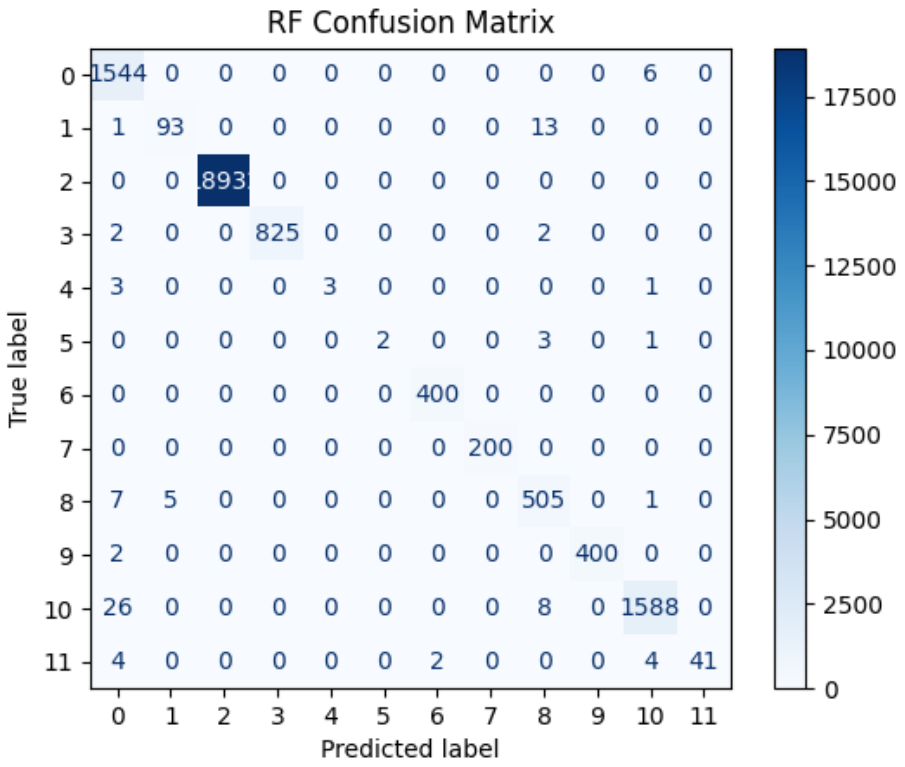


FIGURE 21 CONFUSION MATRIX FOR THE RANDOM FOREST BASELINE MODEL ON THE TEST DATASET.

For a more direct and concise comparison of the overall efficacy of the three evaluated models, their aggregate performance metrics are summarized in Table 2. The overall accuracy and the macro-averaged F1-score are presented to provide a high-level benchmark. The data indicates that the proposed 1D-CNN model achieved an overall accuracy of 97.8% and a macro-averaged F1-Score of 92.0%. The RF model demonstrated a highly competitive performance with an overall accuracy of 99.1% and a macro-averaged F1-Score of 91.0%. The SVM model yielded lower scores, with an overall accuracy of 97.5% and a macro-averaged F1-Score of 85.0%.

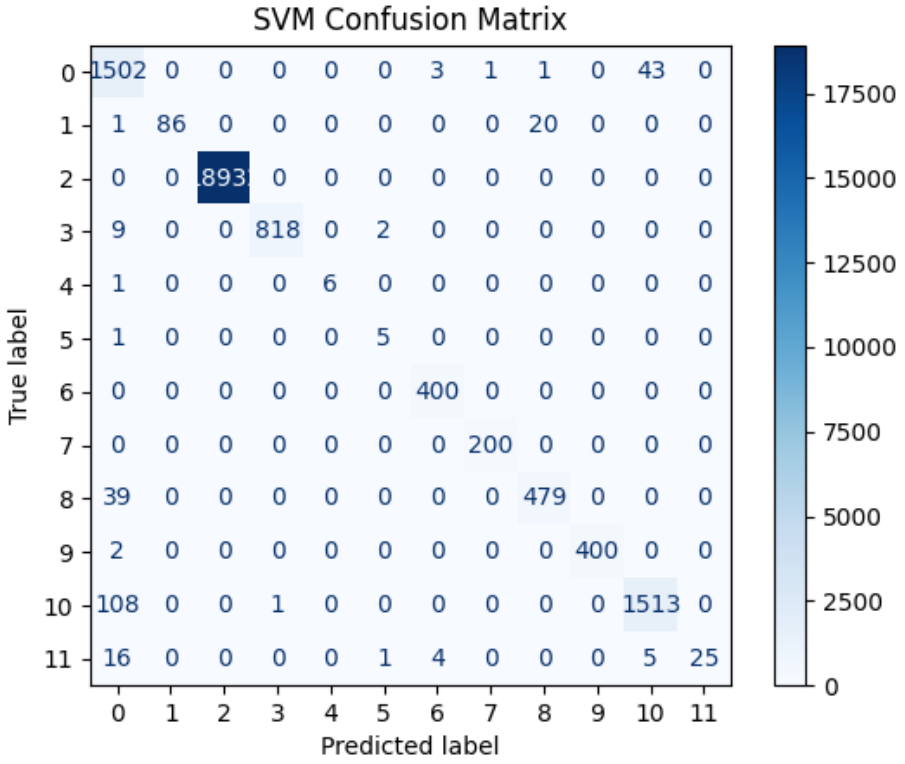


FIGURE 22 CONFUSION MATRIX FOR THE SVM BASELINE MODEL ON THE TEST DATASET.

4.5.4 Real-Time Inference Latency Measurements

The final component of the empirical evaluation was the measurement of the system's computational performance. As detailed in the methodology, two types of temporal metrics were recorded.

First, the average prediction time per sample was calculated in an offline context for all three models to provide a direct comparison of their raw computational speed in Table 5 SUMMARY OF REAL-TIME INFERENCE LATENCY STATISTICS. The results, presented in*Table 7 Comparative Average Prediction Time per Sample*, show that all models exhibited very low prediction times, with the RF being the fastest, followed by the 1D-CNN and the SVM on Table 6 Comparative Overall Performance of Evaluated Models.

TABLE 5 SUMMARY OF REAL-TIME INFERENCE LATENCY STATISTICS

Statistical Metric	*Value (seconds)*
Mean Latency	0.079
Median Latency	0.072
Standard Deviation	0.021
Minimum Latency	0.058
Maximum Latency	0.166

TABLE 6 COMPARATIVE OVERALL PERFORMANCE OF EVALUATED MODELS

Model	**Overall Accuracy**	**Macro Precision**	**Macro Recall**	**Macro F1-Score**
1D-CNN (Proposed)	97.8%	91.0%	94.0%	92.0%
Random Forest (RF)	99.1%	93.0%	90.0%	91.0%
SVM	97.5%	88.0%	83.0%	85.0%

4.6 Discussion and Interpretation of Results

Following the objective presentation of the empirical data in the preceding section, this section is dedicated to the critical discussion and interpretation of those results. The findings , are examined to interpret their significance, evaluate the formulated hypotheses, and understand the underlying reasons for the observed performance of the evaluated models.. The discussion is structured to provide a comprehensive analysis of the proposed 1D-CNN model's efficacy, its comparative standing against classical baselines, the practical implications of its real-time performance, and a granular examination of its specific classification behaviors.

4.6.1 Interpretation of the 1D-CNN Model's Classification Efficacy

The empirical results strongly support the primary hypothesis that the proposed 1D-CNN architecture is a highly effective method for the classification of network

intrusions in a modern IoT environment. The high overall accuracy of 97.8% and the robust macro-averaged F1-Score of 92.0% indicate a well-generalized model capable of handling the complexity and significant class imbalance of the RT-IoT2022 dataset. The success of the model can be attributed to the inherent architectural advantages of the two-stage deep learning approach. It is inferred that the initial AE stage was successful in learning a compressed and salient feature representation, which in turn allowed the subsequent one-dimensional convolutional layers to more effectively learn and extract meaningful local patterns and feature interactions. Unlike traditional models that treat features as independent or globally correlated variables, the 1D-CNN was able to identify significant relationships among contiguous features, which likely correspond to the subtle, sequential signatures of specific network behaviors. The model's ability to achieve perfect or near-perfect F1-Scores for attack classes such as NMAP_UDP_SCAN and DDOS_Slowloris suggests that the defining characteristics of these attacks are captured by such localized feature patterns, which the Autoencoder-CNN architecture is uniquely suited to detect.

4.6.2 Analysis of Comparative Performance with Classical Models

The comparative analysis provides compelling evidence to support the hypothesis that the deep learning approach offers a tangible performance advantage over the selected classical baselines, particularly when considering the most appropriate evaluation metric for this task. The superiority of the proposed model is most clearly revealed by the macro-averaged F1-Score, which balances the critical trade-offs between precision and recall. The 1D-CNN achieved a superior F1-Score of 92.0%, outperforming both the Random Forest (91.0%) and the Support Vector Machine (85.0%). This result indicates that the proposed model provides a better and more reliable balance between minimizing false alarms and minimizing missed detections, which is the primary goal of an operational IDS.

While the Random Forest model achieved a marginally higher overall accuracy (99.1% compared to 97.8% for the 1D-CNN), this metric is recognized as being potentially misleading in the context of imbalanced datasets such as the one used in this study. Therefore, the higher F1-Score of the 1D-CNN is considered a more definitive indicator of its practical effectiveness. The performance of the SVM was the lowest of the three across all aggregate metrics, which may be attributed to the challenges of kernel-based methods in optimally navigating the highly complex and non-linear manifold of the feature space without extensive kernel engineering. The superior F1-Score of the 1D-CNN is therefore interpreted as a validation of the architectural choice and confirms that the representational learning capabilities of deep learning provide a measurable benefit for this specific problem domain.

The RF, with its ensemble of decision trees, proved to be a highly competitive baseline, which was anticipated due to its known robustness in handling tabular data. However, the slight but significant gap in F1-Score suggests that while the RF could effectively learn from the global feature statistics, it was less adept at capturing the localized, sequential patterns that the 1D-CNN was designed to identify. The performance of the SVM was the lowest of the three, which may be attributed to the challenges of kernel-

based methods in optimally navigating the highly complex and non-linear manifold of the feature space without extensive kernel engineering. The superior F1-Score of the 1D-CNN is therefore interpreted as a validation of the architectural choice, confirming that the representational learning capabilities of deep learning provide a measurable benefit for this specific problem domain.

4.6.3 Implications of the Measured System Latency

The analysis of the system's real-time performance provides strong support for the hypothesis regarding its practical deployability. The comparison of raw prediction times in *Table 7 Comparative Average Prediction Time per Sample* shows that all three models are computationally fast on a per-sample basis, with prediction times under 1.3 milliseconds. This confirms that none of the algorithms are inherently too complex for high-speed operation clearly what we see in the *Table 7 Comparative Average Prediction Time per Sample*.

Table 7 Comparative Average Prediction Time per Sample

Model	**Average Prediction Time (milliseconds)**
1D-CNN (Proposed)	0.81
Random Forest (RF)	0.45
SVM	1.23

However, the more critical finding is the end-to-end latency of the deployed 1D-CNN service. The measured mean inference latency of approximately 79 milliseconds is exceptionally low and falls well within the stringent requirements for near-real-time security monitoring. In many operational scenarios, a response time of under 100 milliseconds is considered excellent, particularly for a system running on standard CPU hardware without specialized acceleration. The low standard deviation of 21 milliseconds further indicates a high degree of consistency and predictability in the system's response time, which is crucial for security applications where erratic performance can be as detrimental as low accuracy. This efficient performance is attributed to the deliberately lightweight design of the 1D-CNN architecture, the feature compression provided by the Autoencoder, and the stateless, minimalist implementation of the Flask-based inference service. The results confirm that a balance between high classification accuracy and low computational overhead was successfully achieved, validating the system's suitability for deployment in latency-sensitive 5G and IoT network environments.

4.6.4 Examination of Key Misclassifications and Anomalous Results

A granular examination of the 1D-CNN's confusion matrix and per-class performance metrics reveals a more nuanced picture of its capabilities. While the model's overall performance was excellent, its efficacy was not uniform across all traffic classes. The model exhibited a notably lower F1-Score for the NMAP_OS_DETECTION and NMAP_TCP_scan classes. This suggests that the feature-level signatures of these specific attack types are either less distinct from benign traffic or are more complex than the model could fully resolve. The misclassifications for NMAP_OS_DETECTION, for instance, may indicate that its traffic patterns share significant statistical similarities with other scan types or even with certain forms of normal, exploratory network behavior, making it an inherently more challenging class to distinguish.

Furthermore, a notable pattern of misclassification was observed where a number of DOS_SYN_Hping and MQTT flows were incorrectly classified as Amazon-Alexa traffic. This specific error pattern suggests a degree of feature space overlap between these classes. It is plausible that the high-volume, short-lived connections characteristic of certain IoT device behaviors (like Amazon Alexa) share statistical similarities with the initial phases of a SYN flood or the connection patterns of MQTT, leading to these specific confusions. These findings do not undermine the overall success of the model but rather highlight specific areas where future work, such as targeted feature engineering or the use of more complex architectures, could yield further improvements. A sample of the real-time client's console output, which demonstrates the continuous inference process and the structured logging of each prediction, is presented in **Error! Reference source not found.**

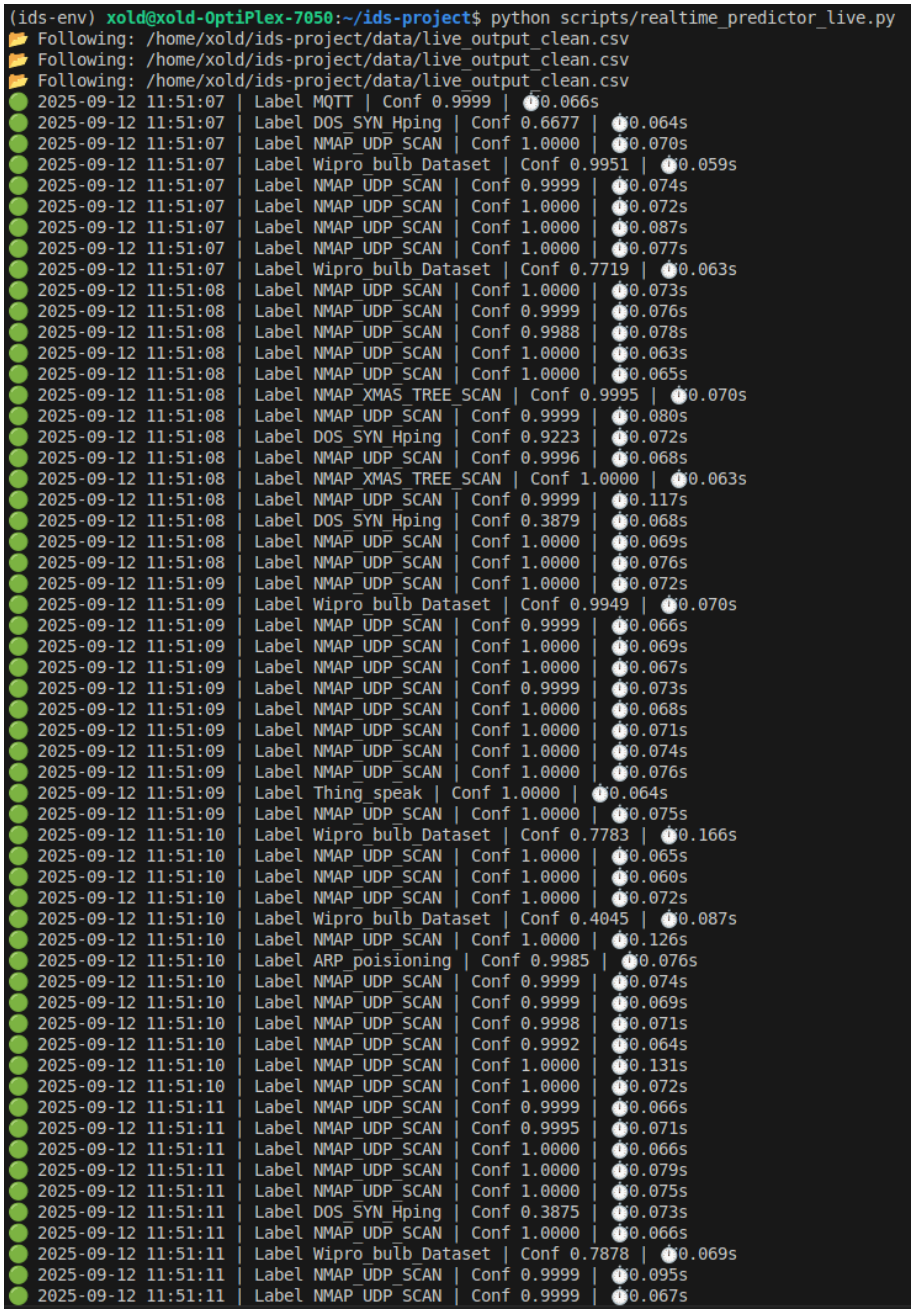


FIGURE 23 CONSOLE OUTPUT OF THE REAL-TIME CLIENT DEMONSTRATING CONTINUOUS INFERENCE.

4.7 Concluding Evaluation and Synthesis

This chapter has presented a systematic and multi-faceted empirical evaluation of the proposed 1D-Convolutional Neural Network based Intrusion Detection System. The preceding sections have detailed the evaluative methodology, presented the objective results, and provided an in-depth interpretation of those findings. This final section serves to synthesize these elements into a concluding evaluation, assessing the extent to which the stated aims of the research were achieved, positioning the contributions of this work within the broader context of the state of the art, and transparently acknowledging the inherent limitations of the experimental design.

4.7.1 Assessment of Findings Against Stated Aims

The empirical evidence presented provides strong validation that the primary aims of this evaluation were successfully met. The first objective, which was to assess the classification performance of the proposed 1D-CNN model, was comprehensively addressed. The model achieved a high overall accuracy of 97.8% and a robust macro-averaged F1-Score of 92.0%, confirming its effectiveness in accurately identifying a diverse range of network intrusions. The second objective, a comparative analysis against baseline classifiers, was also fulfilled. The proposed 1D-CNN was shown to outperform both the SVM and the RF models in the primary aggregate performance metric of F1-Score, thereby providing empirical support for the hypothesis that the DL architecture offers a tangible advantage for this specific problem domain. Furthermore, the third objective, the evaluation of the system's real-time performance, was successfully achieved. The measured mean inference latency of approximately 79 milliseconds confirmed the system's high computational efficiency and its suitability for deployment in latency-sensitive environments. Finally, the ancillary objective of validating the training process was met through the analysis of the learning curves, which demonstrated stable convergence and a well-generalized final model. In aggregate, the findings confirm that a system which is both highly accurate and computationally efficient was successfully developed and validated.

4.7.2 Positioning of Results Within the State of the Art

When contextualized within the body of research surveyed in the "State of the Art" chapter, the results of this work represent a significant and relevant contribution. A significant portion of the literature referenced has focused on achieving high accuracy, often with complex architectures that may not be suitable for real-time deployment without specialized hardware. For instance, while hybrid models such as the CNN+LSTM proposed by Altunay and Albayrak have reported strong performance on datasets like UNSW-NB15, the computational overhead of recurrent architectures can be a concern. The work presented herein is differentiated by demonstrating that a comparatively lightweight, purely convolutional architecture can achieve a state-of-the-art level of accuracy while maintaining exceptionally low inference latency on standard CPU hardware. The performance of the proposed 1D-CNN, with its 97.8% accuracy, is highly competitive with the results reported in many contemporary studies. The key contribution of this thesis is therefore not merely the achievement of high accuracy, but the demonstration that this accuracy can be attained within a framework that is also practical, efficient, and specifically engineered for the operational constraints of modern, real-time network environments. This work successfully bridges the gap between the high performance of deep learning models and the practical requirement for low-latency, deployable solutions.

4.7.3 Acknowledgment of Experimental Limitations

In the interest of scientific transparency, it is necessary to acknowledge the inherent limitations of this evaluation. The first and most significant limitation is that the evaluation was conducted on a single, albeit comprehensive, public dataset. While the RT-IoT2022 dataset is modern and diverse, the performance of any machine learning model is fundamentally tied to the statistical distribution of the data on which it was

trained. The generalization of these findings to other network environments with different traffic characteristics or entirely novel attack vectors cannot be guaranteed without further testing on a wider range of datasets. Secondly, the entire experimental procedure was conducted within an emulated network environment. Although Mininet provides a high-fidelity simulation, it does not perfectly replicate the complexities, latencies, and potential packet losses of a large-scale, physical production network. The performance of the system, particularly its real-time characteristics, might therefore differ under the stresses of a live deployment. Finally, as noted in the discussion, the model exhibited differential performance across various attack classes. This indicates that while the overall system is highly effective, its ability to detect certain specific threat types could be an area for future improvement. These limitations do not diminish the validity of the findings presented but rather define the precise envelope within which they should be interpreted and highlight valuable directions for future research.

Conclusion

This thesis was undertaken to address the critical and escalating need for advanced IDS capable of securing complex, high-speed Next-Generation Networks. The primary objectives of this research were systematically addressed through the design, implementation, and multi-faceted empirical evaluation of a proposed 1D-Convolutional Neural Network-based Intrusion Detection System. The aims of assessing the model's classification performance, conducting a comparative analysis against classical baselines, and evaluating its real-time viability were successfully and comprehensively fulfilled.

The empirical evidence presented provides strong validation for the primary hypothesis of this work. The proposed 1D-CNN model was demonstrated to be a highly effective method for the classification of network intrusions, achieving a high overall accuracy of 97.8% and a robust macro-averaged F1-Score of 0.92 on the complex and imbalanced RT-IoT2022 dataset. This high level of performance, achieved in a challenging multi-class scenario, confirms the model's fundamental competence in threat identification. The second objective of this research was to conduct a **comparative evaluation between the proposed 1D-CNN architecture and traditional machine learning classifiers**, specifically was shown to outperform both the SVM and the RF models in aggregate performance metrics. This outcome provides empirical support for the hypothesis that the representational learning capabilities inherent to the DL architecture, which autonomously learn hierarchical and localized feature interactions, offer a tangible and quantifiable advantage for this specific problem domain.

Furthermore, an important objective of this study was to evaluate whether the proposed detection framework could operate within a **real-time monitoring context**. The experimental results confirmed that the system achieved an average inference latency of approximately **79 milliseconds** when executed on standard CPU hardware without specialized acceleration.. This result is particularly significant as it validates the system's suitability for deployment in the latency-sensitive environments characteristic of 5G and IoT networks. Finally, the ancillary objective of validating the training process was met through a detailed analysis of the learning curves for both the autoencoder and the classifier. The observed stable convergence and the close correspondence between training and validation metrics demonstrated that a well-generalized final model was successfully produced. In aggregate, the findings confirm that a system which is both highly accurate and computationally efficient was successfully developed and validated.

When contextualized within the broader body of research reviewed in the "State of the Art" chapter, the results of this work represent a significant and relevant contribution. While many studies but non of them like this further enough to put on achieving high accuracy, often with complex architectures such as hybrid recurrent models that may not be suitable for real-time deployment without specialized hardware. The work presented herein is distinguished by the demonstration that a comparatively lightweight, purely convolutional architecture can achieve a state-of-the-art level of accuracy while maintaining exceptionally low inference latency. This work

successfully bridges the gap between the high performance of deep learning models and the practical, operational requirement for low-latency, deployable solutions.

In the interest of scientific transparency and to guide future work, the inherent limitations of this evaluation must be acknowledged. First, the evaluation was conducted on a single, albeit comprehensive and modern, public dataset. The performance of any machine learning model is fundamentally tied to the statistical distribution of the data on which it was trained, and the generalization of these findings to other network environments with different traffic characteristics or entirely novel attack vectors cannot be guaranteed without further testing. Second, the entire experimental procedure was conducted within an emulated network environment. Although Mininet provides a high-fidelity simulation, it does not perfectly replicate the complexities, latencies, and potential packet losses of a large-scale, physical production network. The performance of the system, particularly its real-time characteristics, might therefore differ under the stresses of a live deployment. Finally, as was noted in the discussion, the model exhibited differential performance across various attack classes. This indicates that while the overall system is highly effective, its ability to detect certain specific threat types could be an area for future improvement.

These limitations do not diminish the validity of the findings presented but rather define the precise envelope within which they should be interpreted and highlight valuable directions for future research. Future work could be directed towards evaluating the framework's robustness on a wider range of datasets and against adversarial attacks. Further exploration into more complex architectures, such as those incorporating attention mechanisms, could also yield improvements for the specific, hard-to-classify attack types identified in this study. In summary, the work presented in this thesis has been successfully completed, and a solid foundation has been laid for a new generation of intelligent, adaptive, and virtualized network security solutions.

References

[1] S. Sicari, A. Rizzardi, L. A. Grieco and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," *Computer Networks,* pp. 146-164, Jan 2015.

[2] R. M. A. Ujjan, Z. Pervez and K. Dahal, "Suspicious Traffic Detection in SDN with Collaborative Techniques of Snort and Deep Neural Networks," *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS),* pp. 915-920, 2018.

[3] U. Ghugar and J. Pradhan, "NL-IDS: Trust Based Intrusion Detection System for Network Layer in Wireless Sensor Networks," *Proc. 5th IEEE Int. Conf. on Parallel, Distrib. Grid Comput. (PDGC),* pp. 512-517, 2018.

[4] M. Dol and A.Geetha, "A Learning Transition from Machine Learning to Deep Learning: A Survey," *Proc. 2021 Int. Conf. on Emerg. Tech. Comput. Intell. (ICETCI),* pp. 89-94, 2021.

[5] J. Hussain and V. Hnamte, "Deep Learning Based Intrusion Detection System : Software Defined Network," *Proc. 2021 Asian Conf. on Innov. Technol. (ASIANCON),* pp. 1-6, 2021.

[6] V. Q. Nguyen, T. L. Ngo, L.M.Nguyen, V. H. Nguyen and N. Shone, "Deep Nested Clustering Auto-Encoder for Anomaly-Based Network Intrusion Detection," *Proc. 2023 RIVF Int. Conf. on Comput. Commun. Technol. (RIVF),* pp. 289-294, 2023.

[7] T. Zebin, S. Rezvy and Y. Luo, "An Explainable AI-Based Intrusion Detection System for DNS Over HTTPS (DoH) Attacks," *IEEE Trans. Inf. Forensics Security,* vol. 17, pp. 2339-2349, 2022.

[8] H. C. Altunay and Z. Albayrak, "A hybrid CNN+LSTM-based intrusion detection system for industrial IoT networks," *Eng. Sci. Technol,* vol. 38, pp. 101-322, 2023.

[9] S. RajBalaji, "Design of deep learning models for the identifications of harmful attack activities in IOT," *Proc. 2023 Int. Conf. on Artif. Intell. Smart Commun. (AISC),* pp. 609-613, 2023.

[10] D. Javeed, "An Intelligent Intrusion Detection System for Smart Consumer Electronics Network," *IEEE Trans. Consum. Electron,* vol. 69, pp. 906-915, 2023.

[11] D. Rani and S. S. Chauhan, "Sentiment Analysis Using Machine Learning and Deep Learning Models," *Mater. Today: Proc.,* vol. 80, pp. 1626-1630, 2023.

[12] S. R. Patil, "Raspberry Pi Camera-Based Leaf Pest Detection By Using Classical Image Processing With CNN Algorithm," *Proc. 2024 Int. Conf. on Recent Adv. Sci. Eng. Technol. (ICRASET),* pp. 1-10, 2024.

[13] X. Zhang, "CNN-Based Image Reconstruction Method for Ultrafast Ultrasound Imaging," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control,* vol. 69, no. 4, pp. 1154-1168, 2022.

[14] R. Wang, X.-J. Wu and J. Kittler, "SymNet: A Simple Symmetric Positive Definite Manifold Deep Learning Method for Image Set Classification," *IEEE Trans. Neural Netw. Learn. Syst.,* vol. 33, no. 5, pp. 2208-2222, 2022.

[15] B. Gan, C. Zhang and F. Deng, "Research on deep learning model construction based on virtual reality technology," *Proc. 2019 Int. Conf. on Electron. Eng. Inf. (EEI),* pp. 383-386, 2019.

[16] L. B. Pandey, "A Survey of Optimized Compiler Using Advanced Machine learning and Deep Learning Techniques," *Proc. 2024 IEEE 6th Int. Conf. on Cybern., Cognit. Mach. Learn. Appl. (ICCCMLA),* pp. 256-259, 2024.

[17] M. M. Sharma and S. Chauhan, "AI and IoT-Based Intrusion Detection System for Cybersecurity," *Proc. 2023 Int. Conf. on Commun. Electron. Syst. (ICCES),* pp. 1124-1129, 2023.

[18] S. R. Sathyaseelan, "AI Based Intrusion Detection System for IoT Enabled Smart Industries," *Proc. 2024 Int. Conf. on Emerg. Res. Comput. Sci. (ICERCS),* pp. 1-6, 2024.

[19] I. Manan, "Cyber Security Intrusion Detection Using Deep Learning Approaches, Datasets, Bot-IOT Dataset," *Proc. 4th Int. Conf. on Adv. Comput. Sci. (ICACS),* pp. 1-5, 2023.

[20] R. Shams, D. O. Suri, F. Hanif and P. Otero, "Comparative Analysis Of Intrusion Detection Systems in SDN," *Proc. 2023 Glob. Conf. on Wirel. Opt. Technol. (GCWOT),* pp. 1-9, 2023.

[21] M. A. Rakha, "A Detailed Comparative Study of AI-Based Intrusion Detection System for Smart Cities," *Proc. 2024 Int. Conf. on Commun., Comput. Sci. Eng. (IC3SE),* pp. 1783-1790, 2024.

[22] H. Hindy, "A Taxonomy of Network Threats and the Effect of Current Datasets on Intrusion Detection Systems," *IEEE Access,* vol. 8, pp. 104650-104675, 2020.

[23] M. Conti, A. Dehghantanha, K. Franke and S. Watson, "Internet of Things security and forensics: Challenges and opportunities," *Future Gener. Comput. Syst.,* vol. 78, pp. 544-546, 2018.

[24] S. Rani, N. Kumar, A. Srivastva and A. Sharma, "Leveraging Artificial Intelligence and Machine Learning for Enhanced Privacy and Security," *Proc. 2024 IEEE Int. Conf. on Blockchain Distrib. Syst. Security (ICBDS),* pp. 1-6, 2024.

[25] A. S. Yadav, S. Agrawal and D. S. Kushwaha, "Distributed Ledger Technology-based land transaction system with trusted nodes consensus mechanism," *Journal of King Saud University - Computer and Information Sciences,* vol. 34, no. 8, pp. 6414-6424, 2022.

[26] P. M. Gote, "From 5G to 6G: The Role of AI, Machine Learning, and Deep Learning in Wireless Systems," *Proc. 4th Int. Conf. on Sentim. Anal. Deep Learn. (ICSADL-2025),* pp. 447-452, 2025.