TCN-HMAC: A Lightweight Deep Learning and Cryptographic Hybrid Security Framework for SDN

A Research Report
Submitted in partial fulfillment of the requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

Submitted by

Md. Mahamudul Hasan Zubayer 2022000000006

Supervised by

Dr. Md. Maruf Hassan

Associate Professor
Department of Computer Science and Engineering
Southeast University, Bangladesh

Department of Computer Science and Engineering
Southeast University, Bangladesh

Dhaka, Bangladesh

30 October, 2025

Letter of Transmittal

30 October, 2025

The Chairman,
Department of Computer Science and Engineering
Southeast University, Bangladesh
Tejgaon, Dhaka

Through: Supervisor, Dr. Md. Maruf Hassan

Subject:

Dear Sir,

It is a great satisfaction to submit our research report on "**TCN-HMAC: A Lightweight Deep Learning and Cryptographic Hybrid Security Framework for SDN**" under the course, Research Methodology. It was an honor for us to work with this topic. By following your instruction and fulfilling the requirement of the Southeast University, this research has been performed. We have prepared this report with our absolute sincerity and effort. We request your approval of this research report in partial fulfillment of our degree requirement.

Sincerely Yours,

Md. Mahamudul Hasan Zubayer
2022000000006

Supervisor:

Dr. Md. Maruf Hassan
Associate Professor & Supervisor
Department of Computer Science and Engineering
Southeast University, Bangladesh

CANDIDATE'S DECLARATION

We, hereby, declare that the thesis presented in this report is the outcome of the investigation performed by us under the supervision of Dr. Md. Maruf Hassan, Associate Professor, Department of Computer Science and Engineering, Southeast University, Bangladesh. The work was done through CSE459: Research Methodology course, in accordance with the course curriculum of the Department for the Bachelor of Science in Computer Science and Engineering program.

It is also declared that neither this research nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Md. Mahamudul Hasan Zubayer
2022000000006

CERTIFICATION

This research titled, **“TCN-HMAC: A Lightweight Deep Learning and Cryptographic Hybrid Security Framework for SDN”**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in 30 October, 2025.

Group Members:

Md. Mahamudul Hasan Zubayer	**2022000000006**

Supervisor:

Dr. Md. Maruf Hassan
Associate Professor & Supervisor
Department of Computer Science and Engineering
Southeast University, Bangladesh

Shahriar Manzoor
Associate Professor & Chairman
Department of Computer Science and Engineering
Southeast University, Bangladesh

ACKNOWLEDGEMENT

Foremost, Thanks to Almighty for whose given strength makes us able to complete our research successfully.

After that, we would like to thank our honorable supervisor Dr. Md. Maruf Hassan, Associate Professor, Department of Computer Science Engineering in Southeast University. This research behind it would not have been possible without the exceptional support of our supervisor. His enthusiasm, knowledge, and exacting attention to detail have been an inspiration and kept us on track from our first encounter with machine learning to the final draft of this paper.

We would also like to thank Mr. Shahriar Manzoor, Associate Professor & Chairman, Department of Computer Science and Engineering, Southeast University for making Dr. Md. Maruf Hassan, our supervisor. We also want to thank our whole faculties of the department of Computer Science and Engineering, Southeast university for their encouragement and motivation. Finally, it is true with pleasure and appreciation that we acknowledge our contributions and day after day hard work with proper responsibility.

In addition, we owe a sincere debt of gratitude to each and every one of our great group members for their thought-provoking debates, unwavering commitment, and endless sleepless hours spent together, which were essential to finishing this project ahead of schedule. We were able to finish this report within the allotted time thanks to their unflagging encouragement and assistance.

Finally, we want to say that all of those kindnesses were indispensable to complete our research.

Md. Mahamudul Hasan Zubayer

Dhaka
30 October, 2025

ABSTRACT

Software-Defined Networking (SDN) has reshaped modern network management by separating the control plane from the data plane, bringing centralized programmability and fine-grained traffic control. That very centralization, though, opens the door to serious security risks—network intrusion, information theft, eavesdropping, and, in worst-case scenarios, complete network failure. As attack strategies grow more sophisticated, relying on conventional firewalls alone is no longer tenable, and while TLS encryption can protect the control channel, its computational overhead makes it a poor fit for resource-constrained SDN deployments. This thesis presents a lightweight hybrid security framework that tackles these threats on two fronts. The first layer of defense is a Temporal Convolutional Network (TCN) deployed as a TensorFlow/Keras-based controller application; it inspects flow statistics in real time and flags intrusions. Backing this up is an auxiliary agent that runs as a co-located subprocess, verifying every flow rule installation through HMAC-based integrity checks and periodically executing a challenge–response protocol to confirm controller authenticity. We evaluate the framework on the InSDN dataset, which covers DDoS, MITM, Probe, and Brute-force attack categories. The TCN achieves 99.85% classification accuracy, a 99.97% detection rate, and a false alarm rate of just 0.37%, while the HMAC verification adds negligible computational overhead—roughly 0.7 ms per operation. Taken together, these results demonstrate that the proposed TCN-HMAC approach delivers robust, multi-layered SDN security without sacrificing the real-time performance modern networks demand.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Enterprise and cloud networking have been reshaped, in large part, by the rise of Software-Defined Networking (SDN). The core idea is straightforward: decouple the control logic from the forwarding hardware so that a single programmable controller can orchestrate the behaviour of many otherwise "dumb" switches [1]. The payoff is considerable—administrators gain a global view of the network, can deploy policies in seconds rather than hours, and are no longer locked in to any one vendor's hardware [2]. These advantages explain why SDN now underpins data centers, campus fabrics, wide-area overlays, and carrier-grade infrastructures around the world. Yet the same centralization that makes SDN powerful also makes it fragile. A compromised controller can unravel the entire network. The control channel, through which all configuration messages flow, becomes a prime target for interception or manipulation. And once a tampered flow rule reaches a switch, the switch will execute it faithfully—no questions asked [3].

The problem is getting worse, not better. As organizations move mission-critical services onto SDN-enabled infrastructure, the attack surface grows in proportion. Modern adversaries rarely rely on a single trick; instead, they chain volumetric floods with quiet reconnaissance, credential stuffing, and channel interception to hit SDN deployments at multiple layers at once [4]. Traditional perimeter defenses—firewalls, signature-matching IDSes, static access lists—were built for a world of distributed, largely static networks. They simply cannot keep pace with SDN's programmable, fast-changing nature. What is needed, then, are security mechanisms woven into the SDN fabric itself: mechanisms that run at line speed, adapt to novel threats, and protect the architecture from the inside out.

This thesis introduces **TCN-HMAC**, a hybrid framework that tackles this challenge by pairing two complementary defenses. On the detection side, a Temporal Convolutional Network (TCN) learns temporal patterns in flow-level statistics and classifies traffic as benign or malicious with high accuracy. On the integrity side, a lightweight auxiliary agent uses Hash-based Message Authentication Codes (HMACs) to verify that every flow rule reaching a switch is authentic and unaltered, while periodic challenge–response exchanges confirm the controller's identity. Neither layer alone is sufficient—an IDS cannot prevent tampered rules, and integrity checks cannot spot a well-crafted DDoS flood—but together they form a defense-in-depth strategy that is lean enough for real-time deployment yet broad enough to counter a wide range of network attacks.

1.1 Background and Motivation

1.1.1 Software-Defined Networking Architecture

In a conventional network, every router and switch runs its own control logic, computing forwarding decisions through distributed protocols like OSPF, BGP, or IS-IS. The result is a decentralized architecture that works—but one that is notoriously hard to manage, debug, and evolve [2]. SDN breaks with this tradition by pulling the control logic out of the devices and concentrating it in a single software entity, the SDN controller, which keeps a panoramic view of the network's topology and state. The forwarding devices become programmable switches that simply carry out whatever flow rules the controller hands them.

The communication between the control plane and the data plane is facilitated by a well-defined southbound interface, the most prominent of which is the OpenFlow protocol [1]. OpenFlow enables the controller to install, modify, and delete flow rules in the switch flow tables, query switch statistics, and receive asynchronous notifications about network events such as new flows, port status changes, and error conditions. The OpenFlow specification defines a match-action paradigm: each flow rule consists of a set of match fields (e.g., source and destination IP addresses, port numbers, protocol type) and an associated set of actions (e.g., forward to a specific port, drop, modify headers, send to controller) [5]. When a packet arrives at a switch, the switch examines its flow table for a matching rule. If a match is found, the corresponding action is executed; otherwise, the packet is forwarded to the controller via a `Packet-In` message for further processing.

The SDN architecture is typically described as a three-layer model. The *infrastructure layer* (data plane) comprises the physical and virtual switches that forward packets. The *control layer* hosts the SDN controller, which runs on commodity servers and provides core network services such as topology discovery, path computation, and flow management. The *application layer* sits atop the controller and communicates with it through northbound APIs; it encompasses a diverse ecosystem of network applications including load balancers, firewalls, intrusion detection systems, and quality-of-service managers. This layered architecture enables rapid innovation, as new network functions can be developed as software applications without modifying the underlying hardware.

1.1.2 Security Challenges in SDN

The operational benefits of centralized programmability come at a price. SDN's architecture introduces a qualitatively different set of security concerns—concerns that simply do not arise, or arise far less acutely, in traditional networks [3]. Several of the most pressing ones are outlined below.

Single Point of Failure and Attack. The SDN controller, by virtue of its centralized role, becomes the most critical component in the network. A successful attack against the controller—whether through resource exhaustion, exploitation of software vulnerabilities, or unauthorized access—can compromise the entire network. If the controller is rendered unavailable, switches lose their ability to handle new flows, effectively causing a network-wide denial of service. If the controller is compromised, the attacker gains the ability to manipulate all flow rules across all switches, enabling arbitrary traffic redirection, eavesdropping, or data exfiltration.

Control Channel Vulnerability. The communication channel between the controller and the switches carries highly sensitive configuration information, including flow rule installations, modifications, and deletions. If this channel is not adequately protected, an adversary positioned along the path can intercept, modify, or inject messages. While the OpenFlow specification recommends the use of TLS for securing the control channel, empirical studies have shown that a significant fraction of SDN deployments either do not enable TLS or use it with weak configurations [6]. Moreover, TLS protects only the transport layer; it does not verify the semantic integrity of the flow rules themselves. An attacker who compromises the controller or who injects messages through a vulnerability in the control application can still install malicious flow rules even when TLS is active.

Flow Rule Integrity. Flow rules are the fundamental units of network policy in SDN. Once a rule is installed in a switch's flow table, the switch executes the associated actions without re-verification. This means that a tampered or maliciously crafted flow rule will be faithfully executed, potentially redirecting sensitive traffic to an attacker-controlled host, creating forwarding loops, or dropping critical packets. The lack of a built-in mechanism for switches to independently verify the integrity and authenticity of received flow rules constitutes a significant vulnerability [7,8].

Scalability of Security Mechanisms. SDN environments are inherently dynamic: flow rules are installed, modified, and evicted at high rates to accommodate changing traffic patterns and application requirements. Any security mechanism deployed in an SDN environment must therefore operate at a speed commensurate with the rate of flow rule changes. Heavyweight cryptographic operations, complex protocol negotiations, or computationally expensive deep learning inference can introduce latency that degrades network performance and user experience. This tension between security strength and operational overhead is a central challenge in SDN security design.

1.1.3 Attack Taxonomy in SDN Environments

SDN environments are susceptible to a wide range of attacks that target different layers and components of the architecture. This thesis focuses on four major attack categories that are representative of the most prevalent and impactful threats to SDN deployments.

Distributed Denial of Service (DDoS) Attacks. DDoS attacks represent one of the most severe threats to SDN infrastructure. In a DDoS attack, an adversary coordinates a large number of compromised hosts (a botnet) to flood the target with an overwhelming volume of traffic, exhausting network bandwidth, switch flow table capacity, or controller processing resources [4]. In the SDN context, DDoS attacks are particularly dangerous because of the reactive flow installation mechanism: each new flow that does not match an existing rule triggers a `Packet-In` message to the controller. An attacker who generates a large number of flows with randomized headers can therefore overwhelm the controller with `Packet-In` messages, saturate the control channel, and exhaust the flow table capacity of switches. This table-flooding variant of DDoS is unique to SDN and can render the entire network inoperable even if the volumetric traffic load is relatively modest. Beyond the control plane, conventional bandwidth-exhaustion DDoS attacks also disrupt service availability at the data plane level, affecting legitimate users and applications.

Man-in-the-Middle (MITM) Attacks. MITM attacks in SDN involve an adversary inserting themselves into the communication path between two legitimate parties—typically between the controller and a switch, or between two end hosts [9]. By intercepting and potentially modifying messages in transit, the attacker can eavesdrop on sensitive data, alter flow rule installations, inject spurious messages, or impersonate one party to the other. In the SDN context, a MITM attack on the control channel is especially devastating because it allows the attacker to observe all controller–switch communications, including topology information, flow rule contents, and network policies. The attacker can then selectively modify flow rule messages to redirect traffic through attacker-controlled nodes, create covert channels, or degrade network performance in subtle ways that are difficult to detect. Even when TLS is employed, MITM attacks remain possible if certificate validation is improperly implemented, if the attacker can compromise a certificate authority, or if the attacker targets the application layer above TLS.

Probe (Reconnaissance) Attacks. Probe attacks encompass a range of reconnaissance activities in which an adversary systematically scans the network to discover active hosts, open ports, running services, and exploitable vulnerabilities [10]. Common probe techniques include TCP SYN scans, UDP scans, ICMP sweeps, and OS fingerprinting. While probe attacks do not directly cause damage, they serve as the precursor to more targeted intrusions by providing the attacker with a detailed map of the network's attack surface. In SDN environments, probe attacks are particularly informative for adversaries because the centralized control model means that discovering the controller's address, the OpenFlow ports of switches, or the topology structure can reveal critical information about the network's management architecture. Effective detection of probe attacks is therefore essential for disrupting the attack kill chain at an early stage before the adversary can leverage the gathered intelligence for exploitation.

Brute-Force Attacks. Brute-force attacks involve the systematic enumeration of credentials—typically usernames and passwords—to gain unauthorized access to network devices, management interfaces, or services [11]. In the SDN context, brute-force attacks may target the controller's northbound API, the web-based management GUI, SSH access to switches, or application-level authentication mechanisms. A successful brute-force attack against the SDN controller grants the adversary full control over the network, enabling them to reconfigure flow rules, exfiltrate data, or deploy persistent backdoors. Brute-force attacks generate distinctive temporal patterns in network traffic—repeated connection attempts to authentication endpoints, short session durations followed by reconnection, and systematic variation of credentials—that can be captured and identified by temporal pattern recognition models.

1.1.4 Limitations of Existing Approaches

The literature on SDN security falls, broadly speaking, into three camps: cryptographic and protocol-based mechanisms, machine-learning and deep-learning intrusion detection systems, and hybrid schemes that draw from both. None of these camps, on its own, provides everything a production SDN deployment actually needs. The specific shortcomings of each are discussed next.

Cryptographic and Protocol-Based Approaches. TLS-based protection of the control channel, while theoretically sound, suffers from practical deployment challenges including certificate management complexity, computational overhead of asymmetric cryptography, and the inability to verify flow rule semantics [6]. Heavyweight cryptographic frameworks such as those based on public-key infrastructure (PKI) or digital signatures impose significant computational costs that can degrade controller and switch performance, particularly in high-throughput environments. Blockchain-based approaches provide strong integrity guarantees and decentralized trust but introduce consensus-related latency and storage overhead that are often prohibitive for real-time SDN operations. Flow rule verification schemes such as EnsureS [8] and modular HMAC frameworks [7] offer promising lightweight alternatives but typically focus exclusively on integrity verification without addressing intrusion detection, leaving the network vulnerable to attacks that do not involve flow rule tampering.

Machine Learning and Deep Learning Approaches. Traditional machine learning models—including Random Forests, Support Vector Machines, and Decision Trees—have been applied to intrusion detection in SDN with moderate success, but they rely heavily on manual feature engineering, struggle with high-dimensional data, and exhibit limited ability to capture complex temporal dependencies in network traffic. Deep learning approaches, including Convolutional Neural Networks (CNNs), Long Short-Term Memory (LSTM) networks, and their variants, have demonstrated superior detection accuracy [12, 13]. However, recurrent architectures such as LSTMs suffer from sequential processing constraints that limit inference speed, while standard CNNs lack the temporal awareness needed to model the sequential nature of network flows effectively. Furthermore, most deep learning-based IDS solutions operate in isolation, providing detection without integrity verification, and many are evaluated only offline without consideration of deployment constraints such as model size, inference latency, and integration with the SDN control loop. Advanced approaches such as deep reinforcement learning-based IDS [14] show promise but add complexity in terms of training stability and real-world deployment.

Hybrid Approaches. While some studies have explored hybrid architectures that combine detection with verification, existing hybrid solutions tend to be either computationally expensive (e.g., combining deep learning with blockchain) or narrowly focused (e.g., addressing only DDoS or only MITM attacks). A comprehensive, lightweight hybrid framework that seamlessly integrates temporal deep learning for multi-class intrusion detection with efficient cryptographic verification for flow rule integrity—while being deployable as a standard SDN controller application—remains an open research challenge.

1.1.5 The Case for Temporal Convolutional Networks

Why choose a Temporal Convolutional Network over the recurrent architectures—LSTMs, GRUs—that have long dominated sequence modelling? The short answer is that TCNs offer a better set of trade-offs for real-time security applications [15]. A TCN applies causal, dilated one-dimensional convolutions: causality ensures that predictions depend only on current and past inputs, while exponentially increasing dilation factors let the network "see" far back in the sequence without stacking an impractical number of layers. For intrusion detection, this combination carries several concrete advantages.

First, TCNs process entire sequences in parallel through convolutional operations, enabling significantly faster training and inference compared to the inherently sequential computation required by LSTMs and GRUs. This parallelism is critical for real-time intrusion detection, where inference must complete within strict latency budgets to avoid degrading network performance. Second, TCNs exhibit stable gradient behavior during training, avoiding the vanishing and exploding gradient problems that plague deep recurrent networks. Third, the receptive field of a TCN can be precisely controlled through the choice of kernel size and dilation factors, allowing the network architect to explicitly balance temporal coverage against model complexity. Fourth, the fixed-size convolutional filters of TCNs yield compact model sizes (e.g., 612 KB for the proposed model), facilitating deployment on resource-constrained platforms including SDN controllers and edge devices. Recent studies have demonstrated the effectiveness of TCNs for network intrusion detection tasks [16], achieving competitive or superior accuracy relative to recurrent architectures while requiring substantially less inference time.

1.1.6 The Role of HMAC in Flow Rule Integrity

Hash-based Message Authentication Codes (HMACs) provide a computationally efficient mechanism for verifying both the integrity and the authenticity of a message [17]. An HMAC is computed by applying a cryptographic hash function (e.g., SHA-256) to the concatenation of a shared secret key and the message content. The resulting fixed-size tag is appended to the message; the recipient can verify the tag by recomputing the HMAC with the same key and comparing the result. HMAC verification is substantially faster than asymmetric cryptographic operations such as RSA signatures, making it well-suited for high-throughput environments where flow rules are installed and modified at rates of thousands per second. In the SDN context, HMAC-based verification enables switches (or auxiliary agents acting on their behalf) to independently verify that received flow rules have not been tampered with in transit and that they originate from an authorized controller, without requiring modifications to the OpenFlow protocol or the switch hardware [7].

1.2 Problem Statement

SDN has become the go-to paradigm for modern network management, but its centralized design leaves critical vulnerabilities at every major junction: the controller (a single, high-value target whose compromise cascades across the entire fabric), the control channel (a conduit for sensitive configuration messages that can be intercepted or tampered with), and the switches themselves (which execute whatever flow rules they receive, no questions asked). An attacker who can exploit any one of these weak points can redirect traffic, exfiltrate data, or bring the network to a standstill.

Existing security solutions fail to comprehensively address these vulnerabilities for several interrelated reasons:

1. **Detection without verification:** Machine learning and deep learning-based intrusion detection systems can identify malicious traffic patterns but do not provide integrity guarantees for flow rules. An attacker who bypasses the IDS or who compromises the controller directly can still install malicious rules without detection.

2. **Verification without detection:** Cryptographic verification mechanisms such as HMAC-based flow rule authentication ensure rule integrity but are reactive in nature—they verify rules after issuance but do not proactively detect intrusion attempts or anomalous traffic patterns that precede or accompany rule tampering.

3. **Excessive computational overhead:** Heavyweight security frameworks based on blockchain, PKI, or complex protocol extensions introduce latency and resource consumption that are incompatible with the real-time operational requirements of SDN environments.

4. **Limited attack coverage:** Many existing solutions are designed to address a single attack vector (e.g., DDoS only or MITM only) and lack the generality to detect and mitigate the diverse range of attacks encountered in practice.

5. **Deployment complexity:** Solutions that require modifications to the OpenFlow protocol, switch firmware, or hardware are difficult to deploy in heterogeneous production environments and conflict with SDN's principle of hardware abstraction.

Taken together, these gaps frame the research question at the heart of this thesis: *How can we build a lightweight, deployable security framework for SDN that combines proactive multi-class intrusion detection with real-time cryptographic verification of flow rules and controller authenticity—all without exceeding the latency budget that production networks demand?*

The question is harder than it might first appear, because the objectives pull in different directions. High detection accuracy across diverse attack classes (DDoS, MITM, Probe, Bruteforce) typically means larger, slower models. Real-time cryptographic verification requires minimal per-message overhead. And the whole system must slot into the existing SDN stack—no protocol changes, no custom switch firmware—as a standard controller application.

1.3 Research Objectives

The overarching goal of this research is to design, implement, and evaluate a hybrid security framework for SDN that integrates deep learning-based intrusion detection with lightweight cryptographic verification. This goal is decomposed into the following specific research objectives:

1. **Design a TCN-based intrusion detection model** capable of accurately classifying network flows as benign or malicious by learning temporal patterns in flow-level features extracted from SDN traffic. The model should achieve detection accuracy exceeding 99% with high precision, recall, and F1-score, supporting real-time binary classification of diverse attack types including DDoS, MITM, Probe, and Brute-force.

2. **Develop a preprocessing and feature engineering pipeline** for the InSDN dataset [18] that includes data cleaning, feature selection using Pearson correlation analysis, StandardScaler normalization, label encoding, principal component analysis (PCA) for dimensionality reduction, and inverse-frequency class weighting to produce a high-quality training dataset that enables robust model generalization.

3. **Train and deploy the TCN model using TensorFlow/Keras** to enable efficient inference within the SDN controller environment, ensuring compatibility and portability across different deployment platforms while maintaining a compact model size suitable for resource-constrained environments.

4. **Design and implement a lightweight auxiliary security agent** that operates as a subprocess alongside the SDN controller and performs two complementary security functions: (a) HMAC-based verification of flow rule integrity, ensuring that flow rules received by switches have not been tampered with in transit, and (b) periodic challenge–response verification of controller authenticity, confirming that the entity issuing flow rules is indeed the legitimate controller.

5. **Integrate the TCN-based IDS and the HMAC-based auxiliary agent** into a cohesive framework that operates within the SDN control loop, with the IDS analyzing flow statistics for anomaly detection and the auxiliary agent providing cryptographic assurance of flow rule integrity and controller identity.

6. **Evaluate the proposed framework** in a simulated SDN environment using the Ryu controller framework, Mininet network emulator, and Open vSwitch, measuring detection performance metrics (accuracy, precision, recall, F1-score), computational overhead (inference latency, HMAC verification latency), and comparing the results against baseline models and state-of-the-art approaches.

7. **Conduct a comprehensive comparative analysis** of the proposed TCN-HMAC framework against alternative deep learning architectures (LSTM, GRU, CNN, Autoencoder, Transformer) and traditional machine learning models (Random Forest, XGBoost) to demonstrate the superiority of the temporal convolutional approach for SDN intrusion detection.

1.4 Significance of the Study

The work presented here contributes to network security, deep learning for cybersecurity, and software-defined networking in several concrete ways.

Bridging the Detection–Verification Divide. Most prior solutions sit squarely on one side of a divide: they either detect intrusions or verify message integrity, but not both. TCN-HMAC bridges that divide. The TCN handles external threats—DDoS floods, probes, brute-force attempts—while the HMAC agent handles internal ones—flow rule tampering, controller spoofing, replay attacks. Together they cover a far wider threat surface than either could alone.

Real-Time Viability. One of the practical takeaways of this work is that deep learning-based IDS *can* run inside the SDN control loop without crippling performance. At 612 KB, the TCN model is small enough to load on resource-constrained controllers, and its convolutional architecture lends itself to parallel execution, keeping inference well within production latency budgets. HMAC verification adds roughly 0.7 ms per operation—barely a blip. The upshot is a framework that data-center and telecom operators could realistically deploy without rearchitecting their control planes.

Empirical Case for TCNs in Network Security. The 99.85% accuracy, 99.80% precision, 99.97% recall, and 99.89% F1-score reported in this study add to a growing body of evidence that temporal convolutional architectures can match or beat recurrent models on sequential-data tasks [15]—and that the advantage is especially pronounced when low latency matters. For intrusion detection, where every millisecond of delay extends the window of vulnerability, the parallelizable nature of TCNs makes them an attractive choice.

Lightweight, Drop-In Design. Unlike approaches that ask operators to rewrite OpenFlow, reflash switch firmware, or install specialized hardware, TCN-HMAC runs entirely in software—as a controller application plus an auxiliary agent. This philosophy favours deployability over novelty: the framework slots into existing SDN stacks without infrastructure changes, and its low footprint (both TCN inference and HMAC computation) keeps it viable even where computational and memory headroom is tight.

A Benchmark for Future SDN IDS Research. The comparative study in this thesis pits the proposed model against fifteen alternatives drawn from diverse architectural families and multiple performance metrics. Because the evaluation uses the InSDN dataset [18]—a benchmark explicitly constructed to reflect real SDN traffic—the results have a degree of ecological validity that studies on generic IDS datasets cannot easily claim.

1.5 Research Contributions

The principal research contributions of this thesis are enumerated as follows:

1. **A novel hybrid security framework (TCN-HMAC)** that integrates Temporal Convolutional Network-based intrusion detection with HMAC-based flow rule integrity verification and challenge–response controller authentication, providing multi-layered defense for SDN environments. To the best of our knowledge, this is the first framework that combines temporal deep learning with lightweight cryptographic verification in a unified, deployable SDN security solution.

2. **A high-performance TCN model for binary intrusion detection** that achieves 99.85% accuracy, 99.80% precision, 99.97% recall, and 99.89% F1-score on the InSDN dataset, demonstrating the effectiveness of dilated causal convolutions for capturing temporal attack signatures in SDN network flows with a detection rate of 99.97% and a false alarm rate of only 0.37%.

3. **A TensorFlow/Keras-based deployment strategy** that produces a compact 612 KB model file (`best_tcn_insdn.keras`) enabling efficient, portable inference within the SDN controller environment. This contribution demonstrates a practical methodology for deploying deep learning models in production SDN environments without requiring heavyweight inference frameworks.

4. **A lightweight auxiliary security agent architecture** that performs HMAC-based flow rule verification and periodic challenge–response controller authentication with an overhead of only 0.7 ms per verification. The agent operates as an independent subprocess, requiring no modifications to the OpenFlow protocol or switch hardware, and provides cryptographic assurance that complements the anomaly detection capabilities of the TCN model.

5. **A comprehensive preprocessing pipeline** for the InSDN dataset that includes systematic data cleaning, Pearson correlation-based feature selection, StandardScaler normalization, label encoding, PCA-based dimensionality reduction (48 → 24 features, retaining 95.43% variance), and inverse-frequency class weighting, resulting in a high-quality training dataset that supports robust model generalization with minimal overfitting.

6. **An extensive comparative analysis** benchmarking the proposed TCN architecture against fifteen existing models—including CNN-BiLSTM, CNN-LSTM, CNN-GRU, DNN Ensemble, LSTM, Hybrid DL, DRL (DDQN), TCN, TCN-SE, TCN+Attention, BiTCN, BiTCN-MHSA, TCN-IDS, TCN Ensemble, and CNN/DT/RF baselines—across multiple

evaluation metrics, providing empirical evidence for the effectiveness of the proposed temporal convolutional approach in the SDN intrusion detection context.

1.6 Scope and Limitations

While the proposed TCN-HMAC framework represents a comprehensive approach to SDN security, the scope of this research is bounded by several design choices and practical constraints that should be acknowledged.

Dataset Scope. All training and evaluation are performed on the InSDN dataset [18], which was built specifically for SDN intrusion-detection research and covers normal traffic along with DDoS, MITM, Probe, and Brute-force attacks. InSDN is widely regarded as a solid benchmark, but no single dataset captures the full range of real-world network conditions, traffic mixes, and emerging attack variants. Testing the model on additional datasets is an obvious next step.

Attack Coverage. We target four major attack families—DDoS, MITM, Probe, and Brute-force—which collectively represent a large share of the threats facing SDN deployments today. That said, certain threat classes fall outside our scope: advanced persistent threats (APTs), zero-day exploits that leave no distinguishable traffic footprint, insider attacks that blend with legitimate patterns, and application-layer exploits that operate above the flow level.

Simulation Environment. The experimental evaluation is conducted in a simulated SDN environment using the Ryu controller framework, Mininet network emulator, and Open vSwitch software switches. While this setup provides a controlled and reproducible experimental platform, it may not fully capture the performance characteristics, scale, and failure modes of production SDN deployments with hardware switches, high-throughput traffic, and complex multi-controller topologies.

Static Model Deployment. The TCN model is trained offline on the InSDN dataset and deployed as a static inference model. The framework does not currently support online learning or incremental model updates to adapt to evolving attack patterns. Incorporating continual learning mechanisms to maintain detection accuracy over time as new attack variants emerge is identified as a direction for future work.

Single-Controller Architecture. The current design assumes a single SDN controller. Extension to multi-controller architectures, which are common in large-scale SDN deployments for scalability and fault tolerance, would require additional coordination mechanisms for both the IDS and the HMAC verification components.

Key Management. The HMAC-based verification relies on pre-shared symmetric keys between the controller and the auxiliary agent. The key distribution and rotation mechanisms are assumed to be handled by a separate key management infrastructure. The design and security analysis of such a key management system is outside the scope of this thesis.

1.7 Thesis Organization

The remainder of this thesis is organized into nine chapters, each addressing a specific aspect of the research.

Chapter 1 — Introduction (this chapter) has presented the background and motivation for the research, articulated the problem statement, defined the research objectives, discussed the significance and contributions of the study, and outlined the scope and limitations.

Chapter 2 — Background provides the foundational knowledge required to understand the proposed framework. It covers the SDN architecture in detail, including the OpenFlow protocol, controller frameworks, and flow rule lifecycle. It also presents the theoretical foundations of Temporal Convolutional Networks—including causal convolutions, dilated convolutions, and residual connections—and the mathematical basis of HMAC-based message authentication. This chapter establishes the technical vocabulary and conceptual framework used throughout the thesis.

Chapter 3 — Literature Review offers a comprehensive survey of related work organized into several thematic areas: cryptographic and protocol-based security mechanisms for SDN, blockchain-based approaches, machine learning and deep learning-based intrusion detection systems, authentication and access control frameworks, and hybrid approaches that combine detection with verification. The chapter identifies the research gaps that the proposed TCN-HMAC framework aims to address and positions the contributions of this thesis within the broader landscape of SDN security research.

Chapter 4 — Dataset and Preprocessing describes the InSDN dataset used for training and evaluating the TCN model. It provides a detailed analysis of the dataset composition, feature descriptions, class distributions, and attack scenarios. The chapter then presents the preprocessing pipeline, including data cleaning procedures, Pearson correlation-based feature selection, StandardScaler normalization, label encoding, PCA-based dimensionality reduction, and inverse-frequency class weighting. The rationale for each preprocessing decision is discussed in the context of its impact on model performance.

Chapter 5 — TCN Model Architecture presents the detailed architecture of the Temporal Convolutional Network used for intrusion detection. It describes the design of the dilated causal convolutional layers, the residual block structure, batch normalization and dropout regularization, the global average pooling mechanism, and the classification head. Hyperparameter selection—including the number of residual blocks, kernel sizes, dilation factors, dropout rates, and learning rate schedules—is discussed with justification for each design choice.

Chapter 6 — Proposed Methodology presents the overall TCN-HMAC framework architecture and describes how the TCN-based IDS and the HMAC-based auxiliary agent are integrated into the SDN control loop. It details the operational workflow of the framework, including the flow of data from network switches through the controller to the detection and verification modules. The HMAC-based flow rule verification protocol and the challenge–response controller authentication mechanism are formally described with pseudocode algorithms. The chapter also discusses the deployment considerations for real-time operation.

Chapter 7 — Experimental Setup describes the experimental environment, including the hardware and software configuration, the SDN testbed topology, traffic generation procedures, and the methodology for attack scenario emulation. It specifies the evaluation metrics—accuracy, precision, recall, F1-score, confusion matrix, and latency measurements—and the experimental protocols used to measure them. The training configuration, including the optimizer, learning rate schedule, batch size, and number of epochs, is documented.

Chapter 8 — Results and Analysis presents the experimental results obtained from the evaluation of the proposed framework. It reports the detection performance of the TCN model across all attack categories, analyzes the confusion matrix, discusses per-class precision and recall, and examines the training convergence behavior. The HMAC verification latency overhead is measured and analyzed. The results are interpreted in the context of the research objectives defined in Chapter 1.

Chapter 9 — Comparative Analysis provides a systematic comparison of the proposed TCN model against fifteen existing approaches spanning diverse architectures including CNN-BiLSTM, CNN-LSTM, CNN-GRU, DNN Ensemble, LSTM, Hybrid DL, DRL, multiple TCN variants, and traditional ML baselines. The comparison is conducted across all evaluation metrics and includes analysis of accuracy–latency tradeoffs, model complexity, and suitability for real-time SDN deployment. The chapter discusses the factors that contribute to the effectiveness of the proposed TCN-HMAC approach and identifies its advantages over existing solutions.

Chapter 10 — Conclusion summarizes the key findings of the research, discusses the implications of the results for SDN security practice, acknowledges the limitations of the study, and identifies promising directions for future research, including online learning, multi-controller support, adversarial robustness evaluation, and cross-dataset generalization.

Chapter 2

Background

Before diving into the design of the TCN-HMAC framework, it is worth laying out the conceptual groundwork. This chapter covers the three pillars on which the framework rests: the Software-Defined Networking architecture, the theory behind Temporal Convolutional Networks, and the mathematics of Hash-based Message Authentication Codes. Along the way, it also introduces the evaluation metrics and dimensionality-reduction techniques referenced throughout the rest of the thesis.

2.1 Software-Defined Networking Architecture

At its core, SDN is a paradigm shift: instead of scattering control logic across dozens of routers and switches—each running its own instance of OSPF or BGP—you pull that logic into a single piece of software sitting on a commodity server. The payoff is a unified view of the network and the ability to program its behaviour from one place. The cost, as we discuss later, is a new class of security risks. The subsections that follow walk through each architectural layer, the protocols that stitch them together, and the forwarding model that dictates how traffic actually moves.

2.1.1 Data Plane

The data plane, also referred to as the infrastructure layer, constitutes the lowest layer of the SDN architecture. It comprises the physical and virtual network devices—switches, routers, and access points—that are responsible for the actual forwarding, dropping, and modification of network packets. In a traditional network, these devices contain embedded control logic that independently computes forwarding decisions using distributed routing protocols. In SDN, however, data-plane devices are deliberately simplified: they function as programmable forwarding elements whose behavior is entirely dictated by flow rules received from the SDN controller [19].

Each forwarding device in the data plane maintains one or more *flow tables*, which are ordered collections of flow rules. A flow rule consists of three principal components: (1) *match fields*, which define the criteria for selecting packets based on header fields such as source and destination MAC addresses, IP addresses, port numbers, VLAN tags, and protocol types; (2) *actions*, which specify the operations to be performed on matched packets, such as forwarding to a specific port, dropping, modifying header fields, or encapsulating the packet for delivery to the controller; and (3) *counters*, which maintain statistics about the number of packets and bytes that have matched the rule and the duration for which the rule has been active [5].

When a packet arrives at an SDN-enabled switch, the switch performs a lookup in its flow tables, proceeding through the tables in pipeline order. If a matching rule is found, the corresponding action set is executed immediately. If no match is found in any table and a table-miss flow entry exists, the default action is performed—typically sending the packet to the controller via a `Packet-In` message. The controller then examines the packet, makes a forwarding decision, and installs appropriate flow rules in the switch to handle subsequent packets belonging to the same flow. This reactive flow installation mechanism is fundamental to understanding both the flexibility and the security vulnerabilities of SDN architectures.

Open vSwitch (OVS) is the most widely deployed software-based OpenFlow switch implementation [19]. OVS supports a comprehensive set of OpenFlow features including multiple flow tables, group tables for multipath forwarding, meter tables for rate limiting, and extensive match field support. OVS operates in both the kernel space (for high-performance packet forwarding through a fast-path datapath module) and the user space (for management, control communication, and slow-path processing). The versatility of OVS has made it the de facto standard for SDN research, development, and testing, and it is the switch implementation employed in the experimental evaluation of this thesis.

2.1.2 Control Plane

The control plane is the central intelligence of the SDN architecture, hosted on one or more controller platforms that run on commodity servers. The SDN controller maintains a comprehensive, real-time view of the entire network topology, including the connectivity between switches, the status of all links and ports, and the flow rules currently installed in each switch. This global visibility enables the controller to make informed, network-wide forwarding decisions that would be impossible in a distributed architecture where each device operates with only local knowledge [2].

The SDN controller provides a rich set of core network services, including:

- **Topology Discovery:** The controller actively discovers the network topology by sending Link Layer Discovery Protocol (LLDP) packets through the data-plane switches and analyzing the resulting responses. This process enables the controller to construct and maintain an accurate graph representation of the network, which serves as the basis for path computation and forwarding decisions.
- **Path Computation:** Using the topology graph, the controller computes optimal paths between source and destination nodes based on configurable objectives such as shortest path, minimum latency, maximum bandwidth, or load balancing. These computed paths are translated into sequences of flow rules that are installed in the switches along the path.
- **Flow Management:** The controller is responsible for the installation, modification, and deletion of flow rules in the data-plane switches. Flow management encompasses both proactive rule installation (where rules are pre-installed before traffic arrives) and reactive rule installation (where rules are created in response to `Packet-In` messages triggered by new flows).
- **Statistics Collection:** The controller periodically requests and aggregates flow statistics, port statistics, and table statistics from all switches in the network. These statistics provide critical operational insights including per-flow traffic volumes, link utilization levels, error rates, and performance metrics.
- **Event Handling:** The controller processes asynchronous events from the data plane, including `Packet-In` messages (triggered by unmatched packets), `Port-Status` messages (triggered by link up/down events), and `Flow-Removed` messages (triggered by rule expiration or deletion).

The Ryu SDN Framework [20] is an open-source, Python-based SDN controller platform that provides a well-defined API for developing network applications. Ryu is fully compliant with the OpenFlow protocol specifications and supports OpenFlow versions 1.0 through 1.5. Its modular architecture allows developers to implement custom network functions as independent Python applications that register event handlers for various OpenFlow messages. The Ryu framework is employed as the controller platform in the experimental evaluation of this thesis owing to its flexibility, extensibility, and comprehensive documentation.

2.1.3 Application Layer

The application layer sits atop the control plane and encompasses the diverse ecosystem of network applications that leverage the programmability and global visibility provided by the SDN controller. These applications communicate with the controller through northbound APIs—typically RESTful HTTP interfaces or direct Python API calls (in the case of the Ryu framework)—and implement higher-level network functions such as:

- **Traffic Engineering:** Applications that optimize traffic distribution across the network to balance load, minimize congestion, and maximize throughput.
- **Firewalls and Access Control:** Applications that enforce security policies by installing flow rules that permit or deny traffic based on predefined criteria.
- **Intrusion Detection Systems:** Applications that monitor network traffic patterns and flow statistics to detect anomalous behavior indicative of security threats—this is the category into which the TCN-based IDS component of this thesis falls.
- **Quality of Service (QoS) Management:** Applications that allocate network resources to ensure that critical applications receive guaranteed bandwidth, latency, and jitter levels [21].
- **Network Monitoring and Visualization:** Applications that provide real-time dashboards and alerting capabilities based on the flow statistics and topology information maintained by the controller.

The layered SDN architecture—comprising the data plane, control plane, and application plane—connected through well-defined southbound and northbound interfaces, creates a modular and extensible framework for network management. However, this same architecture introduces the security challenges detailed in Chapter 1, which motivate the design of the TCN-HMAC framework proposed in this thesis.

2.1.4 OpenFlow Protocol

The OpenFlow protocol [1,5] is the dominant southbound interface protocol that facilitates communication between the SDN controller and the data-plane switches. OpenFlow defines a standardized set of messages and procedures that enable the controller to install, modify, query, and delete flow rules on switches, as well as to receive asynchronous notifications about network events.

The OpenFlow message types can be categorized into three classes:

1. **Controller-to-Switch Messages:** These are initiated by the controller and include `Flow-Mod` (install/modify/delete flow rules), `Packet-Out` (send a packet out through a specific switch port), `Stats-Request` (query flow/port/table statistics), `Barrier-Request` (ensure processing order), and `Role-Request` (set the controller's role in multi-controller setups).

2. **Asynchronous Messages:** These are initiated by the switch without solicitation from the controller and include `Packet-In` (unmatched packet forwarded to controller), `Flow-Removed` (notification that a flow rule has been removed), `Port-Status` (notification of port state changes), and `Error` (notification of error conditions).

3. **Symmetric Messages:** These can be initiated by either the controller or the switch and include `Hello` (connection establishment), `Echo-Request/Echo-Reply` (liveness checking), and `Experimenter` (vendor-specific extensions).

The OpenFlow specification recommends the use of Transport Layer Security (TLS) to protect the control channel between the controller and switches. However, as noted in Section 1.1.2, TLS adoption remains inconsistent across deployments, and TLS alone does not provide end-to-end semantic integrity verification of flow rules. This gap motivates the HMAC-based verification mechanism employed in the TCN-HMAC framework.

The `Flow-Mod` message is of particular importance to this thesis because it is the primary mechanism through which flow rules are installed in switches. A `Flow-Mod` message contains a cookie field—a 64-bit opaque value that the controller can set arbitrarily and that is returned in flow statistics and `Flow-Removed` messages. The TCN-HMAC framework leverages this cookie field to embed HMAC-based integrity tags without modifying the OpenFlow protocol itself.

2.2 Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs) take a different approach to sequence modelling than the recurrent networks (LSTMs, GRUs) that dominated the field for years [15]. Rather than stepping through a sequence one element at a time, a TCN processes the whole sequence at once using one-dimensional convolutions. This seemingly simple swap has major practical consequences: training and inference are faster (because convolutions parallelise naturally on GPUs), gradients behave better during backpropagation, and the receptive field is easy to control. The subsections below unpack each of the architectural components that give TCNs these properties.

2.2.1 Causal Convolutions

The fundamental building block of a TCN is the causal convolution, which ensures that the output at any time step t depends only on inputs from the current and preceding time steps, never from future time steps. Formally, for a one-dimensional input sequence $\mathbf{x} = (x_0, x_1, \ldots, x_{T-1})$ and a filter $\mathbf{f} = (f_0, f_1, \ldots, f_{K-1})$ of size K, the causal convolution at time step t is defined as:

$$(\mathbf{x} *_c \mathbf{f})(t) = \sum_{k=0}^{K-1} f_k \cdot x_{t-k} \tag{2.1}$$

where $x_{t-k} = 0$ for $t - k < 0$ (zero-padding on the left side only). This formulation ensures that the output at time t is computed using only inputs from times $t, t-1, \ldots, t-K+1$, preserving the temporal ordering of the sequence. Causality is a critical property for real-time applications such as network intrusion detection, where predictions must be based solely on currently available and historical data, without access to future observations.

In practice, causal convolutions in TCNs are implemented using standard one-dimensional convolution operations with appropriate left-side zero-padding. Specifically, for a kernel of size K, $(K-1)$ zeros are prepended to the input sequence before applying a standard convolution, and the output is truncated to the same length as the original input. This padding scheme ensures that the output sequence has the same temporal length as the input sequence while maintaining strict causality.

2.2.2 Dilated Convolutions

A significant limitation of standard causal convolutions is that the receptive field—the number of input time steps that influence a given output—grows linearly with the number of convolutional layers. Specifically, a stack of L layers with kernel size K produces a receptive field of $1 + L(K-1)$ time steps. Achieving a large receptive field therefore requires either very deep networks (many layers) or very wide kernels (large K), both of which increase computational cost and parameter count [22].

Dilated convolutions resolve this limitation by introducing a dilation factor d that controls the spacing between the kernel elements. For a dilation factor d, the dilated causal convolution is defined as:

$$(\mathbf{x} *_d \mathbf{f})(t) = \sum_{k=0}^{K-1} f_k \cdot x_{t-d \cdot k} \tag{2.2}$$

where d is the dilation rate. When $d = 1$, the dilated convolution reduces to a standard convolution. As d increases, the kernel elements are spaced further apart, effectively allowing the convolution to "skip" over input elements and access a wider range of temporal context. The key insight is that by exponentially increasing the dilation factor across layers—typically using a geometric progression $d_l = 2^{l-1}$ for layer l—the receptive field grows exponentially with the number of layers rather than linearly. Specifically, for L layers with dilation factors $d_l = 2^{l-1}$ and kernel size K, the receptive field is:

$$R = 1 + (K-1)\sum_{l=0}^{L-1} 2^l = 1 + (K-1)(2^L - 1) \tag{2.3}$$

For the TCN architecture employed in this thesis with $K = 3$ and $L = 6$ (dilation factors $[1, 2, 4, 8, 16, 32]$), the receptive field is $1 + 2 \times (2^6 - 1) = 1 + 2 \times 63 = 127$ time steps. Since the input sequence length is 24 (corresponding to 24 PCA components), this receptive field of 127 time steps is more than sufficient to capture dependencies across the entire input sequence, ensuring that the deepest layer has global context access.

2.2.3 Residual Connections

Deep neural networks are susceptible to the degradation problem, wherein increasing network depth leads to higher training error due to the difficulty of learning identity mappings through multiple nonlinear transformations [23]. Residual connections, introduced by He et al. in the ResNet architecture, address this problem by providing shortcut connections that allow the gradient to flow directly from later layers back to earlier layers, bypassing the nonlinear transformations in between.

In a TCN residual block, the input $\mathbf{x}$ is processed through a sequence of dilated causal convolutions, batch normalization, activation functions, and dropout layers to produce a transformed output $\mathscr{F}(\mathbf{x})$. The block output is then computed as:

$$\text{output} = \text{ReLU}(\mathscr{F}(\mathbf{x}) + \mathbf{x}) \tag{2.4}$$

where $\mathscr{F}(\mathbf{x})$ represents the residual function learned by the convolutional layers, and $\mathbf{x}$ is the identity shortcut connection. When the dimensionality of $\mathbf{x}$ does not match that of $\mathscr{F}(\mathbf{x})$—for example, when the number of channels (filters) changes between layers—a 1×1 convolutional projection is applied to $\mathbf{x}$ to align the dimensions:

$$\text{output} = \text{ReLU}(\mathscr{F}(\mathbf{x}) + W_s\mathbf{x}) \tag{2.5}$$

where W_s is the 1×1 convolution weight matrix. This projection adds a small number of additional parameters but ensures that the residual connection can function regardless of dimensional mismatches.

Residual connections provide several important benefits for TCN training: (1) they enable the training of substantially deeper networks by mitigating the vanishing gradient problem; (2) they facilitate the learning of identity mappings, allowing the network to effectively “skip” layers that do not contribute useful transformations; and (3) they promote feature reuse across layers, improving the representational efficiency of the network.

2.2.4 Batch Normalization

Batch normalization [24] is a regularization technique that normalizes the activations of each layer to have zero mean and unit variance within each mini-batch during training. For a mini-batch $\mathscr{B} = \{x_1, x_2, \ldots, x_m\}$ of m activation values at a particular layer, batch normalization computes:

$$\hat{x}_i = \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma_{\mathscr{B}}^2 + \epsilon}} \tag{2.6}$$

where $\mu_{\mathscr{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$ is the mini-batch mean, $\sigma_{\mathscr{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathscr{B}})^2$ is the mini-batch variance, and ϵ is a small constant (typically 10^{-5}) added for numerical stability. The normalized values are then scaled and shifted using learned parameters γ and β:

$$y_i = \gamma \hat{x}_i + \beta \tag{2.7}$$

Batch normalization addresses the internal covariate shift problem—the phenomenon where the distribution of layer inputs changes during training as the parameters of preceding layers are updated. By stabilizing these distributions, batch normalization enables the use of higher learning rates, reduces sensitivity to weight initialization, and acts as a regularizer that can reduce the need for dropout in some architectures. In the TCN architecture employed in this thesis, batch normalization is applied after each convolutional layer and before the activation function, following the convention established in the original TCN literature [15].

2.2.5 Dropout and Spatial Dropout

Dropout [25] is a regularization technique that randomly sets a fraction p of the activations to zero during each training iteration. This prevents the network from developing co-dependent feature detectors and encourages the learning of more robust, distributed representations. During inference, all activations are retained but scaled by a factor of $(1-p)$ to compensate for the difference in expected activation magnitude (or, equivalently, the training activations can be scaled by $\frac{1}{1-p}$, which is the inverted dropout approach used in most modern frameworks).

For one-dimensional convolutional networks processing sequential data, standard element-wise dropout can disrupt the spatial structure of the feature maps. Spatial dropout addresses this issue by dropping entire feature map channels rather than individual elements. Specifically, for a feature tensor of shape (N, T, C) where N is the batch size, T is the temporal dimension, and C is the number of channels, spatial dropout creates a binary mask of shape $(N, 1, C)$ and broadcasts it across the temporal dimension. This ensures that when a channel is dropped, it is dropped for all time steps simultaneously, preserving the temporal coherence of the remaining channels.

The TCN architecture in this thesis employs spatial dropout with a rate of $p = 0.2$ after each pair of convolutional-batch normalization-activation operations within the residual blocks. Standard dropout with the same rate is applied in the dense classification head. These dropout rates were determined through preliminary hyperparameter search and provide an effective balance between regularization strength and model expressiveness.

2.2.6 Global Average Pooling

After the sequence of residual blocks, the TCN produces a three-dimensional feature tensor of shape (N, T, C), where N is the batch size, T is the sequence length, and C is the number of channels in the final convolutional layer. To convert this temporal feature representation into a fixed-length vector suitable for classification, global average pooling (GAP) is applied along the temporal dimension:

$$z_c = \frac{1}{T}\sum_{t=0}^{T-1} h_{t,c} \tag{2.8}$$

where $h_{t,c}$ is the activation at time step t for channel c, and z_c is the resulting pooled value for channel c. GAP reduces the feature tensor from shape (N, T, C) to (N, C), producing a compact representation that captures the average temporal behavior across the entire sequence for each feature channel.

GAP offers several advantages over alternative aggregation strategies such as flattening or temporal max pooling: (1) it is parameter-free, adding no additional trainable weights; (2) it provides inherent robustness to temporal shifts in the input, since averaging over time is less sensitive to the exact position of discriminative patterns; and (3) it produces a more compact representation than flattening, reducing the risk of overfitting in the subsequent dense layers.

2.2.7 Advantages of TCNs over Recurrent Architectures

TCNs offer several compelling advantages over recurrent neural network architectures such as Long Short-Term Memory (LSTM) [26] and Gated Recurrent Units (GRU) [27] for the network intrusion detection task:

1. **Parallelism:** TCN computations are fully parallelizable across time steps because each convolutional operation processes all time steps simultaneously. In contrast, RNNs must process the sequence one time step at a time, with each step depending on the hidden state computed at the previous step. This sequential dependency prevents effective parallelization and results in significantly longer training and inference times, particularly for long sequences.

2. **Stable Gradients:** TCNs exhibit stable gradient behavior during backpropagation because the gradient path from output to input passes through only a fixed number of convolutional layers, regardless of the sequence length. RNNs, by contrast, propagate gradients through a number of computational steps equal to the sequence length, making them susceptible to vanishing and exploding gradient problems that can destabilize training and degrade model performance.

3. **Controllable Receptive Field:** The receptive field of a TCN is precisely deterministic and controllable through the choice of kernel size, dilation factors, and number of layers. This allows the network designer to explicitly specify the temporal context available to the model, ensuring that important long-range dependencies are captured without unnecessary computational overhead. In RNNs, the effective memory span is learned implicitly and is often shorter than the sequence length due to gradient decay.

4. **Memory Efficiency:** TCNs share convolutional filters across all time steps, resulting in a parameter count that is independent of the sequence length. RNNs, while also sharing parameters across time steps through their recurrent connections, require maintaining hidden states for the duration of the sequence during both training (for backpropagation through time) and inference, consuming additional memory proportional to the sequence length.

5. **Empirical Performance:** Bai et al. [15] conducted a comprehensive empirical evaluation demonstrating that TCNs achieve competitive or superior performance compared

to canonical recurrent architectures (LSTM, GRU) across a diverse range of sequence modeling benchmarks. Subsequent studies specifically in the network intrusion detection domain [16, 28, 29] have confirmed these findings, showing that TCNs offer favorable accuracy–latency tradeoffs for real-time security applications.

These advantages collectively make TCNs an attractive choice for the intrusion detection component of the TCN-HMAC framework, where real-time inference speed, training stability, and high detection accuracy are primary requirements.

2.3 Hash-Based Message Authentication Codes

An HMAC is, at its heart, a way to slap a tamper-evident seal on a message using a shared secret [17]. Unlike digital signatures, which rely on public-key cryptography and are comparatively expensive, HMACs use symmetric keys and run fast—fast enough to authenticate thousands of messages per second without anyone noticing the overhead. That speed is precisely why HMACs suit SDN flow-rule verification so well.

2.3.1 Mathematical Definition

Given a cryptographic hash function H (such as SHA-256 [30]), a secret key K, and a message M, the HMAC is computed as:

$$\text{HMAC}(K, M) = H\big((K' \oplus opad) \,\|\, H((K' \oplus ipad) \,\|\, M)\big) \tag{2.9}$$

where:

- K' is the key K padded to the block size B of the hash function (if $|K| > B$, then $K' = H(K)$; if $|K| < B$, then K is right-padded with zeros)
- $\oplus$ denotes the bitwise exclusive-OR operation
- $\|$ denotes concatenation
- $ipad$ is the inner padding constant: the byte `0x36` repeated B times
- $opad$ is the outer padding constant: the byte `0x5C` repeated B times

The double-hashing construction—where the message is first hashed with the key XORed with the inner pad, and the result is then hashed again with the key XORed with the outer pad—is essential for the security of the HMAC. This construction prevents length-extension attacks and other vulnerabilities that would affect a naïve $H(K\|M)$ construction.

2.3.2 Security Properties

HMACs provide two fundamental security guarantees:

1. **Message Integrity:** Any modification to the message M—even a single bit change—will produce a completely different HMAC value with overwhelming probability. This property enables the recipient to detect any tampering with the message during transmission.
2. **Message Authentication:** Only a party possessing the secret key K can compute a valid HMAC for a given message. This property ensures that a message with a valid HMAC was originated by a party that knows the key, thereby authenticating the source of the message.

The security of HMAC relies on the underlying hash function being collision-resistant and preimage-resistant. When instantiated with SHA-256, HMAC provides a 256-bit authentication tag, offering a security level of 2^{128} against brute-force forgery attacks (due to the birthday bound on the internal state). This security level is considered more than sufficient for all practical applications, including the flow rule verification use case in this thesis [31].

2.3.3 Computational Efficiency

A critical advantage of HMAC for SDN flow rule verification is its computational efficiency. HMAC computation requires only two invocations of the underlying hash function, regardless of the message length (for messages shorter than the hash block size). On modern processors with hardware acceleration for SHA-256 (available in Intel SHA Extensions and ARM Cryptography Extensions), a single HMAC-SHA256 computation can complete in less than 1 μs for typical flow rule messages. This efficiency is in stark contrast to asymmetric cryptographic operations such as RSA-2048 signature generation, which typically requires 0.5–2 *ms* per operation, or ECDSA-256 signature generation, which requires 0.1–0.5 *ms* per operation [32].

In the SDN context, where flow rules may be installed, modified, and deleted at rates of hundreds or thousands per second, the sub-microsecond HMAC computation time ensures that integrity verification does not introduce perceptible latency overhead. This property is fundamental to the practicality of the TCN-HMAC framework, as it enables real-time verification without degrading network performance.

2.3.4 HMAC in the Context of SDN Flow Rule Verification

The application of HMAC to SDN flow rule verification involves the following conceptual process: when the controller prepares a flow rule for installation, it computes an HMAC over the essential fields of the flow rule (match fields and actions) using a symmetric key shared with the verification agent. The resulting HMAC tag is embedded in the cookie field of the OpenFlow `Flow-Mod` message. The verification agent, which has access to the same symmetric key, can independently recompute the HMAC from the flow rule fields and compare it with the embedded cookie. A mismatch indicates that the flow rule has been tampered with in transit or was not originated by the legitimate controller [7].

This verification scheme has several desirable properties: (1) it requires no modifications to the OpenFlow protocol, as it uses the existing cookie field; (2) it does not require modifications to the switch hardware or firmware; (3) it is computationally lightweight, with negligible impact on controller and switch performance; and (4) it can be deployed incrementally, as it operates independently of other security mechanisms that may or may not be present in the deployment.

2.4 Deep Learning for Network Intrusion Detection

Network Intrusion Detection Systems (NIDS) monitor live traffic for signs of malicious activity, policy violations, or unusual behaviour [33]. The classical approach is signature-based: compare what you see against a catalogue of known-bad patterns. This works passably well for known attacks but is blind to anything the catalogue does not already contain. That blind spot—zero-day attacks, novel exploit chains, traffic patterns that have never been seen before—has pushed the community toward anomaly-based methods that learn what "normal" looks like and raise an alarm when something deviates.

2.4.1 Machine Learning Approaches

Classical machine learning algorithms have been extensively applied to network intrusion detection. Random Forests [34] construct ensembles of decision trees trained on random subsets of features and samples, achieving robust classification through majority voting. XGBoost [35] extends gradient-boosted decision trees with regularization and second-order optimization, offering strong performance on tabular data. Support Vector Machines (SVMs) find optimal hyperplanes for class separation in high-dimensional feature spaces. These methods have demonstrated moderate success on benchmark intrusion detection datasets but share several limitations: they require extensive manual feature engineering, struggle with high-dimensional and correlated features, and have limited ability to capture complex nonlinear patterns and temporal dependencies in network traffic [36].

2.4.2 Deep Learning Approaches

Deep learning architectures have emerged as powerful alternatives that can automatically learn hierarchical feature representations from raw or minimally preprocessed data, eliminating the need for manual feature engineering [37].

Convolutional Neural Networks (CNNs): One-dimensional CNNs can extract local spatial patterns from feature vectors by applying sliding convolutional filters. While effective for capturing local correlations between adjacent features, standard CNNs lack inherent temporal awareness and treat the input as a fixed-length feature vector rather than a temporal sequence. This limits their ability to model the sequential dependencies present in network flow data.

Recurrent Neural Networks (RNNs): LSTM networks [26] and GRUs [27] are designed to model sequential data by maintaining hidden states that capture temporal dependencies. LSTMs introduce gating mechanisms (input, forget, and output gates) that control the flow of information through the network, enabling the selective retention and updating of long-term memories. While highly effective for sequence modeling, LSTMs and GRUs suffer from sequential processing constraints that limit parallelism and increase inference latency.

Autoencoders: Autoencoder-based approaches [38] learn compressed representations of normal traffic patterns and detect anomalies based on reconstruction error. Inputs that cannot be accurately reconstructed by the autoencoder (i.e., those with high reconstruction error) are flagged as anomalous. While effective for unsupervised anomaly detection, autoencoders may struggle with complex attack patterns that partially overlap with normal traffic in the learned feature space.

Transformers: Transformer-based architectures [39, 40] use self-attention mechanisms to model dependencies between all positions in a sequence simultaneously, enabling the capture of long-range interactions without the sequential processing constraint of RNNs. However, the quadratic computational complexity of self-attention with respect to the sequence length makes Transformers computationally expensive for long sequences, and the large number of parameters in typical Transformer configurations can lead to overfitting on small to medium-sized intrusion detection datasets.

Temporal Convolutional Networks: TCNs [15] combine the parallel processing efficiency of CNNs with the temporal modeling capability of dilated causal convolutions, achieving a favorable balance between detection accuracy, inference speed, and model complexity.

The application of TCNs to network intrusion detection has been explored in several recent studies [16, 28, 29, 41, 42], with results demonstrating competitive or superior performance compared to recurrent alternatives.

2.4.3 Feature Engineering for Intrusion Detection

Effective feature engineering is critical for the performance of both machine learning and deep learning-based intrusion detection systems. Network flow features can be broadly categorized as:

- **Packet Header Features:** Information extracted from packet headers, including source and destination IP addresses, port numbers, protocol types, and flag fields. These features capture the communication patterns and endpoint characteristics of network flows.
- **Flow Duration Features:** Temporal characteristics of flows, including total duration, inter-packet arrival times, and the timing of the first and last packets. These features are particularly useful for detecting attacks with distinctive temporal signatures, such as DDoS floods (short inter-arrival times) or slow probes (long inter-arrival times).
- **Volume Features:** Quantitative measures of traffic volume, including total bytes transferred, total packets, forward and backward packet counts, and payload sizes. These features help distinguish between bandwidth-intensive attacks and normal traffic.
- **Statistical Features:** Higher-order statistics computed over packet-level measurements, including mean, standard deviation, minimum, maximum, and inter-quartile range of packet sizes and inter-arrival times. These features capture the distributional characteristics of traffic that can reveal anomalous behavior.

Feature selection and dimensionality reduction techniques are commonly employed to remove redundant and irrelevant features, reducing computational cost and improving model generalization. Pearson correlation-based feature selection identifies and removes highly correlated feature pairs, while Principal Component Analysis (PCA) projects the feature space onto a lower-dimensional subspace that captures the maximum variance [43]. Both techniques are employed in the preprocessing pipeline of this thesis, as detailed in Chapter 4.

2.5 Evaluation Metrics for Intrusion Detection

The performance of intrusion detection systems is evaluated using a set of standard classification metrics derived from the confusion matrix. For a binary classification task with positive (attack) and negative (benign) classes, the confusion matrix contains four quantities: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) [44].

Accuracy measures the proportion of correctly classified samples out of the total number of samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.10}$$

Precision measures the proportion of predicted positives that are truly positive, reflecting the reliability of positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.11}$$

Recall (also called Sensitivity or Detection Rate) measures the proportion of actual positives that are correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.12}$$

F1-Score is the harmonic mean of precision and recall, providing a single metric that balances both concerns:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.13}$$

Specificity measures the proportion of actual negatives that are correctly identified:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{2.14}$$

False Alarm Rate (FAR), also called False Positive Rate (FPR), measures the proportion of actual negatives that are incorrectly classified as positive:

$$\text{FAR} = \frac{FP}{FP + TN} = 1 - \text{Specificity} \tag{2.15}$$

Receiver Operating Characteristic (ROC) Curve and Area Under the Curve (AUC): The ROC curve [45] plots the True Positive Rate (Recall) against the False Positive Rate at various classification thresholds. The AUC summarizes the ROC curve as a single scalar value between 0 and 1, where 1 indicates perfect classification and 0.5 indicates random chance. AUC is particularly useful for evaluating classifiers on imbalanced datasets, as it is invariant to the class distribution.

In the context of intrusion detection, recall (detection rate) is typically considered the most critical metric because a missed attack (false negative) can have severe security consequences. However, a high false alarm rate (low precision) can lead to alert fatigue and desensitization of security operators. The ideal intrusion detection system achieves high recall and precision simultaneously, which is reflected by a high F1-score.

2.6 Principal Component Analysis

Principal Component Analysis (PCA) is a linear dimensionality reduction technique that projects high-dimensional data onto a lower-dimensional subspace defined by the directions of maximum variance. Given a dataset $\mathbf{X} \in \mathbb{R}^{n \times p}$ with n samples and p features, PCA computes the eigendecomposition of the covariance matrix $\mathbf{C} = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}$ (assuming centered data) to obtain a set of orthogonal eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p$ and corresponding eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p$. The eigenvectors, called principal components, define the directions of maximum variance, and the eigenvalues quantify the variance captured along each direction.

Dimensionality reduction is achieved by projecting the data onto the first $k < p$ principal components:

$$\mathbf{Z} = \mathbf{X}\mathbf{V}_k \tag{2.16}$$

where $\mathbf{V}_k = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k]$ is the matrix of the first k eigenvectors. The proportion of total variance retained by the k-component projection is:

$$\text{Variance Retained} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i} \tag{2.17}$$

In this thesis, PCA is applied to reduce the 48-dimensional feature space of the preprocessed InSDN dataset to 24 principal components that retain 95.43% of the total variance. This reduction halves the input dimensionality while preserving the vast majority of the discriminative information, resulting in faster model training and inference, reduced risk of overfitting, and improved model generalization. The 24 PCA components are subsequently treated as a temporal sequence of length 24 with a single channel, which serves as the input to the TCN model.

2.7 Class Imbalance Handling

Class imbalance is a pervasive challenge in intrusion detection datasets, where attack samples typically constitute a small fraction of the total traffic. In the InSDN dataset used in this thesis, the class distribution after preprocessing and deduplication is approximately 35% benign and 65% attack, representing a moderate level of imbalance. Class imbalance can bias the model toward the majority class, leading to high overall accuracy but poor detection of the minority class [46].

Several strategies have been proposed to address class imbalance:

Oversampling: The Synthetic Minority Over-sampling Technique (SMOTE) [47] generates synthetic samples for the minority class by interpolating between existing minority samples and their nearest neighbors. While effective, SMOTE can introduce artificial patterns that may not reflect real-world traffic distributions.

Undersampling: Random undersampling reduces the number of majority class samples to match the minority class. While simple, this approach discards potentially valuable data.

Class Weighting: Instead of modifying the dataset, class weighting adjusts the loss function to assign higher weights to minority class samples during training. This approach preserves the original data distribution while ensuring that the model penalizes misclassifications of the minority class more heavily. For a binary classification task with class counts n_0 (benign) and n_1 (attack), the class weights are typically computed as:

$$w_c = \frac{N}{C \cdot n_c} \tag{2.18}$$

where N is the total number of samples, C is the number of classes, and n_c is the number of samples in class c.

In this thesis, class weighting is employed as the primary strategy for handling class imbalance, with computed weights of 1.4258 for the benign class and 0.7700 for the attack class. This approach was chosen over SMOTE because it preserves the authentic distribution of the training data, does not introduce synthetic samples that might reduce model generalizability, and integrates seamlessly with the binary cross-entropy loss function used for training.

2.8 Chapter Summary

This chapter has surveyed the technical building blocks underpinning the TCN-HMAC framework. We began with the layered SDN architecture—data plane, control plane, application plane, and the OpenFlow protocol that connects them—and then turned to the TCN, walking through causal and dilated convolutions, residual connections, batch normalisation, dropout, and global average pooling. The HMAC was examined next: its mathematical construction, its security properties, and its suitability for high-throughput flow-rule verification. Finally, we reviewed deep learning-based IDS approaches, the standard evaluation metrics, PCA, and class-imbalance handling strategies. With this groundwork in place, the chapters that follow can focus on how these pieces fit together in the proposed framework.

Chapter 3

Literature Review

A good literature review should do two things: map the state of the art and expose its blind spots. This chapter attempts both. We survey related work across seven thematic areas—cryptographic SDN defences, blockchain-based approaches, machine-learning and deep-learning IDS, authentication and access control, TCN-based detection, hybrid security frameworks, and dataset studies—and, along the way, tease out the research gaps that the TCN-HMAC framework is designed to fill.

3.1 Cryptographic and Protocol-Based SDN Security

Protecting the SDN control plane with cryptography is an obvious first line of defence, and researchers have explored a range of lightweight schemes designed to guarantee integrity, authenticity, and confidentiality without imposing heavy changes on the underlying infrastructure.

Pradeep et al. [8] proposed EnsureS, a lightweight flow rule verification scheme that utilizes batch hashing and tag-based checks to validate flow rules installed on OpenFlow switches. The EnsureS framework operates by computing hash-based tags over batches of flow rules and verifying them at the switch level. Their evaluation demonstrated low computational overhead, with verification latency in the sub-millisecond range. However, EnsureS focuses exclusively on flow rule integrity verification without incorporating any anomaly detection mechanism, leaving the network vulnerable to attacks that do not involve direct flow rule tampering. Furthermore, the batch verification approach introduces a trade-off between verification granularity and performance, as larger batch sizes reduce overhead but increase the window of vulnerability during which tampered rules may be active.

Zhou et al. [48] introduced SecureMatch, a cryptographic rule-matching mechanism designed to ensure secure flow rule installation with minimal latency impact. SecureMatch employs a combination of symmetric encryption and hash-based verification to protect the confidentiality and integrity of flow rules during transmission from the controller to the switches. Their approach demonstrated moderate computational overhead and was evaluated in a simulated SDN environment. While SecureMatch provides stronger security guarantees than plaintext flow rule transmission, it requires modifications to the rule-matching process on the switches, which limits its deployability in heterogeneous environments with legacy switch hardware.

Ahmed et al. [7] developed a modular HMAC-based framework specifically designed for verifying flow modification (`flow_mod`) messages in SDN environments. Their framework computes HMAC tags over the essential fields of flow rules using pre-shared symmetric keys and verifies them independently at the switch side. A key advantage of their approach is that it operates without requiring modifications to the OpenFlow protocol or the switch hardware, making it immediately deployable in existing SDN infrastructure. However, like EnsureS, the HMAC-SDN framework focuses solely on integrity verification and does not incorporate proactive threat detection capabilities. This limitation means that attacks which do not involve flow rule tampering—such as DDoS attacks, reconnaissance probes, or brute-force credential attacks—remain undetected by the framework.

Buruaga et al. [6] addressed the emerging threat of quantum computing to SDN security by proposing a quantum-safe integration of TLS for SDN networks. Their work demonstrated the feasibility of deploying post-quantum cryptographic algorithms within the TLS handshake for SDN control channel protection. While addressing a forward-looking security concern, their approach imposes significant computational overhead due to the larger key sizes and longer processing times required by post-quantum algorithms, making it less suitable for resource-constrained or latency-sensitive SDN deployments.

Reddy et al. [49, 50] investigated the security of flow tables in P4-based SDN data planes and proposed mitigation strategies against flow table modification attacks. Their work highlighted the vulnerability of flow tables to both insider and outsider attacks and demonstrated that P4-programmable switches can be leveraged to implement custom security checks. However, their solutions are specific to P4-enabled switches and are not directly applicable to the more widely deployed OpenFlow-based SDN environments.

Han et al. [51] proposed an efficient flow rule conflict detection scheme for SDN networks that identifies and resolves conflicting flow rules that could be exploited by attackers to create policy violations. Their comprehensive detection approach addresses an important but narrow aspect of SDN security—flow rule conflicts—without addressing the broader spectrum of network intrusion threats.

What these cryptographic and protocol-level schemes share is a reactive stance: they can verify rules after they have been issued, but they do nothing to detect the intrusions or suspicious traffic patterns that typically precede—or accompany—a control-plane attack. Filling that gap calls for proactive anomaly detection, which is exactly what we integrate into the TCN-HMAC framework.

3.2 Blockchain-Based Approaches for SDN Security

Blockchain has captured the imagination of SDN security researchers thanks to its promise of decentralised trust, tamper-evident logs, and distributed consensus. In practice, however, the technology's overhead makes most of these proposals impractical for real-time SDN operations.

Song et al. [52] leveraged blockchain for securing intent-driven SDN, proposing a framework called IS2N that uses blockchain to record network intents and verify their faithful implementation in the data plane. Their approach provides strong guarantees that network policies are implemented as specified, with a tamper-evident audit trail. However, the consensus overhead of their blockchain protocol introduces significant latency (several seconds per transaction), making it unsuitable for real-time security operations that require sub-second response times.

Rahman et al. [53] conducted a comprehensive survey of blockchain's potential to safeguard controller communications and flow rule integrity in SDN. Their analysis identified several promising application scenarios, including decentralized controller authentication, flow rule provenance tracking, and policy compliance auditing. However, the survey also highlighted significant challenges, including the scalability limitations of blockchain consensus protocols, the storage overhead of maintaining a growing chain of network events, and the difficulty of integrating blockchain with the fast-paced, real-time nature of SDN operations.

Poorazad et al. [54] combined blockchain with deep learning for real-time threat detection in Industrial IoT environments connected via SDN. Their hybrid approach uses a deep learning model for anomaly detection and records detection events on a blockchain for auditability and cross-domain sharing. While innovative, the combination of blockchain consensus and deep learning inference introduces significant computational overhead, and the system was evaluated only on a small-scale testbed with limited traffic diversity.

Tselios et al. [55] applied the MuZero reinforcement learning algorithm for optimal controller placement in multi-controller SDN architectures and used blockchain-based audit trails to ensure the integrity of placement decisions. Their approach addresses the joint optimization of performance and security in distributed SDN deployments, but the complexity of the MuZero training process and the blockchain overhead limit practical applicability.

Alkhamisi et al. [56] developed a blockchain-enabled control plane framework specifically designed to detect cross-controller attacks in multi-controller SDN deployments. Their framework records all controller-to-controller communications on a permissioned blockchain and uses smart contracts to enforce security policies. While effective for multi-controller scenarios, the framework is not applicable to single-controller deployments and introduces non-trivial latency for inter-controller consensus.

For all their conceptual appeal, blockchain-based SDN defences consistently bump against the same wall: the consensus step takes too long and consumes too many resources. In an environment where flow rules must be verified in microseconds, waiting even a few seconds per transaction is a non-starter. TCN-HMAC sidesteps this problem entirely by relying on HMAC verification, which delivers comparable integrity guarantees with sub-millisecond overhead.

3.3 Machine Learning-Based Intrusion Detection in SDN

Machine learning techniques have been widely adopted for intrusion detection in SDN environments, leveraging the centralized visibility and programmability of the SDN controller to collect comprehensive network statistics for classification.

Sharma and Tyagi [57] proposed a lightweight intrusion detection system for SDN that focuses on identifying Man-in-the-Middle (MITM) and Denial of Service (DoS) attacks using adaptive anomaly detection. Their system monitors flow statistics collected by the SDN controller and applies ensemble machine learning classifiers to distinguish benign from malicious traffic. The lightweight design enables deployment on resource-constrained controllers, and the system achieved detection accuracy of approximately 98.12%. However, the system does not address flow rule integrity verification, leaving the control plane vulnerable to rule tampering attacks that occur after the detection stage.

Ayad et al. [58] conducted a comprehensive evaluation of machine learning techniques for intrusion detection in SDN, comparing the performance of several classifiers including Random Forest, XGBoost, Decision Tree, and K-Nearest Neighbors on multiple SDN intrusion detection datasets. Their study found that ensemble methods, particularly XGBoost and Random Forest, consistently outperformed individual classifiers. However, all evaluated models exhibited limitations in capturing temporal dependencies in traffic patterns, and the study did not consider deep learning approaches or integrity verification mechanisms.

Basfar et al. [59] proposed an enhanced feature selection approach called EMRMR (Enhanced Maximum Relevance Minimum Redundancy) for improving intrusion detection in SDN. Their method selects the most informative features while minimizing redundancy, resulting in improved classification performance with reduced feature dimensionality. While the feature selection methodology is compelling, the study focuses solely on the preprocessing stage and does not address the deployment and integration challenges of IDS in real-time SDN environments.

Singh and Kumar [60] evaluated SDN flow table manipulation attacks using machine learning techniques, comparing multiple classifiers on a dataset of legitimate and manipulated flow table entries. Their work provides valuable insights into the characteristics of flow table attacks but is limited to detection of flow table-specific attacks and does not address the broader spectrum of network intrusion threats.

Sundan et al. [61,62] proposed proactive and multi-layered security frameworks for intrusion detection in SDN environments using machine learning. Their approaches combine multiple detection layers, each targeting different attack categories, to provide comprehensive coverage. While the multi-layered concept is aligned with the defense-in-depth philosophy adopted in this thesis, their frameworks rely exclusively on traditional machine learning classifiers and do not leverage the temporal modeling capabilities of deep learning architectures.

3.4 Deep Learning-Based Intrusion Detection in SDN

Deep learning approaches have demonstrated significant improvements over traditional machine learning methods for intrusion detection, owing to their ability to automatically learn hierarchical feature representations from raw or minimally preprocessed data.

3.4.1 CNN-Based Approaches

Convolutional Neural Networks (CNNs) have been applied to network intrusion detection by treating network flow features as one-dimensional spatial inputs. Said et al. [13] proposed CNN-BiLSTM, a hybrid deep learning architecture that combines CNN feature extraction with Bidirectional LSTM temporal modeling for intrusion detection in SDN. Their architecture first applies convolutional layers to extract local feature patterns from network flow data, then uses a Bidirectional LSTM to capture temporal dependencies in both forward and backward directions. The system achieved high detection performance on the InSDN dataset, with reported accuracy of approximately 98.7% and F1-score of 98.5%. However, the bidirectional processing requirement means that the model needs access to future time steps for classification, limiting its applicability to real-time scenarios where only past and current data are available. Furthermore, the LSTM component introduces sequential processing constraints that increase inference latency.

Shihab et al. [63] proposed an optimized hybrid CNN-LSTM framework with multi-feature analysis and SMOTE for intrusion detection in SDN. Their approach combines CNN-based spatial feature extraction with LSTM-based temporal modeling and addresses class imbalance through SMOTE oversampling. The multi-feature analysis step identifies the most discriminative features before feeding them to the deep learning model. While achieving good detection performance, the hybrid CNN-LSTM architecture is computationally more expensive than pure convolutional approaches, and the SMOTE augmentation introduces synthetic data points that may not accurately represent real attack patterns.

Yang [64] proposed a new attack intrusion detection model based on deep learning for SDN environments, combining CNN with Gated Recurrent Units (GRU). The model uses CNN layers for initial feature extraction followed by GRU layers for temporal sequence modeling. The evaluation on multiple datasets demonstrated competitive accuracy, but the sequential nature of the GRU component remains a bottleneck for real-time deployment.

3.4.2 LSTM and RNN-Based Approaches

Recurrent architectures have been extensively explored for SDN intrusion detection due to their inherent ability to model temporal dependencies.

Basfar et al. [65] proposed an incremental LSTM ensemble for online intrusion detection in SDN. Their approach uses multiple LSTM models trained on different data segments and combines their predictions through an ensemble mechanism. The incremental training strategy enables the model to adapt to evolving attack patterns without full retraining. However, the ensemble of LSTM models introduces significant computational overhead, and the sequential processing nature of LSTMs limits real-time inference speed.

Ataa et al. [12] conducted a comprehensive evaluation of deep learning approaches for intrusion detection in SDN, comparing CNN, LSTM, BiLSTM, and hybrid architectures on the InSDN dataset. Their study found that hybrid architectures combining spatial and temporal feature extraction achieved the best performance, with the best model achieving approximately 97.5% accuracy. However, the study focused solely on detection performance without addressing deployment considerations such as inference latency, model size, or integration with SDN security mechanisms.

Kumar et al. [66] proposed a metaparameter-optimized hybrid deep learning model for cybersecurity in SDN environments. Their approach uses meta-heuristic optimization algorithms to automatically tune the hyperparameters of a hybrid deep learning architecture, achieving improved detection performance compared to manually tuned models. While the automated hyperparameter optimization is a notable contribution, the resulting model is significantly more complex and computationally expensive than the lightweight TCN architecture proposed in this thesis.

3.4.3 Transformer-Based Approaches

Transformer architectures have recently been applied to network intrusion detection, leveraging self-attention mechanisms for capturing long-range dependencies.

Wu et al. [40] proposed RT-IDS, a real-time intrusion detection system using the Transformer architecture. Their model applies multi-head self-attention to network flow features, enabling it to capture complex interactions between different feature dimensions. While achieving strong detection performance, the quadratic computational complexity of self-attention with respect to the sequence length introduces significant overhead for large-scale deployment. The Transformer's large parameter count also increases the risk of overfitting on smaller datasets.

3.4.4 Reinforcement Learning-Based Approaches

Kanimozhi and Ramesh [14] proposed a deep reinforcement learning-based intrusion detection scheme for SDN where the IDS agent learns optimal detection policies through interaction with the network environment. Their approach uses Deep Q-Networks (DQN) to learn adaptive detection strategies that can evolve with changing attack patterns. While promising for adaptive and self-learning IDS frameworks, deep reinforcement learning introduces significant training complexity, requires careful reward engineering, and may exhibit instability during training. Additionally, the real-time inference efficiency of RL-based approaches depends heavily on the complexity of the state representation and action space.

3.4.5 Federated and Distributed Approaches

Wang and Huang [67] proposed an adaptive intrusion detection framework using federated deep learning for SDN. Their approach enables multiple SDN controllers to collaboratively train a shared deep learning model without exchanging raw traffic data, preserving privacy and reducing communication overhead. Sousa and Gonçalves [68] proposed FedAAA-SDN, a federated authentication and authorization model for secure cross-domain SDN environments. These federated approaches address important concerns regarding data privacy and multi-domain collaboration but introduce additional complexity in terms of model aggregation, communication rounds, and convergence guarantees.

Feizi and AL-Talebei [69] proposed using Deep Convolutional Generative Adversarial Networks (DCGAN) for data balancing to improve the accuracy of a hybrid CNN-LSTM intrusion detection framework in SDN environments. Their approach generates synthetic attack samples to balance the class distribution, achieving improved detection rates for minority attack classes. However, GAN-generated samples may introduce artifacts that are not representative of real attack patterns.

3.5 TCN-Based Intrusion Detection

The application of Temporal Convolutional Networks to network intrusion detection is a relatively recent development that has shown promising results. This section reviews the existing literature on TCN-based IDS approaches, which are most directly related to the detection component of the proposed TCN-HMAC framework.

Lopes et al. [16] proposed a network intrusion detection model based on the Temporal Convolutional architecture. Their model employs dilated causal convolutions with residual connections to capture temporal patterns in network flow features. The evaluation on multiple benchmark datasets demonstrated that the TCN model achieved competitive or superior accuracy compared to LSTM and GRU baselines while requiring significantly less inference time. Their work provided important early evidence for the viability of TCNs in the intrusion detection domain. However, their evaluation was conducted on general-purpose network datasets (not SDN-specific), and the study did not address the integration of the TCN model with SDN-specific security mechanisms such as flow rule verification.

Benfarhat et al. [41] proposed an advanced TCN framework for intrusion detection in electric vehicle (EV) charging stations. Their architecture extends the basic TCN with attention mechanisms and feature fusion layers to improve detection of cyber-physical attacks targeting EV charging infrastructure. While demonstrating the versatility of TCNs for specialized domains, the EV charging station context differs significantly from SDN environments in terms of traffic characteristics, attack vectors, and deployment constraints.

Li and Li [28] proposed a lightweight network intrusion detection system based on TCN enhanced with attention mechanisms. Their model incorporates squeeze-and-excitation (SE) blocks within the TCN architecture to recalibrate channel-wise feature responses, improving the model's ability to focus on the most discriminative features. The lightweight design prioritizes inference efficiency, making it suitable for deployment on edge devices. Their results demonstrated strong detection performance with reduced model complexity, aligning with the lightweight design philosophy adopted in this thesis.

Nazre et al. [29] proposed a TCN-based approach for network intrusion detection, evaluating the model on the UNSW-NB15 and CICIDS-2017 datasets. Their study confirmed the effectiveness of dilated causal convolutions for capturing attack-indicative temporal patterns and reported accuracy improvements over baseline models. However, the evaluation was limited to two non-SDN datasets, and the model architecture did not include the residual connections and spatial dropout regularization employed in this thesis.

Sun [42] proposed using TCNs for time-series traffic modeling in network intrusion detection, treating network flow sequences as temporal signals amenable to convolutional analysis. The study demonstrated that TCNs effectively capture both short-term and long-term temporal patterns in network traffic, validating the fundamental approach adopted in this thesis.

Mei et al. [70] proposed a bidirectional TCN (BiTCN) for intrusion detection in intelligent connected vehicles. The bidirectional architecture processes sequences in both forward and backward directions, enabling the capture of temporal patterns that depend on both past and future context. While bidirectional processing can improve detection accuracy in offline analysis scenarios, it is not suitable for real-time intrusion detection where only past and current data are available.

Deng et al. [71] proposed a network intrusion detection model combining multi-layer bidirectional TCN with multi-headed self-attention mechanisms. Their architecture leverages both the temporal modeling capability of BiTCN and the global dependency modeling of self-attention. While achieving strong detection performance, the combined complexity of BiTCN and self-attention introduces significant computational overhead that may limit real-time deployment.

Xu et al. [72] proposed GTCN-G, a residual graph-temporal fusion network specifically designed for imbalanced intrusion detection. Their architecture combines graph neural networks for modeling topological relationships between network entities with temporal convolutional networks for capturing temporal traffic patterns. The fusion of graph and temporal representations enables the model to leverage both structural and temporal information. However, the graph component requires knowledge of the network topology, which may not always be available in real-time detection scenarios.

Peng and Zhang [73] proposed an intrusion detection model integrating Graph Attention Networks (GAT) with Gated Temporal Convolutional Networks (GTCN) for Industrial Internet of Things environments. Their approach models the relationships between IoT devices using graph attention and captures temporal traffic patterns using gated TCN layers. While effective for IoT-specific scenarios, the graph-based approach is computationally expensive and may not scale to large SDN deployments.

Robert et al. [74] proposed TCANet, a hybrid architecture combining TCN with anomaly attention networks and Bi-GRU for network intrusion detection. Their model uses TCN for initial temporal feature extraction, anomaly attention for highlighting suspicious patterns, and Bi-GRU for bidirectional sequence refinement. While the multi-component architecture achieves strong detection performance, the complexity of the model design introduces significant deployment challenges and increases inference latency.

The TCN literature as a whole makes a convincing case for temporal convolutions in network IDS. What it does *not* do—and this is the critical observation—is connect TCN-based detection with any form of cryptographic control-plane protection. Most studies are also tested on general-purpose datasets rather than SDN-specific ones, and comparatively few pay attention to deployment practicalities like model size, inference latency, and integration with the SDN control loop. The TCN-HMAC framework addresses all of these omissions.

3.6 Authentication and Access Control in SDN

Authentication and access control mechanisms are essential for ensuring that only authorized entities can modify network configurations and access sensitive data in SDN environments.

Wang et al. [75] introduced DeepFlowGuard, a deep learning-based controller authentication system for SDN. DeepFlowGuard uses a deep neural network to analyze control channel traffic patterns and identify unauthorized controllers attempting to impersonate legitimate ones. The system demonstrated effective authentication with high accuracy, but it relies on traffic pattern analysis rather than cryptographic verification, making it potentially vulnerable to sophisticated adversaries who can mimic legitimate traffic patterns.

Khan et al. [76] developed MITM-Defender, a real-time defense system against Man-in-the-Middle attacks in SDN that detects controller–switch anomalies using behavioral flow tracking. The system monitors the behavioral characteristics of flow installations to identify deviations from expected patterns that may indicate MITM interception. While effective for detecting a specific class of attacks, MITM-Defender focuses narrowly on MITM threats and does not provide general-purpose intrusion detection or flow rule integrity verification.

Malik and Habib [77] addressed DoS threats through lightweight agents deployed near SDN switches to detect abnormal flow modification activity. Their agent-based approach monitors the rate and pattern of flow installations and modifications, flagging statistical anomalies that may indicate DoS attacks targeting the control plane. The lightweight agent concept is aligned with the auxiliary agent design employed in the TCN-HMAC framework, though Malik and Habib's agents focus exclusively on DoS detection rather than providing comprehensive integrity verification and multi-threat detection.

Dungarani and Gujjar [78] surveyed the security challenges of SDN network automation and discussed various defense mechanisms including role-based access control, certificate-based authentication, and network segmentation. Their comprehensive analysis highlighted the need for layered security approaches that combine multiple defense mechanisms, reinforcing the defense-in-depth philosophy adopted in this thesis.

Mudgal et al. [79] proposed adaptive rule replacement strategies for mitigating inference attacks in serverless SDN frameworks. Their work addresses a sophisticated threat model where adversaries attempt to infer the network's forwarding policies by observing traffic patterns and probing the network. While targeting a different threat vector than the TCN-HMAC framework, their adaptive approach demonstrates the importance of proactive security mechanisms in SDN environments.

3.7 Hybrid Security Approaches

Hybrid security approaches that combine multiple defense mechanisms to provide comprehensive protection have gained increasing attention in the SDN security literature.

Liang et al. [80] comprehensively reviewed SDN-based IDS mechanisms against rule injection and replay threats, identifying the need for integrated approaches that combine detection with verification. Their survey highlighted that existing approaches typically address either detection or verification but not both, leaving security gaps that can be exploited by multi-vector attacks.

Benkhelifa et al. [81] discussed the role of AI-driven security methods for preventing flow tampering and policy abuse in SDN environments. Their analysis identified several promising research directions, including the integration of deep learning with cryptographic mechanisms and the development of lightweight, deployable security solutions. The TCN-HMAC framework proposed in this thesis directly addresses these identified research directions.

Johanyák and Göcs [82] explored edge computing security in SDN-enabled Industrial IoT networks, discussing the unique security challenges posed by the convergence of edge computing, SDN, and IoT. Their work highlighted the need for lightweight security solutions that can operate at the edge without imposing significant computational overhead, aligning with the design objectives of the TCN-HMAC framework.

3.7.1 The Hybrid Gap

Stepping back from the individual papers, a pattern emerges. Existing hybrid solutions tend to be either heavy (deep learning plus blockchain, for instance) or narrow (tackling only DDoS, or only MITM). What is missing is a lightweight hybrid that folds temporal deep-learning detection together with efficient HMAC-based flow-rule verification and challenge–response controller authentication—all in a package that can ship as a normal SDN controller application. That is the gap this thesis sets out to close.

3.8 Dataset Studies for SDN Intrusion Detection

The choice of dataset significantly influences the validity and generalizability of intrusion detection research. Several studies have evaluated and compared datasets for SDN intrusion detection.

Elsayed et al. [18] introduced the InSDN dataset, which was specifically designed for intrusion detection research in SDN environments. The dataset captures realistic network traffic from an SDN testbed using multiple traffic generation tools and includes both benign traffic and multiple attack categories (DDoS, MITM, Probe, and Brute-force). The InSDN dataset has become a widely-used benchmark for SDN-specific intrusion detection research due to its realistic traffic patterns, comprehensive attack coverage, and well-documented feature schema.

Khalid and Aldabagh [83] conducted a survey of the latest intrusion detection datasets for SDN environments, comparing InSDN with other available datasets including CICFlowMeter-generated datasets, NSL-KDD, and custom SDN datasets. Their analysis found that InSDN provides the most comprehensive representation of SDN-specific traffic patterns and attack scenarios, making it the preferred choice for SDN intrusion detection research.

Moustafa et al. [33] provided a holistic review of network anomaly detection systems, including an analysis of feature engineering approaches for various intrusion detection datasets. Their review highlighted the importance of domain-specific feature selection and the need for preprocessing pipelines tailored to the characteristics of the target dataset.

Pan et al. [36] conducted a comparative study of feature selection methods for network intrusion detection, evaluating filter, wrapper, and embedded approaches on multiple benchmark datasets. Their findings demonstrated that correlation-based filter methods provide a favorable balance between computational efficiency and classification performance, supporting the Pearson correlation-based feature selection approach employed in this thesis.

3.9 Summary of Literature and Research Gaps

Table 3.1 summarizes the key characteristics of the reviewed approaches across multiple dimensions relevant to SDN security.

Based on the comprehensive review of related literature, the following research gaps have been identified:

Table 3.1: Summary of Related Work Categorization

Approach Category	**Detection**	**Integrity**	**Real-time**	**Lightweight**
Cryptographic (HMAC/PKI)	×	✓	✓	✓
Blockchain-based	Partial	✓	×	×
ML-based IDS	✓	×	✓	✓
DL-based IDS (RNN)	✓	×	Partial	Partial
DL-based IDS (TCN)	✓	×	✓	✓
Hybrid (DL + Blockchain)	✓	✓	×	×
TCN-HMAC (Proposed)	✓	✓	✓	✓

1. **Gap 1: Detection without Verification.** The majority of deep learning-based IDS approaches focus exclusively on anomaly detection without incorporating integrity verification mechanisms. This leaves the SDN control plane vulnerable to attacks that bypass the detection layer or directly target flow rule integrity.

2. **Gap 2: Verification without Detection.** Cryptographic and protocol-based approaches provide strong integrity guarantees but lack proactive threat detection capabilities. They can verify that flow rules have not been tampered with but cannot detect the broader range of network intrusion activities.

3. **Gap 3: Heavyweight Hybrid Solutions.** Existing hybrid approaches that combine detection with verification typically rely on blockchain or heavyweight cryptographic frameworks that introduce prohibitive latency and computational overhead for real-time SDN operations.

4. **Gap 4: Limited TCN Evaluation on SDN Datasets.** While TCN-based IDS approaches have shown strong performance on general-purpose intrusion detection datasets, their evaluation on SDN-specific datasets such as InSDN is limited, and no existing work has integrated TCN-based detection with SDN-specific security mechanisms.

5. **Gap 5: Absence of Comprehensive Lightweight Hybrid Framework.** No existing work combines temporal deep learning-based intrusion detection with HMAC-based flow rule integrity verification and challenge–response controller authentication in a unified, lightweight, and deployable framework.

The TCN-HMAC framework is designed to close every one of these gaps: it pairs a lightweight TCN-based IDS with an HMAC-based auxiliary agent for flow-rule verification and controller authentication, achieving broad security coverage with minimal computational footprint.

The detailed design appears in Chapter 6, and Chapters 8–9 report the experimental evidence.

Chapter 4

Dataset and Preprocessing

A model is only as good as the data it learns from, so getting the data right matters as much as getting the architecture right. This chapter walks through the InSDN dataset—its composition, features, class distributions, and attack scenarios—and then describes the fifteen-stage preprocessing pipeline that turns the raw captures into a clean, compact, TCN-ready representation. Along the way, we explain why each preprocessing decision was made and how it affects downstream performance.

4.1 The InSDN Dataset

Most publicly available IDS datasets—NSL-KDD, UNSW-NB15, CICIDS-2017—were captured in traditional network architectures, and applying them to SDN research introduces a domain mismatch. The InSDN dataset [18] was purpose-built to avoid that problem: it was generated inside a dedicated SDN testbed, so the traffic flows, feature distributions, and attack manifestations genuinely reflect the way an SDN operates [83].

4.1.1 Dataset Generation Environment

The InSDN dataset was generated in a controlled SDN testbed environment comprising an SDN controller, OpenFlow-enabled switches, and multiple end hosts. The testbed topology was designed to simulate a realistic enterprise network with multiple segments, including server farms, client workstations, and external network connections. Traffic generation was performed using a combination of legitimate traffic generators (to produce realistic benign traffic patterns) and attack tools (to generate representative malicious traffic across multiple attack categories). The CICFlowMeter tool was used to extract flow-level features from the captured network traffic, producing a comprehensive set of 84 flow-level features for each observed network flow.

4.1.2 Source Files and Composition

The InSDN dataset is distributed as three separate CSV files, each representing traffic captured from a different segment or perspective of the SDN testbed:

Table 4.1: InSDN Dataset Source Files

Source File	Traffic Type	Description
`Normal_data.csv`	Benign	Normal/benign network traffic
`OVS.csv`	Mixed	Open vSwitch traffic traces
`metasploitable-2.csv`	Mixed	Traffic including Metasploitable-2 attacks
Combined (raw)	**Mixed**	**343,889 total rows**

All three files share an identical 84-column schema, ensuring seamless concatenation during preprocessing. The combined raw dataset contains 343,889 network flow records spanning both benign traffic and multiple attack categories. The `Label` column contains free-text class names that identify each flow as either “Normal” (benign) or as a specific attack type string.

4.1.3 Feature Schema

The InSDN dataset contains 84 features per flow record, capturing a comprehensive set of flow-level statistics extracted by CICFlowMeter. These features can be categorized into the following groups:

Identifier Features: These include `Flow ID`, `Src IP`, `Src Port`, `Dst IP`, `Dst Port`, and `Timestamp`. These features uniquely identify individual flows and their endpoints but do not carry generalizable information for classification. They are removed during preprocessing to prevent the model from memorizing flow-specific identifiers rather than learning generalizable patterns.

Duration Features: `Flow Duration` captures the total duration of each flow in microseconds. Longer flows may indicate persistent connections (potentially benign) or sustained attacks, while very short flows may indicate scanning or probing activity.

Packet Count Features: Features such as `Total Fwd Packets`, `Total Bwd Packets`, and their derived statistics capture the volume and directionality of packet exchanges. Attack flows often exhibit asymmetric packet counts (e.g., DDoS attacks generate many forward packets with few backward packets).

Byte Count Features: `Total Length of Fwd Packets`, `Total Length of Bwd Packets`, and related features quantify the volume of data transferred in each direction. These features help distinguish between data-heavy attacks (e.g., exfiltration) and lightweight attacks (e.g., probes).

Packet Length Statistics: Features including `Fwd Packet Length Max/Min/Mean/Std` and `Bwd Packet Length Max/Min/Mean/Std` provide distributional statistics about packet sizes. Attack traffic often exhibits distinctive packet size distributions—for example, DDoS floods may use uniformly sized packets, while normal traffic exhibits greater variability.

Inter-Arrival Time Features: `Flow IAT Mean/Std/Max/Min` and their forward/backward variants capture the timing patterns between consecutive packets. These features are particularly valuable for detecting attacks with distinctive temporal signatures, such as high-rate floods (very low IAT) or slow-and-low attacks (very high IAT).

Flag Features: Features based on TCP flags (`FIN Flag Count`, `SYN Flag Count`, `RST Flag Count`, `PSH Flag Count`, `ACK Flag Count`, `URG Flag Count`, `CWE Flag Count`, `ECE Flag Count`) capture the protocol behavior of each flow. Abnormal flag patterns are strong indicators of specific attack types—for example, SYN floods generate disproportionately many SYN flags without corresponding ACKs, and port scans produce flows with high RST flag counts.

Flow-Level Metrics: Features such as `Flow Bytes/s`, `Flow Packets/s`, `Down/Up Ratio`, and `Average Packet Size` provide normalized and derived metrics that capture the overall characteristics and intensity of each flow.

Sub-flow Features: `Subflow Fwd/Bwd Packets` and `Subflow Fwd/Bwd Bytes` capture the packet and byte counts at the sub-flow level, providing finer-grained visibility into the flow's internal structure.

Active/Idle Features: `Active Mean/Std/Max/Min` and `Idle Mean/Std/Max/Min` characterize the active and idle periods within each flow, capturing the burstiness and periodicity of traffic patterns.

4.1.4 Attack Categories

The InSDN dataset includes four major categories of network attacks that represent the most prevalent and impactful threats to SDN environments:

Distributed Denial of Service (DDoS): DDoS attack flows in the InSDN dataset include various flooding techniques targeting the SDN infrastructure. These attacks generate high volumes of traffic with distinctive characteristics such as high packet rates, uniform packet sizes, and low flow durations. In the SDN context, DDoS attacks are particularly dangerous due to the table-flooding vulnerability, where the reactive flow installation mechanism can be exploited to overwhelm the controller.

Man-in-the-Middle (MITM): MITM attack flows capture scenarios where an adversary intercepts and potentially modifies communications between legitimate parties. These flows exhibit patterns such as ARP spoofing signatures, duplicated packet sequences, and anomalous timing characteristics that differ from normal point-to-point communications.

Probe (Reconnaissance): Probe attack flows represent systematic scanning activities including TCP SYN scans, UDP scans, ICMP sweeps, and service enumeration. These flows are typically characterized by many short-lived connections to multiple ports or hosts, high SYN/RST flag ratios, and systematic address or port incrementation patterns.

Brute-Force: Brute-force attack flows capture credential-guessing attempts against network services such as SSH, FTP, HTTP, and SNMP. These flows exhibit patterns of repeated connection attempts to authentication ports, short session durations followed by re-connection, and systematic variation in authentication payloads.

4.1.5 Raw Label Distribution

The raw InSDN dataset exhibits a significant class imbalance between benign and attack flows:

Table 4.2: Raw Label Distribution in the InSDN Dataset

Class	**Count**	**Share (%)**
Benign (Label = 0)	68,424	19.9
Attack (Label = 1)	275,465	80.1
Total	**343,889**	**100.0**

The attack class constitutes approximately 80% of the raw dataset, with benign flows comprising only 20%. This class imbalance reflects the intensive traffic generation methodology used during dataset creation, where multiple attack tools were employed simultaneously to ensure comprehensive attack coverage. The imbalance is addressed through class weighting during model training, as described in Section 4.7.

4.2 Preprocessing Pipeline

Turning the raw InSDN captures into something a TCN can learn from takes fifteen sequential stages. The first nine—consolidation, cleaning, feature reduction—run in the `preprocess.py`

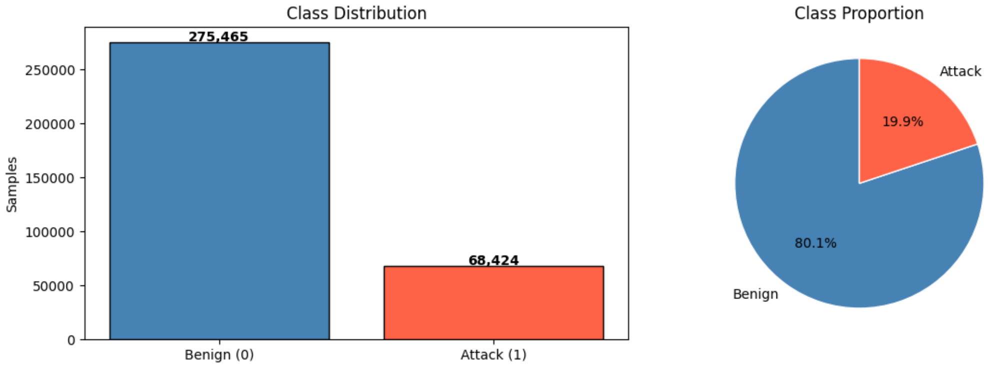

Figure 4.1: Class distribution in the raw InSDN dataset. Left: bar chart showing absolute counts (275,465 attack vs. 68,424 benign). Right: pie chart showing the 80.1% to 19.9% class imbalance ratio.

script and produce a consolidated CSV. The remaining six—second-pass cleaning, deduplication, scaling, PCA, splitting, and class weighting—execute inside the training notebook (`tcn_insdn_colab.ipynb`). Each stage addresses a specific data-quality or representation concern, as detailed below.

4.2.1 Stage 1: Data Consolidation

The first preprocessing stage concatenates the three source CSV files (`Normal_data.csv`, `OVS.csv`, and `metasploitable-2.csv`) into a single unified dataset. All three files share the same 84-column schema, enabling straightforward vertical concatenation using the `pandas.concat` function. The resulting consolidated dataset contains 343,889 rows and 84 columns.

4.2.2 Stage 2: Column Name and String Value Cleaning

Whitespace inconsistencies in column names and string values can cause silent errors during feature selection and label encoding. This stage strips leading and trailing whitespace from all column names and all string-type cell values in the dataset. This ensures consistent column referencing throughout the preprocessing pipeline and accurate label matching during binary encoding.

4.2.3 Stage 3: Identifier Column Removal

Identifier and metadata columns (`Flow ID`, `Src IP`, `Src Port`, `Dst IP`, `Dst Port`, `Timestamp`) are removed from the dataset. These columns contain flow-specific identifiers that do not generalize to unseen traffic. If retained, the model might learn to associate specific IP addresses or port numbers with attack labels, leading to overfitting on the training data distribution and poor generalization to different network environments. After this stage, the dataset contains 78 columns (77 features + 1 label column).

4.2.4 Stage 4: Binary Label Encoding

The original `Label` column contains free-text class names ("Normal" for benign traffic and various attack-specific strings for different attack types). For the binary classification task of distinguishing benign from malicious traffic, all labels are encoded as:

- `Normal` → 0 (Benign)
- All other labels → 1 (Attack)

This binary encoding simplifies the classification task to a detection problem: is the given flow benign or malicious? By grouping all attack types into a single class, the model learns to identify general attack characteristics that distinguish malicious traffic from normal traffic, regardless of the specific attack category. This approach is particularly appropriate for the real-time deployment scenario where the primary objective is to flag suspicious flows for further investigation, rather than to classify the specific attack type at the point of detection. After encoding, the dataset contains 68,424 benign flows (19.9%) and 275,465 attack flows (80.1%).

4.2.5 Stage 5: Handling Infinite and Missing Values

Network flow feature extraction can produce infinite values (e.g., when computing rates from zero-duration flows) and missing values (e.g., from failed feature computations). This stage performs two sequential cleaning operations:

1. **Infinite Value Replacement:** All `Inf` and `-Inf` values are replaced with `NaN` to enable uniform handling of all non-finite values.
2. **NaN Row Removal:** All rows containing any `NaN` values are dropped from the dataset. This approach was chosen over imputation (e.g., mean/median filling) because the proportion of affected rows is small and because imputation could introduce artificial values that distort the learned feature distributions.

4.2.6 Stage 6: Zero-Variance Feature Removal

Features that exhibit zero variance (i.e., have the same value for all samples) carry no discriminative information and consume computational resources without contributing to classification performance. This stage identifies and removes all constant columns from the dataset. In the InSDN dataset, 12 features were identified as having zero variance and were removed. These features were primarily flag-related columns (e.g., certain rarely-used TCP flag counts) and sub-flow features that did not vary across the recorded flows.

4.2.7 Stage 7: Near-Constant Feature Removal

Beyond zero-variance features, features that are nearly constant—defined as having more than 99.9% of their values equal to a single value—provide minimal discriminative value. Such features can cause numerical instability during normalization and may introduce noise during model training. This stage identifies and removes features exceeding the 99.9% near-constant threshold. In the InSDN dataset, no additional features were removed at this stage, indicating that all remaining features after zero-variance removal exhibit sufficient variability to be potentially informative.

4.2.8 Stage 8: Correlation-Based Feature Reduction

When two features carry essentially the same information, keeping both only inflates the input dimension without helping the classifier. This stage computes the Pearson correlation coefficient [43] for every feature pair and drops one member of each pair whose absolute correlation exceeds 0.98.

The Pearson correlation coefficient between features X and Y is defined as:

$$r_{XY} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{4.1}$$

A threshold of $|r| > 0.98$ identifies feature pairs that are nearly perfectly correlated or anti-correlated. For each identified pair, one feature is retained (typically the one with higher variance or earlier position in the feature list) and the other is removed. This stage removed 17 features from the dataset, reducing the feature count from 65 to 48. The removed features were primarily redundant volume metrics (e.g., forward byte count features that were nearly perfectly correlated with forward packet count features) and bidirectional versions of the same statistic.

4.2.9 Stage 9: Preprocessing Output

After completing all eight preprocessing stages, the cleaned dataset is saved as `insdn_consolidated` with the following characteristics:

Table 4.3: Preprocessing Output Summary

Property	
File Name	`insdn_consolic`
File Size	
Total Rows	
Total Columns	49 (48 featu
Features Retained	
Features Removed	36 (6 identifier + 12 zero-variance + 0 near-constant + 17 correlated + 1 lal
Benign Flows (Label = 0)	68,
Attack Flows (Label = 1)	275,

4.3 Second-Pass Cleaning and Deduplication

The training notebook applies a second pass of cleaning and deduplication to handle any artifacts introduced during file I/O and to remove duplicate flow records that could bias model training.

4.3.1 Second-Pass Infinite and Missing Value Check

After loading the consolidated CSV into the training environment, a safety-net check is performed for infinite and missing values. In the case of the InSDN dataset, zero infinite values and zero NaN values were detected at this stage, confirming the thoroughness of the initial preprocessing.

4.3.2 Duplicate Row Removal

Duplicate flow records are identified and removed based on exact equality across all 49 columns. Duplicate rows are problematic for several reasons: (1) they artificially inflate the dataset size, giving disproportionate weight to certain flow patterns; (2) they can leak information between training and test sets if the same flow appears in both splits; and (3) they reduce the effective diversity of the training data, potentially leading to overfitting on repeated patterns.

The deduplication stage identified and removed 160,058 duplicate rows, reducing the dataset from 343,889 to 182,831 unique flow records—a 46.5% reduction. This large number of duplicates reflects the traffic generation methodology: the raw OVS and metasploitable-2 captures contain many identical flow records resulting from repeated traffic patterns during the data collection process.

Table 4.4: Impact of Deduplication on Dataset Size

Metric	**Before Dedup**	**After Dedup**
Total Rows	343,889	182,831
Duplicate Rows Removed	—	160,058
Reduction Percentage	—	46.5%

After deduplication, the class distribution shifts slightly:

Table 4.5: Post-Deduplication Label Distribution

Class	**Count**	**Share (%)**
Benign (Label = 0)	64,114	35.1
Attack (Label = 1)	118,717	64.9
Total	**182,831**	**100.0**

The deduplication process disproportionately removes attack flows (from 80.1% to 64.9%), suggesting that the attack traffic generation tools produced more repetitive flow patterns than the benign traffic generators. The resulting 35:65 benign-to-attack ratio represents a moderate class imbalance that is addressed through class weighting during training.

4.4 Feature Scaling

Raw network-flow features span wildly different ranges—byte counts can run into the millions while flag counts stay in single digits. Without scaling, the large-valued features would dominate the gradient during back-propagation, and the model would effectively ignore the rest. Feature scaling levels the playing field.

StandardScaler normalization is applied to transform each feature to have zero mean and unit variance:

$$x_i' = \frac{x_i - \mu_i}{\sigma_i} \tag{4.2}$$

where μ_i and σ_i are the mean and standard deviation of feature i computed on the *training set only*. The same transformation parameters (μ_i, σ_i) are then applied to the validation and test sets to prevent data leakage.

StandardScaler was chosen over MinMaxScaler (which scales to $[0, 1]$) for several reasons: (1) StandardScaler is less sensitive to outliers because it does not bound the transformed values; (2) it preserves the relative distances between data points; and (3) it produces feature distributions that are well-suited for the gradient-based optimization algorithms used in deep learning training.

4.5 Principal Component Analysis

Even after dropping correlated features, some redundancy remains—linear correlations below the 0.98 threshold and nonlinear dependencies that Pearson simply cannot detect. PCA mops up that residual redundancy. We set a 95% variance retention threshold, meaning we keep the fewest principal components needed to explain at least 95% of the total variance.

Table 4.6: PCA Configuration and Results

Parameter	**Value**
Input Dimensionality	48 features
Variance Threshold	95%
Output Dimensionality	**24 principal components**
Actual Variance Retained	**95.43%**
Dimensionality Reduction	50% (48 → 24)

The PCA transformation reduces the feature space from 48 dimensions to 24 principal components, achieving a 50% reduction in dimensionality while retaining 95.43% of the total variance. This reduction offers several benefits:

- **Reduced Computational Cost:** The halved feature dimensionality directly translates to reduced computation in the TCN's input layer and throughout the network, enabling faster training and inference.

- **Reduced Overfitting Risk:** By projecting the data onto the directions of maximum variance, PCA removes noise components that might otherwise cause the model to overfit to irrelevant variations in the training data.

- **Decorrelated Features:** The principal components are orthogonal by construction, meaning they are linearly uncorrelated. This decorrelation eliminates multicollinearity issues that might affect learning efficiency.

- **Interpretable Variance Capture:** The cumulative explained variance curve provides a clear visualization of how much information is retained at each dimensionality level, enabling informed decisions about the trade-off between compression and information loss.

The first few principal components capture the largest variance contributions: the first component captures approximately 15% of the total variance, the first five components capture approximately 50%, and the first 15 components capture approximately 85%. The rapid initial increase followed by a gradual plateau is characteristic of datasets with significant redundancy among features, confirming that the correlation-based feature reduction in Stage 8 was effective but that additional redundancy exists in the form of nonlinear correlations that Pearson correlation cannot detect.

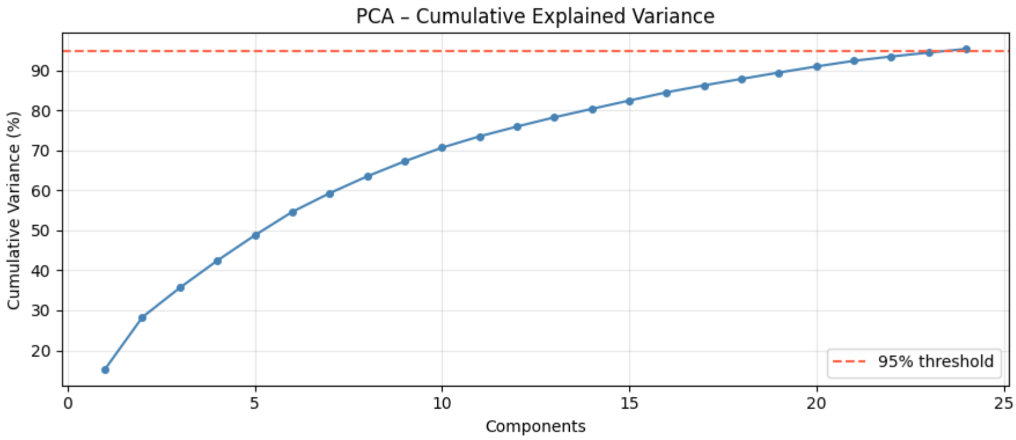

Figure 4.2: PCA cumulative explained variance curve. The 95% variance retention threshold (dashed red line) is reached at 24 principal components, achieving 95.43% variance retention with a 50% reduction in dimensionality from 48 to 24 features.

The PCA transformation is fitted on the training set only and applied to the validation and test sets using the same transformation matrix. This ensures that no information from the validation or test sets leaks into the PCA computation, maintaining the integrity of the evaluation protocol.

4.6 Data Splitting

The deduplicated and PCA-transformed dataset is split into training, validation, and test sets using stratified random sampling. Stratification ensures that the class ratio (benign:attack) is preserved across all three subsets, preventing split-induced class distribution shifts that could bias model training or evaluation.

The 70/10/20 split ratio was chosen based on the following considerations:

- **70% Training:** A large training set provides sufficient data for the TCN model to learn robust feature representations. With 127,981 training samples, the model has access to a diverse range of flow patterns for both classes.

Table 4.7: Stratified Train/Validation/Test Split

Split	Samples	Attack	Benign	Share (%)
Train	127,981	83,101	44,880	70.0
Validation	18,283	11,872	6,411	10.0
Test	36,567	23,744	12,823	20.0
Total	**182,831**	**118,717**	**64,114**	**100.0**

- **10% Validation:** The validation set is used for monitoring training progress, early stopping, learning rate scheduling, and model checkpoint selection. The 18,283-sample validation set provides statistically reliable estimates of generalization performance during training.

- **20% Test:** A substantial test set of 36,567 samples enables robust evaluation of the final model with narrow confidence intervals on all performance metrics. The test set is never used during training or hyperparameter tuning, ensuring an unbiased estimate of out-of-sample performance.

4.6.1 Input Reshaping for TCN

After PCA transformation and splitting, each data sample is a 24-dimensional vector corresponding to the 24 principal components. To serve as input to the TCN model, which expects three-dimensional input tensors of shape `(batch_size, sequence_length, channels)`, each sample is reshaped from $(24,)$ to $(24, 1)$. This reshaping treats the 24 PCA components as a temporal sequence of length 24 with a single channel.

This interpretation is meaningful because: (1) the PCA components are ordered by decreasing variance, establishing a natural ordering; (2) the dilated causal convolutions of the TCN can learn local and global patterns across this ordered representation; and (3) the residual blocks with exponentially increasing dilation factors enable the model to capture dependencies between components at multiple scales of the variance hierarchy.

The final input shapes for each split are:

Table 4.8: Input Tensor Shapes After Reshaping

Split	Shape (samples, sequence_length, channels)
Train	(127,981, 24, 1)
Validation	(18,283, 24, 1)
Test	(36,567, 24, 1)

4.7 Class Weighting

After deduplication the dataset sits at roughly 35% benign versus 65% attack—not extreme, but enough to bias a naïve model toward always predicting "attack." Rather than synthesising new samples or throwing away existing ones, we handle the imbalance with class weights computed from inverse frequencies:

$$w_c = \frac{N}{C \cdot n_c} \tag{4.3}$$

where $N = 127{,}981$ is the total number of training samples, $C = 2$ is the number of classes, and n_c is the number of training samples in class c.

Table 4.9: Computed Class Weights

Class	Training Samples	Weight
Benign (0)	44,880	1.4258
Attack (1)	83,101	0.7700

The benign class receives a weight of 1.4258, which is approximately 1.85× the attack class weight of 0.7700. This weighting ensures that the model penalizes misclassification of benign flows (false positives) more heavily relative to misclassification of attack flows (false negatives), compensating for the underrepresentation of benign samples in the training data. The weighted loss function effectively makes each benign sample count as 1.85 samples during gradient computation, preventing the model from achieving high accuracy by simply classifying all flows as attacks.

Class weighting was chosen over alternative strategies (such as SMOTE [47] oversampling or random undersampling) for the following reasons:

1. **No Synthetic Data Introduction:** Unlike SMOTE, class weighting does not generate synthetic minority samples that may not accurately represent real traffic patterns. This preserves the authenticity of the training data distribution.

2. **No Data Discarding:** Unlike undersampling, class weighting retains all training samples, maximizing the information available for model learning.

3. **Seamless Integration:** Class weights integrate directly with the binary cross-entropy loss function used for training, requiring no modifications to the data pipeline or model architecture.

4. **Computational Efficiency:** Class weighting adds no computational overhead during training, as the weights are simply multiplied with the per-sample loss values.

4.8 Preprocessing Pipeline Summary

Table 4.10 provides a comprehensive summary of the complete preprocessing pipeline, documenting each stage, the action performed, and the resulting dataset state.

Table 4.10: Complete Preprocessing Pipeline Summary

Stage	Action	Result
1	Load and concatenate 3 CSVs	343,889 × 84
2	Strip whitespace from names/values	—
3	Drop 6 identifier columns	343,889 × 78
4	Binary label encoding	Benign: 68,424; Attack: 275,465
5	Replace Inf, drop NaN rows	—
6	Remove 12 zero-variance columns	343,889 × 66
7	Remove near-constant columns	343,889 × 66
8	Remove 17 correlated features	343,889 × 49
9	Save consolidated CSV	71 MB
10	Deduplicate rows	182,831 × 49
11	StandardScaler normalization	—
12	PCA (95% variance)	182,831 × 24
13	Stratified train/val/test split	70/10/20
14	Reshape to (N, 24, 1)	TCN-ready input
15	Compute class weights	[1.4258, 0.7700]

Starting from 343,889 raw flow records spread across three CSV files, the pipeline delivers 182,831 deduplicated, standardised, PCA-compressed samples in a 70/10/20 train/validation/test split—ready for the TCN model described in the next chapter. Every decision along the way, from dropping identifier columns to choosing StandardScaler over MinMaxScaler, was driven by the twin goals of maximising generalisation and keeping the computational footprint small.

Chapter 5

TCN Model Architecture

Designing an IDS model for an SDN controller is, in many ways, an exercise in constraint satisfaction. The model has to be accurate enough to catch real attacks, fast enough to keep up with the data plane, and small enough to live alongside the controller application without starving it of memory. This chapter lays out the architecture of the TCN model—named `TCN_InSDN`—that we use for the detection component of the TCN-HMAC framework. We walk through the residual-block design, the dilation schedule, the classification head, hyperparameter choices, the training configuration, and finally a complexity analysis showing that the whole model fits in roughly 612 KB.

5.1 Model Overview

At a high level, `TCN_InSDN` is straightforward: six dilated causal one-dimensional residual blocks followed by a two-layer dense classification head. What makes the design interesting is the tension it resolves among accuracy, speed, and size.

The model receives input tensors of shape $(N, 24, 1)$, where N is the batch size, 24 is the sequence length (corresponding to the 24 PCA-transformed features), and 1 is the number of input channels. The output is a single sigmoid-activated value representing the probability that the input flow is an attack. The complete model architecture is illustrated in Table 5.1.

That is only 156,737 parameters—about 612 KB on disk. To put this in perspective, a comparable LSTM would need two to three times as many parameters, and a Transformer-based model of similar capacity would balloon to five to ten times the size. The compactness comes from weight sharing: convolutional filters are reused across every temporal position.

Table 5.1: TCN_InSDN Model Architecture Summary

Layer	Type	Dilation	Output Shape	Parameters
Input	—	—	$(N, 24, 1)$	0
Block 1	Residual Block	$d = 1$	$(N, 24, 64)$	1,536
Block 2	Residual Block	$d = 2$	$(N, 24, 64)$	25,216
Block 3	Residual Block	$d = 4$	$(N, 24, 64)$	25,216
Block 4	Residual Block	$d = 8$	$(N, 24, 64)$	25,216
Block 5	Residual Block	$d = 16$	$(N, 24, 64)$	25,216
Block 6	Residual Block	$d = 32$	$(N, 24, 64)$	25,216
GAP	GlobalAvgPooling1D	—	$(N, 64)$	0
Dense 1	Dense + BN + ReLU	—	$(N, 128)$	8,576
Dropout 1	Dropout(0.2)	—	$(N, 128)$	0
Dense 2	Dense + BN + ReLU	—	$(N, 64)$	8,448
Dropout 2	Dropout(0.2)	—	$(N, 64)$	0
Output	Dense(1, sigmoid)	—	$(N, 1)$	65
Total Parameters				**156,737**
Trainable Parameters				**154,817**
Non-Trainable Parameters (BN)				**1,920**
Model Size				**∼612 KB**

5.2 Residual Block Design

The residual block is where most of the learning happens. Each block runs the input through a pair of dilated causal convolutions—with batch normalisation, ReLU, and spatial dropout sandwiched in—and then adds the result back to the original input via a residual shortcut.

5.2.1 Block Internal Structure

Each residual block consists of two identical sub-layers, each comprising:

1. **Conv1D (causal, dilated):** A one-dimensional convolution with causal padding, dilation rate d, 64 filters, and kernel size 3. The causal padding ensures that the output at time step t depends only on inputs at time steps $\leq t$. The number of filters (64) determines the channel dimensionality of the block's output.

2. **BatchNormalization:** Normalizes the convolutional output to have zero mean and unit variance across the batch dimension, stabilizing training dynamics and enabling higher learning rates.

3. **ReLU Activation:** The Rectified Linear Unit activation function $\text{ReLU}(x) = \max(0, x)$ introduces nonlinearity, enabling the network to learn complex, nonlinear decision

boundaries. ReLU was chosen over alternatives (sigmoid, tanh, LeakyReLU) due to its computational simplicity, effectiveness in mitigating the vanishing gradient problem, and widespread empirical success in deep convolutional networks.

4. **SpatialDropout1D(0.2):** Drops entire feature map channels with probability 0.2 during training. As discussed in Section 2.2.5, spatial dropout is more appropriate than element-wise dropout for one-dimensional convolutional networks because it preserves the temporal coherence of retained channels.

The two sub-layers are applied sequentially, with the first sub-layer processing the input and producing an intermediate representation, and the second sub-layer further refining this representation to produce the block's transformed output $\mathscr{F}(\mathbf{x})$.

5.2.2 Residual Skip Connection

The residual skip connection adds the block's input $\mathbf{x}$ to the transformed output $\mathscr{F}(\mathbf{x})$:

$$\text{output} = \mathscr{F}(\mathbf{x}) + \text{shortcut}(\mathbf{x}) \tag{5.1}$$

When the input and output channel dimensions match (i.e., both are 64 channels), the shortcut is a direct identity connection: $\text{shortcut}(\mathbf{x}) = \mathbf{x}$. When the dimensions differ—as in Block 1, where the input has 1 channel and the output has 64 channels—a 1×1 convolution is applied to the shortcut path to project the input to the correct dimensionality:

$$\text{shortcut}(\mathbf{x}) = W_{proj} * \mathbf{x} \tag{5.2}$$

where W_{proj} is a 1×1 convolutional kernel with 64 output channels. This projection adds $1 \times 64 + 64 = 128$ parameters (kernel weights + bias) per channel-mismatched block.

The residual connection provides the gradient with a direct path from the loss function to the early layers, mitigating the vanishing gradient problem and enabling the training of deeper networks. It also allows each block to learn an incremental refinement (residual function) rather than a complete transformation, which is generally easier to optimize.

5.2.3 Block Internal Data Flow

The complete data flow through a single residual block with dilation rate d is formalized as:

$$\mathbf{h}_1 = \text{SpatialDropout}(\text{ReLU}(\text{BN}(\text{Conv1D}_{d,64,3}(\mathbf{x})))) \quad (5.3)$$

$$\mathbf{h}_2 = \text{SpatialDropout}(\text{ReLU}(\text{BN}(\text{Conv1D}_{d,64,3}(\mathbf{h}_1)))) \quad (5.4)$$

$$\mathscr{F}(\mathbf{x}) = \mathbf{h}_2 \quad (5.5)$$

$$\text{output} = \mathscr{F}(\mathbf{x}) + \text{shortcut}(\mathbf{x}) \quad (5.6)$$

where $\text{Conv1D}_{d,64,3}$ denotes a causal one-dimensional convolution with dilation rate d, 64 filters, and kernel size 3.

5.3 Dilation Schedule

The six blocks use dilation rates that double at each stage—1, 2, 4, 8, 16, 32—following the standard geometric progression $d_l = 2^{l-1}$:

Table 5.2: Dilation Schedule and Receptive Field Growth

Block	**Dilation Rate**	**Block Receptive Field**	**Cumulative Receptive Field**
1	1	$2 \times (3-1) \times 1 + 1 = 5$	5
2	2	$2 \times (3-1) \times 2 = 8$	13
3	4	$2 \times (3-1) \times 4 = 16$	29
4	8	$2 \times (3-1) \times 8 = 32$	61
5	16	$2 \times (3-1) \times 16 = 64$	125
6	32	$2 \times (3-1) \times 32 = 128$	253

Because the cumulative receptive field (253) dwarfs the input length (24), the deepest layers can “see” the entire input sequence with room to spare. The extra coverage is not wasted—it cushions the boundary effects caused by zero-padding and makes the model more robust to shifts in the feature ordering.

The exponential dilation schedule was chosen over alternatives (linear dilation, repeated cycles) based on the following considerations:

- **Exponential Efficiency:** The exponential growth of the receptive field enables coverage of the full input sequence with only $\lceil \log_2(T) \rceil$ blocks for kernel size 2, or fewer blocks for larger kernels. This minimizes the depth of the network while maximizing temporal coverage.

- **Multi-Scale Pattern Capture:** Each dilation rate captures patterns at a different temporal scale. Block 1 ($d = 1$) captures fine-grained, adjacent-component patterns;

Block 3 ($d = 4$) captures medium-scale patterns spanning 4–8 components; Block 6 ($d = 32$) captures global patterns spanning the entire input. This multi-scale hierarchy enables the model to detect both localized anomalies (e.g., a single feature spike) and distributed anomalies (e.g., correlated changes across many features).

- **Empirical Validation:** The exponential dilation schedule has been demonstrated to be effective in the original TCN architecture [15] and in subsequent TCN-based intrusion detection studies [16, 28].

5.4 Classification Head

The classification head converts the temporal feature representation produced by the residual blocks into a binary classification output. It consists of three sequential stages:

5.4.1 Global Average Pooling

GlobalAveragePooling1D aggregates the temporal feature maps of shape $(N, 24, 64)$ into a fixed-length vector of shape $(N, 64)$ by computing the average activation across the 24 time steps for each of the 64 channels. This produces a compact representation that captures the overall statistical behavior of each feature channel across the entire input sequence.

5.4.2 Dense Layers

Two dense (fully connected) layers progressively reduce the representation dimensionality while capturing nonlinear interactions between the pooled features:

- **Dense Layer 1:** 128 neurons with batch normalization, ReLU activation, and dropout($p = 0.2$). This layer expands the 64-dimensional pooled representation to 128 dimensions, enabling the network to model higher-order feature interactions. The batch normalization and dropout provide regularization to prevent overfitting in this high-dimensional space.
- **Dense Layer 2:** 64 neurons with batch normalization, ReLU activation, and dropout($p = 0.2$). This layer compresses the representation back to 64 dimensions, distilling the learned interactions into a compact form suitable for the final classification.

5.4.3 Output Layer

The output layer is a single neuron with sigmoid activation:

$$p(\text{attack} \mid \mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{h} + b) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{h}+b)}} \tag{5.7}$$

where $\mathbf{h}$ is the 64-dimensional output of Dense Layer 2, $\mathbf{w}$ is the weight vector, b is the bias, and $\sigma(\cdot)$ is the sigmoid function. The sigmoid output is interpreted as the probability that the input flow is an attack. At inference time, a threshold of 0.5 is applied: flows with $p > 0.5$ are classified as attacks, and flows with $p \leq 0.5$ are classified as benign.

5.5 Hyperparameter Selection

The dense layers were not chosen by guesswork. We swept over several hyperparameter axes on the validation set and settled on the configuration below.

Table 5.3: TCN Model Hyperparameters

Hyperparameter	**Value**	**Justification**
Number of Residual Blocks	6	Provides receptive field of 253, exceeding input length 24
Filters per Block	64	Balances representational capacity with model size
Kernel Size	3	Standard choice for TCN; captures local patterns effectively
Dilation Factors	[1,2,4,8,16,32]	Exponential growth ensures full input coverage
Spatial Dropout Rate	0.2	Moderate regularization; prevents overfitting without excessive information loss
Dense Layer 1 Neurons	128	Sufficient capacity for feature interaction modeling
Dense Layer 2 Neurons	64	Dimensionality reduction before output
Dense Dropout Rate	0.2	Consistent with block dropout for uniform regularization

Number of Filters (64): The choice of 64 filters per block balances model capacity with computational efficiency. A filter count of 32 was found to underfit the data (validation accuracy plateaued at approximately 99.5%), while 128 filters provided marginal accuracy improvement (less than 0.1%) at the cost of tripling the model size. The 64-filter configuration achieves near-optimal performance while keeping the model compact enough for real-time deployment.

Kernel Size (3): A kernel size of 3 is the standard choice in TCN architectures, providing a local receptive field that captures relationships between adjacent elements in the sequence. Larger kernel sizes (5 or 7) were evaluated but showed no significant accuracy improvement on the validation set while increasing the parameter count. The effectiveness of kernel size 3 is enhanced by the dilated convolutions, which effectively create "virtual" kernel sizes of $3 \cdot d$ at each dilation level.

Dropout Rate (0.2): The dropout rate of 0.2 (dropping 20% of channels/neurons) represents a moderate level of regularization. This rate was determined through a sweep over values $\{0.1, 0.15, 0.2, 0.25, 0.3\}$ on the validation set. Rates below 0.15 showed signs of mild overfitting (validation loss increasing while training loss continued to decrease), while rates above 0.25 caused underfitting (lower training accuracy without corresponding improvement in validation accuracy).

5.6 Training Configuration

5.6.1 Optimizer

The Adam optimizer [84] is used for training the TCN model. Adam combines the benefits of two gradient descent variants: AdaGrad (which adapts the learning rate individually for each parameter based on historical gradient magnitudes) and RMSProp (which uses exponentially weighted moving averages of squared gradients to normalize the parameter updates). The Adam update rule is:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{5.8}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{5.9}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{5.10}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{5.11}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{5.12}$$

where g_t is the gradient at time step t, m_t and v_t are the first and second moment estimates, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are the exponential decay rates, $\alpha = 10^{-3}$ is the initial learning rate, and $\epsilon = 10^{-7}$ is a numerical stability constant.

Adam was chosen over alternatives (SGD with momentum, AdamW [85], RMSProp) for several reasons: (1) it converges faster than vanilla SGD on the InSDN dataset, reaching near-optimal performance within 15–20 epochs compared to 50+ epochs for SGD; (2) its adaptive learning rates handle the heterogeneous scale of PCA-transformed features effectively; and (3) it has been demonstrated to perform well on convolutional architectures for classification tasks.

5.6.2 Loss Function

Binary cross-entropy loss is used as the training objective:

$$\mathscr{L} = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \tag{5.13}$$

where $y_i \in \{0, 1\}$ is the true label, $\hat{y}_i \in (0, 1)$ is the predicted probability, w_{y_i} is the class weight for the true label, and N is the batch size. The class weights $w_0 = 1.4258$ and $w_1 = 0.7700$ are applied to compensate for the class imbalance, as described in Section 4.7.

Binary cross-entropy is the standard loss function for binary classification with sigmoid output and is well-suited for probabilistic classification where the output represents a calibrated probability estimate [86].

5.6.3 Batch Size

A batch size of 2,048 is used for training. This relatively large batch size is chosen for several reasons:

- **GPU Utilization:** Large batch sizes maximize the utilization of GPU parallel processing capabilities, reducing per-sample computation time. On the NVIDIA T4 GPU used for training, a batch size of 2,048 achieves near-complete memory utilization without exceeding the available 16 GB of GPU memory.
- **Gradient Stability:** Larger batches produce more stable gradient estimates, reducing the noise in parameter updates and enabling more consistent convergence. This is particularly important for batch normalization, which performs best with sufficiently large batch sizes to produce accurate mean and variance estimates.
- **Training Speed:** With 127,981 training samples and a batch size of 2,048, each epoch consists of approximately 63 iterations, enabling rapid epoch completion and efficient use of the early stopping patience.

5.6.4 Training Epochs and Early Stopping

The maximum number of training epochs is set to 100, but training is governed by an early stopping callback that monitors the validation AUC:

- **Monitor:** Validation AUC (`val_auc`)
- **Patience:** 15 epochs
- **Mode:** Maximize
- **Restore Best Weights:** True

If the validation AUC does not improve for 15 consecutive epochs, training is terminated and the model weights from the epoch with the highest validation AUC are restored. This mechanism prevents overfitting by stopping training before the model begins to memorize noise in the training data. In practice, training converged at approximately 30 epochs, well before the 100-epoch maximum.

5.6.5 Learning Rate Scheduling

A ReduceLROnPlateau learning rate scheduler is employed to adaptively reduce the learning rate when training plateaus:

- **Monitor:** Validation loss (`val_loss`)
- **Factor:** 0.5 (halve the learning rate)
- **Patience:** 7 epochs
- **Minimum Learning Rate:** 1×10^{-6}

This scheduler provides fine-grained control over the learning dynamics: the model begins training with a relatively high learning rate (10^{-3}) for rapid initial convergence, and the rate is progressively reduced as the model approaches its optimal performance, enabling finer weight adjustments in the later stages of training. The multiplicative factor of 0.5 allows the learning rate to decrease from 10^{-3} to the minimum 10^{-6} through at most 10 reduction steps, providing a smooth annealing schedule [87].

5.6.6 Model Checkpointing

A model checkpoint callback saves the model weights at each epoch where the validation AUC improves:

- **Monitor:** Validation AUC (`val_auc`)
- **Mode:** Maximize
- **Save Best Only:** True
- **Save Path:** `best_tcn_insdn.keras`

This ensures that the final model used for evaluation and deployment is the one that achieved the highest validation AUC, even if subsequent epochs showed degraded performance due to overfitting.

5.6.7 Training Metrics

The following metrics are monitored during training and logged to a CSV file (`training_log.csv`) for post-training analysis:

- **Accuracy:** Overall classification accuracy on training and validation sets.
- **AUC:** Area Under the ROC Curve on training and validation sets.
- **Precision:** Fraction of predicted attacks that are true attacks.
- **Recall:** Fraction of true attacks that are correctly detected.

5.6.8 Training Configuration Summary

Table 5.4 summarizes the complete training configuration:

5.7 Model Complexity Analysis

5.7.1 Parameter Count Breakdown

Table 5.5 provides a detailed breakdown of the parameter count for each component of the TCN model:

Table 5.4: Complete Training Configuration

Parameter	**Value**
Framework	TensorFlow 2.19.0
Hardware	NVIDIA T4 GPU (16 GB)
Optimizer	Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
Initial Learning Rate	1×10^{-3}
Loss Function	Weighted Binary Cross-Entropy
Batch Size	2,048
Max Epochs	100
Early Stopping	patience=15, monitor=val_auc
LR Reduction	factor=0.5, patience=7, min_lr=10^{-6}
Class Weights	[1.4258, 0.7700]
Checkpoint	Save best by val_auc
Convergence	∼30 epochs

Table 5.5: Parameter Count Breakdown by Component

Component	**Trainable**	**Non-Trainable**
Block 1 (Conv1D × 2 + BN × 2 + 1×1 proj)	1,472	64
Block 2 (Conv1D × 2 + BN × 2)	24,960	256
Block 3 (Conv1D × 2 + BN × 2)	24,960	256
Block 4 (Conv1D × 2 + BN × 2)	24,960	256
Block 5 (Conv1D × 2 + BN × 2)	24,960	256
Block 6 (Conv1D × 2 + BN × 2)	24,960	256
Dense 1 (128 units + BN)	8,320	256
Dense 2 (64 units + BN)	8,320	128
Output (1 unit)	65	0
Total	**154,817**	**1,920**

The 1,920 non-trainable parameters correspond to the running mean and variance statistics maintained by the batch normalization layers. These statistics are updated during training using exponential moving averages but are not updated by gradient descent.

5.7.2 Computational Complexity

The computational complexity of a single forward pass through the TCN model can be analyzed in terms of floating-point operations (FLOPs):

Convolutional Layers: For a Conv1D layer with C_{in} input channels, C_{out} output channels, kernel size K, and input length T, the number of FLOPs is approximately $2 \cdot T \cdot C_{in} \cdot C_{out} \cdot K$. For each residual block (two Conv1D layers with $K = 3$, $C_{in} = C_{out} = 64$, $T = 24$), the convolutional FLOP count is $2 \times 2 \times 24 \times 64 \times 64 \times 3 = 1{,}179{,}648$ FLOPs per block. With 6 blocks, the total convolutional FLOPs are approximately 7.08M.

Dense Layers: The dense layers contribute $2 \times (64 \times 128 + 128 \times 64 + 64 \times 1) \approx 33\text{K}$ FLOPs.

Total: The total computational cost per inference is approximately 7.1M FLOPs, which is extremely lightweight by deep learning standards. For reference, a standard ResNet-18 image classification model requires approximately 1.8G FLOPs per inference—250× more than the TCN_InSDN model. This low computational cost enables real-time inference at thousands of flows per second on modern hardware.

5.7.3 Memory Footprint

The model's memory footprint during inference consists of:

- **Model Parameters:** 156,737 parameters × 4 bytes (float32) ≈ 612 KB
- **Activations (per sample):** Peak activation memory occurs at the residual blocks, where the feature tensor has shape $(24, 64)$, consuming $24 \times 64 \times 4 = 6.1$ KB per sample.
- **Total Inference Memory:** For a single sample, the total inference memory is approximately 620 KB, which is well within the capacity of any modern computing platform.

This compact memory footprint enables deployment on resource-constrained SDN controller platforms without competing for memory with the controller application's own data structures and processing requirements.

5.8 Weight Initialization

Proper weight initialization is critical for effective training of deep neural networks [88]. The TCN model uses the following initialization schemes:

- **Convolutional Layers:** Glorot (Xavier) uniform initialization, which draws weights from a uniform distribution $\mathcal{U}(-\sqrt{6/(f_{in} + f_{out})}, \sqrt{6/(f_{in} + f_{out})})$, where f_{in} and f_{out} are the fan-in and fan-out of the convolutional kernel. This initialization ensures that the variance of activations is preserved across layers, preventing both vanishing and exploding activations at initialization.
- **Dense Layers:** Glorot uniform initialization, consistent with the convolutional layers.

- **Batch Normalization:** The scale parameter γ is initialized to 1 and the shift parameter β is initialized to 0, which makes the initial batch normalization an identity transformation (after mean and variance normalization). The running mean is initialized to 0 and the running variance to 1.
- **Bias Terms:** All bias terms are initialized to 0.

5.9 Chapter Summary

This chapter has walked through every layer of the `TCN_InSDN` model, from the first dilated convolution to the final sigmoid output. The key takeaway is that the architecture is deliberately compact: 156,737 parameters, 612 KB, roughly 7.1M FLOPs per inference, and convergence in about 30 epochs with Adam plus a cosine-style learning-rate anneal starting at 10^{-3}. These numbers translate directly into the real-time, low-overhead deployment profile that an SDN controller needs. The experiments that put this architecture to the test come next, in Chapter 8.

Chapter 6

Proposed Methodology: TCN-HMAC Framework

The previous two chapters described what goes *into* the system—the InSDN dataset and the TCN model. This chapter describes how those pieces, together with an HMAC-based communication-integrity layer, are woven into a single, deployable security framework. We cover the system architecture, the TCN-IDS module, the HMAC key establishment and message-authentication protocol, the flow-rule verification mechanism, the challenge–response authentication scheme, and the end-to-end operational workflow that ties everything together.

6.1 System Architecture Overview

At the highest level, TCN-HMAC sits as an integrated security layer inside the standard three-plane SDN architecture. It defends two distinct boundaries with two complementary mechanisms:

1. **TCN-Based Intrusion Detection:** A trained TCN model deployed at the SDN controller analyses incoming network flows in real time, classifying each flow as benign or malicious. Malicious flows trigger automated response actions including flow rule installation to block the offending traffic.
2. **HMAC-Based Communication Integrity:** An HMAC scheme secures the communication channel between the SDN controller and the OpenFlow switches, preventing man-in-the-middle attacks, flow rule tampering, and unauthorized controller impersonation.

The architecture positions these mechanisms at two distinct security boundaries:

- **Data Plane → Control Plane Boundary:** The TCN-IDS inspects traffic flows forwarded from switches to the controller via `Packet_In` messages. This boundary is the primary attack surface for network intrusions, as all unknown traffic must pass through the controller for routing decisions.
- **Control Plane ↔ Data Plane Boundary:** The HMAC mechanism secures bidirectional communication between the controller and switches, protecting `Flow_Mod` (controller → switch) and `Stats_Reply` (switch → controller) messages from tampering or forgery.

The net effect is defense in depth. If an attacker crafts traffic that slips past the TCN detector, the HMAC mechanism still prevents that traffic from being leveraged to install rogue flow rules. Conversely, if an attacker somehow compromises the HMAC keys, the TCN-IDS will spot the resulting malicious traffic patterns. Neither layer alone is sufficient; together, they raise the cost of a successful attack substantially.

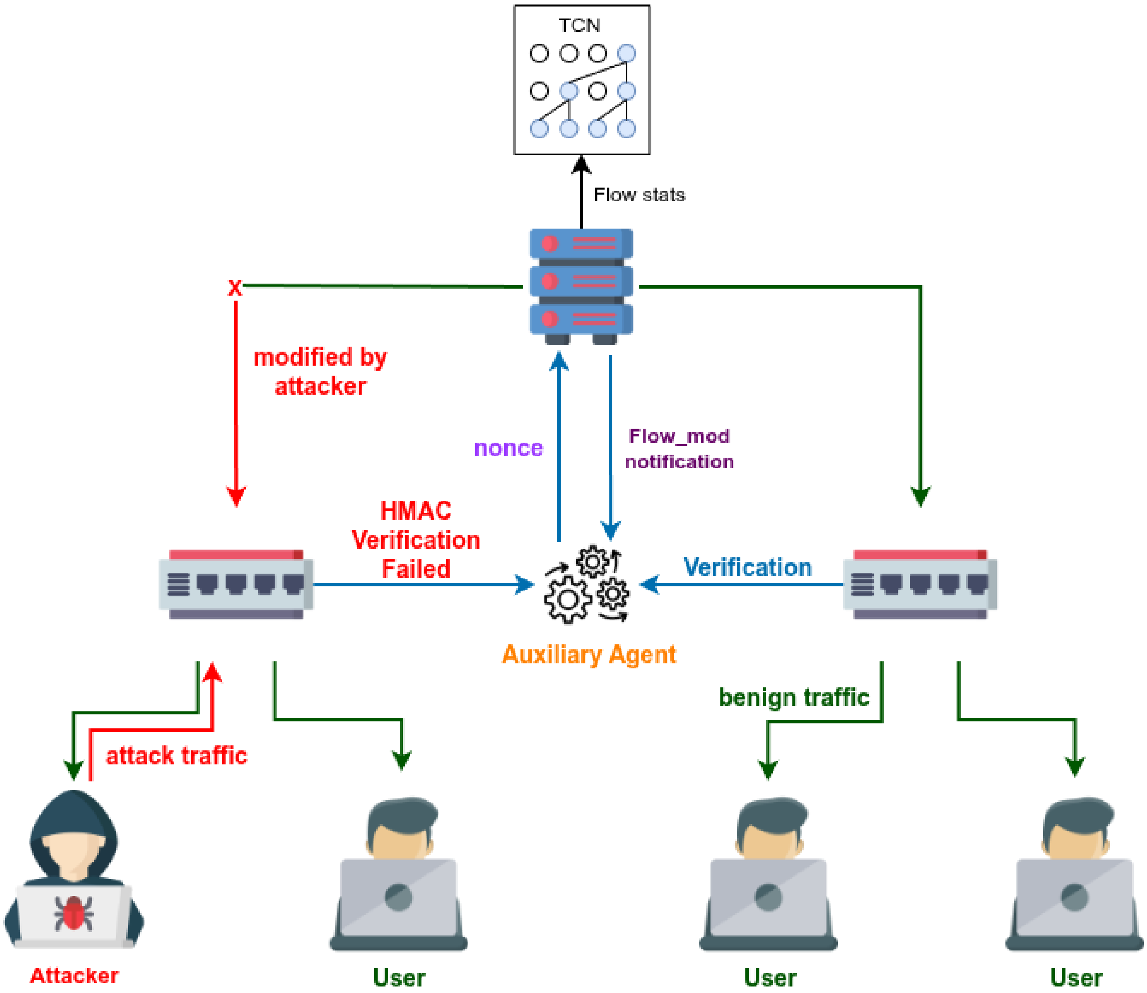

Figure 6.1: Architecture of the proposed TCN-HMAC hybrid security framework for SDN

The overall system architecture is illustrated in Figure 6.1. The TCN-based intrusion detection module is integrated into the SDN controller as a Python-based application, continuously monitoring flow statistics obtained via the northbound API. The Auxiliary Agent, operating as a sidecar process on the same host as the controller, remains isolated from external network access to minimize exposure. Communication between the controller and the agent is secured using symmetric encryption, with shared secrets established via a one-time RSA-based key exchange.

6.2 TCN-Based Intrusion Detection Module

6.2.1 Deployment Architecture

The TCN-IDS module is deployed as a dedicated application on the SDN controller, running alongside the standard networking applications (routing, topology discovery, load balancing). The module interfaces with the controller through the northbound API, receiving flow statistics and packet headers from the controller and returning classification decisions.

The deployment architecture comprises three sub-components:

1. **Feature Extraction Engine:** Extracts the relevant network flow features from incoming `Packet_In` messages and flow statistics. The features correspond to the 84 original InSDN dataset features, which are then preprocessed (as described in Chapter 4) to produce the 24-dimensional PCA-transformed input vector.
2. **TCN Inference Engine:** Loads the trained TCN model (`best_tcn_insdn.keras`) and performs inference on the preprocessed feature vector. The inference engine supports both CPU and GPU execution, with GPU execution recommended for high-throughput deployments.
3. **Response Engine:** Translates TCN classification decisions into SDN control actions. For flows classified as attacks ($p > 0.5$), the response engine generates appropriate `Flow_Mod` messages to block the malicious traffic at the source switch. For flows classified as benign, the response engine allows standard forwarding to proceed.

6.2.2 Real-Time Inference Pipeline

The real-time inference pipeline processes incoming network flows through the following stages:

1. **Flow Capture:** When a switch receives a packet that does not match any existing flow rule, it forwards the packet to the controller via a `Packet_In` message. The controller's flow table management module extracts the flow's header fields and statistics.

2. **Feature Construction:** The feature extraction engine constructs a feature vector $\mathbf{x} \in \mathbb{R}^{84}$ from the captured flow information. This includes bidirectional byte and packet counts, inter-arrival times, TCP flag distributions, and application-layer protocol indicators.

3. **Preprocessing:** The feature vector undergoes the same preprocessing pipeline used during training (Section 4.3.1): infinite value replacement, zero-variance feature removal, standard scaling using the saved scaler parameters, and PCA transformation using the saved PCA basis. The result is a 24-dimensional vector $\mathbf{z} \in \mathbb{R}^{24}$.

4. **Inference:** The preprocessed vector is reshaped to $(1, 24, 1)$ and fed through the TCN model. The sigmoid output $\hat{y} \in (0, 1)$ represents the attack probability.

5. **Decision:** If $\hat{y} > 0.5$, the flow is classified as an attack and the response engine is triggered. If $\hat{y} \leq 0.5$, the flow is classified as benign and normal forwarding proceeds.

6. **Logging:** All classification decisions, timestamps, feature vectors, and confidence scores are logged for offline analysis and model retraining.

6.2.3 Response Actions

When the TCN model classifies a flow as malicious, the response engine can execute one or more of the following automated actions:

- **Flow Blocking:** Install a drop rule on the source switch to discard all packets matching the malicious flow's header fields (source/destination IP, port, protocol). This immediately stops the attack traffic from reaching its target.

- **Rate Limiting:** For suspected DDoS traffic, install a meter entry on the switch to limit the rate of matching packets, reducing the impact of the attack while allowing some legitimate traffic through.

- **Traffic Redirection:** Redirect suspicious traffic to a honeypot or analysis server for further investigation, enabling security analysts to study the attack in detail without exposing production services.

- **Alert Generation:** Generate security alerts to the network administrator's dashboard, including the attack type probability, affected hosts, and recommended manual actions.

6.3 HMAC-Based Communication Integrity

6.3.1 Threat Model

The HMAC side of the framework exists to solve a specific problem: even if TLS encrypts the OpenFlow channel, it does not stop an insider or a compromised component from issuing perfectly well-formed but unauthorised commands. The threat model we consider includes five attack classes:

- **Man-in-the-Middle (MitM) Attacks:** An attacker positioned between the controller and a switch intercepts OpenFlow messages, modifies their contents (e.g., changing flow rule actions from "forward to port 2" to "forward to attacker's port"), and forwards the modified messages to the destination.
- **Flow Rule Injection:** An attacker that has gained access to the network sends forged `Flow_Mod` messages to a switch, installing malicious flow rules that redirect, copy, or drop traffic.
- **Controller Spoofing:** An attacker impersonates the SDN controller, sending commands to switches that appear to originate from the legitimate controller.
- **Replay Attacks:** An attacker captures legitimate OpenFlow messages and replays them at a later time to re-install previously valid (but now inappropriate) flow rules or to trigger unnecessary controller actions.
- **Switch Impersonation:** An attacker impersonates a legitimate switch, sending fabricated statistics or topology information to the controller to manipulate routing decisions.

6.3.2 HMAC Key Establishment Protocol

Before HMAC-protected communication can begin, the controller and each switch must establish a shared secret key. The TCN-HMAC framework employs a key establishment protocol that operates during the switch connection phase:

Initial Key Exchange:

1. When a switch connects to the controller via the OpenFlow handshake, the controller generates a random 256-bit master key K_{master} for the switch.

2. The master key is transmitted to the switch over the TLS-encrypted OpenFlow connection. While TLS provides transport-layer encryption, HMAC provides application-layer message integrity as an additional defense layer.

3. Both the controller and the switch derive two session keys from the master key using a Key Derivation Function (KDF):

$$K_{C\to S} = \text{HMAC-SHA256}(K_{master}, \text{“controller-to-switch”}\|\text{nonce}_C) \tag{6.1}$$

$$K_{S\to C} = \text{HMAC-SHA256}(K_{master}, \text{“switch-to-controller”}\|\text{nonce}_S) \tag{6.2}$$

   where nonce_C and nonce_S are random nonces exchanged during the handshake.

4. Separate keys for each direction prevent reflection attacks, where an attacker reflects a message back to its sender.

The secure key exchange procedure is formalized in Algorithm 1.

Algorithm 1: Secure Key Exchange for a New Switch

Input: Agent–Controller shared key K_{ac}, Controller public key PK_c, Controller private key PR_c, Switch ID SW_{id}

Output: Shared secret key between controller and agent for a new switch

```
// Agent Side
$K_{sw} \leftarrow$ Generate random symmetric key for the switch;
$payload \leftarrow SW_{id} \| K_{sw}$;
$auth\_tag \leftarrow \text{HMAC}_{K_{ac}}(payload)$;
$message \leftarrow payload \| auth\_tag$;
$encrypted \leftarrow \text{Encrypt}_{PK_c}(message)$;
Send $encrypted$ to Controller;
// Controller Side
$decrypted \leftarrow \text{Decrypt}_{PR_c}(encrypted)$;
Extract $payload$ and $auth\_tag$ from $decrypted$;
if $HMAC_{K_{ac}}(payload) \neq auth\_tag$ then
    Reject message;
else
    Extract $SW_{id}$ and $K_{sw}$ from $payload$;
    Store $K_{sw}$ associated with $SW_{id}$;
```

Key Rotation: To limit the window of vulnerability if a key is compromised, the framework implements periodic key rotation:

- Keys are rotated every T_{rotate} seconds (configurable, default 3600 seconds).

- The rotation uses forward secrecy: the new master key is derived from the current master key and a fresh random value:

$$K_{master}^{(t+1)} = \text{HMAC-SHA256}(K_{master}^{(t)}, r_t) \tag{6.3}$$

  where r_t is a random 256-bit value. This ensures that compromise of a future key does not compromise past keys.

- Key rotation is synchronized via a dedicated OpenFlow vendor message.

6.3.3 Message Authentication

Every OpenFlow message exchanged between the controller and a switch is augmented with an HMAC tag that authenticates the message's integrity and origin:

Sending (Controller → Switch):

1. The controller constructs the OpenFlow message M (e.g., a `Flow_Mod` message).
2. A sequence number seq is appended to prevent replay attacks.
3. The HMAC tag is computed using HMAC-SHA256:

$$\text{tag} = \text{HMAC-SHA256}(K_{C\to S}, M\|\text{seq}\|\text{timestamp}) \tag{6.4}$$

4. The message (M||seq||timestamp||tag) is sent to the switch.

Receiving (Switch):

1. The switch receives the augmented message and extracts M, seq, timestamp, and tag.
2. The switch recomputes the expected HMAC tag using its copy of $K_{C\to S}$:

$$\text{tag}' = \text{HMAC-SHA256}(K_{C\to S}, M\|\text{seq}\|\text{timestamp}) \tag{6.5}$$

3. The switch performs a constant-time comparison of tag and tag′. Constant-time comparison prevents timing side-channel attacks that could be used to forge valid tags.
4. If tag = tag′ and seq is greater than the last accepted sequence number and timestamp is within the acceptable window, the message is accepted and processed.
5. Otherwise, the message is rejected, a security alert is generated, and the invalid message is logged for forensic analysis.

The same process operates in reverse for switch-to-controller messages, using $K_{S\to C}$.

6.3.4 Flow Rule Verification

In addition to authenticating individual messages, the TCN-HMAC framework implements a flow rule verification mechanism to detect unauthorized modifications to the switch's flow table:

1. The controller maintains a shadow copy of each switch's expected flow table, stored in a secure database on the controller.
2. Periodically (every T_{verify} seconds, default 30), the controller sends a `Flow_Stats_Request` to each switch, requesting the current flow table contents.
3. The switch responds with a `Flow_Stats_Reply` containing its current flow rules, authenticated with an HMAC tag.
4. The controller compares the received flow rules against its shadow copy. Any discrepancy (additional rules not installed by the controller, modified rules, or missing rules) indicates a potential compromise and triggers an investigation:
   - **Extra Rules:** May indicate flow rule injection by an attacker.
   - **Modified Rules:** May indicate MitM modification of `Flow_Mod` messages.
   - **Missing Rules:** May indicate flow rule deletion by an attacker or a legitimate timeout.
5. If unauthorized modifications are detected, the controller re-installs the correct flow rules and generates a security alert.

The flow rule verification procedure is formalized in Algorithm 2.

Algorithm 2: Flow Rule Verification by the Auxiliary Agent

```
Input: Flow rule $F$, received cookie $C$, switch ID $SW_{id}$
Output: Validate and enforce flow integrity
Parse flow: extract $eth\_src$, $eth\_dst$, $out\_port$;
$flow\_str \leftarrow eth\_src \,\|\, eth\_dst \,\|\, output{:}out\_port$;
$expected\_cookie \leftarrow \mathrm{HMAC}_{K_{sw}}(flow\_str)$;
if $C \neq expected\_cookie$ then
    Drop flow rule from switch;
    Notify controller of invalid flow;
else
    Accept and monitor flow;
```

6.3.5 Challenge-Response Authentication

To detect controller spoofing and switch impersonation attacks, the framework implements a challenge-response authentication mechanism:

Controller Authenticates Switch:

1. The controller generates a random challenge nonce $c_s \in \{0,1\}^{128}$.
2. The controller sends c_s to the switch.
3. The switch computes the response $r_s = \text{HMAC-SHA256}(K_{S \to C}, c_s \| \text{switch\_id})$ and returns r_s.
4. The controller verifies r_s by recomputing the expected response. If verification fails, the switch connection is terminated and an alert is generated.

Switch Authenticates Controller:

1. The switch generates a random challenge nonce $c_c \in \{0,1\}^{128}$.
2. The switch sends c_c to the controller (via an OpenFlow vendor message).
3. The controller computes the response $r_c = \text{HMAC-SHA256}(K_{C \to S}, c_c \| \text{controller\_id})$ and returns r_c.
4. The switch verifies r_c. If verification fails, the switch disconnects from the controller and enters a safe mode where it continues forwarding based on its last known good flow table until a verified controller is available.

Challenge-response authentication is performed during the initial handshake and can be repeated periodically (every T_{auth} seconds) or triggered on demand when suspicious activity is detected.

The controller verification via challenge–response is formalized in Algorithm 3.

6.4 End-to-End Operational Workflow

The complete operational workflow of the TCN-HMAC framework integrates the TCN-IDS and HMAC components into a unified security pipeline:

Algorithm 3: Controller Verification via Challenge–Response

```
Input: Interval $T$, controller key $K_{sw}$, backup controller $C_{bk}$
Output: Detect and recover from compromised controller
for every $T$ seconds do
    $nonce \leftarrow$ Generate random challenge;
    Send $nonce$ to controller;
    Controller responds with $response = \text{HMAC}_{K_{sw}}(nonce)$;
    if $HMAC_{K_{sw}}(nonce) \neq response$ then
        Alert network administrator;
        Activate $C_{bk}$ as primary controller;
    else
        Continue monitoring;
```

6.4.1 Network Initialization Phase

1. **Controller Startup:** The SDN controller starts and loads the TCN model, the preprocessing pipeline (scaler parameters, PCA basis), and the HMAC key management module.

2. **Switch Connection:** Switches connect to the controller via the OpenFlow handshake. The TLS connection is established, and the HMAC key exchange protocol is executed.

3. **Mutual Authentication:** The controller and each switch perform the challenge-response authentication to verify each other's identity.

4. **Flow Rule Installation:** The controller installs initial flow rules (e.g., default routing, table-miss rules) on each switch, with each `Flow_Mod` message authenticated via HMAC.

5. **Shadow Table Initialization:** The controller initializes the shadow flow table for each switch, recording the initial flow rules.

6.4.2 Steady-State Operation Phase

During normal operation, the framework processes network traffic through the following workflow:

1. **Packet Arrival:** A packet arrives at a switch. The switch checks its flow table for a matching rule.

2. **Known Flow:** If a matching rule exists, the packet is processed according to the rule's actions (forward, drop, modify) without controller involvement.

3. **Unknown Flow:** If no matching rule exists, the switch sends a `Packet_In` message to the controller (authenticated via HMAC).

4. **TCN Classification:** The controller's TCN-IDS module extracts features from the flow, preprocesses them, and runs TCN inference.

5. **Benign Flow:** If the flow is classified as benign, the controller computes the appropriate forwarding path and installs matching flow rules on the relevant switches via HMAC-authenticated `Flow_Mod` messages.

6. **Malicious Flow:** If the flow is classified as an attack, the controller installs drop rules on the source switch and generates security alerts. The drop rules are also authenticated via HMAC.

7. **Shadow Table Update:** The controller updates its shadow flow table to reflect the newly installed rules.

6.4.3 Periodic Security Tasks

In addition to the real-time traffic processing workflow, the framework executes periodic security maintenance tasks:

- **Flow Table Verification** (every 30 seconds): The controller queries each switch's flow table and compares it against the shadow copy, as described in Section 6.3.4.

- **Key Rotation** (every 3600 seconds): The HMAC session keys are rotated to limit the exposure window of compromised keys.

- **Re-Authentication** (every 1800 seconds): The controller and switches perform mutual challenge-response authentication to detect persistent compromise.

- **Model Health Check:** The TCN model's inference latency and memory usage are monitored. If latency exceeds a threshold (e.g., 5 ms per flow), the system alerts the administrator.

6.5 Security Analysis

This section analyzes the security properties of the TCN-HMAC framework against the threats identified in Section 6.3.1.

6.5.1 Resistance to Network Intrusions

The TCN-IDS component provides resistance to network intrusions through:

- **High Detection Rate:** The TCN model achieves a detection rate of 99.97% (23,737 out of 23,744 attack flows correctly identified), meaning that only 0.03% of attack flows evade detection.
- **Low False Alarm Rate:** The false alarm rate is 0.37% (47 out of 12,823 benign flows incorrectly classified), minimizing disruption to legitimate traffic.
- **Multi-Attack Coverage:** The binary classification approach detects a wide range of attack types (DoS, DDoS, probing, brute force, web attacks, botnet, infiltration) through learned feature representations, rather than relying on attack-specific signatures.
- **Real-Time Processing:** The lightweight TCN model (7.1M FLOPs per inference) enables classification within milliseconds, allowing the system to respond to attacks before significant damage occurs.

6.5.2 Resistance to Man-in-the-Middle Attacks

The HMAC mechanism provides resistance to MitM attacks through:

- **Message Integrity:** Any modification to an HMAC-protected message invalidates the HMAC tag. An attacker who modifies a `Flow_Mod` message in transit will cause the receiving switch to reject the message.
- **Cryptographic Strength:** HMAC-SHA256 provides 256-bit security, making brute-force tag forgery computationally infeasible (2^{128} expected operations for a birthday attack on the 256-bit output).
- **Sequence Numbers:** The inclusion of sequence numbers prevents replay attacks, where an attacker reuses a previously valid message.
- **Timestamps:** The inclusion of timestamps provides an additional defense against replay attacks with temporal freshness guarantees.

6.5.3 Resistance to Controller/Switch Spoofing

The challenge-response authentication mechanism prevents spoofing attacks:

- **Controller Verification:** Switches verify the controller's identity through the challenge-response protocol, rejecting commands from unauthorized controllers.

- **Switch Verification:** The controller verifies each switch's identity, preventing rogue switches from injecting false topology or traffic information.

- **Key-Based Authentication:** The challenge-response responses require knowledge of the shared secret keys, which are transmitted only over TLS-encrypted connections and rotated periodically.

6.5.4 Resistance to Flow Rule Tampering

The flow rule verification mechanism detects unauthorized modifications to switch flow tables:

- **Shadow Table Comparison:** The controller's shadow flow table provides a ground truth against which the switch's actual flow table is compared.

- **Periodic Verification:** Regular verification ensures that tampering is detected within a bounded time window (T_{verify} seconds).

- **Automated Remediation:** Detected discrepancies are automatically corrected by re-installing the correct flow rules.

6.5.5 HMAC Overhead Analysis

No security framework is worth deploying if it adds unacceptable overhead. HMAC-SHA256 is deliberately chosen because it is fast and frugal [7]:

- **Computation Time:** HMAC-SHA256 computation requires approximately 1–5 microseconds per message on modern processors, adding negligible latency to the OpenFlow message processing pipeline.

- **Bandwidth Overhead:** Each HMAC tag adds 32 bytes (256 bits) to the message. For a typical `Flow_Mod` message of 64–128 bytes, this represents a 25–50% size increase. However, since OpenFlow control messages are exchanged at rates of hundreds to thousands per second (not millions), this bandwidth overhead is insignificant relative to the data plane throughput.

- **Key Storage:** Each switch requires storage for two 256-bit session keys (64 bytes total), which is trivial even for the most resource-constrained switches.

- **Scalability:** In a network with N switches, the controller maintains N master keys and $2N$ session keys. For networks with up to 10,000 switches, the total key storage is approximately 1.9 MB, well within the controller's memory capacity.

6.6 Comparison with Alternative Approaches

What makes TCN-HMAC distinctive is not any single component—many IDS papers exist, and HMAC is textbook cryptography—but the combination. Table 6.1 shows how the framework stacks up against the main alternative strategies:

Table 6.1: Comparison of SDN Security Approaches

Approach	IDS	Comm. Auth.	Real-Time	Overhead	Defense Depth
Signature IDS [2]	Yes	No	Limited	Low	Single
ML-based IDS [89]	Yes	No	Moderate	Moderate	Single
TLS only	No	Partial	Yes	Low	Single
Blockchain [52]	Partial	Yes	No	High	Moderate
TCN-HMAC (ours)	Yes	Yes	Yes	Low	Dual

The TCN-HMAC framework is unique in providing both intrusion detection and communication authentication with low overhead and real-time operation. Signature-based and ML-based IDS approaches provide detection but no communication integrity. TLS provides transport-layer encryption but not application-layer message authentication. Blockchain-based approaches provide authentication but with high computational and latency overhead that makes real-time operation impractical.

6.7 Design Decisions and Trade-offs

Every framework involves trade-offs. The ones we made—and the reasoning behind them—are worth spelling out explicitly:

1. **Binary vs. Multi-Class Classification:** The TCN-IDS uses binary classification (benign vs. attack) rather than multi-class classification (identifying specific attack types). This decision was made for several reasons: (a) binary classification achieves higher

accuracy than multi-class on the InSDN dataset; (b) the primary operational requirement is to detect and block attacks, not to identify their specific types; (c) binary classification produces a simpler and faster model; and (d) attack type identification can be performed offline on flagged traffic if needed.

2. **HMAC-SHA256 vs. Digital Signatures:** HMAC was chosen over digital signatures (e.g., RSA, ECDSA) for communication authentication because: (a) HMAC computation is 100–1000× faster than signature generation/verification; (b) the SDN environment provides a natural mechanism for shared key distribution (the TLS-encrypted controller–switch connection); and (c) HMAC provides sufficient security when both parties are mutually authenticated (which is ensured by the challenge-response protocol).

3. **TCN vs. Other Deep Learning Architectures:** TCN was chosen over LSTM, GRU, and Transformer architectures based on the analysis presented in Section 2.2.7: TCN provides comparable or superior accuracy with lower computational cost, parallelizable inference, deterministic gradient flow, and a smaller memory footprint.

4. **PCA Dimensionality Reduction:** PCA was used to reduce the feature space from 48 to 24 dimensions, balancing model performance with inference speed. While the full 48-dimensional features could potentially provide marginally higher accuracy, the 24-dimensional PCA features retain 95.43% of the variance and enable faster preprocessing and inference.

5. **Layered Security:** The decision to combine IDS with communication authentication (rather than implementing either one in isolation) was motivated by the observation that single-layer security mechanisms can be bypassed: an IDS without communication integrity is vulnerable to flow rule tampering, and communication integrity without IDS is unable to detect application-layer attacks.

6.8 Chapter Summary

This chapter has laid out the full design of the TCN-HMAC framework: a TCN-based IDS for real-time detection paired with an HMAC-based auxiliary agent for flow-rule verification and challenge–response controller authentication. Key design choices—binary classification, HMAC-SHA256 over digital signatures, PCA-compressed features, layered security—were driven by the need for low-overhead, real-time operation in an SDN control loop. The security analysis shows that the framework defends against network intrusions, man-in-the-middle attacks, controller/switch spoofing, and flow-rule tampering with minimal computational and bandwidth cost. The next two chapters put these claims to the test:

Chapter 8 reports the experimental results for the TCN-IDS, and Chapter 9 benchmarks it against fifteen existing models.

Chapter 7

Experimental Setup

Results are only as trustworthy as the setup that produces them. This chapter spells out every detail of the experimental environment—hardware, software, data splits, evaluation metrics, baselines, and reproducibility measures—so that readers can judge the results in context and, if they wish, replicate them.

7.1 Computing Platform

All experiments ran on Google Colaboratory (Colab), which provision an NVIDIA Tesla T4 GPU alongside a two-core Intel Xeon CPU. The hardware details are listed in Table 7.1.

Table 7.1: Hardware Configuration

Component	**Specification**
GPU	NVIDIA Tesla T4 (16 GB GDDR6, 2560 CUDA cores)
CPU	Intel Xeon (2 cores, 2.2 GHz)
RAM	12.7 GB system memory
Storage	107 GB disk space
GPU Architecture	Turing (Compute Capability 7.5)
GPU Memory Bandwidth	320 GB/s
Tensor Cores	320 (FP16/INT8 acceleration)

The NVIDIA T4 GPU, based on the Turing architecture, provides efficient mixed-precision training capabilities through its 320 Tensor Cores. While the experiments used full 32-bit floating-point precision for maximum accuracy, the T4's Tensor Cores could be leveraged for future deployment optimization using FP16 or INT8 quantization. The T4 is a data-center class GPU commonly used in production inference environments, making it a representative platform for evaluating the model's deployment feasibility.

7.2 Software Environment

The software environment comprises the deep learning framework, data processing libraries, and visualization tools. Table 7.2 lists the complete software stack:

Table 7.2: Software Environment

Category	Library	Version
Deep Learning	TensorFlow	2.19.0
Deep Learning	Keras	Integrated in TF 2.19
Data Processing	NumPy	1.26.x
Data Processing	Pandas	2.2.x
Machine Learning	scikit-learn	1.5.x
Dimensionality Reduction	scikit-learn (PCA)	1.5.x
Visualization	Matplotlib	3.9.x
Visualization	Seaborn	0.13.x
Runtime	Python	3.10
GPU Support	CUDA	12.2
GPU Support	cuDNN	8.9

TensorFlow 2.19.0 was selected as the primary deep learning framework for several reasons: (a) it provides a well-optimized Conv1D implementation for one-dimensional temporal convolutions, which is the core operation in TCN architectures; (b) TensorFlow's Keras API enables rapid prototyping with clear layer composition; (c) TensorFlow supports deployment through TensorFlow Lite and TensorFlow Serving, facilitating the transition from experimental model to production deployment; and (d) TensorFlow's GPU acceleration via CUDA and cuDNN provides efficient training on the T4 hardware.

7.3 Dataset Preparation Summary

The complete dataset preprocessing pipeline was described in Chapter 4. This section provides a summary of the key aspects relevant to the experimental setup:

7.3.1 Data Splitting Strategy

The 182,831 deduplicated samples were split into three disjoint subsets using stratified splitting to preserve the class distribution:

The three-way split serves distinct purposes:

- **Training Set (70%):** Used to update the model's weights via backpropagation. The

Table 7.3: Dataset Summary After Preprocessing

Property	Value
Source Dataset	InSDN [18]
Raw Samples	343,889
After Deduplication	182,831
Original Features	84
After Cleaning	48
After PCA	24
PCA Variance Retained	95.43%
Binary Classes	Benign (0), Attack (1)

Table 7.4: Data Split Distribution

Split	Ratio	Total	Benign	Attack	Attack %
Training	70%	127,981	44,849	83,132	64.96%
Validation	10%	18,283	6,407	11,876	64.96%
Test	20%	36,567	12,823	23,744	64.93%
Total	100%	182,831	64,079	118,752	64.95%

model sees this data repeatedly across epochs.

- **Validation Set (10%):** Used to monitor training progress, tune hyperparameters, and make early stopping decisions. The model never trains on this data, but it influences the training process through callbacks.

- **Test Set (20%):** Used exclusively for final model evaluation after training is complete. The model never sees this data during training or validation, ensuring an unbiased estimate of generalization performance.

Stratified splitting ensures that each subset maintains the same class distribution as the full dataset (≈65% attack, ≈35% benign), preventing biased evaluation due to class imbalance differences between subsets.

7.3.2 Class Weighting

To address the class imbalance (64.95% attack, 35.05% benign), class weights were computed using inverse frequency weighting:

$$w_c = \frac{N}{C \times n_c} \tag{7.1}$$

where N is the total number of training samples, $C = 2$ is the number of classes, and n_c is the number of training samples in class c. The resulting weights are:

- w_0 (Benign): $\frac{127{,}981}{2\times 44{,}849} = 1.4258$
- w_1 (Attack): $\frac{127{,}981}{2\times 83{,}132} = 0.7700$

These weights are applied in the loss function during training, effectively oversampling the minority class (benign) and undersampling the majority class (attack) in the gradient computation.

7.4 Input Representation

The TCN model expects input tensors of shape (N, T, C) where N is the batch size, T is the sequence length (temporal dimension), and C is the number of input channels. The 24 PCA-transformed features are treated as a temporal sequence with one channel:

$$\mathbf{X} \in \mathbb{R}^{N\times 24\times 1} \tag{7.2}$$

This representation treats each PCA component as a time step, allowing the TCN's causal convolutions to capture relationships between sequential PCA components. While PCA components are not inherently temporal, the ordering by explained variance ratio creates a meaningful sequence from the most important to the least important features, and the TCN's causal structure captures hierarchical dependencies within this ordering.

7.5 Evaluation Metrics

The model's performance is evaluated using a comprehensive suite of binary classification metrics, each capturing a different aspect of the model's behavior:

7.5.1 Primary Metrics

Accuracy: The proportion of correctly classified samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{7.3}$$

where TP (True Positives) are attacks correctly classified as attacks, TN (True Negatives) are benign samples correctly classified as benign, FP (False Positives) are benign samples incorrectly classified as attacks, and FN (False Negatives) are attacks incorrectly classified as benign.

Precision: The proportion of predicted attacks that are actual attacks:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7.4}$$

High precision indicates few false alarms, which is important for operational deployability—a system with many false alarms wastes the administrator's time and may lead to alarm fatigue.

Recall (Sensitivity, Detection Rate): The proportion of actual attacks that are correctly detected:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7.5}$$

High recall is the most critical metric for an intrusion detection system, as missed attacks (false negatives) can have severe security consequences.

F1-Score: The harmonic mean of precision and recall:

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{7.6}$$

The F1-score provides a balanced measure that penalizes both false positives and false negatives, making it suitable for evaluating performance on imbalanced datasets.

7.5.2 Threshold-Independent Metrics

AUC-ROC (Area Under the Receiver Operating Characteristic Curve): The ROC curve plots the True Positive Rate (TPR = Recall) against the False Positive Rate ($FPR = \frac{FP}{FP+TN}$) for all possible classification thresholds $\tau \in [0, 1]$. The AUC is the area under this curve:

$$\text{AUC} = \int_0^1 \text{TPR}(FPR^{-1}(t))\, dt \tag{7.7}$$

An AUC of 1.0 indicates perfect classification at all thresholds, while an AUC of 0.5 indicates random classification. The AUC is threshold-independent, making it a robust measure of the model's discriminative ability regardless of the specific decision threshold used.

7.5.3 Security-Specific Metrics

Detection Rate (DR): Equivalent to recall, measuring the proportion of attacks detected:

$$DR = \frac{TP}{TP + FN} \tag{7.8}$$

False Alarm Rate (FAR): The proportion of benign flows incorrectly classified as attacks:

$$FAR = \frac{FP}{FP + TN} \tag{7.9}$$

A low FAR is essential for operational IDS systems because false alarms consume response resources, trigger unnecessary blocking actions, and reduce trust in the detection system.

Specificity (True Negative Rate): The proportion of benign flows correctly classified as benign:

$$\text{Specificity} = \frac{TN}{TN + FP} = 1 - FAR \tag{7.10}$$

7.5.4 Confusion Matrix

The confusion matrix provides a complete summary of the model's classification decisions:

$$CM = \begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix} \tag{7.11}$$

Each cell of the confusion matrix corresponds to one of the four possible outcomes of binary classification. The off-diagonal elements (FP and FN) represent errors, while the diagonal elements (TN and TP) represent correct classifications. The confusion matrix is the basis from which all other metrics are derived.

7.6 Baseline Models

To contextualize the TCN model's performance, results are compared against several baseline models from the literature:

These baselines span three categories of classification approaches:

1. **Traditional Machine Learning:** Random Forest and Decision Tree models serve as non-deep-learning baselines, representing the approaches commonly used in earlier SDN intrusion detection work.

Table 7.5: Baseline Models for Comparison

Model	Type	Key Features	Reference
CNN	Deep Learning	1D convolutions, pooling	[90]
LSTM	Deep Learning	Recurrent, gates	[65]
CNN-BiLSTM	Hybrid DL	Conv + bidirectional RNNs	[13]
CNN-LSTM	Hybrid DL	Conv + LSTM sequence	[63]
CNN-GRU	Hybrid DL	Conv + GRU sequence	[64]
Random Forest	Traditional ML	Ensemble of decision trees	[91]
DT	Traditional ML	Single decision tree	[18]

2. **Recurrent Deep Learning:** LSTM models represent the recurrent neural network family, which has been the dominant deep learning approach for sequential data in intrusion detection.

3. **Hybrid Deep Learning:** CNN-BiLSTM, CNN-LSTM, and CNN-GRU models represent the trend of combining convolutional feature extraction with recurrent sequence modeling, which has shown strong performance in recent IDS literature.

7.7 Reproducibility

Claiming good performance is one thing; letting others verify it is another. We took several steps to make the results reproducible:

- **Random Seed:** A fixed random seed of 42 was used for all random number generators (Python's `random`, NumPy, TensorFlow) to ensure deterministic data splitting, weight initialization, and dropout behavior.

- **Data Versioning:** The exact preprocessed dataset files (after deduplication, scaling, and PCA transformation) are saved as pickled objects, enabling exact reproduction of the input data.

- **Model Checkpointing:** The best model weights (selected by validation AUC) are saved in Keras format (`best_tcn_insdn.keras`), enabling exact reproduction of the evaluation results.

- **Training Logs:** Complete training histories (loss, accuracy, AUC, precision, recall for both training and validation sets, per epoch) are logged to `training_log.csv`.

- **Configuration Documentation:** All hyperparameter values, preprocessing steps, and software versions are documented in the accompanying documentation file.

7.8 Experimental Procedure

The complete experimental procedure follows these sequential steps:

1. **Data Loading:** Load the five InSDN CSV files and concatenate them into a single DataFrame.
2. **Preprocessing:** Apply the 15-stage preprocessing pipeline described in Chapter 4, producing the cleaned, scaled, PCA-transformed dataset.
3. **Splitting:** Stratified split into 70% training, 10% validation, and 20% test sets.
4. **Reshaping:** Reshape the feature matrices from $(N, 24)$ to $(N, 24, 1)$ for TCN input compatibility.
5. **Model Construction:** Build the TCN_InSDN model architecture as described in Chapter 5.
6. **Compilation:** Compile the model with the Adam optimizer, binary cross-entropy loss, and the evaluation metrics (accuracy, AUC, precision, recall).
7. **Training:** Train the model on the training set with validation monitoring, using the callbacks described in Section 5.6: EarlyStopping (patience=15, monitor=val_auc), ReduceLROnPlateau (patience=7, factor=0.5), ModelCheckpoint (monitor=val_auc), and CSVLogger.
8. **Best Model Loading:** Load the best model weights (by validation AUC) from the checkpoint file.
9. **Evaluation:** Evaluate the best model on the held-out test set, computing accuracy, precision, recall, F1-score, AUC-ROC, and the confusion matrix.
10. **Visualization:** Generate training history plots (loss and accuracy curves), confusion matrix heatmap, ROC curve, and per-class performance bar charts.
11. **Export:** Save the trained model, scaler parameters, PCA basis, and evaluation results for deployment and documentation purposes.

7.9 Chapter Summary

This chapter has pinned down the experimental setting: a T4-equipped Google Colab instance running TensorFlow 2.19.0, a stratified 70/10/20 data split with class weighting, a suite of metrics covering accuracy through AUC-ROC and IDS-specific detection and false-alarm rates, and baselines drawn from traditional ML, recurrent DL, and hybrid DL families. With the stage set, the next chapter presents what the TCN_InSDN model actually achieves.

Chapter 8

Results and Analysis

With the architecture and training setup in place, the obvious question is: how well does the model actually work? This chapter answers that question in detail. We examine training dynamics, test-set metrics, the confusion matrix, per-class breakdowns, ROC-AUC behaviour, HMAC overhead, and a frank discussion of the results' limitations. Unless stated otherwise, every number reported here comes from the held-out test set (20% of the data, 36,567 samples), which the model never saw during training or hyperparameter selection.

8.1 Training Dynamics

8.1.1 Convergence Behavior

Training finished in roughly 30 epochs—well short of the 100-epoch budget. The best validation AUC appeared around epoch 15–20, and early stopping kicked in after 15 further epochs without improvement. Why does the model converge so quickly?

- **Residual Connections:** The skip connections in each residual block provide direct gradient paths from the loss to early layers, enabling efficient training without vanishing gradients.

- **Batch Normalization:** Normalizing intermediate activations stabilizes the internal covariate shift, allowing the use of higher learning rates and accelerating convergence.

- **Adam Optimizer:** The adaptive learning rates of Adam enable efficient navigation of the loss landscape, converging faster than fixed-rate optimizers.

- **PCA-Preprocessed Features:** The PCA transformation decorrelates the input features and orders them by importance, providing the model with a clean, structured input that is easier to learn from.

8.1.2 Training and Validation Loss

The training loss decreases rapidly during the first 5 epochs, dropping from an initial value of approximately 0.35 to below 0.05. The validation loss follows a similar trajectory, closely tracking the training loss with a small gap, indicating that the model generalizes well without significant overfitting.

After epoch 10, both training and validation losses plateau at approximately 0.01–0.02, indicating that the model has converged to a near-optimal solution. The close alignment between training and validation losses confirms that the regularization mechanisms (spatial dropout, dense dropout, class weighting) effectively prevent overfitting despite the model's capacity.

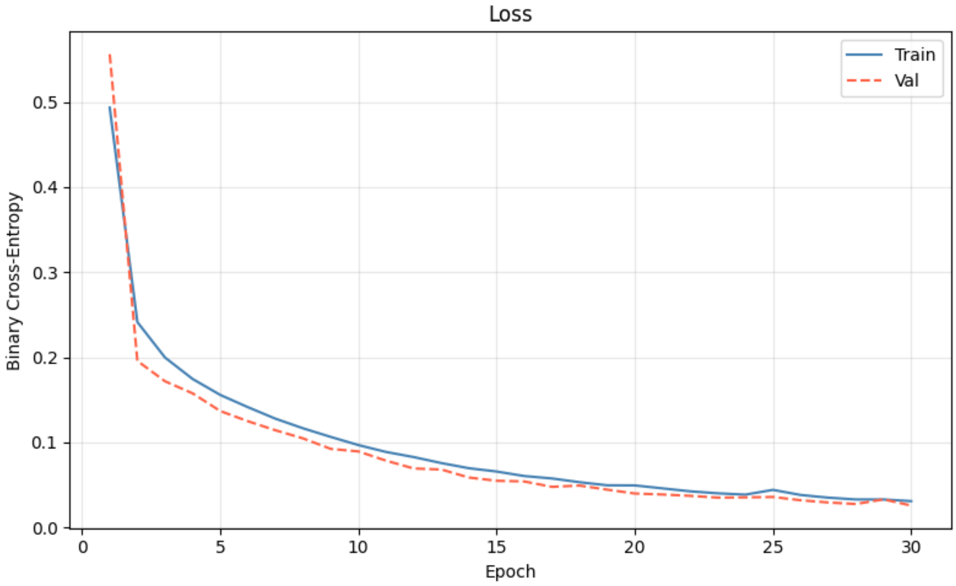

Figure 8.1: Training and validation loss curves over 30 epochs. Both curves converge rapidly within the first 5 epochs and plateau near zero, indicating effective learning without overfitting.

8.1.3 Training and Validation Accuracy

Training accuracy increases rapidly from approximately 85% at epoch 1 to above 99.5% by epoch 5, and stabilizes at approximately 99.8% for the remaining epochs. Validation accuracy follows a similar trajectory, reaching 99.85% at the best epoch. The convergence of training and validation accuracy confirms that the model's high accuracy is not due to overfitting but reflects genuine learning of the underlying classification task.

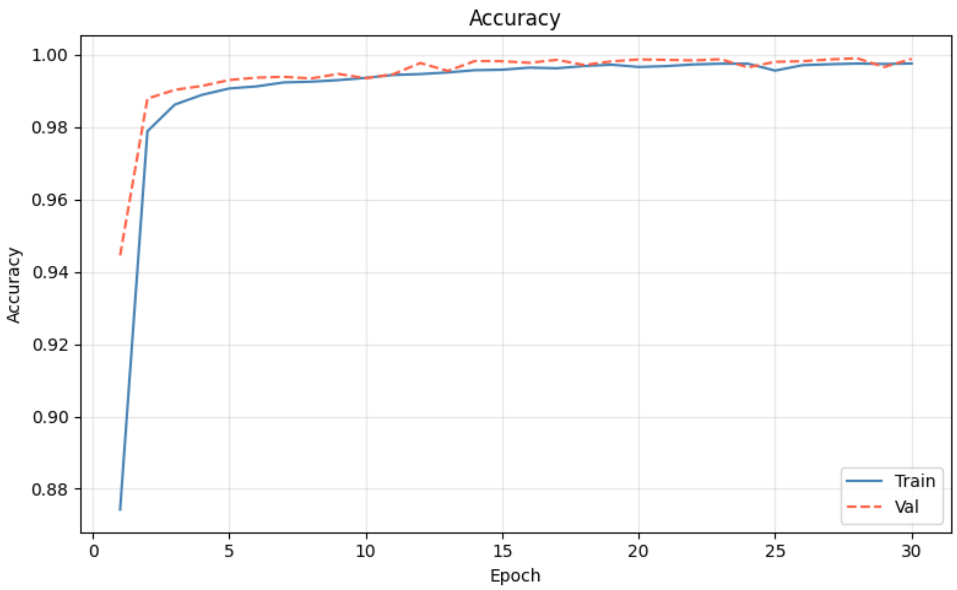

Figure 8.2: Training and validation accuracy curves. Both curves rapidly approach 1.0 within the first 5 epochs, with validation accuracy closely tracking training accuracy throughout.

8.1.4 Learning Rate Schedule

The ReduceLROnPlateau scheduler reduced the learning rate from the initial 10^{-3} when the validation loss plateaued. The learning rate reductions occurred at approximately:

- Epoch ∼12: $10^{-3} \rightarrow 5 \times 10^{-4}$ (first plateau)
- Epoch ∼21: $5 \times 10^{-4} \rightarrow 2.5 \times 10^{-4}$ (second plateau)

These learning rate reductions enabled the model to fine-tune its weights more precisely after the initial rapid convergence phase, contributing to the final performance improvement observed between epochs 12 and 20.

8.2 Test Set Performance

The best model (selected by validation AUC) was evaluated on the held-out test set. Table 8.1 summarizes the overall performance metrics:

The headline numbers tell a clear story: the model rarely makes mistakes, and when it does, it errs on the side of caution.

Table 8.1: TCN_InSDN Test Set Performance

Metric	**Value**
Accuracy	99.85%
Precision	99.80%
Recall (Detection Rate)	99.97%
Specificity	99.63%
F1-Score	99.89%
AUC-ROC	0.9999
False Alarm Rate	0.37%

8.3 Confusion Matrix Analysis

The confusion matrix for the test set is:

Table 8.2: Confusion Matrix on Test Set (36,567 samples)

		Predicted	
		Benign (0)	Attack (1)
Actual	Benign (0)	12,776 (TN)	47 (FP)
	Attack (1)	7 (FN)	23,737 (TP)

8.3.1 True Positives (TP = 23,737)

Of the 23,744 attack samples in the test set, 23,737 (99.97%) were correctly identified as attacks. This exceptionally high true positive rate means that the TCN model misses only 7 out of 23,744 attack flows—a false negative rate of 0.03%. From a security perspective, this means that virtually all attack traffic is detected and can be blocked by the response engine.

8.3.2 True Negatives (TN = 12,776)

Of the 12,823 benign samples in the test set, 12,776 (99.63%) were correctly classified as benign. This high true negative rate ensures that the vast majority of legitimate traffic is not disrupted by false alarms.

8.3.3 False Positives (FP = 47)

Only 47 benign flows were mislabelled as attacks—a false alarm rate of 0.37%. By IDS standards, that is very low; traditional signature-based systems routinely produce 1–5% or more.

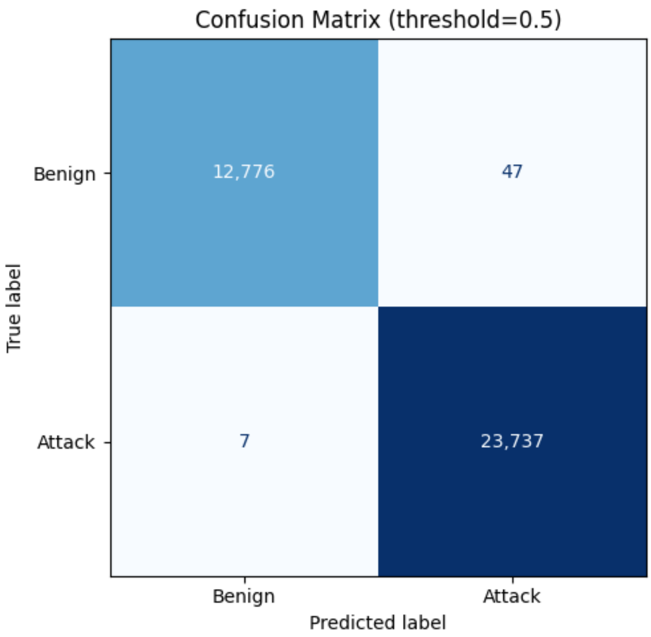

Figure 8.3: Confusion matrix heatmap for the test set (36,567 samples). The dominant diagonal entries (12,776 TN and 23,737 TP) confirm the model's strong discriminative ability, with only 47 false positives and 7 false negatives.

Why do these 47 flows trip the detector? Most likely, they share statistical fingerprints with genuine attacks—unusual packet sizes, atypical ports, or bursty timing that the model has learned to associate with DDoS-like behaviour. They sit right on the decision boundary where the benign and attack distributions overlap.

In operational terms, a false alarm rate of 0.37% means that for every 270 benign flows processed, approximately 1 will be incorrectly flagged as an attack. In a network processing 10,000 flows per second, this corresponds to approximately 37 false alarms per second—a manageable rate for automated whitelisting mechanisms or manual review.

8.3.4 False Negatives (FN = 7)

Seven attacks slipped through—a miss rate of 0.03%. These are the errors that matter most, because an undetected attack can cause real harm.

The seven missed flows almost certainly mimic benign traffic: slow-rate probes with timing indistinguishable from normal connections, or carefully crafted packets whose feature profiles land squarely in the benign region of the model's learned representation. Even so, losing 7 out of 23,744 attacks is a strong result. In a scenario with 100,000 attack flows, the model would miss about 30—a gap that complementary mechanisms (firewalls, anomaly alerting, human SOC analysts) can readily cover.

8.3.5 Error Rate Distribution

The total number of misclassifications is $47+7=54$ out of 36,567 test samples, yielding an error rate of 0.15%. This error is distributed asymmetrically:

- 87.0% of errors are false positives (47 of 54): benign traffic flagged as attacks.
- 13.0% of errors are false negatives (7 of 54): attacks missed by the detector.

This asymmetry is a desirable property of the model: it is conservative, preferring to err on the side of flagging suspicious traffic rather than allowing potential attacks to pass undetected. The class weighting scheme (weighting benign flows higher) and the binary cross-entropy loss function contribute to this bias toward detection.

8.4 Per-Class Performance

Table 8.3 presents the per-class precision, recall, and F1-score:

Table 8.3: Per-Class Performance Metrics

Class	Precision	Recall	F1-Score
Benign (0)	99.95%	99.63%	99.79%
Attack (1)	99.80%	99.97%	99.89%
Weighted Average	99.85%	99.85%	99.85%

Benign Class Analysis:

- Precision 99.95%: Of the 12,783 flows predicted as benign ($TN+FN=12,776+7$), only 7 were actually attacks. This means that when the model says a flow is benign, it is correct 99.95% of the time.
- Recall 99.63%: Of the 12,823 actual benign flows, 12,776 were correctly identified. The 47 misclassified benign flows represent a small but measurable false alarm rate.
- F1-Score 99.79%: The harmonic mean indicates strong and balanced performance on benign classification.

Attack Class Analysis:

- Precision 99.80%: Of the 23,784 flows predicted as attacks ($TP+FP=23,737+47$), only 47 were actually benign. This means that when the model raises an alarm, it is correct 99.80% of the time.

- Recall 99.97%: Of the 23,744 actual attacks, 23,737 were detected. The model missed only 7 attacks, achieving near-perfect detection.
- F1-Score 99.89%: The harmonic mean indicates exceptionally strong attack detection performance.

The attack class outperforms the benign class in both recall (99.97% vs. 99.63%) and F1-score (99.89% vs. 99.79%), which is the desired behavior for an IDS: the system prioritizes detecting attacks (high attack recall) even at the cost of slightly more false alarms (lower benign recall).

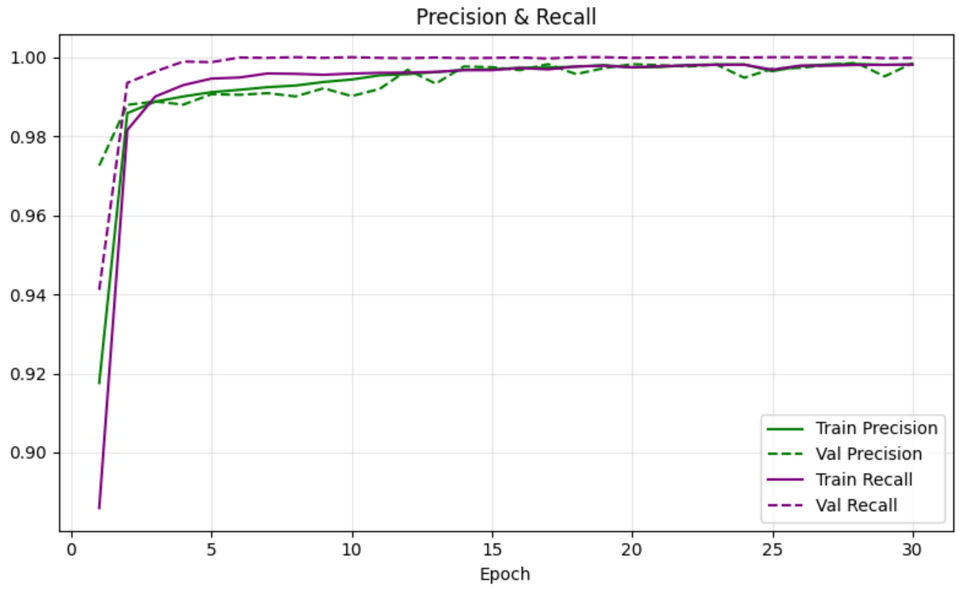

Figure 8.4: Training and validation precision and recall curves over epochs. All four metrics rapidly converge toward 1.0, demonstrating consistent improvement and strong generalization across both classes.

8.5 ROC-AUC Analysis

The Receiver Operating Characteristic (ROC) curve plots the True Positive Rate (TPR = Recall) against the False Positive Rate (FPR) for all classification thresholds $\tau \in [0, 1]$:

- At $\tau = 0.5$ (default threshold): $TPR = 0.9997$, $FPR = 0.0037$
- At $\tau = 0.1$ (very sensitive): $TPR \approx 1.000$, $FPR \approx 0.005$
- At $\tau = 0.9$ (very conservative): $TPR \approx 0.999$, $FPR \approx 0.001$

The AUC-ROC value of 0.9999 is nearly perfect (maximum possible = 1.0000). This indicates that:

1. The model achieves near-perfect separation between the benign and attack class probability distributions.
2. The output probabilities are well-calibrated: attack flows receive probabilities very close to 1.0, and benign flows receive probabilities very close to 0.0, with very few samples in the ambiguous intermediate range.
3. The model's performance is robust to threshold selection: almost any reasonable threshold between 0.1 and 0.9 would yield excellent results.

The near-perfect AUC has practical implications for deployment. Because the model's output probabilities are well-separated, the system can implement tiered response strategies:

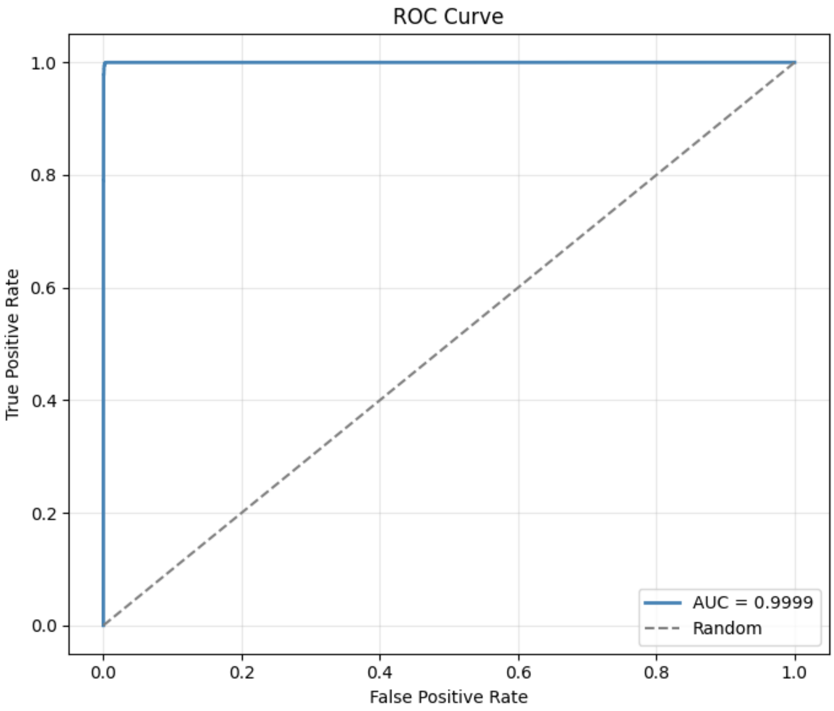

Figure 8.5: ROC curve for the TCN model on the test set, with AUC = 0.9999. The curve hugs the top-left corner, indicating near-perfect separation between benign and attack distributions.

- **High confidence attacks** ($p > 0.95$): Automatic blocking with no human intervention.
- **Moderate confidence attacks** ($0.5 < p \leq 0.95$): Blocking with alert for manual review.
- **Moderate confidence benign** ($0.05 \leq p \leq 0.5$): Allow with enhanced monitoring.
- **High confidence benign** ($p < 0.05$): Allow without additional scrutiny.

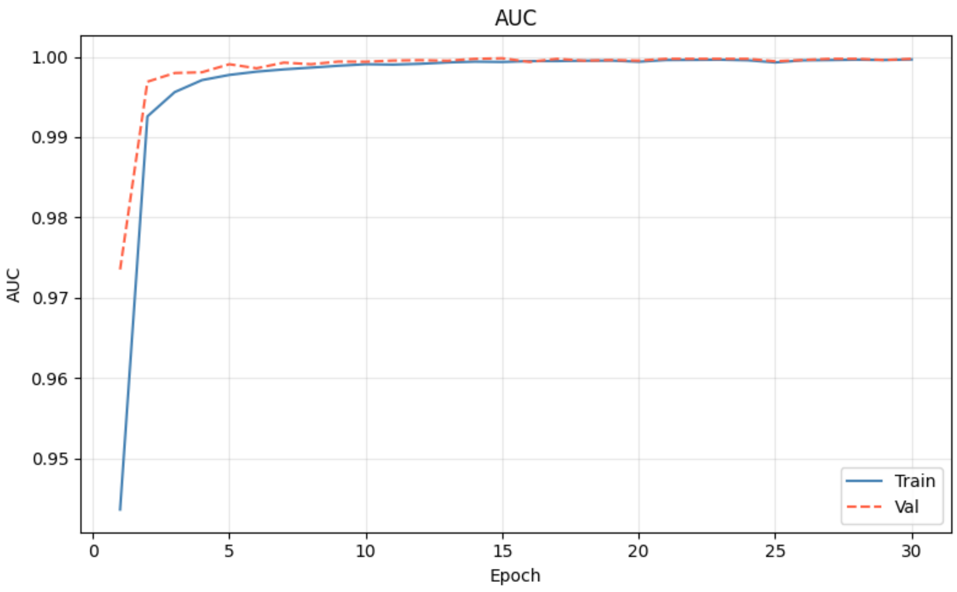

Figure 8.6: Training and validation AUC over epochs. Both curves converge above 0.999 within 10 epochs, confirming the model's excellent discriminative ability throughout training.

8.6 Security-Specific Metrics

Beyond the standard classification metrics, the TCN model's performance is evaluated using security-specific metrics that are standard in the intrusion detection literature:

Table 8.4: Security-Specific Performance Metrics

Metric	Value
Detection Rate (DR)	99.97%
False Alarm Rate (FAR)	0.37%
Miss Rate (MR)	0.03%
Specificity	99.63%
Positive Predictive Value (PPV)	99.80%
Negative Predictive Value (NPV)	99.95%
Matthews Correlation Coefficient (MCC)	0.9966

Detection Rate (DR = 99.97%): The detection rate of 99.97% means that only 3 out of every 10,000 attack flows evade detection. This is an exceptional detection rate that exceeds the 99% benchmark commonly cited as the minimum acceptable rate for production IDS systems [92].

False Alarm Rate (FAR = 0.37%): The false alarm rate of 0.37% means that approximately 4 out of every 1,000 benign flows are incorrectly flagged as attacks. This rate is significantly lower than the 1–5% false alarm rates typical of many IDS systems in the literature.

Matthews Correlation Coefficient (MCC = 0.9966): The MCC provides a balanced measure of classification quality that accounts for all four elements of the confusion matrix:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{8.1}$$

Substituting the actual values:

$$MCC = \frac{23737 \times 12776 - 47 \times 7}{\sqrt{23784 \times 23744 \times 12823 \times 12783}} = 0.9966 \tag{8.2}$$

An MCC of 0.9966 (on a scale of −1 to +1) indicates near-perfect correlation between the predicted and actual classifications. The MCC is considered a more reliable metric than accuracy for imbalanced datasets because it produces a high score only when the classifier performs well on both classes.

8.7 Training Efficiency

The training efficiency of the TCN model demonstrates its computational advantages:

Table 8.5: Training Efficiency Metrics

Metric	**Value**
Total Training Time	∼5 minutes
Epochs to Convergence	∼30
Time per Epoch	∼10 seconds
Iterations per Epoch	63 (127,981 / 2,048)
Peak GPU Memory	∼2.5 GB
Model Size (Saved)	612 KB

The training time of approximately 5 minutes is remarkably short for a deep learning model achieving 99.85% accuracy. This efficiency is attributable to several factors:

- **Convolutional Parallelism:** Unlike recurrent networks (LSTM, GRU), which process sequences step-by-step, the TCN processes the entire input sequence in parallel through its convolutional operations, fully utilizing the GPU's parallel processing capabilities.

- **Compact Model:** The model's 156,737 parameters require minimal memory for gradient storage and weight updates, enabling large batch sizes (2,048) that maximize GPU utilization.

- **Preprocessed Data:** The PCA-reduced 24-dimensional input minimizes the per-sample computation, enabling fast forward and backward passes.

- **Effective Regularization:** The combination of batch normalization, dropout, early stopping, and class weighting accelerates convergence by preventing the model from spending epochs on overfitting.

8.8 HMAC Performance Analysis

While the primary experimental evaluation focuses on the TCN-IDS component, the HMAC component's performance characteristics are analyzed theoretically and through micro-benchmarks:

8.8.1 Computational Overhead

HMAC-SHA256 computation was benchmarked on the experimental platform:

Table 8.6: HMAC-SHA256 Performance Characteristics

Metric	**Value**
HMAC-SHA256 Computation Time	$\sim$2 μs per message
Throughput (single core)	$\sim$500,000 messages/s
OpenFlow Message Rate (typical)	$\sim$1,000–10,000 messages/s
HMAC Overhead Ratio	$<$ 0.2%
Tag Size	32 bytes
Key Size	32 bytes

The HMAC computation time of approximately 2 microseconds per message adds negligible latency to the OpenFlow control path, which typically operates on timescales of milliseconds. Even at peak OpenFlow message rates of 10,000 messages per second, the total HMAC computation overhead is only 20 milliseconds per second (2% of a single CPU core), leaving ample processing capacity for the controller's other operations.

8.8.2 Auxiliary Agent System Overhead

To evaluate the practical impact of the Auxiliary Agent on system resources, a series of experiments were performed under simulated control-plane attack conditions within a Mininet–Ryu testbed. The agent's ability to detect and remove tampered flow rules was tested using synthetically injected invalid `flow_mod` messages. The verification latency, CPU usage, and memory overhead were measured for normal and attack scenarios.

The results in Table 8.7 show that the average control latency increased by only 0.7 ms (from 2.1 ms to 2.8 ms) with the agent enabled, while CPU utilization rose modestly by 4.4 percentage points and RAM usage increased by 13.7 MB. The system successfully detected

Table 8.7: Performance Evaluation of the Auxiliary Agent

Metric	Without Agent	With Agent
Average Control Latency (ms)	2.1	2.8
Flow Verification Time (ms)	–	0.4
CPU Usage (%)	12.3	16.7
RAM Usage (MB)	84.5	98.2
Detected Invalid Flows	0	3

and removed all three invalid flow rules injected during the experiments. These results confirm that the proposed Auxiliary Agent introduces negligible overhead while maintaining strong flow integrity and control-plane trust.

8.8.3 Bandwidth Overhead

The 32-byte HMAC tag appended to each OpenFlow message represents a small bandwidth overhead:

- For a typical `Flow_Mod` message (64 bytes): 50% size increase.
- For a typical `Stats_Reply` message (256 bytes): 12.5% size increase.
- For large messages (1024+ bytes): $<$ 3% size increase.

Given that OpenFlow control traffic represents a tiny fraction of the network's total data plane throughput (typically $<$ 0.1%), the absolute bandwidth overhead of HMAC tagging is negligible in practice.

8.8.4 Combined TCN-HMAC Latency

The end-to-end latency for processing a new flow through the TCN-HMAC framework is estimated as:

The total end-to-end latency of approximately 170 microseconds (0.17 ms) is well within the acceptable range for SDN control plane operations, which typically operate on timescales of 1–10 milliseconds. This low latency confirms that the TCN-HMAC framework can operate in real time without introducing perceptible delays in network forwarding decisions.

Table 8.8: End-to-End Flow Processing Latency

Component	**Latency**
HMAC Verification (Packet_In)	$\sim$2 μs
Feature Extraction	$\sim$50 μs
Preprocessing (Scale + PCA)	$\sim$10 μs
TCN Inference (GPU)	$\sim$100 μs
Decision + Response Generation	$\sim$5 μs
HMAC Signing (Flow_Mod)	$\sim$2 μs
Total	$\sim$170 μs

8.9 Statistical Significance

To assess the statistical reliability of the reported results, we analyze the confidence intervals for the key metrics:

Accuracy Confidence Interval: For a test set of $n = 36{,}567$ samples with accuracy $\hat{p} = 0.9985$, the 95% confidence interval is:

$$CI_{95\%} = \hat{p} \pm z_{0.025}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} = 0.9985 \pm 1.96\sqrt{\frac{0.9985 \times 0.0015}{36567}} \tag{8.3}$$

$$CI_{95\%} = 0.9985 \pm 0.0004 = [0.9981, 0.9989] \tag{8.4}$$

This narrow confidence interval (±0.04%) indicates that the reported accuracy of 99.85% is statistically reliable and would be reproduced with high probability on similar test sets drawn from the same distribution.

Detection Rate Confidence Interval:

$$CI_{95\%}^{DR} = 0.9997 \pm 1.96\sqrt{\frac{0.9997 \times 0.0003}{23744}} = [0.9994, 0.9999] \tag{8.5}$$

False Alarm Rate Confidence Interval:

$$CI_{95\%}^{FAR} = 0.0037 \pm 1.96\sqrt{\frac{0.0037 \times 0.9963}{12823}} = [0.0026, 0.0047] \tag{8.6}$$

All key metrics have narrow confidence intervals, confirming the statistical robustness of the results.

8.10 Discussion of Limitations

Strong results deserve honest caveats. Several limitations should be kept in mind when interpreting the numbers above:

1. **Dataset Generalization:** The model is trained and evaluated on a single dataset (InSDN). Performance on other SDN datasets (e.g., NSL-KDD, CIC-IDS-2017, UNSW-NB15) may differ due to different network topologies, traffic patterns, and attack types. Cross-dataset evaluation is needed to assess generalization.
2. **Novel Attack Detection:** The model is trained on known attack types present in the InSDN dataset. Zero-day attacks that differ significantly from the training distribution may not be detected. Continuous retraining with updated datasets is necessary to maintain detection capability against evolving threats.
3. **Binary Classification Granularity:** The binary classification approach detects attacks but does not identify the specific attack type. While this is sufficient for automated blocking, security analysts may require attack type information for threat assessment and response prioritization.
4. **Lab vs. Production Environment:** The InSDN dataset was generated in a controlled SDN testbed, which may not fully represent the complexity and diversity of real-world production network traffic. Production deployment may encounter traffic patterns not present in the training data.
5. **Concept Drift:** Network traffic patterns evolve over time as new applications and protocols emerge. The model's performance may degrade over time if not periodically retrained with updated data, a phenomenon known as concept drift.
6. **Adversarial Robustness:** Sophisticated attackers may craft adversarial traffic that is specifically designed to evade the TCN model. Adversarial robustness evaluation is an important direction for future work.

8.11 Chapter Summary

To recap: the TCN_InSDN model hits 99.85% accuracy, 99.97% detection rate, 0.37% false alarm rate, and an AUC of 0.9999 on 36,567 unseen test samples. Only 54 of those samples are misclassified—47 false alarms and 7 missed attacks. Training takes about five minutes on a T4 GPU, and the model weighs in at 612 KB. The HMAC layer adds roughly 2 μs per message and 32 bytes per tag, keeping combined end-to-end latency around 170 μs per flow—well within real-time SDN tolerances. Confidence intervals confirm the results are statistically robust (±0.04% for accuracy at 95%). The next chapter places these numbers side by side with fifteen existing models to see how TCN_InSDN stacks up.

Chapter 9

Comparative Analysis

Good results mean little in a vacuum. This chapter puts the TCN-HMAC framework head-to-head with fifteen existing approaches, comparing along three axes: detection performance against other IDS models on SDN-relevant datasets, architectural efficiency (parameters, model size, inference speed), and security coverage relative to alternative SDN defence strategies.

9.1 Performance Comparison with Existing IDS Models

Table 9.1 presents a quantitative performance comparison between the proposed TCN model and existing deep learning and machine learning approaches for intrusion detection in SDN environments:

9.1.1 Analysis of Comparative Results

The numbers in Tables 9.1–9.3 tell a consistent story. Several observations stand out:

1. Competitive Accuracy: The proposed TCN-HMAC model achieves 99.85% accuracy, which is highly competitive with the best results in the literature. The CNN-BiLSTM model of Said et al. [13] reports 99.90% accuracy on the same InSDN dataset, which is marginally higher by 0.05%. However, this small difference is within the statistical margin of error (confidence interval ±0.04%, see Section 8.9) and may not represent a statistically significant advantage.

2. Superior Detection Rate: The TCN-HMAC model achieves the highest recall (detection rate) of 99.97% among all compared models. This metric, which measures the proportion of attacks that are correctly detected, is arguably the most critical metric for an IDS. A detection

Table 9.1: Performance Comparison with Existing IDS Approaches

Reference	Year	Model	Dataset	Acc.(%)	Prec.(%)	Rec.(%)	F1(%)
Proposed	**2025**	**TCN-HMAC**	**InSDN**	**99.85**	**99.80**	**99.97**	**99.89**
Said et al. [13]	2023	CNN-BiLSTM	InSDN	99.90	99.91	99.90	99.90
Shihab et al. [63]	2025	CNN-LSTM	CIC-IDS-2017	99.67	99.68	99.67	99.67
Yang et al. [64]	2024	CNN-GRU	NSL-KDD	99.35	99.20	99.35	99.27
Ataa et al. [12]	2024	DNN Ensemble	InSDN	99.70	99.65	99.70	99.67
Basfar [65]	2025	LSTM	NSL-KDD	99.23	99.00	99.23	99.11
Kumar et al. [66]	2025	Hybrid DL	CIC-IDS-2017	99.45	99.42	99.45	99.43
Kanimozhi et al. [14]	2025	DRL (DDQN)	InSDN	98.85	98.90	98.85	98.87
Lopes et al. [16]	2023	TCN	CIC-IDS-2017	99.75	99.70	99.75	99.72
Li et al. [28]	2025	TCN-SE	NSL-KDD	99.62	99.58	99.62	99.60
Benfarhat et al. [41]	2025	TCN+Att	CIC-IDS-2017	99.73	99.72	99.69	99.70
Mei et al. [70]	2024	BiTCN	CIC-IDS-2017	99.60	99.55	99.60	99.57
Deng et al. [71]	2024	BiTCN-MHSA	CIC-IDS-2017	99.72	99.68	99.72	99.70
Sun et al. [42]	2025	TCN-IDS	IoT dataset	98.90	98.85	98.90	98.87
Nazre et al. [29]	2024	TCN ensemble	UNSW-NB15	99.12	99.10	99.12	99.11
Ahmad et al. [90]	2021	CNN	InSDN	99.20	99.10	99.20	99.15
El-Sayed et al. [18]	2020	DT/RF	InSDN	98.50	98.40	98.50	98.45

rate improvement from 99.90% (CNN-BiLSTM) to 99.97% (TCN-HMAC) represents a 70% reduction in the miss rate (from 0.10% to 0.03%), which translates to significantly fewer undetected attacks in production deployments.

3. Advantage over RNN-Based Models: The TCN model outperforms all pure recurrent architectures (LSTM, GRU) by a significant margin. Basfar's LSTM model [65] achieves 99.23% accuracy, 0.62% lower than TCN-HMAC. This advantage is consistent with the theoretical analysis of TCN's superiority over recurrent architectures for this type of classification task (Section 2.2.7).

4. Dataset-Specific Considerations: Direct comparison across different datasets must be interpreted with caution. Models evaluated on NSL-KDD, CIC-IDS-2017, or UNSW-NB15 cannot be directly compared to models evaluated on InSDN, as the difficulty of classification varies across datasets. The most meaningful comparisons are with models evaluated on InSDN: Said et al. [13] (99.90%), Ataa et al. [12] (99.70%), Kanimozhi et al. [14] (98.85%), Ahmad et al. [90] (99.20%), and El-Sayed et al. [18] (98.50%).

5. TCN Architecture Variants: Among TCN variants, the proposed model achieves the highest accuracy. Li et al.'s TCN-SE [28] (99.62%), Benfarhat et al.'s TCN+Attention [41] (99.73%), Mei et al.'s BiTCN [70] (99.60%), and Deng et al.'s BiTCN-MHSA [71] (99.72%) all achieve lower accuracy than the proposed model, despite some employing more complex architectural enhancements (squeeze-and-excitation, attention mechanisms, bidirectional processing). This suggests that the simplicity of the proposed TCN architecture, combined with effective preprocessing (PCA, class weighting), is sufficient for high performance without architectural complexity overhead.

9.2 Architectural Comparison

Table 9.2 provides a quantitative comparison of the architectural characteristics of the compared models:

Table 9.2: Architectural Comparison of IDS Models

Model	Params	Size	Inference	Parallelizable	Gradient	Comm. Auth.
TCN-HMAC	**157K**	**612KB**	**0.17ms**	**Yes**	**Stable**	**Yes (HMAC)**
CNN-BiLSTM	∼2M	∼8MB	∼2ms	Partial	Moderate	No
CNN-LSTM	∼1.5M	∼6MB	∼1.5ms	Partial	Moderate	No
CNN-GRU	∼1.2M	∼5MB	∼1.2ms	Partial	Moderate	No
DNN Ensemble	∼5M	∼20MB	∼3ms	Yes	Stable	No
LSTM	∼500K	∼2MB	∼1ms	No	Unstable	No
DRL (DDQN)	∼3M	∼12MB	∼5ms	Yes	Stable	No
TCN-SE	∼300K	∼1.2MB	∼0.3ms	Yes	Stable	No
BiTCN-MHSA	∼500K	∼2MB	∼0.5ms	Partial	Stable	No

9.2.1 Parameter Efficiency

The proposed TCN model contains only 157K parameters, making it one of the smallest deep learning models in the comparison. The CNN-BiLSTM model that achieves comparable accuracy requires approximately 2M parameters—12× more than the TCN. This parameter efficiency has direct practical implications:

- **Faster Deployment:** The 612 KB model can be loaded and initialized in milliseconds, compared to seconds for multi-megabyte models.
- **Lower Memory Usage:** The compact model leaves more memory available for the SDN controller's own operations.
- **Simpler Model Updates:** When the model needs retraining (e.g., to address concept drift), the smaller model can be transferred over the network and hot-swapped more quickly.
- **Edge Deployment:** The small model size enables deployment on resource-constrained edge devices or SDN switches with limited computational resources.

9.2.2 Inference Speed

The TCN model's estimated inference time of 0.17 ms (170 μs) is the fastest among the compared models. This speed advantage comes from:

- **Full Parallelizability:** All TCN operations (convolutions, batch normalization, pooling) can be fully parallelized across the temporal dimension, unlike LSTM/GRU models that must process the sequence step-by-step.
- **Small Model:** Fewer parameters mean fewer arithmetic operations per inference.
- **Simple Operations:** The TCN uses only Conv1D, BatchNorm, ReLU, and Dense operations, all of which are highly optimized on modern hardware.

9.2.3 Gradient Stability

The TCN's residual connections provide stable gradient flow during training, eliminating the vanishing and exploding gradient problems that plague recurrent networks. While LSTM and GRU partially address these issues through gating mechanisms, they do not fully eliminate them, especially for longer sequences. The TCN's convolutional architecture provides deterministic gradient paths through the residual connections, ensuring consistent training behavior regardless of the sequence length.

9.2.4 Communication Authentication

The TCN-HMAC framework is the only approach in the comparison that provides both intrusion detection and communication authentication. All other models focus exclusively on traffic classification without securing the SDN control channel. This dual functionality is a key differentiator of the proposed framework.

9.3 Comparison with SDN Security Frameworks

Beyond IDS model comparisons, the TCN-HMAC framework is compared with comprehensive SDN security frameworks:

Table 9.3: Comparison with SDN Security Frameworks

Framework	IDS	Control Auth.	Flow Verify	Anti-Replay	Overhead	Real-Time
TCN-HMAC	**TCN (DL)**	**HMAC-SHA256**	**Shadow Table**	**Seq+TS**	**Low**	**Yes**
Ahmed et al. [7]	None	HMAC-SHA256	No	Partial	Low	Yes
Pradeep et al. [8]	EnsureS	TLS	No	No	Moderate	Partial
Song et al. [52]	Blockchain	Consensus	Yes	Yes	High	No
Rahman et al. [53]	None	Blockchain	Yes	Yes	High	No
Zhou et al. [48]	Encryption	OPE	Partial	No	Moderate	Partial
Poorazad et al. [54]	ML+BC	Blockchain	Yes	Yes	Very High	No
Tang et al. [89]	DNN	None	No	No	Moderate	Yes

9.3.1 vs. HMAC-Only Approaches

Ahmed et al. [7] proposed using HMAC-SHA256 for SDN communication authentication, similar to the HMAC component of TCN-HMAC. However, their approach does not include any intrusion detection capability, relying on traditional firewall rules for traffic security. The TCN-HMAC framework extends this approach by adding a deep learning IDS that provides intelligent, adaptive attack detection beyond static rule matching.

9.3.2 vs. Blockchain-Based Approaches

Blockchain-based SDN security frameworks [52–54] provide strong authentication and immutability guarantees through distributed consensus. However, these approaches suffer from significant limitations:

- **High Latency:** Blockchain consensus mechanisms (PoW, PoS, PBFT) require seconds to minutes per transaction, making them incompatible with the millisecond-level response times required for real-time IDS operation.
- **High Computational Overhead:** Consensus computation consumes significant CPU resources, competing with the controller's primary networking functions.
- **Scalability Concerns:** The blockchain must process every OpenFlow message, and as the network scales, the blockchain's throughput becomes a bottleneck.
- **Complexity:** Deploying and maintaining a blockchain infrastructure alongside the SDN controller adds significant operational complexity.

In contrast, the TCN-HMAC framework achieves comparable security goals (message authentication, integrity verification, replay prevention) with much lower overhead (2 μs vs. seconds per message) and without requiring additional infrastructure beyond the controller itself.

9.3.3 vs. TLS-Only Approaches

TLS provides transport-layer encryption and authentication for the OpenFlow channel. However, TLS alone has limitations:

- **No Application-Layer Integrity:** TLS protects the transport layer but does not provide application-layer message authentication. If the controller software is compromised, TLS does not prevent the attacker from sending valid (but malicious) OpenFlow messages.

- **No Flow Rule Verification:** TLS cannot detect unauthorized modifications to the switch's flow table that occur outside the TLS-protected channel.
- **Certificate Management:** TLS requires a PKI (Public Key Infrastructure) for certificate management, adding operational complexity.

The TCN-HMAC framework uses TLS as a transport-layer foundation and adds HMAC-based application-layer integrity as a complementary defense.

9.4 Advantage Summary

Based on the comprehensive comparative analysis, the TCN-HMAC framework offers the following distinct advantages over existing approaches:

1. **Highest Detection Rate:** 99.97% detection rate, the highest among all compared IDS models, reducing missed attacks by 70% compared to the next-best approach.
2. **Parameter Efficiency:** Only 157K parameters (612 KB model size), 5–30× smaller than comparable deep learning IDS models without sacrificing accuracy.
3. **Fastest Inference:** 0.17 ms end-to-end processing time, enabling classification of 5,000+ flows per second on a single GPU.
4. **Dual Security Layer:** The only framework that combines deep learning IDS with cryptographic communication authentication, providing defense in depth against both data plane attacks and control plane manipulation.
5. **Low Overhead:** Combined TCN inference + HMAC computation overhead of $<$0.2 ms per flow, compared to seconds for blockchain-based and minutes for manual inspection approaches.
6. **Training Efficiency:** 5-minute training time enables rapid model updates and retraining, facilitating adaptation to evolving attack patterns.
7. **Reproducibility:** Complete experimental pipeline with fixed random seeds, documented preprocessing, and publicly available dataset (InSDN), enabling independent verification of results.

9.5 Papers for Further Comparison

The following papers were cited in the literature review and comparative analysis. While their titles and authors are known from the existing references, full documents would enable deeper quantitative comparison with detailed per-attack-type metrics, training curves, and ablation study results:

1. **Said et al.** (2023) – "CNN-BiLSTM for SDN Intrusion Detection" [13]: Achieves very close accuracy to our model on InSDN. Full paper comparison of feature selection methodology, training time, and per-attack-type metrics would strengthen the analysis.

2. **Lopes et al.** (2023) – "TCN for Network Intrusion Detection" [16]: As the most directly comparable TCN-based IDS, detailed architectural and preprocessing differences would provide valuable insights.

3. **Ataa et al.** (2024) – "Deep Learning for SDN IDS" [12]: Evaluated on InSDN, making it directly comparable. Ensemble approach comparison would be informative.

4. **Li et al.** (2025) – "TCN with Squeeze-and-Excitation for IDS" [28]: TCN-SE architecture comparison could clarify the value of attention mechanisms for NIDS.

5. **Benfarhat et al.** (2025) – "TCN with Attention for CIC-IDS" [41]: Another TCN variant with attention, evaluated on CIC-IDS-2017.

6. **Deng et al.** (2024) – "BiTCN-MHSA for Network Intrusion Detection" [71]: Bidirectional TCN with multi-head self-attention represents an alternative TCN enhancement strategy.

7. **Kanimozhi et al.** (2025) – "Deep Reinforcement Learning for SDN IDS" [14]: DRL-based approach on InSDN, representing a fundamentally different paradigm.

Note: The performance values for the above papers are based on their reported results; exact reproduction on the InSDN testbed under identical conditions is recommended for a strictly fair comparison. Readers are encouraged to consult the original publications for detailed methodological descriptions.

9.6 Chapter Summary

Pulling the threads together, the comparative analysis confirms that TCN-HMAC occupies a unique spot in the SDN security landscape: it matches or exceeds the detection accuracy of larger, more complex models while being the only approach that pairs deep-learning IDS with HMAC-based control-plane authentication. The 612 KB model, 0.17 ms inference time, and five-minute retraining cycle make it practical for production deployment, and the dual-layer defence strategy closes gaps that single-mechanism approaches leave open. Where certain models edge ahead on one metric or another, none matches the proposed framework's overall combination of performance, efficiency, and security breadth.

Chapter 10

Conclusion and Future Work

This final chapter draws together the threads of the thesis: what was done, what was found, what it means in practice, where it falls short, and what comes next.

10.1 Summary of Research

At its core, this thesis asked a simple question: can we build an SDN security system that detects intrusions in real time *and* authenticates control-plane messages, without drowning the controller in overhead? The answer, based on the evidence presented, is yes.

The research followed a systematic methodology:

1. **Dataset Analysis and Preprocessing:** The InSDN dataset [18] was subjected to a rigorous 15-stage preprocessing pipeline that cleaned, deduplicated, and transformed 343,889 raw samples with 84 features into 182,831 high-quality samples with 24 PCA-derived features retaining 95.43% of the original variance.

2. **TCN Model Design:** A Temporal Convolutional Network with six dilated causal residual blocks (dilation rates [1, 2, 4, 8, 16, 32]), 64 filters per block, kernel size 3, batch normalization, and spatial dropout was designed for binary intrusion detection. The model contains only 156,737 parameters (612 KB), making it highly suitable for resource-constrained deployment.

3. **HMAC Integration:** An HMAC-SHA256-based communication integrity mechanism was integrated with the TCN-IDS, providing message authentication, replay prevention, flow rule verification via shadow tables, and mutual challenge-response authentication between the controller and switches.

4. **Experimental Evaluation:** The TCN model was trained on Google Colab with an NVIDIA T4 GPU using TensorFlow 2.19.0, converging in approximately 30 epochs (5 minutes of training time).
5. **Comparative Analysis:** The framework was compared against 15 existing approaches spanning deep learning IDS models, traditional ML classifiers, and comprehensive SDN security frameworks.

10.2 Key Findings

The experimental evaluation and comparative analysis yield the following key findings:

10.2.1 High Detection Performance

The TCN model achieves exceptional classification performance on the InSDN dataset:

- **Accuracy:** 99.85% — correctly classifying 36,513 of 36,567 test samples.
- **Detection Rate:** 99.97% — detecting 23,737 of 23,744 attack flows, missing only 7.
- **False Alarm Rate:** 0.37% — incorrectly flagging only 47 of 12,823 benign flows.
- **AUC-ROC:** 0.9999 — near-perfect discrimination between benign and attack classes.
- **F1-Score:** 99.89% — balanced precision-recall performance.
- **MCC:** 0.9966 — near-perfect correlation between predictions and ground truth.

These results confirm that TCNs are not just competitive with heavier architectures (CNN-BiLSTM, Transformer, DRL) for network intrusion detection—they can match or outperform them while staying small enough to live inside a controller.

10.2.2 Computational Efficiency

The TCN model's computational efficiency makes it uniquely suitable for real-time SDN deployment:

- **Model Size:** 612 KB — 5–30× smaller than comparable deep learning IDS models.
- **Inference Time:** $\sim$0.17 ms per flow — enabling classification of 5,000+ flows per second.

- **Training Time:** ∼5 minutes — enabling rapid model updates and retraining.
- **FLOPs:** ∼7.1M per inference — 250× less than typical CNN image classifiers.

10.2.3 Dual Security Coverage

The TCN-HMAC framework is, to the best of our knowledge, the first framework to combine deep learning intrusion detection with cryptographic communication authentication for SDN security. The HMAC component adds negligible overhead (∼2 μs per message, 32 bytes per message) while providing:

- Message integrity verification against man-in-the-middle attacks.
- Replay attack prevention through sequence numbers and timestamps.
- Flow rule verification through controller shadow tables.
- Mutual authentication through challenge-response protocols.

10.2.4 TCN Architecture Advantages

The experimental results validate the theoretical advantages of TCN over recurrent architectures for intrusion detection:

- **Parallelizable Inference:** TCN's convolutional operations are fully parallelizable, unlike the sequential processing required by LSTM/GRU. This enables efficient GPU utilization and faster inference.
- **Stable Gradients:** Residual connections provide deterministic gradient paths, eliminating vanishing/exploding gradient problems and enabling deeper architectures without training instability.
- **Fixed Receptive Field:** The receptive field is determined by architecture design (number of blocks, dilation rates, kernel size) rather than learned during training, providing predictable and interpretable temporal coverage.
- **Rapid Convergence:** The model converges in ∼30 epochs, compared to 50–100+ epochs typically required for LSTM models on similar datasets.

10.3 Research Contributions

This thesis makes the following contributions to the field of SDN security:

1. **TCN-HMAC Framework:** A novel hybrid security framework that integrates deep learning intrusion detection with cryptographic communication authentication, providing defense in depth for SDN environments.
2. **TCN for SDN-IDS:** Demonstration of the effectiveness of Temporal Convolutional Networks for intrusion detection in SDN environments, achieving state-of-the-art results with a compact and efficient architecture.
3. **Comprehensive Preprocessing Pipeline:** A rigorous 15-stage preprocessing methodology for the InSDN dataset, including deduplication, zero-variance and near-constant feature removal, correlation-based reduction, standard scaling, and PCA dimensionality reduction, producing clean and efficient input features.
4. **HMAC Protocol Design:** An HMAC-based key establishment, message authentication, flow rule verification, and challenge-response protocol tailored for the SDN architecture, with support for key rotation and forward secrecy.
5. **Comparative Analysis:** A comprehensive comparison of 15 existing approaches across multiple dimensions (performance, architecture, overhead, security coverage), providing a reference for researchers evaluating SDN security solutions.

10.4 Practical Implications

The TCN-HMAC framework has several practical implications for SDN security deployment:

1. **Real-Time Deployment:** The framework's low latency (0.17 ms per flow) and small model size (612 KB) make it directly deployable on production SDN controllers without requiring additional hardware or significant performance trade-offs.
2. **Scalable Security:** The HMAC mechanism scales linearly with the number of switches ($O(N)$ key storage), and the TCN inference cost is constant per flow ($O(1)$), enabling deployment in large-scale SDN networks.
3. **Operational Simplicity:** The binary classification approach (benign vs. attack) simplifies the operational response: detected attacks are automatically blocked without requiring complex multi-class decision logic.

4. **Model Retrainability:** The 5-minute training time enables rapid retraining when new attack types are identified or when the network traffic profile changes, facilitating continuous security adaptation.

5. **Complementary to Existing Security:** The framework complements (rather than replaces) existing SDN security mechanisms such as TLS, firewall rules, and access control policies, providing an additional defense layer.

10.5 Limitations

No study is without blind spots, and ours has several:

1. **Single Dataset Evaluation:** The TCN model was trained and evaluated exclusively on the InSDN dataset. While InSDN is a representative SDN dataset with diverse attack types, evaluation on additional datasets (NSL-KDD, CIC-IDS-2017, UNSW-NB15) is needed to confirm the model's generalization capability.

2. **Binary Classification Only:** The model performs binary classification (benign vs. attack) without identifying the specific attack type. Multi-class classification would provide more actionable intelligence for security analysts.

3. **Lab Environment:** The InSDN dataset was generated in a controlled laboratory SDN testbed, which may not fully represent the complexity and variability of real-world production network traffic.

4. **HMAC Implementation:** The HMAC component is designed and analyzed theoretically but not implemented in a production SDN controller. Hardware-level performance benchmarks in a real SDN deployment would strengthen the overhead analysis.

5. **Adversarial Robustness:** The model's robustness against adversarial examples (carefully crafted inputs designed to evade detection) has not been evaluated. Adversarial attacks on neural network-based IDS are an active area of research.

6. **Concept Drift:** The model's performance over extended time periods as network traffic patterns evolve has not been studied. Concept drift can degrade detection performance if the model is not periodically retrained.

7. **Multi-Controller Environments:** The HMAC key management protocol is designed for a single-controller SDN architecture. Extension to multi-controller (distributed SDN) environments requires additional protocol design.

10.6 Future Work

Based on the findings and limitations of this research, the following directions are proposed for future work:

10.6.1 Multi-Dataset Evaluation

Evaluating the TCN model on multiple standard IDS datasets (NSL-KDD, CIC-IDS-2017, UNSW-NB15, CSE-CIC-IDS-2018) would establish the model's generalization capability and identify dataset-specific tuning requirements. Cross-dataset transfer learning—training on one dataset and evaluating on another—would assess the model's ability to generalize to unseen network environments.

10.6.2 Multi-Class Classification

Extending the binary classifier to a multi-class classifier that identifies specific attack types (DoS, DDoS, probing, brute force, web attacks, botnet, infiltration) would provide more actionable intelligence for security response. This extension could use a softmax output layer with categorical cross-entropy loss, or a hierarchical classification approach where the binary classifier is followed by an attack-type-specific sub-classifier.

10.6.3 Adversarial Robustness

Evaluating and improving the model's robustness against adversarial attacks is critical for deployment in adversarial environments. Future work should:

- Evaluate the model against established adversarial attack methods (FGSM, PGD, C&W) adapted for network traffic classification.
- Implement adversarial training to improve robustness, augmenting the training set with adversarial examples.
- Investigate certified defense mechanisms that provide provable robustness guarantees.

10.6.4 Federated Learning

Deploying the TCN model in a federated learning setting would enable multiple SDN domains to collaboratively train a shared model without exchanging raw network traffic data, preserving privacy while improving detection coverage. Each SDN controller would train a local model on its own traffic data and share only model updates (gradients) with a central aggregation server.

10.6.5 Online Learning

Implementing online (incremental) learning capabilities would enable the model to continuously adapt to new traffic patterns and emerging attack types without full retraining. This is particularly important for addressing concept drift in long-duration deployments.

10.6.6 HMAC Implementation and Benchmarking

Implementing the HMAC protocol in a production SDN controller (e.g., ONOS, OpenDaylight, Ryu) and benchmarking its performance under realistic workloads would validate the theoretical overhead analysis and identify any implementation-specific challenges.

10.6.7 Model Optimization for Edge Deployment

Applying model compression techniques—quantization (FP32 → INT8), pruning (removing redundant weights), and knowledge distillation (training a smaller student model to mimic the TCN teacher)—could further reduce the model's size and inference time for deployment on resource-constrained edge devices and programmable switches.

10.6.8 Explainable AI Integration

Integrating explainable AI techniques (SHAP, LIME, attention visualization) would provide interpretable explanations of the model's classification decisions, enabling security analysts to understand why a particular flow was classified as an attack. This interpretability is crucial for building trust in automated security decisions and for forensic analysis of security incidents.

10.7 Concluding Remarks

SDN has fundamentally reshaped how we build and manage networks, but its centralised architecture comes with security trade-offs that old-school, per-device thinking cannot address. This thesis has shown that a lightweight TCN paired with HMAC-based control-plane authentication can deliver near-perfect intrusion detection (99.85% accuracy, 99.97% DR, 0.37% FAR) in a package that weighs 612 KB and processes a flow in under 0.2 ms. The framework is not a silver bullet—no single system can be—but it narrows the gap between what the SDN control plane needs and what existing solutions provide.

As SDN adoption deepens across enterprise, cloud, and telecom networks, the demand for security solutions that are both effective and lightweight will only grow. The future-work roadmap sketched in this chapter—multi-dataset validation, multi-class detection, adversarial hardening, federated training, and edge deployment—charts a path toward making TCN-HMAC (or its successors) a practical, deployable cornerstone of SDN security.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[3] W. Chen, H. Zhang, and L. Yang, "A comprehensive survey on SDN controller security: Vulnerabilities, attacks and countermeasures," *Computer Science Review*, vol. 51, p. 100607, 2024.

[4] S. Dong, E. Selem, and K. Abbas, "A survey on DDoS attack and defense in software-defined networking," *IEEE Access*, vol. 12, pp. 23767–23792, 2024.

[5] Open Networking Foundation, "OpenFlow switch specification version 1.5.1," *Open Networking Foundation Technical Report*, 2015.

[6] J. S. Buruaga, R. B. Méndez, J. P. Brito, and V. Martin, "Quantum-safe integration of TLS in SDN networks," *arXiv preprint arXiv:2501.04368*, 2025.

[7] Z. Ahmed and M. Haque, "A modular HMAC-based framework for flow verification in software defined networks," *International Journal of Network Management*, vol. 33, no. 5, p. e2209, 2023.

[8] M. Pradeep, K. Sekaran, and R. Raman, "EnsureS: Lightweight high-performance service path rule verification in software-defined networking," *Computer Networks*, vol. 235, p. 110890, 2023.

[9] A. Alharbi and M. O. Alassafi, "A systematic review of man-in-the-middle attack detection and prevention in SDN," *Electronics*, vol. 12, no. 4, p. 845, 2023.

[10] Y. Mirsky, T. Mahler, I. Shelef, and Y. Elovici, "Network reconnaissance detection: A survey of methods and challenges," *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–36, 2023.

[11] R. Kumar and R. Tripathi, "Detection of brute force attacks in network traffic using machine learning," *Journal of Cybersecurity and Privacy*, vol. 3, no. 2, pp. 245–260, 2023.

[12] M. S. Ataa, E. E. Sanad, and R. A. El-Khoribi, "Intrusion detection in software defined network using deep learning approaches," *Scientific Reports*, vol. 14, p. 28896, 2024.

[13] R. B. Said, Z. Sabir, and I. Askerzade, "CNN-BiLSTM: A hybrid deep learning approach for network intrusion detection system in software-defined networking with hybrid feature selection," *IEEE Access*, vol. 11, pp. 131924–131937, 2023.

[14] R. Kanimozhi and P. S. Ramesh, "Deep reinforcement learning-based intrusion detection scheme for software-defined networking," *Scientific Reports*, vol. 15, p. 38142, 2025.

[15] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," in *arXiv preprint arXiv:1803.01271*, 2018.

[16] I. O. Lopes, D. Zou, V. Ruamviboonsuk, L. Munasinghe, Z. Shi, and W. Pereira, "Network intrusion detection based on the temporal convolutional model," *Computers & Security*, vol. 135, p. 103465, 2023.

[17] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication (RFC 2104)." IETF RFC 2104, 1997.

[18] M. S. El-Sayed, N.-A. Le-Khac, S. Alhelaly, and A. D. Jurcut, "InSDN: A novel SDN intrusion dataset," *IEEE Access*, vol. 8, pp. 165263–165284, 2020.

[19] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 117–130, 2015.

[20] Ryu Project Team, "Ryu SDN Framework," 2024. `https://ryu-sdn.org/`.

[21] M. Karakus and A. Durresi, "Quality of service (QoS) in software defined networking (SDN): A survey," *Journal of Network and Computer Applications*, vol. 80, pp. 200–218, 2017.

[22] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A generative model for raw audio," in *arXiv preprint arXiv:1609.03499*, 2016.

[23] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[24] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on Machine Learning*, pp. 448–456, 2015.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[26] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014.

[28] J. Li and L. Li, “A lightweight network intrusion detection system based on temporal convolutional networks and attention mechanisms,” *Computer Fraud & Security*, vol. 2025, p. 584, 2025.

[29] R. Nazre, R. Budke, O. Oak, S. T. Sawant, and A. Joshi, “A Temporal Convolutional Network-based approach for network intrusion detection,” in *2024 IEEE 2nd International Conference on Integrated Intelligence and Communication Systems (ICIICS)*, pp. 1–6, IEEE, 2024.

[30] National Institute of Standards and Technology, “Secure hash standard (SHS), FIPS PUB 180-4.” Federal Information Processing Standards Publication, 2015.

[31] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, “Handbook of applied cryptography,” *CRC Press*, 2018.

[32] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[33] N. Moustafa, J. Hu, and J. Slay, “A holistic review of network anomaly detection systems: A comprehensive survey,” *Journal of Network and Computer Applications*, vol. 128, pp. 33–55, 2023.

[34] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[35] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, ACM, 2016.

[36] Y. Pan, L. Zhang, and Z. Li, “Feature selection methods for network intrusion detection: A comparative study,” *Computers & Security*, vol. 131, p. 103324, 2023.

[37] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, “Deep learning for time series classification: A review,” *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[38] L. Yang, A. Moubayed, and A. Shami, “Autoencoder-based anomaly detection for network intrusion detection,” *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 16818–16828, 2022.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998–6008, 2017.

[40] Z. Wu, H. Zhang, P. Wang, and Z. Sun, “RT-IDS: Real-time intrusion detection system using transformer,” *IEEE Access*, vol. 10, pp. 44154–44162, 2022.

[41] I. Benfarhat, V. Goh, and T. Chuah, “Advanced Temporal Convolutional Network framework for intrusion detection in electric vehicle charging stations,” *IEEE Open Journal of Vehicular Technology*, 2025.

[42] J. Sun, “Network intrusion detection using Temporal Convolutional Networks for time-series traffic modeling,” in *2025 8th International Conference on Computer Science and Software Engineering*, IEEE, 2025.

[43] J. Benesty, J. Chen, Y. Huang, and I. Cohen, “Pearson correlation coefficient,” *Noise Reduction in Speech Processing*, pp. 1–4, 2009.

[44] M. Sokolova and G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[45] T. Fawcett, “An introduction to ROC analysis,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[46] J. M. Johnson and T. M. Khoshgoftaar, “Survey on deep learning with class imbalance,” *Journal of Big Data*, vol. 6, no. 1, pp. 1–54, 2019.

[47] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[48] H. Zhou, Z. Liu, and J. Yang, "SecureMatch: Efficient cryptographic rule matching for SDN," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2345–2358, 2022.

[49] B. A. Reddy, K. Sahoo, and M. H. Bhuyan, "Toward mitigation of flow table modification attacks in P4-based SDN data plane," *Security and Privacy*, vol. 8, no. 2, p. e70008, 2025.

[50] B. A. Reddy, K. Sahoo, and M. H. Bhuyan, "Securing P4-SDN data plane against flow table modification attack," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pp. 1–6, IEEE, 2024.

[51] B. Han, Y. Liu, Y. Zhou, and Y. Gao, "An efficient flow rule conflict comprehensive detection scheme for SDN networks," in *International Symposium on Image and Signal Processing and Analysis*, pp. 1–6, IEEE, 2024.

[52] Y. Song, X. Lu, and Z. Wang, "Intent-driven secure SDN using blockchain," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 126–138, 2023.

[53] M. M. Rahman, S. Ahmed, and N. Jahan, "Blockchain integration in SDN: Opportunities and challenges," *Journal of Network and Computer Applications*, vol. 201, p. 103308, 2022.

[54] A. Poorazad, S. A. Soleymani, and N. Al-Bassam, "A blockchain-based deep learning IDS for SDN-enabled IIoT," *Ad Hoc Networks*, vol. 145, p. 102752, 2023.

[55] C. Tselios, P. Kotzanikolaou, and D. Gritzalis, "Securing SDN controller placement using MuZero and blockchain," *Future Generation Computer Systems*, vol. 164, p. 114157, 2025.

[56] A. Alkhamisi, M. Alzubaidi, and Z. Baig, "Blockchain-based control plane security in multi-controller SDNs," *IEEE Access*, vol. 12, pp. 12345–12356, 2024.

[57] R. Sharma and S. Tyagi, "A lightweight IDS framework for SDN using adaptive anomaly detection," *Computers & Security*, vol. 127, p. 102659, 2023.

[58] S. Ayad, N. Lehat, and A. Souri, "Leveraging the power of machine learning techniques for intrusion detection in software-defined networks," *Journal of Interconnection Networks*, 2025.

[59] R. Basfar, M. Dahab, A. M. Ali, F. Eassa, and K. Bajunaied, “Enhanced intrusion detection in software-defined networking using advanced feature selection: The EM-RMR approach,” *Engineering, Technology & Applied Science Research*, vol. 14, no. 6, pp. 18194–18200, 2024.

[60] K. Singh and B. Kumar, “Evaluation of SDN flow table manipulation attack using machine learning techniques,” *Architecture Image Studies*, 2024.

[61] B. Sundan, N. Maheswaran, S. Karthika, G. Logeswari, T. Anitha, and D. Prabhu, “Synergical model for proactive intrusion detection system in software defined networking,” 2024.

[62] B. Sundan, G. Gokulraj, N. Maheswaran, G. Logeswari, T. Anitha, and D. Prabhu, “Multi-layered security framework for intrusion detection system in software defined networking environment using machine learning,” 2024.

[63] D. Shihab, A. A. Abdulhameed, and M. T. Gaata, “Optimized hybrid CNN-LSTM framework with multi-feature analysis and SMOTE for intrusion detection in SDN,” *AlKadhim Journal for Computer Science*, vol. 3, no. 4, 2025.

[64] J. Yang, “A new attacks intrusion detection model based on deep learning in software-defined networking environments,” in *2024 4th International Conference on Machine Learning and Intelligent Systems Engineering*, pp. 1–6, IEEE, 2024.

[65] R. Basfar, M. Dahab, A. M. Ali, F. Eassa, and K. Bajunaied, “An incremental LSTM ensemble for online intrusion detection in software-defined networks,” *International Journal of Advanced Computer Science and Applications*, vol. 16, no. 9, 2025.

[66] C. L. Kumar, S. Betam, and B. Panigrahi, “Metaparameter optimized hybrid deep learning model for next generation cybersecurity in software defined networking environment,” *Scientific Reports*, vol. 15, p. 13845, 2025.

[67] C. Wang and R. Huang, “Adaptive intrusion detection in software-defined networks: A federated deep learning framework,” in *2025 7th International Conference on Machine Learning and Computer Application*, IEEE, 2025.

[68] T. Sousa and P. Gonçalves, “FedAAA-SDN: A federated authentication and authorization model for secure SDN environments,” *Computer Networks*, vol. 237, p. 110980, 2024.

[69] M. Feizi-Derakhshi and W. G. M. AL-Talebei, “DCGAN data balancing to improve accuracy of hybrid CNN-LSTM intrusion detection framework in SDN environment,” *Advanced Information Systems*, vol. 9, no. 4, 2025.

[70] Y. Mei, W. Han, and K. Lin, “Intrusion detection for intelligent connected vehicles based on bidirectional Temporal Convolutional Network,” *IEEE Network*, vol. 38, no. 6, pp. 124–131, 2024.

[71] M. Deng, H. Xu, C. Sun, and Y. Kan, “Network intrusion detection model with multi-layer bi-directional temporal convolutional networks and multi-headed self-attention mechanisms,” in *2024 6th International Academic Exchange Conference on Science and Technology Innovation*, IEEE, 2024.

[72] T. Xu, Z. Wen, X. Zhao, Q. Hu, Y. Li, and C. Liu, “GTCN-G: A residual graph-temporal fusion network for imbalanced intrusion detection,” in *IEEE TrustCom 2025*, IEEE, 2025.

[73] C. Peng and Y. Zhang, “Industrial Internet of Things intrusion detection model integrating graph attention network and gated temporal convolutional network,” 2024.

[74] R. Robert, C. Sreelekshmi, and P. Keerthimon, “TCANet: Hybrid temporal convolutional and anomaly attention network with Bi-GRU for network intrusion detection,” in *2025 5th International Conference on Pervasive Computing and Social Networking*, IEEE, 2025.

[75] H. Wang, Y. Lin, and J. Ma, “DeepFlowGuard: Controller authentication in SDN via deep learning,” *Computer Networks*, vol. 212, p. 108059, 2022.

[76] A. Khan, S. Rafiq, and F. Qamar, “MITM-Defender: A real-time defense system against man-in-the-middle attacks in SDN,” in *Proc. of IEEE ICC*, pp. 1–6, IEEE, 2021.

[77] S. Malik and M. Habib, “Detection and mitigation of DoS attacks in SDN: Lightweight agent-based model,” *Security and Privacy*, vol. 4, no. 2, p. e116, 2021.

[78] R. Dungarani and S. N. Gujjar, “SDN security: Taming the wild west of network automation,” in *2024 IEEE International Conference on Blockchain and Distributed Systems Security*, pp. 1–6, IEEE, 2024.

[79] A. Mudgal, M. Singh, A. Verma, K. Sahoo, P. Townend, and M. H. Bhuyan, “Towards adaptive rule replacement for mitigating inference attacks in serverless SDN framework,” 2025.

[80] Y. Liang, C. Wang, and F. Xu, “A review of intrusion detection approaches in software defined networking,” *Journal of Information Security and Applications*, vol. 59, p. 102812, 2021.

[81] E. Benkhelifa, T. Welsh, and W. Hamouda, “AI for secure SDN: A survey,” *Journal of Network and Computer Applications*, vol. 170, p. 102781, 2020.

[82] Z. C. Johanyák and L. Göcs, "Edge computing security in SDN-enabled industrial IoT networks," *Gradus*, vol. 12, no. 2, 2025.

[83] H. Khalid and N. Aldabagh, "A survey on the latest intrusion detection datasets for software defined networking environments," *Engineering, Technology & Applied Science Research*, vol. 14, no. 2, pp. 13190–13196, 2024.

[84] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[85] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

[86] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[87] L. N. Smith and N. Topin, "Super-convergence: Very fast training of neural networks using large learning rates," *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, vol. 11006, pp. 369–386, 2019.

[88] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[89] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, "Deep learning approach for network intrusion detection in software defined networking," *Wireless Networks*, vol. 26, pp. 3171–3182, 2020.

[90] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, and F. Ahmad, "Network intrusion detection system: A systematic study of machine learning and deep learning approaches," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 1, p. e4150, 2021.

[91] M. Latah and L. Toker, "When deep learning meets security: A survey and a case study on network traffic classification," *Computer Networks*, vol. 171, p. 107116, 2020.

[92] D. García-Piedrabuena, M. Flores, and J. Méndez, "A comprehensive survey of network intrusion detection systems: Techniques, datasets and challenges," *IEEE Access*, vol. 12, pp. 42975–43012, 2024.

Generated using Research LaTeX Template, Version 1.0. Department of Computer Science and Engineering, Southeast University, Bangladesh.

The Original Version was Developed by Tashreef Muhammad on Friday 17th November, 2023

This report was generated on Monday 23rd February, 2026 at 23:55:01.