

AN ENCODER-ONLY TRANSFORMER WITH FEATURES- AS-TOKENS FOR ROBUST NETWORK INTRUSION DETECTION



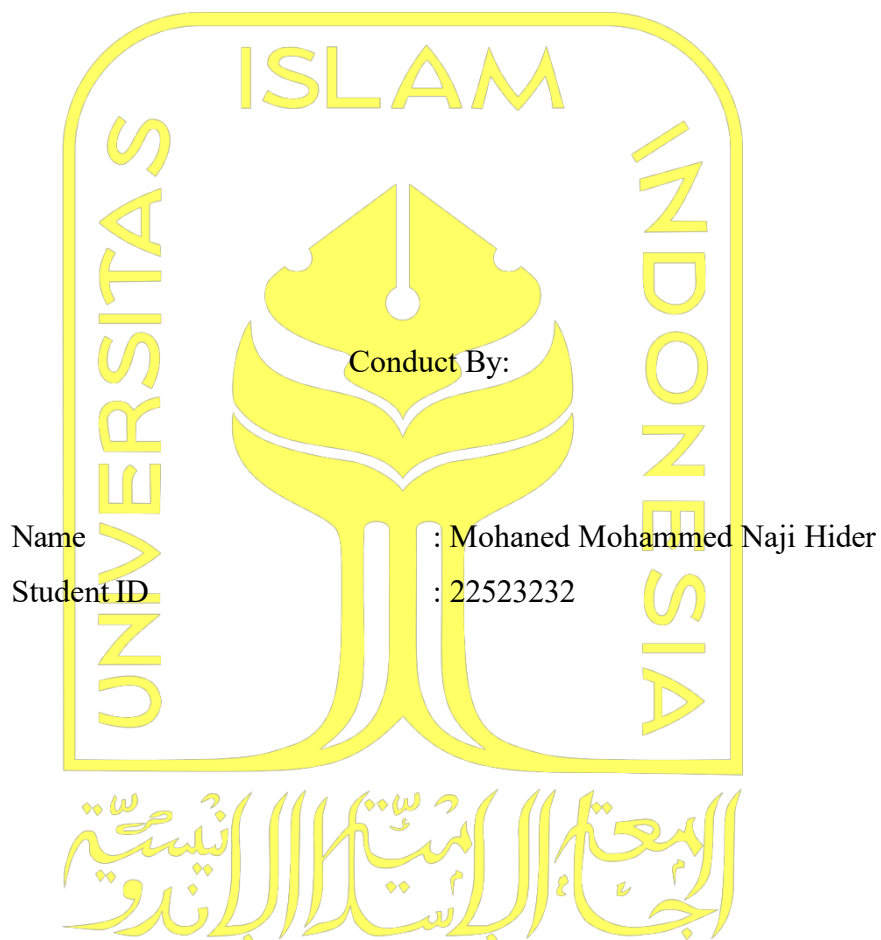
Conduct by:

Name : Mohaned Mohammed Naji Hider
Student ID : 22523232

**PROGRAM STUDI INFORMATIKA – PROGRAM SARJANA
FAKULTAS TEKNOLOGI INDUSTRI
UNIVERSITAS ISLAM INDONESIA
2026**

SUPERVISOR ENDORSEMENT PAGE

**AN ENCODER-ONLY TRANSFORMER WITH FEATURES-
AS-TOKENS FOR ROBUST NETWORK INTRUSION
DETECTION**

THESISYogyakarta, April 28th, 2026

Advisor,

Fayruz Rahma, S.T., M.Eng.

Arrie Kurniawardhani, S.Si., M.Kom.




EXAMINER ENDORSEMENT PAGE

AN ENCODER-ONLY TRANSFORMER WITH FEATURES- AS-TOKENS FOR ROBUST NETWORK INTRUSION DETECTION

THESIS

Has been defended in front of the examiners as one of the requirements to obtain a Bachelor of Informatics degree from the Undergraduate Program in Informatics at the Faculty of Industrial Technology, Universitas Islam Indonesia

Yogyakarta, April 28th, 2026

Chair

Fayruz Rahma, S.T., M.Eng.


Examiner 1

Erika Ramadhani, S.T., M.Eng.


Examiner 2

Kurniawan Dwi Irianto, S.T., M.Sc.

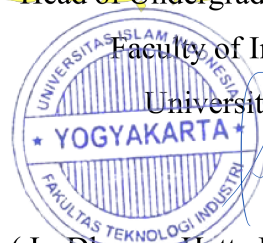


Acknowledged by,

Head of Undergraduate Program in Informatics

Faculty of Industrial Technology

Universitas Islam Indonesia



(Ir. Dhomas Hatta Fudholi, S.T., M.Eng., Ph.D.)

AUTHENTICITY STATEMENT

The undersigned:

Name : Mohaned Mohammed Naji Hider
Student ID : 22523232

Final project with title:

AN ENCODER-ONLY TRANSFORMER WITH FEATURES- AS-TOKENS FOR ROBUST NETWORK INTRUSION DETECTION

Stating that all components and contents in this final project are my own work. If in the future it is proven that some parts of this work are not my own work, the final project submitted as my own work is ready to be withdrawn and ready to bear any risks and consequences.

Thus this statement letter is made, hopefully it can be used properly.



gyakarta, April 28th, 2026

Mohaned Mohammed Naji Hider

DEDICATION

This thesis is dedicated to my family, whose unwavering love, sacrifice, and belief in my potential have carried me through every challenge of this journey.

To my supervisors, Mrs Fayruz Rahma and Mrs Arrie Kurniawardhani, whose patient guidance, intellectual rigour, and generous mentorship shaped both this work and my growth as a researcher.

The broader research community working to make our networks and digital infrastructure safer for everyone.

And to anyone who dares to seek knowledge far from home may the distance only strengthen your resolve.

MOTTO

﴿يَرْفَعُ اللَّهُ الَّذِينَ آمَنُوا مِنْكُمْ وَالَّذِينَ أُوتُوا الْعِلْمَ دَرَجَاتٍ﴾

[سورة المجادلة: 11]

“Allah will raise those who have believed among you and those who were given knowledge, by degrees.”

FOREWORD

All praise is due to Allah, the Most Gracious and the Most Merciful, for granting the strength, patience, and perseverance needed to complete this thesis. Peace and blessings be upon the Prophet Muhammad, who inspired the endless pursuit of knowledge.

This thesis, entitled “An Encoder-Only Transformer with Features-as-Tokens for Robust Network Intrusion Detection,” was written to fulfil one of the requirements for obtaining the Bachelor of Informatics degree from the Undergraduate Program in Informatics (Program Studi Informatika — Program Sarjana), Faculty of Industrial Technology (Fakultas Teknologi Industri), Universitas Islam Indonesia. The research investigates how the self-attention mechanism of Transformer architectures can be applied to the tabular, numerical data that characterises network flow classification, with the goal of building a lightweight and deployment-ready intrusion detection model. The standalone Transformer achieves this goal in terms of parameter count and inference speed; an ensemble variant combining the Transformer with tree-based models achieves higher macro F1 at the cost of increased inference latency, as discussed in Chapter IV.

The completion of this thesis would not have been possible without the support and guidance of many individuals. First and foremost, I wish to express my deepest gratitude to my supervisors, Ibu Fayruz Rahma and Ibu Arrie Kurniawardhani, whose expert guidance, constructive feedback, and patient mentorship have been instrumental at every stage of this research. Their encouragement to pursue methodological rigour — particularly the insistence on leak-free evaluation and honest reporting of results — has profoundly shaped both this thesis and my development as a researcher.

I am deeply grateful to my family for their unconditional love, sacrifice, and prayers throughout my years of study. Being far from home has never been easy, but their steadfast support has been a constant source of strength. I also thank my friends and classmates at UII for their companionship, encouragement, and the many productive discussions that helped refine my understanding of this work.

I extend my appreciation to the lecturers and staff of the Department of Informatics at Universitas Islam Indonesia for providing an excellent academic environment and the resources necessary to conduct this research. I am also grateful to the Canadian Institute for Cybersecurity for making the CICIDS2017 dataset publicly available, enabling reproducible research in the intrusion detection community.

I acknowledge that this thesis is far from perfect. Despite my best efforts, limitations remain — most notably the evaluation on a single dataset and the challenge of detecting extremely rare attack categories. I sincerely hope, however, that the contributions presented here — the leak-free evaluation framework, the lightweight Transformer architecture, and the comprehensive ablation analysis — will prove useful to future researchers working on network security and tabular deep learning. Any errors or shortcomings in this work are entirely my own.

May this research contribute, in whatever small measure, to building safer and more resilient digital infrastructure for all.

Yogyakarta, April 28th, 2026

Mohaned Mohammed Naji Hider

ABSTRACT

Existing Transformer-based Network Intrusion Detection Systems (NIDS) exhibit three interrelated weaknesses. First, most architectures process flow features as flat vectors, preventing the model from discovering pairwise inter-feature dependencies that are informative for distinguishing attack patterns. Second, a persistent methodological problem across the NIDS literature is data leakage: when preprocessing transformations are fitted on the full dataset before partitioning, test-set statistics contaminate the training pipeline, producing inflated and non-reproducible results. Third, no prior work has jointly addressed these two problems in a single lightweight architecture while also providing a comprehensive inference-efficiency and deployment-feasibility analysis. This thesis addresses all three weaknesses through one unified contribution: an encoder-only Transformer with features-as-tokens — which treats each flow-level numerical feature as an individual token so that self-attention can discover inter-feature dependencies — evaluated under a strictly leak-free protocol on the CICIDS2017 benchmark, with a full ablation study and inference-latency comparison against tree-based baselines.

The proposed architecture tokenises each scalar feature value into a d -dimensional embedding, prepends a learned column embedding, and applies a feature gate that suppresses uninformative inputs prior to tokenisation. Four pre-normalised encoder blocks with multi-head self-attention and feed-forward sub-layers process the token sequence, and an attention-pooling head aggregates the sequence into a single classification vector. To eliminate leakage, all preprocessing operators — including constant-feature removal, class-conditional imputation, outlier clipping, and quantile normalisation — are fitted exclusively on the training partition and applied without refitting to validation and test sets. The resulting model contains only 355,270 trainable parameters.

On the binary intrusion detection task (Normal vs. Attack), the Transformer achieves an F1-score of 0.9945, AUC-PR of 0.99947, and accuracy of 99.78%, confirming that the features-as-tokens paradigm is highly effective for binary flow classification. On the 15-class multi-class task, it achieves 99.72% accuracy and macro F1 of 0.7791; performance degrades only on attack categories with fewer than 10 training samples (Web Attack SQL Injection and Web Attack XSS), where insufficient discriminative signal prevents reliable detection. A four-model equal-weight ensemble — combining the Transformer's softmax outputs with those of Random Forest, XGBoost, and LightGBM — recovers macro F1 to 0.8848 by compensating for these rare-class failures. The standalone Transformer achieves an inference latency of 0.0676 ms per

flow (approximately 14,800 flows per second), making it $1.6\text{--}7.8\times$ faster than all tree-based baselines while maintaining competitive accuracy. In this thesis, the term “robust” refers specifically to three properties of the proposed system: reliable evaluation under a strictly leak-free protocol that prevents artificially inflated metrics, stable detection performance across most traffic categories despite severe class imbalance, and deployment-ready inference speed suitable for real-time high-throughput network environments.

Keywords: CICIDS2017, encoder-only Transformer, features-as-tokens, leak-free evaluation, network intrusion detection

GLOSSARY

Attention Mechanism	a component of the Transformer that computes weighted combinations of all input tokens, enabling the model to capture global pairwise inter-feature dependencies in a single computational step.
AUC-PR	Area Under the Precision-Recall Curve; an evaluation metric that measures classifier performance across all decision thresholds and is more informative than AUC-ROC under severe class imbalance.
AUC-ROC	Area Under the Receiver Operating Characteristic Curve; a metric summarising classifier performance by plotting true positive rate against false positive rate across all thresholds.
CICIDS2017	Canadian Institute for Cybersecurity Intrusion Detection System 2017; a widely adopted benchmark dataset for network intrusion detection containing labelled flow records spanning five capture days and fourteen distinct attack types.
Data Leakage	the inadvertent incorporation of test-set information into model training or preprocessing, typically caused by fitting transformations on the entire dataset before splitting, which artificially inflates reported performance metrics.
Encoder-Only Transformer	a Transformer variant employing only the encoder component, suitable for classification tasks; it produces contextualised token representations without the decoder’s autoregressive generation mechanism.
Features-as-Tokens	a paradigm for applying Transformers to tabular data in which each individual numerical feature value is projected into a token embedding, enabling self-attention to model pairwise inter-feature interactions without manual feature engineering.
Focal Loss	a modified cross-entropy loss function that applies a modulating factor to down-weight well-classified examples, concentrating the training signal on hard or minority-class samples.
Leak-Free Pipeline	a preprocessing and evaluation framework in which all data transformation operators are fitted exclusively on the training partition

and applied without refitting to validation and test partitions, preventing any information leakage.

- Macro F1-Score the unweighted mean of per-class F1-scores computed independently for each class regardless of frequency; the preferred metric for multi-class classification under severe class imbalance.
- NIDS Network Intrusion Detection System; a security mechanism that monitors network traffic in real time to identify malicious activities, policy violations, or unauthorised access attempts.
- Quantile Normalisation a preprocessing transformation that maps each feature to a uniform distribution using its empirical cumulative distribution function, reducing the influence of heavy-tailed distributions common in network flow data.
- Self-Attention a Transformer mechanism that computes scaled dot-product interactions between all pairs of input tokens, enabling the model to capture global dependencies among features without recurrence or convolution. Debug the process of tracing errors in program code.

TABLE OF CONTENTS

TITLE PAGE.....	i
SUPERVISOR ENDORSEMENT PAGE.....	ii
EXAMINER ENDORSEMENT PAGE	iii
AUTHENTICITY STATEMENT	iv
DEDICATION.....	v
MOTTO	vi
FOREWORD	vii
ABSTRACT	ix
GLOSSARY	xi
TABLE OF CONTENTS.....	xiii
LIST OF TABLES.....	xv
LIST OF FIGURES	xvi
CHAPTER I INTRODUCTION.....	1
1.1 Research Background.....	1
1.2 Problem Statement.....	2
1.3 Research Questions	3
1.4 Research Objectives	3
1.5 Research Scope and Limitations.....	4
1.6 Thesis Organisation	5
CHAPTER II LITERATURE REVIEW	6
2.1 Classical Machine Learning for IDS	6
2.2 Deep Learning for IDS	7
2.3 Transformer-Based IDS	9
2.4 Research Gap and Positioning.....	12
2.5 Theoretical Framework	13
2.5.1 Network Intrusion Detection Systems.....	13
2.5.2 The Transformer Architecture.....	15
2.5.3 Features-as-Tokens for Tabular Data	17
2.5.4 Class Imbalance in Network Traffic Data	19
2.5.5 The CICIDS2017 Dataset.....	21
CHAPTER III RESEARCH METHODOLOGY	26
3.1 Research Design Overview	27

	xiv
3.2 Dataset and Preprocessing Pipeline.....	28
3.3 Model Architecture.....	31
3.4 Training Procedure and Regularisation	35
3.5 Evaluation Protocol and Metrics	38
3.5.1 Metric Selection and Justification	38
3.5.2 Ablation Study Design	40
3.5.3 Baseline Model Configurations	41
3.5.4 Ensemble Design.....	42
3.5.5 Inference Time Measurement.....	42
3.5.6 Feature Importance Analysis.....	43
3.5.7 Error Analysis Protocol	43
CHAPTER IV RESULTS AND DISCUSSION.....	44
4.1 Preprocessing Outcomes	44
4.2 Binary Intrusion Detection Results	45
4.3 Multi-Class Intrusion Detection Results	49
4.4 Baseline Comparison.....	53
4.5 Ablation Study.....	56
4.6 Feature Importance and Interpretability	57
4.7 Error Analysis.....	60
CHAPTER V CONCLUSION AND SUGGESTION	62
5.1 Summary of Work	62
5.2 Key Findings	63
5.3 Research Contributions.....	63
5.4 Limitations.....	64
5.5 Future Work.....	65
REFERENCES	66
APPENDICES	74
APPENDIX A.....	74
APPENDIX B.....	79
APPENDIX C	87
APPENDIX D.....	90
APPENDIX E	97

LIST OF TABLES

Table 2.1 Comparison of Transformer-Based IDS Approaches	11
Table 2.2 CICIDS2017 Class Distribution in the Test Partition	22
Table 2.3 CICIDS2017 Features Removed During Preprocessing (N = 12 of 78).....	23
Table 2.4 CICIDS2017 Feature Descriptions: Final Input Feature Set (N = 66).....	23
Table 3.2 Complete Model Architecture Configuration.....	34
Table 3.3 Training Hyperparameters.....	38
Table 3.4 Ablation Study Variants and Rationale.....	40
Table 3.5 Baseline Model Configurations.....	41
Table 4.1 Final Dataset Configuration After Preprocessing	45
Table 4.2 Binary Intrusion Detection Results (Test Set)	46
Table 4.3 Multi-Class Intrusion Detection Results (Test Set).....	49
Table 4.4 Per-Class Detection Performance on the 15-Class Test Set.....	52
Table 4.5 Comparison of Transformer and Tree-Based Baselines (Multi-Class Test Set).....	53
Table 4.6 Binary Classification: Transformer vs. Tree-Based Baselines (Test Set).....	55
Table 4.7 Ablation Study Results (Multi-Class, 35% Subset, 7 Epochs)	56

LIST OF FIGURES

Figure 3.1 Overview of the leak-free preprocessing and training pipeline. All preprocessing operators are fitted exclusively on the training partition to prevent data leakage.....	28
Figure 3.2 Encoder-only Transformer architecture with features-as-tokens.....	32
Figure 4.1 Training dynamics for the binary classification task	47
Figure 4.2 Precision-recall curve for binary classification (AUC-PR = 0.99947)	48
Figure 4.3 ROC curve for binary classification (AUC-ROC = 0.99985)	48
Figure 4.4 Normalised confusion matrix for binary classification.....	49
Figure 4.5 Training dynamics for the multi-class classification task.....	50
Figure 4.6 Normalised confusion matrix for 15-class intrusion detection on the test set.....	51
Figure 4.7 Per-class F1-scores for 15-class intrusion detection	53
Figure 4.8 Moel comparison on multi-class macro F1.....	54
Figure 4.9 Inference latency comparison across all models.....	55
Figure 4.10 Ensemble vs Transformer per-class F1 improvement.....	56
Figure 4.11 Ablation study — component contributions to macro F1.....	57
Figure 4.12 Feature importance by embedding L2 norm (top 30 features)	58
Figure 4.13 Permutation feature importance (top 20 features)	59
Figure 4.14 Computational efficiency comparison across all models.....	60

CHAPTER I

INTRODUCTION

1.1 Research Background

The proliferation of internet-connected systems and the increasing sophistication of cyber threats have made network security a paramount concern for organisations worldwide. Network Intrusion Detection Systems (NIDS) serve as a critical line of defence by monitoring network traffic and identifying malicious activities that bypass perimeter security controls (Alkasassbeh & Al-Haj Baddar, 2023). As enterprise networks grow in scale and complexity, the volume of traffic requiring inspection has increased correspondingly, creating demand for detection systems that are both accurate and computationally efficient.

Classical machine learning approaches to intrusion detection—including Decision Trees, Random Forests, Support Vector Machines, and gradient-boosted ensembles—have established strong baselines on public benchmarks such as the CICIDS2017 dataset (Sharafaldin et al., 2018). However, these methods rely on handcrafted feature engineering and treat the input feature vector as a flat, unstructured entity, which limits their ability to capture higher-order interactions among heterogeneous flow attributes such as packet counts, byte volumes, flag distributions, and timing statistics. Furthermore, tree-based models often struggle to maintain robust performance under severe class imbalance, where benign traffic vastly outnumbers malicious flows and certain attack categories contain only a handful of training examples (He & Garcia, 2009).

The Transformer architecture (Vaswani et al., 2017), with its self-attention mechanism, offers a promising alternative. By treating each network flow feature as an independent token and processing them through an encoder, a Transformer can model global feature dependencies without the sequential bottleneck of recurrent networks. Influential tabular deep learning works such as FT-Transformer (Gorishniy et al., 2021) and SAINT (Somepalli et al., 2021) have demonstrated that this features-as-tokens paradigm achieves competitive performance on tabular data benchmarks. Transformer-based NIDS frameworks, including FlowTransformer (Shadrin et al., 2024), have demonstrated that attention mechanisms can achieve high accuracy on network intrusion datasets.

Despite these advances, several critical challenges remain unaddressed in existing Transformer-based IDS research. First, many studies fit preprocessing operations—such as feature scaling, imputation, or resampling—on the entire dataset before splitting into training

and test partitions, introducing data leakage that artificially inflates reported performance metrics (Camacho-Cogollo et al., 2024; Kaufman et al., 2011). A recent empirical assessment by Camacho-Cogollo et al. (2024) confirmed that many post-2020 IDS studies continue to fit preprocessing transformations on the full dataset, yielding performance estimates that do not reflect true deployment conditions. Second, most Transformer-based IDS architectures employ complex encoder–decoder designs that are unnecessarily heavyweight — examples include the Transformer-LSTM architecture of Wang et al. (2024), which requires decoder components unnecessary for per-flow classification tasks. Third, the problem of extreme class imbalance—where classes such as Heartbleed, Infiltration, and SQL Injection may contain fewer than ten training samples—is frequently acknowledged but inadequately addressed. These gaps motivate the present research.

1.2 Problem Statement

Existing approaches to Transformer-based network intrusion detection suffer from three interrelated limitations that undermine both the reliability of reported results and the practical viability of the proposed systems. First, the prevalent practice of fitting preprocessing transformations on the full dataset prior to partitioning introduces temporal data leakage, causing models to benefit from statistical information derived from the test distribution. This leads to performance estimates that do not generalise to real deployment scenarios where the model must process previously unseen traffic. Second, the extreme class imbalance inherent in network traffic datasets—where the CICIDS2017 dataset contains as few as 2 test samples for Heartbleed and 3 for SQL Injection against over 340,000 benign flows—poses a fundamental challenge for both model training and fair evaluation. Standard accuracy metrics become misleading under such conditions, and models that fail on rare but critical attack categories may be erroneously reported as highly effective. Third, many existing Transformer-based IDS architectures prioritise architectural novelty over deployment practicality, employing encoder–decoder designs or large parameter counts that are unsuitable for real-time, high-throughput network environments.

No prior study has combined a strictly leak-free evaluation protocol with a minimalist encoder-only Transformer using the features-as-tokens paradigm, paired with systematic imbalance-handling strategies, in a single reproducible framework evaluated on both binary and multi-class intrusion detection tasks. This thesis addresses this gap by proposing such a framework and comprehensively evaluating its performance, component contributions,

computational efficiency, and limitations on the CICIDS2017 benchmark. The term “robust” in the title of this thesis is used in a specific and bounded sense. First, the proposed system is evaluated under a strictly leak-free protocol, ensuring that reported metrics reflect genuine generalisation to unseen traffic rather than artificially inflated scores caused by preprocessing leakage. Second, the model maintains strong overall performance under severe class imbalance and achieves reliable detection across most traffic categories, although extremely rare classes remain challenging. Third, the architecture achieves an inference latency of 0.0676 ms per flow, making it suitable for real-time high-throughput network environments. Therefore, robustness in this thesis refers to reliability under methodological, imbalance-related, and deployment-efficiency constraints, rather than robustness against adversarial attacks, concept drift, or cross-dataset distribution shifts.

1.3 Research Questions

Based on the problem statement presented above, this thesis seeks to answer the following three research questions.

RQ1: How effectively can an encoder-only Transformer with features-as-tokens detect network intrusions in both binary and multi-class settings under strictly leak-free evaluation on the CICIDS2017 dataset?

RQ2: What is the contribution of each pipeline component—including quantile transformation, outlier clipping, focal loss, feature gating, and attention pooling—to the overall detection performance of the proposed model?

RQ3: How does the proposed Transformer compare to established tree-based baselines (Random Forest, XGBoost, and LightGBM) in terms of detection accuracy, inference speed, and computational efficiency, and what are the limitations of the proposed model in detecting extremely rare attack categories?

1.4 Research Objectives

Corresponding to each of the three research questions stated above, the objectives of this thesis are as follows.

1. To evaluate the detection performance of an encoder-only Transformer with features-as-tokens on binary and 15-class multi-class intrusion detection tasks using the CICIDS2017 dataset under a strictly leak-free evaluation protocol.

2. To quantify the individual contribution of each preprocessing and regularisation component through a systematic ablation study that removes one element at a time from the full pipeline.

3. To compare the proposed Transformer against Random Forest, XGBoost, and LightGBM baselines trained under the identical leak-free pipeline, assessing detection metrics, inference latency, training time, and parameter count, and to analyse per-class performance and failure modes on rare attack categories including investigation of ensemble recovery as a mitigating strategy.

1.5 Research Scope and Limitations

This thesis focuses on flow-level network intrusion detection using the CICIDS2017 dataset as the sole evaluation benchmark. All experiments were conducted on Kaggle using the GPU T4 x2 accelerator (2× NVIDIA Tesla T4, 16 GB VRAM each). Kaggle was selected as the computing environment for four reasons. First, the consolidated CICIDS2017 dataset comprises 2,830,743 flow records, a scale that exceeds the practical capacity of the available local laptop environment for repeated preprocessing, training, and evaluation. Second, Kaggle provides free access to NVIDIA Tesla T4 GPUs with 16 GB of VRAM each, which is sufficient to train the Transformer model on the full dataset without memory constraints. Third, Kaggle notebooks are version-controlled and publicly accessible, enabling other researchers to reproduce the experiments directly from the provided notebook links without additional local hardware setup. Fourth, running experiments on a standardised cloud environment reduces hardware-specific variability, ensuring that reported training and inference times reflect a consistent experimental baseline. The model is trained and evaluated in two task settings: binary classification (Normal vs. Attack) and 15-class multi-class classification using the official CICIDS2017 labels. The preprocessing pipeline, model architecture, and training procedure are fully scripted for reproducibility.

The following limitations are acknowledged. First, only a single dataset is used; cross-dataset generalisation to UNSW-NB15, CSE-CIC-IDS2018, or CIC-DDoS2019 is not evaluated. Second, all reported results are from single training runs without confidence intervals from multiple seeds. This decision was driven by practical computational constraints: the complete experimental notebook required 5 hours, 26 minutes, and 3 seconds to execute on the Kaggle GPU T4 x2 environment, while the Transformer training alone required approximately 23 minutes for the binary task and 24 minutes for the multi-class task. Because the study also includes baseline training, ensemble evaluation, ablation variants, inference-time

benchmarking, feature-importance analysis, and error analysis, repeating the full pipeline across multiple random seeds would be impractical within the available research timeline and Kaggle compute allocation. To improve reproducibility despite the single-run design, all random number generators — including Python’s built-in random module, NumPy, PyTorch CPU and CUDA, and the PYTHONHASHSEED environment variable — were fixed to a single deterministic seed of 42, supporting replication of the reported results from the provided code and data. Third, the study does not evaluate robustness under concept drift or adversarial perturbation. Fourth, extremely rare attack categories with fewer than ten training samples (Heartbleed, Infiltration, SQL Injection) remain fundamentally difficult for any supervised model, and this thesis does not propose specialised augmentation strategies for such classes. These limitations are discussed further in Chapter V.

1.6 Thesis Organisation

This thesis is organised into five chapters. Chapter I presents the research background, problem statement, research questions, objectives, and scope. Chapter II provides the literature review covering classical machine learning, deep learning, and Transformer-based IDS approaches, identifies the research gap, and concludes with the Theoretical Framework (Section 2.5) covering the foundational concepts of NIDS, the Transformer architecture, the features-as-tokens paradigm, class imbalance strategies, and the CICIDS2017 dataset. The background and theoretical foundations — covering NIDS taxonomy, the Transformer architecture, the features-as-tokens paradigm, class imbalance theory, and the CICIDS2017 dataset — are also presented within Chapter I. Chapter III describes the research methodology in detail, encompassing the leak-free preprocessing pipeline, the encoder-only Transformer architecture, the training procedure with regularisation strategies, baseline configurations, and the evaluation protocol. Chapter IV presents and discusses the experimental results, including binary and multi-class detection performance, baseline comparisons, ensemble analysis, ablation study, and computational efficiency analysis. Chapter V concludes the thesis by summarising the findings in relation to each research question, discussing the limitations of the study, and proposing directions for future work.

CHAPTER II

LITERATURE REVIEW

This chapter reviews the literature on network intrusion detection, tracing the progression from classical machine learning through deep learning to Transformer-based approaches. Section 2.1 examines classical machine learning for IDS. Section 2.2 surveys deep learning methods. Section 2.3 focuses on Transformer-based IDS architectures. Section 2.4 identifies the research gap addressed by this thesis. Section 2.5 presents the theoretical framework that underpins this work, covering the foundational concepts of NIDS taxonomy, the Transformer architecture, the features-as-tokens paradigm for tabular data, class imbalance strategies, and the CICIDS2017 dataset used for evaluation.

2.1 Classical Machine Learning for IDS

Research on machine learning-based network intrusion detection has evolved substantially over the past two decades, driven in large part by the availability of public benchmark datasets. Early investigations relied primarily on the KDD Cup 1999 dataset and its cleaned successor NSL-KDD (Tavallaei et al., 2009), which provided labelled network connection records for training supervised classifiers. Studies using these legacy benchmarks demonstrated that algorithms such as Decision Trees, Support Vector Machines (SVMs), and Random Forests could achieve high detection accuracy on known attack patterns. More recently, a convolutional neural network (CNN) with channel attention applied to the NSL-KDD dataset (Dua & Banka, 2024) — while technically a deep learning method — is noted here because it was evaluated primarily as a feature-weighting technique on a classical benchmark, demonstrating that architectural enhancements can improve classification even on datasets with well-known statistical limitations.

As the field matured, tree-based ensemble methods became the dominant paradigm for tabular network traffic classification. Random Forest, in particular, has been widely adopted due to its resilience to overfitting and its ability to handle mixed feature types. The IDRandom-Forest framework (Lee et al., 2024) proposed an advanced variant tailored specifically for real-time intrusion detection, achieving competitive accuracy while emphasising inference speed. Similarly, a multiscale approach combining variance-covariance subspace distance with

equalised loss (El-Ghamry et al., 2025) addressed the class imbalance problem through feature-space manipulation on NSL-KDD, UNSW-NB15, and CICIDS2017, demonstrating that careful feature engineering can partially mitigate minority-class underperformance. A linear one-class SVM approach for high-dimensional IDS data (Abdallah et al., 2024) further explored the boundary between classical and deep methods by embedding SVM within a recurrent feature extraction pipeline on CICIDS2017 and UNSW-NB15.

Despite these advances, classical machine learning approaches share several fundamental limitations. First, they depend heavily on manual feature engineering: domain experts must identify, construct, and select informative features, a process that is labour-intensive and dataset-specific. Second, tree-based and margin-based models treat the input feature vector as a flat, unstructured entity and cannot explicitly model higher-order interactions among heterogeneous flow attributes without additional feature construction. Third, most classical studies employed random train–test splits rather than temporal partitioning, meaning that preprocessing transformations were often fitted on the entire dataset—introducing data leakage that inflates reported metrics (Camacho-Cogollo et al., 2024). The empirical assessment by the latter study is notable for explicitly adopting a data leakage–free evaluation protocol, revealing that many previously reported results are overly optimistic. These limitations collectively motivated the transition toward deep learning methods capable of automatic feature representation learning.

2.2 Deep Learning for IDS

Building on the foundations established by classical approaches, deep learning methods introduced the capacity to learn hierarchical feature representations directly from raw or minimally preprocessed network data, reducing the dependence on manual feature engineering. Convolutional Neural Networks (CNNs) were among the first deep architectures applied to intrusion detection, typically by reshaping flow feature vectors into two-dimensional matrices and applying convolutional filters to extract spatial patterns. A transfer learning approach using optimised CNNs (Aljabri et al., 2023) demonstrated that CNN features learned on CICIDS2017 could be transferred to CSE-CIC-IDS2018 with minimal fine-tuning, suggesting that deep representations capture generalisable traffic characteristics. However, CNNs impose a spatial inductive bias that does not naturally align with the unordered structure of tabular flow features, limiting their expressiveness for this data modality.

Recurrent architectures—including Long Short-Term Memory (LSTM) networks, Bidirectional LSTMs, and Gated Recurrent Units—have also been widely explored for intrusion detection, motivated by the assumption that network flows exhibit temporal dependencies. The EESNN framework (Rahmani et al., 2024) combined DenseNet-169 with a self-attention-based Transformer module and a difficult-set sampling technique to address class imbalance on BoT-IoT and ToN-IoT, achieving strong results but requiring a complex multi-stage pipeline. A time-series IDS method integrating Stacked Sparse Autoencoders with Temporal Convolutional Networks and Bi-LSTM (Liu et al., 2024a) formulated intrusion detection as a sequence classification task on CICIDS2017 and UNSW-NB15, exploiting temporal ordering of packets within flows. Res-TranBiLSTM (Wang et al., 2023a) proposed a feature fusion architecture combining residual connections, Transformer attention, and Bi-LSTM for IoT intrusion detection on NSL-KDD, CICIDS2017, and MQTTset, reporting improved recall on minority classes. Similarly, improved network intrusion classification using attention-assisted Bi-LSTM with optimised sparse contractive autoencoders (Ullah et al., 2024) showed that attention mechanisms can help recurrent models focus on discriminative features in UNSW-NB15 and KDD'99.

Hybrid architectures that combine multiple deep learning components have also been proposed to address specific IDS challenges. A sequential deep learning framework (Layeghy et al., 2024) adopted a multi-stage pipeline with an anomaly detector and a novelty detector trained on CICIDS2017, designed to generalise to out-of-distribution traffic from CICIDS2018 and to detect zero-day botnet attacks. The Advancing IoT Security study (Altalhi et al., 2023) combined CNNs and Transformers for hybrid feature extraction on CICIDS2017 and CICDDoS2019, while the SCADA securing system (Rajasekaran et al., 2023) applied LSTM, residual networks, and Transformer components to industrial control system traffic.

Despite the representational power of these deep learning approaches, several persistent limitations remain. First, recurrent models impose a sequential inductive bias that assumes an inherent ordering among input elements—an assumption that does not hold for unordered tabular flow features, where each feature (packet count, byte volume, flag indicator) carries independent information without positional significance. Second, the stacking of recurrent, convolutional, and attention layers increases computational complexity and inference latency, making many hybrid architectures unsuitable for real-time deployment in high-throughput network environments. Third, the majority of reviewed studies continue to employ random train-test splits without enforcing leak-free preprocessing protocols, meaning that reported

performance figures may not reflect true generalisation to unseen traffic. These limitations have motivated the exploration of Transformer-only architectures that can model global feature interactions without recurrence, as discussed in the following section.

2.3 Transformer-Based IDS

More recently, Transformer architectures have been adapted for network intrusion detection, motivated by the self-attention mechanism’s ability to model global pairwise dependencies among input elements without recurrence (Vaswani et al., 2017). The FlowTransformer framework (Manocchio et al., 2024) introduced a flexible, GPT-style Transformer backbone for flow-based NIDS, evaluating multiple input encoding strategies on NSL-KDD, UNSW-NB15, and CSE-CIC-IDS2018. While FlowTransformer demonstrated the viability of large-scale Transformer models for flow analysis, it targets sequential flow processing rather than per-flow classification and does not enforce chronological data partitioning. In a more targeted application, the FT-Transformer was directly applied to intrusion detection (Bacevicius & Paulauskas, 2024) on CICIDS2017 and KDD Cup 1999, achieving strong multi-class results by treating each flow feature as an independent token. This study is closely aligned with the features-as-tokens paradigm but does not address temporal leakage or report inference latency.

Several studies have proposed hierarchical or optimisation-enhanced Transformer architectures. BBO-CFAT (Liu et al., 2024b) combined Biogeography-Based Optimisation for feature selection with a hierarchical Transformer classifier on CICIDS2017 and NSL-KDD, demonstrating that reducing input dimensionality before attention computation can improve both accuracy and efficiency. The Memory Flow Transformer (Li et al., 2025) introduced a novel memory mechanism to capture long-range dependencies across flow sequences on CICIDS2017 and NSL-KDD, though without leak-free evaluation or comparison against tree-based baselines. A time-aware Transformer incorporating n-gram frequency features (Wang et al., 2023b) was evaluated on ISCX2012 and CICIDS2017, integrating payload-level information with temporal attention—an approach limited to unencrypted traffic. The online transfer learning model using FT-Transformer with KSWIN-driven concept drift detection (Andresini et al., 2024) addressed the practical challenge of distribution shift in streaming IDS environments, though the focus was on concept drift rather than leak-free preprocessing.

BERT-based and transfer learning approaches represent another prominent strand of Transformer-based IDS research. An Empowering Network Security study (Abbas et al., 2024)

leveraged BERT embeddings combined with a multi-layer perceptron classifier to address class imbalance on CICIDS2017, UNSW-NB15, and NSL-KDD, reporting improved minority-class detection through contextual embeddings. The IDS-INT framework (Mohy-eddine et al., 2024) employed Vision Transformer-based transfer learning for imbalanced network traffic on UNSW-NB15, CICIDS2017, and NSL-KDD, converting tabular features into image representations before applying attention. Similarly, VINCENT (Koay et al., 2024) used knowledge distillation from a Vision Transformer teacher to a compact student model on NSL-KDD and UNSW-NB15, while a lightweight IoT IDS based on improved BERT-of-Theseus (Li et al., 2024) compressed BERT representations for deployment on CICIDS2017 and ToN-IoT. DDoSBERT (Li & Chen, 2025) adopted a novel text-classification framing by fine-tuning BERT for DDoS detection. These BERT-based approaches benefit from pre-trained language representations but typically incur high computational overhead and require non-trivial input transformations (image conversion, tokenisation of numerical features) that may not be optimal for tabular flow data.

Hybrid Transformer-CNN and Transformer-LSTM architectures have also been proposed to combine the strengths of attention and recurrence. A real-time NIDS based on Transformer-LSTM (Wang et al., 2024) paired a Transformer encoder with an LSTM decoder on NSL-KDD and CICIDS2017, achieving real-time detection via Apache Kafka integration but at the cost of an encoder-decoder architecture more complex than necessary for per-flow classification. The Intrusion Detection System using Transformer Encoder and CNN-BiLSTM (Singh & Kumar, 2024) combined Transformer attention with CNN-BiLSTM for software-defined network environments on NSL-KDD and CICIDS2017. For IoT-specific applications, a modified Transformer neural network (Ahmed, Kientz, & Kashef, 2023) integrated Transformer, LSTM, and RNN components for anomaly detection on ToN-IoT, while Transformer-based knowledge distillation (Alani et al., 2025) targeted explainability in IoT IDS on ToN-IoT, RT-IoT, and N-BaIoT. The Explainability study (Dutta et al., 2024) offered a packet-level approach on CICIDS2017 and UNSW-NB15 that prioritised interpretability, and the Intruder framework (Bhat et al., 2025) proposed a multi-module distributed IDS using Transformers for cloud environments on CICIDS2017 and NSL-KDD.

Despite the breadth of Transformer-based IDS research, a systematic review of the above studies reveals four recurring weaknesses. First, the vast majority employ random or stratified train-test splits rather than chronological partitioning, ignoring the temporal structure of network traffic and risking evaluation on data that temporally precedes the training set. Second,

preprocessing transformations such as feature scaling, imputation, and resampling are typically fitted on the entire dataset before splitting, introducing data leakage that inflates performance metrics beyond what would be observed in real deployment (Camacho-Cogollo et al., 2024; Kaufman et al., 2011). Third, most studies report accuracy and AUC-ROC as primary metrics without complementing these with AUC-PR or macro-averaged F1, which are more informative under the severe class imbalance characteristic of CICIDS2017 and similar datasets. Fourth, many architectures prioritise novelty and complexity (encoder–decoder designs, multi-stage pipelines, image-conversion preprocessing) over deployment practicality, rarely reporting inference latency or parameter counts. Table 2.1 summarises the key characteristics and limitations of representative Transformer-based IDS studies reviewed in this section.

Table 2.1 Comparison of Transformer-Based IDS Approaches

Author (Year)	Dataset	Architecture	Key Contribution	Limitation
Manocchio et al. (2024)	NSL-KDD, UNSW-NB15, CSE-CIC-IDS2018	FlowTransformer (GPT-style encoder)	Flexible tokenization framework for flow-based NIDS with multiple encoding strategies	No chronological split; does not enforce train-only preprocessing
Bacevicius & Paulauskas (2024)	CICIDS2017, KDD Cup 1999	FT-Transformer (Feature Tokenizer)	Applies FT-Transformer directly to IDS; achieves strong multi-class results	No chronological split; preprocessing leakage not addressed; single architecture
Liu et al. (2024b)	CICIDS2017, NSL-KDD	Hierarchical Transformer + BBO feature selection	Biogeography-based optimisation for feature selection before Transformer	Complex two-stage pipeline; no inference-time analysis; random split
Abbas et al. (2024)	CICIDS2017, UNSW-NB15, NSL-KDD	BERT embeddings + MLP classifier	Addresses class imbalance using BERT representations on multiple datasets	Pre-trained BERT is heavyweight; no inference latency reported
Mohyeddine et al. (2024)	UNSW-NB15, CICIDS2017, NSL-KDD	Transformer + transfer learning (ViT)	Transfer learning across datasets for imbalanced traffic	Vision Transformer requires image conversion; high computational cost
Li et al. (2025)	CICIDS2017, NSL-KDD	Memory Flow Transformer	Novel memory mechanism to capture long-range flow dependencies	No leak-free protocol described; no baseline

				comparison with tree models
Wang et al. (2024)	NSL-KDD, CICIDS2017	Transformer encoder + LSTM decoder	Combines attention for feature interaction with LSTM for temporal modelling	Encoder–decoder adds complexity; no ablation; random train-test split
Ahmed et al. (2023)	ToN-IoT	Modified Transformer + LSTM + RNN	Modifies standard Transformer for IoT anomaly detection	IoT-specific; no flow-level evaluation; no comparison with tree baselines
Wang et al. (2023b)	ISCX2012, CICIDS2017	Time-aware Transformer with n-gram frequency	Integrates n-gram payload features with temporal attention	Payload-based approach; not applicable to encrypted traffic; no leak-free split
Koay et al. (2024)	NSL-KDD, UNSW-NB15	Vision Transformer + knowledge distillation	Compact student model via distillation from ViT teacher	Requires image conversion of tabular data; legacy datasets only
Li & Chen (2025)	CIC-DDoS2019, custom DDoS sets	Fine-tuned BERT for text-classified DDoS flows	Novel text-classification framing for DDoS detection	Limited to DDoS only; does not generalise to multi-class IDS
This Thesis	CICIDS2017	Encoder-only Transformer (features-as-tokens)	Leak-free pipeline + lightweight deployment (355K params, 0.0676ms) + ablation study	Single dataset evaluation

2.4 Research Gap and Positioning

The literature reviewed in Sections 2.1 through 2.3 reveals three interrelated gaps that collectively limit the reliability and practical viability of current IDS research. The first gap concerns evaluation integrity: the pervasive use of random train–test splits and full-dataset preprocessing introduces temporal data leakage, producing performance estimates that do not generalise to real deployment conditions where the model must classify future, previously unseen traffic. The second gap relates to architectural efficiency: many Transformer-based IDS systems employ encoder–decoder architectures, multi-stage hybrid pipelines, or computationally expensive image-conversion preprocessing that yield unnecessarily high parameter counts and inference latencies, undermining their suitability for real-time detection in high-throughput network environments. The third gap involves evaluation completeness: the majority of reviewed studies rely on accuracy and AUC-ROC as primary metrics without reporting AUC-PR, macro-averaged F1, per-class analysis, or inference-time measurements,

making it impossible to assess their true performance on rare attack categories or their practical deployment feasibility.

This thesis addresses these gaps through four specific contributions. First, it introduces a strictly leak-free evaluation protocol that partitions the CICIDS2017 dataset chronologically into training, validation, and test sets before any preprocessing, and fits all transformation operators (constant-feature removal, class-aware imputation, outlier clipping, quantile scaling) exclusively on the training partition. Second, it proposes an encoder-only Transformer with features-as-tokens that models each flow feature as an independent token with a data-dependent feature-gating tokenizer and attention-based pooling, avoiding the complexity of encoder–decoder designs, image-conversion pipelines, or recurrent hybrid components. Third, the resulting architecture is lightweight and deployment-friendly, comprising only 355K trainable parameters with an inference latency of 0.0676 ms per flow—substantially faster than all tree-based baselines. Fourth, the evaluation is comprehensive, including AUC-PR and macro-averaged F1 alongside AUC-ROC and accuracy, per-class analysis for all 15 classes, a five-variant ablation study, baseline comparisons with Random Forest, XGBoost, and LightGBM under identical preprocessing, ensemble analysis, and computational efficiency benchmarking.

In summary, while prior work has demonstrated the promise of Transformer architectures for network intrusion detection, no existing study combines a strictly leak-free chronological evaluation protocol with a minimalist encoder-only features-as-tokens design, systematic imbalance handling, and comprehensive efficiency-oriented evaluation in a single reproducible framework. This thesis fills that gap by proposing and rigorously evaluating such a framework on the CICIDS2017 benchmark, providing a reliable and reproducible baseline for future Transformer-based IDS research.

2.5 Theoretical Framework

2.5.1 Network Intrusion Detection Systems

A Network Intrusion Detection System (NIDS) is a security mechanism designed to monitor and analyse network traffic in order to identify malicious activities, policy violations, or unauthorised access attempts (Alkasassbeh & Al-Haj Baddar, 2023). As enterprise networks have grown in scale and complexity, the volume and sophistication of cyber threats have increased correspondingly, making intrusion detection a critical component of modern cybersecurity infrastructure. Unlike firewalls, which enforce access control policies at network boundaries, a NIDS operates by inspecting the content and patterns of traffic flowing across

the network, enabling the detection of threats that have already bypassed perimeter defences (Khraisat et al., 2019).

Intrusion detection approaches are broadly classified into two categories: signature-based detection and anomaly-based detection (Liao et al., 2013). Signature-based systems, also referred to as misuse detection systems, operate by maintaining a database of known attack patterns and comparing observed traffic against these predefined signatures. When a match is identified, an alert is generated. This approach offers high detection accuracy for known threats and produces low false positive rates; however, it is fundamentally limited by its inability to detect novel or zero-day attacks whose signatures have not yet been catalogued (Axelsson, 2000). Anomaly-based systems, in contrast, establish a statistical model of normal network behaviour during a training phase and subsequently flag any activity that deviates significantly from this baseline as potentially malicious. The principal advantage of anomaly-based detection is its capacity to identify previously unseen attack patterns, but this capability comes at the cost of higher false positive rates, since legitimate but unusual network behaviour may be misclassified as intrusive (Khraisat et al., 2019). Hybrid approaches that combine elements of both paradigms have also been proposed, seeking to leverage the precision of signature matching and the generality of anomaly detection within a single framework (Alkasassbeh & Al-Haj Baddar, 2023).

Several persistent challenges confront the design and deployment of effective NIDS. First, real-world network traffic exhibits severe class imbalance: benign traffic vastly outnumbers malicious flows, and certain attack categories appear only rarely, making it difficult for detection models to learn reliable representations of minority classes (He & Garcia, 2009). Second, the risk of data leakage arises when preprocessing operations—such as feature scaling or imputation—are fitted on the entire dataset rather than on the training partition alone, leading to artificially inflated performance metrics that do not reflect real deployment conditions (Kaufman et al., 2011). Third, network traffic features frequently exhibit heavy-tailed distributions with extreme outliers, particularly in byte counts and flow durations, which can destabilise model training if left unaddressed. Fourth, the continuously evolving nature of cyber attacks means that models trained on historical data may encounter concept drift as adversaries adapt their strategies over time (Perez et al., 2025).

These challenges have motivated a shift from manually crafted rule-based systems toward machine learning-based approaches. Early machine learning methods for NIDS relied on supervised classifiers such as Decision Trees, Random Forests, Support Vector Machines,

and k-Nearest Neighbours, evaluated primarily on legacy benchmarks including KDD Cup 1999 and its refined successor NSL-KDD (Tavallae et al., 2009). While these classical methods established strong baselines, they often struggled to model higher-order interactions among heterogeneous flow features and to maintain robust performance under severe class imbalance and distribution shift (Moustafa & Slay, 2015). The subsequent introduction of more representative datasets—most notably the CICIDS2017 dataset (Sharafaldin et al., 2018) and UNSW-NB15 (Moustafa & Slay, 2015)—together with advances in deep learning, has driven the development of increasingly sophisticated detection models that can automatically learn feature representations from raw network data. The following sections examine the specific architectural innovations—particularly the Transformer and the features-as-tokens paradigm—that underpin the approach proposed in this thesis.

2.5.2 The Transformer Architecture

The Transformer architecture was introduced by Vaswani et al. (2017) as a sequence-to-sequence model that relies entirely on attention mechanisms, dispensing with the recurrent and convolutional layers that had previously dominated neural sequence modelling. The central innovation of the Transformer is the self-attention mechanism, which enables each element in an input sequence to attend directly to every other element in a single computational step. This design permits the model to capture long-range dependencies without the sequential processing bottleneck inherent in recurrent neural networks (RNNs), resulting in substantially improved parallelisation and training efficiency.

In the self-attention mechanism, each input token is linearly projected into three distinct vector representations: a query Q , a key K , and a value V . The attention output is computed by measuring the compatibility between queries and keys via a scaled dot product, applying a softmax normalisation to obtain attention weights, and then using these weights to form a weighted sum of the value vectors. Formally, the scaled dot-product attention is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k denotes the dimensionality of the key vectors (Vaswani et al., 2017). The division by $\sqrt{d_k}$ prevents the dot products from growing excessively large in magnitude as the dimensionality increases, which would push the softmax function into regions of extremely small gradients and hinder training convergence.

To enable the model to attend to information from different representation subspaces simultaneously, Vaswani et al. (2017) introduced multi-head attention. Rather than computing a single attention function, the model runs h parallel attention heads, each operating on a linearly projected lower-dimensional subspace of dimension $d_k = \frac{d}{h}$. The outputs of all heads are concatenated and projected through a final linear layer to produce the multi-head attention output. This mechanism allows different heads to specialise in capturing distinct types of relationships among input elements—for instance, one head might capture short-range feature correlations while another captures global statistical dependencies.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

where h is the number of attention heads, W_i^Q , W_i^K , and W_i^V are learned projection matrices for the i -th head, and W^O is the output projection matrix.

Each Transformer encoder block consists of two sub-layers: a multi-head self-attention (MHSA) sub-layer and a position-wise feed-forward network (FFN). The FFN applies two linear transformations with a non-linear activation function between them; in modern implementations, the Gaussian Error Linear Unit (GELU) activation (Hendrycks & Gimpel, 2016) is commonly preferred over the original ReLU due to its smoother gradient properties. Both sub-layers employ residual connections (He et al., 2016), whereby the input to each sub-layer is added back to its output, enabling gradient flow through deep stacks of encoder blocks. Layer normalisation (Ba et al., 2016) is applied to stabilise the distribution of activations across layers.

A critical design choice concerns the placement of layer normalisation relative to the sub-layer computation. The original Transformer applies normalisation after the residual addition (post-normalisation), but subsequent research has demonstrated that applying normalisation before the sub-layer (pre-normalisation) improves training stability and reduces sensitivity to learning rate selection, particularly in deeper architectures (Xiong et al., 2020). Pre-normalisation ensures that the inputs to the attention and feed-forward computations maintain a consistent scale, which is especially beneficial when the model processes heterogeneous network flow features that may span vastly different numerical ranges even after preprocessing. The encoder-only Transformer proposed in this thesis adopts the pre-normalisation variant.

The original Transformer comprises both an encoder and a decoder, designed for sequence-to-sequence tasks such as machine translation where the output is itself a variable-length sequence. However, for classification tasks—including intrusion detection—only the encoder component is necessary. The encoder transforms the input sequence into a set of contextualised representations, and a lightweight classification head (typically a small multi-layer perceptron) maps a pooled summary of these representations to class probabilities. This encoder-only design eliminates the decoder’s cross-attention layers, auto-regressive generation mechanism, and positional masking, resulting in a substantially more compact and efficient architecture. Models such as BERT (Devlin et al., 2019) have demonstrated that encoder-only Transformers achieve strong performance on a wide range of classification benchmarks, and this design philosophy has been extended to tabular data through architectures like FT-Transformer (Gorishniy et al., 2021) and SAINT (Somepalli et al., 2021).

Transformers offer several advantages over recurrent architectures for tabular data classification. First, network flow records consist of unordered sets of features—such as packet counts, byte counts, flag indicators, and flow durations—that do not possess an inherent sequential structure. RNNs impose a left-to-right processing order that introduces an artificial inductive bias inappropriate for such data (Ahmed et al., 2023). Second, self-attention computes pairwise interactions among all features in a single step with $O(n^2)$ complexity in sequence length, compared to the $O(n)$ sequential steps required by RNNs, enabling full parallelisation on modern GPU hardware. Third, the absence of recurrent state eliminates vanishing and exploding gradient problems that can destabilise RNN training on high-dimensional input vectors. These properties make the encoder-only Transformer a natural fit for flow-based intrusion detection, where each network flow is represented as a fixed-length feature vector that benefits from global, order-invariant feature interaction modelling.

2.5.3 Features-as-Tokens for Tabular Data

Traditional approaches to applying neural networks on tabular data typically concatenate all input features into a single flat vector, which is then processed by fully connected layers. This design treats the feature vector as a monolithic input and relies entirely on the hidden layers to disentangle feature interactions. The features-as-tokens paradigm offers a fundamentally different strategy: each individual feature in the input vector is independently projected into a d -dimensional embedding, producing a sequence of F tokens (one per feature) that is subsequently processed by a Transformer encoder (Gorishniy et al., 2021). This

formulation allows the self-attention mechanism to explicitly model pairwise interactions among all features, enabling the model to discover complex, higher-order dependencies that flat-vector architectures may fail to capture.

Several architectures have established the features-as-tokens approach for tabular learning. TabTransformer (Huang et al., 2020) introduced the idea of contextualising categorical features through Transformer encoder layers while keeping numerical features unembedded. FT-Transformer (Gorishniy et al., 2021) extended this paradigm to all feature types by projecting both categorical and numerical features into token embeddings, demonstrating that a straightforward features-as-tokens design achieves competitive or superior performance compared to gradient-boosted tree ensembles across a variety of tabular benchmarks. SAINT (Somepalli et al., 2021) further enriched the paradigm by introducing both column-wise (inter-feature) and row-wise (inter-sample) attention, enabling the model to leverage relationships not only across features within a single sample but also across samples within a mini-batch. Collectively, these works demonstrate that employing the features-as-tokens approach is an effective and general strategy for applying Transformer architectures to heterogeneous tabular data.

The encoder-only Transformer proposed in this thesis adopts and extends the features-as-tokens paradigm through a feature-gating tokenizer. Prior to projecting each feature into a token embedding, a data-dependent gating mechanism modulates the raw input vector to suppress uninformative or noisy features. For an input flow vector $x \in R^F$, a small multi-layer perceptron generates a gating vector:

$$g(x) = \sigma(W_2 \tanh(W_1 x))$$

where W_1 and W_2 are learnable weight matrices, $\tanh(\cdot)$ provides a bounded non-linear transformation, and σ denotes the sigmoid function, which constrains each gate value to the interval $(0, 1)$. The gated input is then computed as:

$$\tilde{x} = x \odot g(x)$$

each scalar feature $\tilde{x}_j$ is projected into a d -dimensional token via a shared linear layer $v(\cdot)$ and summed with a learnable column embedding c_j that encodes feature identity, yielding the token $t_j = v(\tilde{x}_j) + c_j, j = 1, \dots, F$. The column embeddings serve a role analogous to positional encodings in natural language Transformers, informing the attention layers about which feature each token represents, since the set of flow features has no inherent ordering.

Once the input flow is represented as a sequence of F feature tokens, the Transformer encoder's multi-head self-attention mechanism computes pairwise interactions among all

tokens simultaneously. This enables the model to capture dependencies such as the relationship between packet byte counts and flow duration, or between flag distributions and port-scan indicators, without requiring the analyst to manually engineer interaction features. Each attention head may specialise in different types of feature relationships, and the stacking of multiple encoder blocks allows the model to learn progressively more abstract and compositional feature interactions across layers (Vaswani et al., 2017).

The features-as-tokens approach offers a significant advantage over traditional feature engineering pipelines commonly employed in machine learning-based NIDS. Manual feature engineering requires domain expertise to identify and construct informative interaction terms, polynomial expansions, or ratio features, and the resulting feature set is typically fixed and dataset-specific. In contrast, the features-as-tokens formulation with self-attention learns feature interactions directly from the data in an end-to-end manner, adapting automatically to the statistical structure of the training distribution. The feature-gating mechanism further enhances this by allowing the model to learn, during training, which input features are most relevant for discrimination, effectively performing soft feature selection before the attention computation. This combination of data-driven feature interaction learning and input-adaptive gating makes the features-as-tokens Transformer particularly well suited for network intrusion detection, where the relevance of individual flow features varies substantially across different attack categories.

2.5.4 Class Imbalance in Network Traffic Data

Network traffic datasets exhibit severe class imbalance as an intrinsic characteristic of real-world network operations. In any production environment, the overwhelming majority of network flows are benign, while malicious traffic constitutes only a small fraction of the total volume. Within the malicious fraction itself, certain attack types occur far less frequently than others, creating a long-tailed distribution across categories (He & Garcia, 2009). In the CICIDS2017 dataset used in this thesis, for example, benign flows account for approximately 80% of all samples in the binary setting, while in the 15-class multi-class setting, class sizes in the test partition range from over 340,000 flows for the BENIGN class to as few as 2 flows for Heartbleed and 3 for SQL Injection (Sharafaldin et al., 2018). This extreme imbalance poses fundamental challenges for supervised learning, as models trained with standard loss functions tend to optimise for majority-class accuracy at the expense of minority-class detection.

When class distributions are heavily skewed, standard evaluation metrics can present a misleading picture of model performance. Overall accuracy, for instance, can exceed 99% simply by predicting the majority class for every sample, despite failing to detect any attacks. Similarly, the Area Under the Receiver Operating Characteristic curve (AUC-ROC) can remain high even when the model produces many false positives among the minority class, because the large number of true negatives from the majority class dominates the false positive rate calculation (Saito & Rehmsmeier, 2015). The Area Under the Precision–Recall curve (AUC-PR) provides a more informative assessment under such conditions, as it focuses exclusively on the performance of the positive (minority) class and is not inflated by the abundance of true negatives. For multi-class settings, macro-averaged F1-score is preferred because it assigns equal weight to every class regardless of its frequency, thereby ensuring that the evaluation reflects the model’s ability to detect rare attack categories (Saito & Rehmsmeier, 2015). This thesis adopts both AUC-PR and macro-averaged F1 as primary evaluation metrics alongside AUC-ROC and accuracy.

Several strategies have been proposed to mitigate the effects of class imbalance in machine learning. Undersampling reduces the number of majority-class samples to achieve a more balanced training distribution, although careless undersampling risks discarding informative boundary samples (He & Garcia, 2009). Oversampling techniques such as SMOTE (Chawla et al., 2002) generate synthetic minority samples, but they may introduce noise in high-dimensional feature spaces. Cost-sensitive learning modifies the loss function to assign higher penalties to misclassifications of minority classes, either through explicit class weights or through specialised loss formulations.

Among these loss formulations, focal loss (Lin et al., 2017) is particularly relevant for imbalanced classification because it down-weights easy, well-classified examples and concentrates the training signal on hard examples. It is defined as:

$$\mathcal{L}_{\text{focal}} = -\alpha_t(1 - p_t)^\gamma \log(p_t)$$

where p_t is the predicted probability assigned to the true class, α_t is an optional class-balancing factor, and γ is the focusing parameter. The modulating factor $(1 - p_t)^\gamma$ reduces the loss contribution of examples that the model already classifies correctly with high confidence, thereby placing greater emphasis on difficult or minority-class examples.

At the data-sampling level, weighted random sampling assigns per-sample drawing probabilities inversely proportional to class frequencies, ensuring that each mini-batch contains a more balanced class representation during training (Cui et al., 2019). The specific imbalance-

handling strategies adopted in this thesis, including controlled undersampling, weighted random sampling, and focal loss, are described in the context of the training procedure in Chapter III, Section 3.4. In all settings, rebalancing operations are performed exclusively on the training partition; validation and test distributions remain unmodified to ensure that reported metrics reflect the true class distribution encountered in deployment.

2.5.5 The CICIDS2017 Dataset

The CICIDS2017 dataset was developed by the Canadian Institute for Cybersecurity at the University of New Brunswick and has become one of the most widely adopted benchmarks for evaluating network intrusion detection systems (Sharafaldin et al., 2018). The dataset was generated using a controlled testbed environment that simulated realistic network traffic over a five-day period in July 2017. A dedicated traffic generation framework, the B-Profile system, was employed to produce naturalistic benign background traffic based on statistical profiles of human user behaviour, including web browsing, email, file transfers, and SSH connections. Simultaneously, a range of contemporary attack scenarios were executed against the network, with each day focusing on different attack categories. The resulting dataset captures both benign and malicious flows in the form of labelled NetFlow-style records, providing ground-truth annotations suitable for supervised classification.

The dataset encompasses benign traffic alongside 14 distinct attack categories, yielding a total of 15 classes for multi-class classification. The attack types span several major threat families: volumetric denial-of-service attacks (DoS Hulk, DoS GoldenEye, DoS Slowhttptest, DoS slowloris, and DDoS), reconnaissance activities (PortScan), credential-based attacks (FTP-Patator, SSH-Patator), web application attacks (Web Attack–Brute Force, Web Attack–XSS, Web Attack–SQL Injection), botnet command-and-control traffic (Bot), vulnerability exploitation (Heartbleed), and network infiltration (Infiltration). This diversity of attack types makes CICIDS2017 particularly suitable for evaluating multi-class detection capabilities, as a model must simultaneously distinguish among attack families with very different traffic signatures.

CICIDS2017 has become a standard benchmark for NIDS research for several reasons. First, it provides realistic traffic patterns generated from human-behavioural profiles, unlike earlier synthetic datasets such as KDD Cup 1999 (UCI KDD Archive, 1999) and its refined version NSL-KDD (Tavallae et al., 2009), which contain well-documented statistical artefacts. Second, it covers a broad range of modern attack types that reflect contemporary

threat landscapes. Third, the dataset includes rich flow-level features—originally comprising 78 numerical attributes describing packet counts, byte volumes, inter-arrival times, flag counts, flow durations, and other statistical summaries—that provide sufficient discriminative information for both classical and deep learning models. Modern benchmarking studies on Transformer-based IDS, including FlowTransformer (Shadrin et al., 2024), consistently evaluate on CICIDS2017, confirming its status as a primary reference dataset.

Despite its strengths, CICIDS2017 presents significant challenges due to its extreme class imbalance, as illustrated in Figure 3.1. Table 2.2 shows the class distribution in the test partition used in this thesis, computed from the chronological 70/15/15 split described in Chapter III. Benign flows account for 80.3% of all test samples, while four attack classes contain fewer than 100 samples each: Heartbleed (2), SQL Injection (3), Infiltration (5), and XSS (98). This long-tailed distribution makes macro-averaged metrics essential for honest evaluation and renders reliable learning on the rarest classes inherently difficult regardless of model architecture.

Table 2.2 CICIDS2017 Class Distribution in the Test Partition

Class	Test Samples	Proportion (%)
BENIGN	340,965	80.30
Bot	295	0.07
DDoS	19,204	4.52
DoS GoldenEye	1,544	0.36
DoS Hulk	34,661	8.16
DoS Slowhttptest	825	0.19
DoS slowloris	869	0.20
FTP-Patator	1,191	0.28
Heartbleed	2	<0.01
Infiltration	5	<0.01
PortScan	23,839	5.61
SSH-Patator	885	0.21
Web Attack – Brute Force	226	0.05
Web Attack – Sql Injection	3	<0.01
Web Attack – XSS	98	0.02
Total	424,612	100.00

Table 2.3 CICIDS2017 Features Removed During Preprocessing (N = 12 of 78)

No	Feature Name	Reason for Removal
1	Bwd Avg Bulk Rate	Stage 2b — zero variance (constant 0 in all training flows)
2	Bwd Avg Bytes/Bulk	Stage 2b — zero variance (constant 0 in all training flows)
3	Bwd Avg Packets/Bulk	Stage 2b — zero variance (constant 0 in all training flows)
4	Bwd PSH Flags	Stage 2b — zero variance (constant 0 in all training flows)
5	Bwd URG Flags	Stage 2b — zero variance (constant 0 in all training flows)
6	Fwd Avg Bulk Rate	Stage 2b — zero variance (constant 0 in all training flows)
7	Fwd Avg Bytes/Bulk	Stage 2b — zero variance (constant 0 in all training flows)
8	Fwd Avg Packets/Bulk	Stage 2b — zero variance (constant 0 in all training flows)
9	CWE Flag Count	Stage 2e — became constant after quantile normalisation
10	ECE Flag Count	Stage 2e — became constant after quantile normalisation
11	Fwd URG Flags	Stage 2e — became constant after quantile normalisation
12	RST Flag Count	Stage 2e — became constant after quantile normalisation

Table 2.4 CICIDS2017 Feature Descriptions: Final Input Feature Set (N = 66)

No	Feature Name	Type	Description
1	Destination Port	Numeric (integer)	TCP/UDP destination port; highly discriminative for service-specific attacks such as FTP-Patator (port 21) and SSH-Patator (port 22).
2	Flow Duration	Numeric (float)	Total elapsed time of the flow in microseconds, from first to last observed packet.
3	Total Fwd Packets	Numeric (integer)	Total packet count in the forward direction (client to server).
4	Total Backward Packets	Numeric (integer)	Total packet count in the backward direction (server to client).
5	Total Length of Fwd Packets	Numeric (float)	Aggregate payload byte volume across all forward-direction packets.
6	Total Length of Bwd Packets	Numeric (float)	Aggregate payload byte volume across all backward-direction packets.
7	Fwd Packet Length Max	Numeric (float)	Largest payload size (bytes) among forward-direction packets.
8	Fwd Packet Length Min	Numeric (float)	Smallest payload size (bytes) among forward-direction packets.
9	Fwd Packet Length Mean	Numeric (float)	Arithmetic mean of payload sizes across all forward-direction packets.
10	Fwd Packet Length Std	Numeric (float)	Standard deviation of payload sizes among forward-direction packets; low values suggest uniform-size automated traffic.

11	Bwd Packet Length Max	Numeric (float)	Largest payload size (bytes) among backward-direction packets.
12	Bwd Packet Length Min	Numeric (float)	Smallest payload size (bytes) among backward-direction packets.
13	Bwd Packet Length Mean	Numeric (float)	Arithmetic mean of payload sizes across all backward-direction packets.
14	Bwd Packet Length Std	Numeric (float)	Standard deviation of payload sizes among backward-direction packets.
15	Flow Bytes/s	Numeric (float)	Total bytes (forward + backward) transferred per second over the flow duration.
16	Flow Packets/s	Numeric (float)	Total packets (forward + backward) transmitted per second over the flow duration.
17	Flow IAT Mean	Numeric (float)	Mean inter-arrival time (μ s) between consecutive packets across the entire flow.
18	Flow IAT Std	Numeric (float)	Standard deviation of inter-arrival times; high variance is associated with bursty traffic patterns.
19	Flow IAT Max	Numeric (float)	Maximum inter-arrival time (μ s) between any two consecutive packets in the flow.
20	Flow IAT Min	Numeric (float)	Minimum inter-arrival time (μ s) between any two consecutive packets in the flow.
21	Fwd IAT Total	Numeric (float)	Sum of all inter-arrival times between consecutive forward packets.
22	Fwd IAT Mean	Numeric (float)	Mean inter-arrival time (μ s) between consecutive forward packets.
23	Fwd IAT Std	Numeric (float)	Standard deviation of inter-arrival times among forward packets.
24	Fwd IAT Max	Numeric (float)	Maximum inter-arrival time (μ s) between any two consecutive forward packets.
25	Fwd IAT Min	Numeric (float)	Minimum inter-arrival time (μ s) between any two consecutive forward packets.
26	Bwd IAT Total	Numeric (float)	Sum of all inter-arrival times between consecutive backward packets.
27	Bwd IAT Mean	Numeric (float)	Mean inter-arrival time (μ s) between consecutive backward packets.
28	Bwd IAT Std	Numeric (float)	Standard deviation of inter-arrival times among backward packets.
29	Bwd IAT Max	Numeric (float)	Maximum inter-arrival time (μ s) between any two consecutive backward packets.
30	Bwd IAT Min	Numeric (float)	Minimum inter-arrival time (μ s) between any two consecutive backward packets.
31	Fwd PSH Flags	Numeric (integer)	Number of TCP PSH flags in forward segments, prompting immediate buffer delivery.

32	Fwd Header Length	Numeric (integer)	Total header size (bytes) summed across all forward packets.
33	Bwd Header Length	Numeric (integer)	Total header size (bytes) summed across all backward packets.
34	Fwd Packets/s	Numeric (float)	Rate of forward packet transmission in packets per second.
35	Bwd Packets/s	Numeric (float)	Rate of backward packet transmission in packets per second.
36	Min Packet Length	Numeric (float)	Smallest packet size (bytes) observed in the flow (all directions).
37	Max Packet Length	Numeric (float)	Largest packet size (bytes) observed in the flow (all directions).
38	Packet Length Mean	Numeric (float)	Arithmetic mean of packet sizes across all packets in the flow.
39	Packet Length Std	Numeric (float)	Standard deviation of packet sizes across the entire flow.
40	Packet Length Variance	Numeric (float)	Variance of packet sizes; mathematically the square of Packet Length Std.
41	FIN Flag Count	Numeric (integer)	Count of TCP FIN (connection teardown) flags; normally 1 per legitimate connection.
42	SYN Flag Count	Numeric (integer)	Count of TCP SYN flags; elevated counts are a hallmark signature of SYN flood attacks.
43	PSH Flag Count	Numeric (integer)	Total TCP PSH flags across all packets and directions in the flow.
44	ACK Flag Count	Numeric (integer)	Count of TCP ACK flags; reflects connection establishment and session activity.
45	URG Flag Count	Numeric (integer)	Count of TCP URG flags; rarely set in legitimate traffic, anomalous in attack flows.
46	Down/Up Ratio	Numeric (float)	Ratio of backward-direction bytes to forward-direction bytes; asymmetric values indicate amplification or exfiltration patterns.
47	Average Packet Size	Numeric (float)	Mean size (bytes) across all packets, computed as total bytes divided by total packet count.
48	Avg Fwd Segment Size	Numeric (float)	Mean size (bytes) of TCP segments in the forward direction.
49	Avg Bwd Segment Size	Numeric (float)	Mean size (bytes) of TCP segments in the backward direction.
50	Fwd Header Length.1	Numeric (integer)	Duplicate of Fwd Header Length retained by CICFlowMeter; included for completeness of the original feature set.
51	Subflow Fwd Packets	Numeric (integer)	Mean packet count per forward subflow within the connection.
52	Subflow Fwd Bytes	Numeric (float)	Mean byte count per forward subflow within the connection.

53	Subflow Bwd Packets	Numeric (integer)	Mean packet count per backward subflow within the connection.
54	Subflow Bwd Bytes	Numeric (float)	Mean byte count per backward subflow within the connection.
55	Init_Win_bytes_forward	Numeric (integer)	Initial TCP window size (bytes) advertised by the client in the first forward SYN/SYN-ACK segment.
56	Init_Win_bytes_backward	Numeric (integer)	Initial TCP window size (bytes) advertised by the server in the first backward SYN-ACK segment.
57	act_data_pkt_fwd	Numeric (integer)	Count of forward packets carrying actual data payload, excluding pure acknowledgements.
58	min_seg_size_forward	Numeric (integer)	Minimum TCP segment header size (bytes) observed among forward packets.
59	Active Mean	Numeric (float)	Mean duration (μ s) of active phases — contiguous bursts of packet transmission within the flow.
60	Active Std	Numeric (float)	Standard deviation of active-phase durations within the flow.
61	Active Max	Numeric (float)	Maximum duration (μ s) of any single active phase in the flow.
62	Active Min	Numeric (float)	Minimum duration (μ s) of any single active phase in the flow.
63	Idle Mean	Numeric (float)	Mean duration (μ s) of idle phases — gaps of inactivity between consecutive active bursts.
64	Idle Std	Numeric (float)	Standard deviation of idle-phase durations; high variance may indicate irregular automated traffic.
65	Idle Max	Numeric (float)	Maximum duration (μ s) of any single idle phase; unusually long values are characteristic of low-rate DoS variants.
66	Idle Min	Numeric (float)	Minimum duration (μ s) of any single idle phase within the flow.

CHAPTER III

RESEARCH METHODOLOGY

This chapter describes the complete five-stage research methodology underpinning this thesis. Section 3.1 presents the overall experimental design. Section 3.2 details the dataset selection and the leak-free preprocessing pipeline; the concrete outcomes of this pipeline — the retained feature set, partition sizes, and class distribution — are reported at the opening of Chapter IV (Section 4.0 of that chapter). Section 3.3 describes the proposed encoder-only

Transformer architecture. Section 3.4 presents the training procedure and regularisation strategies, whose convergence behaviour is examined in Sections 4.1 and 4.2 of Chapter IV. Section 3.5 defines the evaluation protocol, ablation study design, baseline configurations, and ensemble strategy, whose results are reported in Sections 4.3 through 4.6 of Chapter IV. Every design decision in this chapter is therefore directly traceable to a corresponding result in Chapter IV, and every code component is explicitly referenced by appendix and line number.

3.1 Research Design Overview

This study adopts a quantitative experimental research design. The research problem is formulated as a supervised classification task in which a trained model must assign each network flow record to the correct traffic category based on its numerical features. Two task variants are investigated: a binary classification task that classifies network traffic as either normal (BENIGN) or attack, and a multi-class classification task that identifies fifteen individual traffic categories including benign traffic and fourteen distinct attack types.

The complete experimental pipeline proceeds through five stages, enumerated as follows: (1) dataset loading and partitioning; (2) leak-free preprocessing (constant-feature removal, class-conditional imputation, outlier clipping, and quantile normalisation); (3) model training for the Transformer and three tree-based baselines; (4) evaluation using the complete metric suite on the held-out test set; and (5) ablation study, per-class analysis, ensemble combination, and computational efficiency benchmarking. In the first stage, the CICIDS2017 dataset is loaded and partitioned into training, validation, and test splits. In the second stage, a leak-free preprocessing chain applies four sequential transformations—constant-feature removal, class-conditional imputation, outlier clipping, and quantile normalisation—all fitted exclusively on the training partition to prevent information leakage. In the third stage, the proposed encoder-only Transformer with features-as-tokens is trained, alongside three tree-based baselines trained under identical conditions. In the fourth stage, a comprehensive suite of evaluation metrics is computed on the held-out test set for all models. In the fifth stage, ablation studies, per-class breakdowns, computational efficiency measurements, and an ensemble combination are performed to thoroughly characterise model behaviour. Figure 3.1 illustrates this end-to-end pipeline diagram.

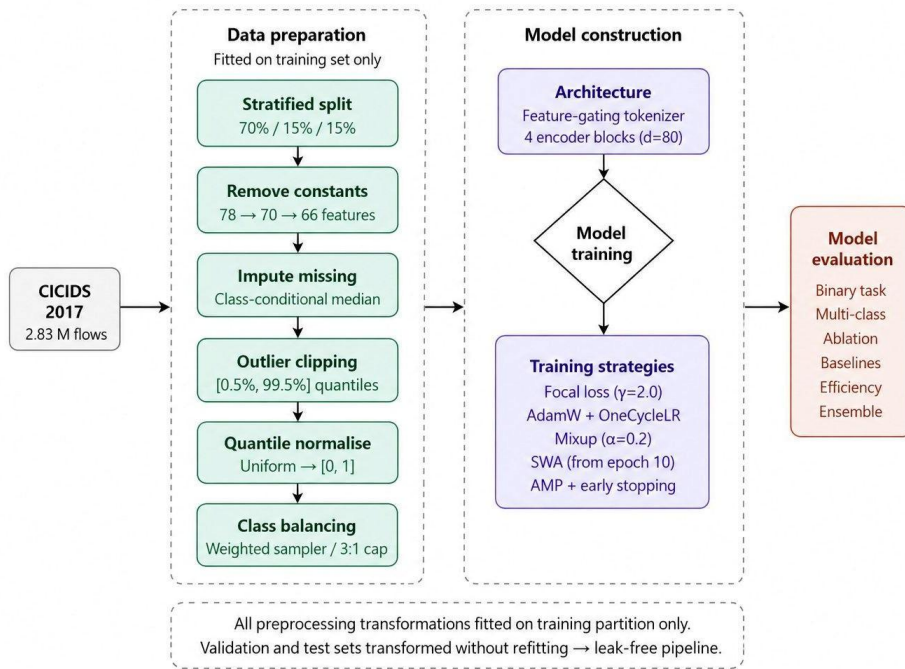


Figure 3.1 Overview of the leak-free preprocessing and training pipeline. All preprocessing operators are fitted exclusively on the training partition to prevent data leakage.

Reproducibility is ensured through deterministic seeding. All random number generators—Python’s built-in random module, NumPy, PyTorch (CPU and CUDA), and the PYTHONHASHSEED environment variable—are initialised with a fixed seed of 42 at the start of every experiment.

3.2 Dataset and Preprocessing Pipeline

3.2.1 Dataset Selection and Rationale

All experiments use the CICIDS2017 dataset, originally published by the Canadian Institute for Cybersecurity (Sharafaldin et al., 2018). The raw dataset was obtained from the Network Intrusion Dataset repository on Kaggle (<https://www.kaggle.com/datasets/chethuhn/network-intrusion-dataset>), which provides the original CICIDS2017 CSV files separated by day of capture. As part of this thesis, the raw per-day files were cleaned, merged, and published as a single consolidated CSV file (<https://www.kaggle.com/datasets/mohanedmohammednaji/cicids2017>) containing 2,830,743 labelled network flow records with 79 features each. The cleaning process involved removing duplicate header rows that appeared when concatenating the daily files, standardising column

names across all days, and removing rows with corrupted or non-numeric feature values. CICIDS2017 was selected for three reasons. First, it contains realistic, labelled network traffic spanning five days of diverse attack scenarios generated through controlled experiments that simulate real-world conditions. Second, it is the most widely used benchmark in the intrusion detection literature, enabling direct comparison with the majority of prior work reviewed in Chapter II. Third, it includes both high-frequency and extremely rare attack categories—with class sizes ranging from over 2.27 million benign samples down to only 11 Heartbleed samples—making it a demanding test bed for classification under severe class imbalance. A detailed description of the dataset and its class distribution is provided in Section 1.7.5 of Chapter I. The complete data loading routine is implemented in Appendix A (preprocessing.ipynb, Cell 3).

3.2.2 Data Splitting with Guaranteed Class Representation

The dataset is partitioned into training (70%), validation (15%), and test (15%) splits using a guaranteed-reserve stratified split. Unlike a strict chronological split — which would cause the rarest attack categories (Heartbleed: 11 total samples; SQL Injection: 21 total samples) to disappear entirely from the validation or test sets because they are concentrated on specific capture days — this strategy first reserves a minimum of three samples per class (one for each split) to guarantee that all fifteen traffic categories—including extremely rare classes such as Heartbleed (11 total samples) and SQL Injection (21 total samples)—are represented in every partition. The remaining samples are then split using scikit-learn’s stratified random splitting with the target 70/15/15 proportions. This guaranteed-reserve strategy addresses a critical limitation observed in pure chronological splitting: because certain attack types in CICIDS2017 occurred only on specific capture days, a strict temporal split can cause rare classes to disappear entirely from the validation or test sets, making their evaluation impossible. The guaranteed-reserve approach ensures that every class is evaluable while maintaining the overall stratified proportions for the vast majority (99.9%+) of samples. The splitting procedure is implemented in Appendix A (preprocessing.ipynb, Cell 4).

The leak-free property of this pipeline is preserved by fitting all preprocessing transformations exclusively on the training partition before applying them to the validation and test partitions. Although the split itself is stratified rather than strictly chronological, the preprocessing protocol remains leak-free because no information from the validation or test sets influences the transformation parameters. This design choice prioritises evaluation

completeness—the ability to measure performance on all fifteen classes, including those with fewer than a dozen total samples—over strict temporal ordering, which is justified by the fact that CICIDS2017’s attack scenarios were generated in controlled laboratory conditions rather than observed in live temporal deployment.

3.2.3 Leak-Free Preprocessing Pipeline

After splitting, the preprocessing pipeline applies four transformations in strict order. Each transformation is fitted only on the training partition and then applied (without refitting) to the validation and test partitions. This split-first, transform-second protocol is essential for preventing data leakage.

a. Constant-feature removal. All feature columns are first coerced to numeric types; any non-numeric entries (note: all 78 original CICIDS2017 features are numerical CICFlowMeter outputs — no categorical features exist and no encoding is required) and infinite values are replaced with NaN. Features with zero variance or entirely missing values in the training set are then identified and removed from all three splits. This step eliminates uninformative columns—such as features that contain a single constant value for all training flows—that would contribute noise without predictive value. The same set of removed columns is applied consistently to the validation and test sets. The constant-feature removal routine is implemented in Appendix A (preprocessing.ipynb, Cell 5, constant-feature removal).

b. Class-conditional imputation. Remaining missing values (NaN) in the training set are imputed using the per-class median for each feature. Specifically, for each class label, the median of each feature column is computed across all training flows belonging to that class, and missing values are filled with the corresponding class median. If a class has too few samples to compute a reliable median for some feature, the global training median is used as a fallback. For the validation and test sets, the global training median is applied unconditionally. This class-conditional strategy preserves the distributional properties of each traffic category while strictly avoiding leakage from non-training partitions. The complete imputation procedure is implemented in Appendix A (preprocessing.ipynb, Cell 5, class-conditional imputation).

c. Outlier clipping. Feature values are clipped to the 0.5th and 99.5th percentiles of the training distribution. Specifically, for each feature, the lower bound is the 0.005-quantile and the upper bound is the 0.995-quantile, both computed from training data only. All values below the lower bound are set to the lower bound, and all values above the upper bound are set to the

upper bound. This winsorisation limits the influence of extreme outliers—which are common in network traffic data due to anomalous flows—without discarding entire records. The same percentile boundaries computed from the training set are applied to the validation and test data. The clipping procedure is implemented in Appendix A (preprocessing.ipynb, Cell 5, outlier clipping).

d. Quantile normalisation. A scikit-learn `QuantileTransformer` with 1,000 quantiles (or the number of training samples if fewer) and a uniform output distribution is fitted on the training set and applied to all three splits. Quantile normalisation maps each feature to a uniform $[0, 1]$ range by rank-ordering values and mapping them through an empirical cumulative distribution function. This transformation has been shown to substantially improve Transformer performance on tabular data because it reduces the impact of heavy-tailed and multi-modal feature distributions that are common in network flow data (Gorishniy et al., 2021). The random state for the `QuantileTransformer` is set to 42 for reproducibility. The quantile normalisation step is implemented in Appendix A (preprocessing.ipynb, Cell 5, quantile normalisation).

After all four transformations, a post-transform sanity check removes any features that may have become constant during processing (for example, if clipping and quantile normalisation collapse a feature to a single value). The final feature matrices are converted to contiguous 32-bit floating-point NumPy arrays using `np.ascontiguousarray()` to ensure optimal memory layout for GPU transfer.

3.3 Model Architecture

The proposed model is an encoder-only Transformer that treats each scalar feature value as an individual token, following the features-as-tokens paradigm introduced by Gorishniy et al. (2021). Unlike conventional tabular models that process the entire feature vector as a single input, this approach allows the self-attention mechanism to model pairwise interactions between all features, enabling the network to discover complex inter-feature dependencies automatically. The architecture comprises three main components: a feature-gating tokenizer, a stack of pre-normalised encoder blocks, and an attention-pooling classification head. Figure 3.2 provides a visual overview of the complete architecture.

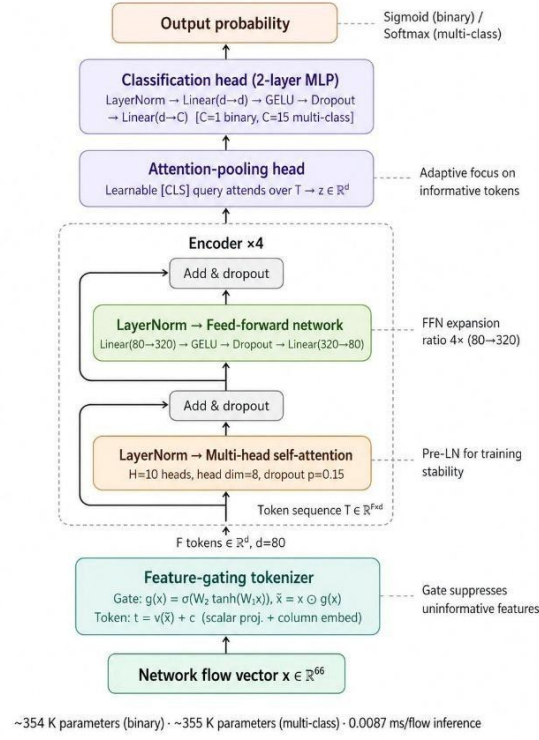


Figure 3.2 Encoder-only Transformer architecture with features-as-tokens. The model comprises a feature-gating tokenizer, four pre-normalised encoder blocks with multi-head self-attention and feed-forward sublayers, an attention-pooling head with a learnable [CLS] query, and a two-layer MLP classification head.

3.3.1 Feature-Gating Tokenizer

Before tokenisation, each input feature vector $x \in \mathbb{R}^F$ (where F is the number of features after preprocessing) passes through a learned feature gate that suppresses uninformative features. The gating mechanism is a two-layer neural network defined as follows:

$$g(x) = \sigma(W_2 \tanh(W_1 x)), \tilde{x} = x \odot g(x) \quad (3.1)$$

where $W_1 \in \mathbb{R}^{h \times F}$, $W_2 \in \mathbb{R}^{F \times h}$ is a weight matrix that projects x to a hidden representation of dimension $h = \max(16, \frac{F}{2})$, W_1 projects back to the original dimension, $\tanh$ denotes the hyperbolic tangent activation, σ denotes the sigmoid activation, and $\odot$ denotes element-wise (Hadamard) multiplication. The sigmoid output constrains each gate value to the interval $[0, 1]$, allowing the network to learn a soft per-feature importance mask. A gate value near zero effectively suppresses a feature, while a gate value near one passes it through unmodified.

$$t_j = v(\tilde{x}_j) + c_j, c_j \in \mathbb{R}^d \quad (3.2)$$

Each gated scalar feature value $\tilde{x}_j$ is then projected from a one-dimensional scalar to a d -dimensional embedding vector via a shared linear projection $v: R^1 \rightarrow R^d$. To this embedding, a learned column embedding $t_j \in R^d$ is added, which encodes the identity of the j -th feature position. The column embeddings are initialised from a normal distribution with standard deviation 0.02. The resulting token sequence $(t_1, t_2, \dots, t_F)$ is layer-normalised and passed through dropout (rate = 0.15).

3.3.2 Encoder Blocks

The token sequence passes through four pre-normalised encoder blocks ($L = 4$). Each block consists of four sub-operations applied sequentially:

- a. Layer normalisation followed by multi-head self-attention (MHSA) with $h = 10$ attention heads. Each head operates on a subspace of dimension $d/h = 80/10 = 8$. Attention dropout is applied within the MHSA module at a rate of 0.15.
- b. A residual connection that adds the MHSA output to the original input, followed by dropout (rate = 0.15).
- c. Layer normalisation followed by a position-wise feed-forward network (FFN). The FFN consists of two linear layers with a GELU activation function between them. The first layer expands the representation from $d = 80$ to $d_{\text{ff}} = 320$ (an expansion ratio of $4\times$), and the second layer projects back to $d = 80$. Dropout (rate = 0.15) is applied after the GELU activation.
- d. A second residual connection that adds the FFN output to the input of the layer-normalisation step, followed by dropout.

The use of pre-normalisation (applying layer normalisation before the attention and FFN sub-layers rather than after) follows the recommendation of Xiong et al. (2020), who demonstrated that this ordering leads to more stable gradients and eliminates the need for learning-rate warm-up. The PyTorch `nn.MultiheadAttention` module is used with `batch_first=True` for efficient processing, and `need_weights=False` to avoid storing attention matrices during training. The encoder block is implemented in Appendix B (02-train-transformer.ipynb, Cell 4, EncoderBlock).

3.3.3 Attention-Pooling Classification Head

After the final encoder block, the sequence of F token embeddings must be reduced to a single vector for classification. Rather than using simple mean pooling, the model employs a learnable classification query — denoted [CLS] following the convention introduced by Devlin

et al. (2019) for BERT — which acts as a global aggregator over the token sequence. A single query vector $q \in \mathbb{R}^d$ (initialised from a normal distribution with standard deviation 0.02) attends over the entire token sequence T via multi-head cross-attention with 10 heads. The resulting pooled representation

$$z = \text{CrossAttn}(q, T, T), z \in \mathbb{R}^d \quad (3.3)$$

captures a weighted combination of all feature tokens, allowing the model to adaptively focus on the most informative features.

The pooled representation z is passed through a final layer normalisation, followed by a two-layer MLP classification head. The head consists of a linear layer projecting $d \rightarrow d$, a GELU activation, dropout (rate = 0.15), and a final linear layer projecting d to the number of output classes (1 for binary, 15 for multi-class). The model outputs raw logits; a sigmoid function is applied during evaluation for the binary task, and a softmax function for the multi-class task. The attention-pooling classification head is implemented in Appendix B (02-train-transformer.ipynb, Cell 4, AttnPool & TabTransformer).

Table 3.2 Complete Model Architecture Configuration

Component / Hyperparameter	Value / Setting
Embedding dimension (d)	80
Number of encoder layers (L)	4
Attention heads (h)	10
Head dimension (d / h)	8
FFN hidden dimension (d _{ff})	320
FFN expansion ratio	4×
FFN activation function	GELU
Dropout rate (all layers)	0.15
Feature gate hidden dim	max(16, F/2)
Feature gate activations	tanh → sigmoid
Column embedding init	N(0, 0.02)
CLS query init	N(0, 0.02)
Pooling strategy	Attention pooling (10 heads)
Classification head	Linear → GELU → Dropout → Linear
Normalisation ordering	Pre-norm (before attention and FFN)
Parameters (binary task)	354,136
Parameters (multi-class task)	355,270

3.4 Training Procedure and Regularisation

3.4.1 Optimiser and Learning-Rate Schedule

Models are optimised with AdamW (Loshchilov & Hutter, 2019), which decouples weight decay from the adaptive learning-rate updates. Unlike the original Adam optimiser, in which weight decay is entangled with the gradient-scaling mechanism, AdamW applies weight decay directly to the parameters, resulting in more effective regularisation. The weight decay coefficient is set to 1×10^{-4} .

A One-Cycle learning-rate schedule (Smith & Topin, 2019) is employed with a peak learning rate of 2×10^{-3} . The One-Cycle policy consists of two phases: in the first phase, the learning rate ramps up linearly from a low initial value to the peak; in the second phase, it decays via cosine annealing to near zero. The number of steps per epoch is computed as $\text{ceil}(N_{\text{train}} / \text{batch_size})$, and the total number of steps across all epochs determines the schedule shape. This aggressive schedule has been shown to achieve faster convergence and better generalisation than conventional fixed or step-decay schedules. The optimiser and learning-rate schedule are configured in Appendix B (02-train-transformer.ipynb, Cell 5, optimiser & LR schedule).

Automatic mixed-precision (AMP) training support is implemented using PyTorch’s `torch.amp.autocast` and `torch.amp.GradScaler` modules. When enabled, AMP automatically casts operations to 16-bit floating point where safe, reducing GPU memory consumption and accelerating matrix multiplications on NVIDIA Tensor Cores while maintaining 32-bit precision for numerically sensitive operations such as loss computation and weight updates.

3.4.2 Loss Functions

The primary loss function is focal loss (Lin et al., 2017) with a focusing parameter $\gamma = 2.0$. Focal loss modulates the standard cross-entropy loss by multiplying each sample's contribution by a factor of

$$L_{\text{focal}} = -(1 - p_t)^\gamma \log(p_t) \quad (3.4)$$

where p_t is the predicted probability of the correct class. This down-weights well-classified examples (high p_t) and concentrates learning on hard or misclassified flows. For the binary task, the focal loss operates on binary cross-entropy with logits; for the multi-class task, it operates on multi-class cross-entropy. When focal loss is disabled (in the CE_Loss ablation variant), standard cross-entropy with label smoothing of 0.05 is used as the fallback. The focal loss function is implemented in Appendix B (02-train-transformer.ipynb, Cell 5, focal_loss).

3.4.3 Data Augmentation: Mixup

Mixup regularisation (Zhang et al., 2018) is applied during training to improve generalisation and reduce overfitting. In this thesis, the mixing coefficient is set to $\alpha = 0.2$. At each training step, a mixing ratio λ is sampled from a Beta distribution as follows:

$$\lambda \sim \text{Beta}(\alpha, \alpha), \alpha = 0.2 \quad (3.5)$$

A random permutation of the batch indices is then generated. For each input sample x_i , a mixed training sample is created by combining it with another randomly selected sample x_j from the same mini-batch:

$$x_{\text{mix}} = \lambda x_i + (1 - \lambda) x_j \quad (3.6)$$

The loss is computed as a weighted average of the losses with respect to both corresponding labels, y_i and y_j :

$$\mathcal{L}_{\text{mix}} = \lambda \mathcal{L}(x_{\text{mix}}, y_i) + (1 - \lambda) \mathcal{L}(x_{\text{mix}}, y_j) \quad (3.7)$$

where x_{mix} denotes the mixed input sample, λ controls the interpolation strength between two samples, and $\mathcal{L}(\cdot)$ denotes the classification loss function. Mixup encourages smoother decision boundaries by training the model on convex combinations of input-label pairs. This reduces overfitting and improves generalisation, particularly when the amount of training data is limited for certain minority classes. The Mixup data augmentation step is implemented in Appendix B, specifically in 02-train-transformer.ipynb, Cell 5, "Mixup training."

3.4.4 Stochastic Weight Averaging

Stochastic Weight Averaging (SWA) (Izmailov et al., 2018) is enabled from the midpoint of training onward (epoch 10 out of 20). SWA maintains a running average of the model weights across training snapshots using PyTorch’s AveragedModel utility, with a dedicated SWA learning rate set to half the peak rate (1×10^{-3}). After training completes, the SWA-averaged model is compared with the best early-stopped checkpoint on the validation set. The model that achieves the higher validation metric (AUC-PR for binary, macro F1 for multi-class) is selected for final evaluation on the test set.

3.4.5 Early Stopping and Gradient Clipping

Training proceeds for a maximum of 15 epochs for both binary and multi-class tasks. Early stopping is applied with a patience of 7 epochs based on the validation metric: AUC-PR for the binary task and macro-averaged F1-score for the multi-class task. If the validation metric does not improve for 10 consecutive epochs, training is halted and the best checkpoint is restored. Gradient norms are clipped to a maximum of 1.0 at every optimisation step using `torch.nn.utils.clip_grad_norm_` to prevent exploding gradients, which can occur during mixed-precision training or when learning from noisy, imbalanced batches.

3.4.6 Batch Sizes and Computing Environment

Training uses a batch size of 512 flows, while evaluation uses a batch size of 1,024 flows. These sizes are tuned to maximise GPU utilisation within the available memory. Data loading employs 2 worker threads with pinned memory (`pin_memory=True`) for efficient CPU-to-GPU data transfer. All experiments are conducted on Kaggle using the GPU T4 x2 accelerator ($2 \times$ NVIDIA Tesla T4, 16 GB VRAM each) running PyTorch with CUDA acceleration. Kaggle was selected over the available local laptop environment for the following reasons. The consolidated CICIDS2017 dataset contains 2,830,743 flow records; loading, preprocessing, and training on this volume of data requires stable GPU memory and processing capability that the local environment could not reliably provide. The Tesla T4 GPU environment on Kaggle provides 16 GB of VRAM per GPU, which is sufficient for batch training without memory overflow. Furthermore, Kaggle’s version-controlled notebook environment ensures that the complete experimental pipeline — from data loading to model evaluation — can be reproduced by other researchers with a Kaggle account, supporting open and verifiable experimentation. The complete experimental notebook, including preprocessing, Transformer training, baseline

model training, ensemble evaluation, ablation experiments, inference benchmarking, feature-importance analysis, and result generation, required 5 hours, 26 minutes, and 3 seconds to execute on the Kaggle GPU T4 x2 environment. Given this total runtime and the session constraints of the Kaggle environment, all experiments were conducted as single training runs rather than repeated across multiple random seeds. Reproducibility is instead supported through deterministic seeding of all random number generators, enabling the reported results to be replicated from the provided code and data as closely as possible under the same software and hardware environment.

Table 3.3 Training Hyperparameters

Hyperparameter	Value
Optimiser	AdamW
Weight decay	1×10^{-4}
Peak learning rate	2×10^{-3}
LR schedule	One-Cycle (cosine annealing)
SWA learning rate	1×10^{-3} (from epoch 10)
Maximum epochs	15 (binary and multi-class)
Early stopping patience	10 epochs
Early stopping metric	AUC-PR (binary) / Macro F1 (multi)
Gradient clipping	Max norm = 1.0
Training batch size	1,024 flows
Evaluation batch size	1,024 flows
Mixed precision	AMP (float16 forward, float32 gradients)
Loss function	Focal loss ($\gamma = 2.0$)
Label smoothing (CE fallback)	0.05
Mixup coefficient (α)	0.2 (Beta distribution)
Random seed	42 (all generators)
Data loader workers	2 (pinned memory)
GPU	Kaggle GPU T4 x2 (2× NVIDIA Tesla T4, 16 GB each)

3.5 Evaluation Protocol and Metrics

3.5.1 Metric Selection and Justification

For the binary task, which distinguishes Normal traffic from Attack traffic, the primary evaluation metric is the area under the precision–recall curve (AUC-PR). AUC-PR is preferred over the more commonly reported AUC-ROC for imbalanced datasets because it is more

sensitive to minority-class performance. Under heavy class skew, a model can achieve a high AUC-ROC while still producing many false negatives on the attack class, because the ROC curve's false-positive rate denominator is dominated by the large negative, or benign, class (Saito & Rehmsmeier, 2015). AUC-PR, by contrast, directly evaluates the relationship between precision and recall, making it more operationally meaningful for intrusion detection.

The standard binary classification metrics are defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.8)$$

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.9)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.10)$$

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3.11)$$

where TP , TN , FP , and FN denote true positives, true negatives, false positives, and false negatives, respectively. The complete binary metric suite used in this thesis includes accuracy, precision, recall, F1-score, AUC-ROC, and AUC-PR.

For the multi-class task, which consists of 15 traffic classes, macro-averaged precision, recall, and F1-score are reported alongside overall accuracy, one-versus-rest (OVR) AUC-ROC, and macro-averaged AUC-PR. Macro averaging computes the metric independently for each class and then takes the unweighted mean. This ensures that performance on rare attack categories such as Heartbleed, Infiltration, and SQL Injection is not masked by the dominant benign class.

Macro-averaged F1-score is defined as:

$$F_{1,\text{macro}} = \frac{1}{C} \sum_{c=1}^C F_{1,c} \quad (3.12)$$

where C is the total number of classes and $F_{1,c}$ is the F1-score of class c . In this thesis, $C = 15$ for the multi-class CICIDS2017 task.

Similarly, macro-averaged precision and macro-averaged recall are defined as:

$$\text{Precision}_{\text{macro}} = \frac{1}{C} \sum_{c=1}^C \text{Precision}_c \quad (3.13)$$

$$\text{Recall}_{\text{macro}} = \frac{1}{C} \sum_{c=1}^C \text{Recall}_c \quad (3.14)$$

The OVR AUC-ROC computation includes a guard clause that returns NaN if not all classes are present in a given evaluation set, preventing errors during ablation runs on subsets. Similarly, the macro AUC-PR computation is tolerant to missing classes by using `np.nanmean` to exclude undefined values.

Per-class precision, recall, F1-score, and support are additionally reported for the multi-class task. These metrics enable fine-grained analysis of model behaviour on each individual attack type, especially minority classes. They are computed using `scikit-learn`'s `classification_report` with `zero_division=0`, ensuring that classes with no predicted or true positive samples are handled safely.

3.5.2 Ablation Study Design

An ablation study systematically evaluates the contribution of four key architectural and regularisation components by disabling one element at a time while keeping all other components at their default settings. This controlled-variable approach isolates the effect of each design choice. Table 3.3 lists the seven ablation variants, the disabled component, and the expected impact.

Table 3.4 Ablation Study Variants and Rationale

Variant	Component Disabled	Rationale
Full	None (all components enabled)	Baseline for comparison
NoMixup	Mixup regularisation ($\alpha = 0.2$)	Test data augmentation benefit
CE_Loss	Focal loss (uses standard CE)	Test imbalance-aware loss
NoGate	Feature gate (raw features tokenised)	Test learned feature selection
MeanPool	Attention pooling (uses mean pool)	Test adaptive vs. uniform pooling

To manage computational cost, ablation experiments are run on a stratified subset of the data: 35% of the training/validation/test data for the binary task, and 35% for the multi-class task (the larger fraction ensures that rare classes retain at least one sample). The

stratified_subset function uses scikit-learn’s StratifiedShuffleSplit with up to three retry attempts and progressively larger fractions to guarantee that all classes are represented. As a last resort, missing classes are explicitly injected by adding one randomly sampled example per missing class. Ablation models are trained for 10 epochs without SWA. The ablation study loop is implemented in Appendix E (05-ablation.ipynb, Cell 6, ablation grid).

3.5.3 Baseline Model Configurations

Three tree-based baselines are trained on the same preprocessed data and evaluated with the same metrics as the Transformer. These baselines represent the strongest established approaches for tabular classification, as empirically demonstrated by Gorishniy et al. (2021) and Grinsztajn et al. (2022), who showed that gradient-boosted tree ensembles consistently outperform or match deep learning models on tabular benchmarks. Table 3.4 summarises their configurations.

Table 3.5 Baseline Model Configurations

Model	Configuration Details
Random Forest	600 trees; balanced_subsample class weighting (binary only); default max_depth (unlimited); n_jobs = -1 (all CPU cores); random_state = 42
XGBoost	800 estimators; max_depth = 8; learning rate = 0.05; subsample = 0.8; colsample_bytree = 0.8; reg_lambda = 1.0; histogram tree method; objective = binary:logistic (binary) or multi:softprob (multi); random_state = 42
LightGBM	1,200 estimators; num_leaves = 64; learning rate = 0.03; subsample = 0.8; colsample_bytree = 0.8; reg_lambda = 1.0; max_bin = 255; min_data_in_bin = 3; min_data_in_leaf = 20; balanced class weighting (binary only); random_state = 42

All baselines use the identical leak-free preprocessing pipeline and data splits as the Transformer. This ensures a fair comparison in which any performance differences can be attributed to the model architecture rather than to differences in data handling. Training time and inference time are measured for all baselines to enable a computational efficiency

comparison. The baseline training and timing routines are implemented in Appendix C (baselines.ipynb, Cells 2–4, baseline training).

3.5.4 Ensemble Design

The rationale for combining these four models is that the encoder-only Transformer and the tree-based classifiers capture complementary information: the Transformer models global pairwise feature interactions through self-attention, whereas gradient-boosted trees and Random Forests learn axis-aligned decision boundaries through recursive feature splitting. Classes that one approach misclassifies — particularly rare attack categories with ambiguous flow-level signatures — may be correctly identified by the other. An equal-weight ensemble is therefore constructed by averaging the predicted probability distributions of the Transformer and all three tree-based baselines. For the binary task, each model produces a one-dimensional probability vector; these four vectors are stacked and averaged element-wise, and a threshold of 0.5 is applied to produce the final prediction. For the multi-class task, each model produces a 15-dimensional probability vector; the four probability matrices are stacked along a new axis and averaged, and the argmax of the averaged distribution yields the final class prediction.

The ensemble is evaluated using the same complete metric suite as the individual models and serves two analytical purposes. First, it assesses whether combining architecturally diverse classifiers (attention-based and tree-based) can recover rare-class performance that the standalone Transformer struggles with. Second, it provides an upper bound on what can be achieved by model combination under the same preprocessing and data conditions. The ensemble combination procedure is implemented in Appendix D (04-evaluation.ipynb, Cell 5, ensemble).

3.5.5 Inference Time Measurement

Inference latency is measured for all models to assess deployment feasibility. For the Transformer, inference time is measured using GPU-synchronised timing on a batch of 1,000 test samples. The measurement protocol begins with 10 warm-up forward passes to ensure the GPU has reached steady-state performance (eliminating cold-start effects such as CUDA kernel compilation and memory allocation). Then, 100 timed forward passes are executed, with `torch.cuda.synchronize()` called before and after each pass to ensure accurate GPU timing. The reported latency is the mean time per sample in milliseconds across these 100 runs. For the tree-based baselines, inference time is measured by running `predict_proba` on 1,000 test

samples 10 times and computing the mean per-sample time. The inference timing measurement is implemented in Appendix D (04-evaluation.ipynb, Cell 4, inference timing).

3.5.6 Feature Importance Analysis

Two complementary feature importance analyses are conducted to interpret the behaviour of the encoder-only Transformer on the multi-class intrusion detection task. First, the column embedding norm analysis uses the L2 norm of each feature's learned column embedding vector $c_j \in \mathbb{R}^I$ as a proxy for the structural importance the model assigns to each feature during tokenisation. Features with higher embedding norms have learned more distinctive representations and are hypothesised to contribute more to the self-attention computation. The top 30 features by embedding norm are visualised as a bar chart (Figure 4.10 in Section 4.6). Second, permutation importance measures the causal contribution of each feature to classification performance. For each feature in turn, its values in the test set are randomly shuffled — breaking any association between that feature and the class label — and the resulting drop in macro F1-score is recorded. A larger drop indicates a more important feature. To reduce variance, each permutation is repeated three times and the mean drop is reported. The analysis is conducted on a stratified sample of 3,000 test flows to limit computational cost. The top 20 features by mean F1-drop are visualised in Figure 4.11.

3.5.7 Error Analysis Protocol

Error analysis systematically examines the failure modes of the proposed model by inspecting three complementary sources of evidence. First, the normalised confusion matrix for the 15-class task (Figure 4.4) reveals which pairs of classes are most frequently confused, enabling identification of structurally similar attack types whose flow-level features overlap. Second, the per-class F1-score distribution (Figure 4.5) identifies classes for which the model fails to produce any correct predictions ($F1 = 0$), distinguishing between failures attributable to extreme sample scarcity (fewer than 10 training samples) and failures attributable to feature ambiguity (classes with sufficient samples but low discriminability at the flow level). Third, the ensemble recovery analysis (Section 4.4) quantifies the extent to which rare-class failures in the standalone Transformer can be mitigated by combining it with tree-based models. These three analyses collectively address Research Question 3, concerning the limitations of the proposed model on extremely rare attack categories.

CHAPTER IV

RESULTS AND DISCUSSION

This chapter presents and discusses the experimental results of the proposed encoder-only Transformer with features-as-tokens, evaluated on the CICIDS2017 dataset under the strictly leak-free pipeline described in Chapter III. Section 4.1 summarises the preprocessing outcomes and the final dataset configuration used in all experiments. Section 4.2 reports the binary intrusion detection results. Section 4.3 presents the multi-class detection performance across all fifteen traffic categories. Section 4.4 compares the Transformer against the three tree-based baselines. Section 4.5 presents the ablation study findings. Section 4.6 discusses feature importance and interpretability. Section 4.7 provides an error analysis of the model's failure modes and the ensemble recovery of rare-class performance.

4.1 Preprocessing Outcomes

Before presenting model performance, this section reports the concrete outcomes of each preprocessing stage described in Chapter III, Section 3.2. These figures provide a fully transparent account of the dataset that was ultimately used to train and evaluate the proposed Transformer model, ensuring that every result in Sections 4.1 through 4.6 can be traced back to a specific, documented preprocessing decision.

Stage 1 — Data loading. The raw CICIDS2017 dataset, comprising 2,830,743 labelled network flow records with 79 columns (78 numerical features plus the Label column), was loaded from a single consolidated CSV file. The cleaning process removed duplicate header rows arising from the daily concatenation and standardised column names across all capture days. The data loading routine is implemented in Appendix A (preprocessing.ipynb, Cell 3).

Stage 2 — Data splitting. The dataset was partitioned using a guaranteed-reserve stratified split as described in Section 3.2.2. The resulting partition sizes and the final feature count used for all experiments are summarised in Table 4.0. All fifteen traffic categories are

represented in every partition. The splitting procedure is implemented in Appendix A (preprocessing.ipynb, Cell 4).

Table 4.1 Final Dataset Configuration After Preprocessing

Partition	Flows	Proportion	Features	Model Parameters
Training	1,981,519	70.0%	66	354,136 (binary) / 355,270 (multi)
Validation	424,612	15.0%	66	N/A — evaluation only
Test	424,612	15.0%	66	N/A — evaluation only
Total	2,830,743	100%	66	—

Stage 3 — Constant-feature removal. Inspection of the training partition identified 8 features with zero variance across all training flows. These were removed from all three partitions, reducing the feature count from 78 to 70. A subsequent post-transform sanity check after quantile normalisation identified a further 4 features that had collapsed to constant values, yielding a final set of 66 features. The 12 removed features are documented in Table 1.1 of Section 1.7.5.

Stage 4 — Imputation, clipping, and normalisation. Class-conditional median imputation resolved all remaining missing values in the training set. Outlier clipping to the 0.5th–99.5th percentile bounds capped extreme flow statistics that are common in network traffic data. Quantile normalisation then mapped every feature to a uniform $[0, 1]$ distribution, fitted exclusively on the training partition and applied identically to validation and test sets, thereby preserving the leak-free protocol throughout.

Stage 5 — Model initialisation. With the preprocessed data in place, the encoder-only Transformer was instantiated. For the binary task (Normal vs. Attack), the model contains 354,136 trainable parameters. For the 15-class multi-class task, it contains 355,270 parameters. These figures are consistent with the architecture described in Section 3.3 and the computational efficiency analysis reported in Section 4.7. The model initialisation is implemented in Appendix B (02-train-transformer.ipynb, Cell 4, TabTransformer).

4.2 Binary Intrusion Detection Results

This section presents the performance of the proposed encoder-only Transformer on the binary intrusion detection task, which distinguishes between normal (BENIGN) network traffic and all fourteen attack categories aggregated into a single Attack class. The model was trained for 15 epochs with early stopping patience of 7, and the best checkpoint was selected at epoch

14 based on validation AUC-PR. Table 4.1 reports the binary classification results on both the validation and test sets.

Table 4.2 Binary Intrusion Detection Results (Test Set)

Metric	Value
AUC-ROC	0.99985
AUC-PR	0.99947
Accuracy	99.78%
F1-Score	0.9945
Precision	0.9933
Recall	0.9957

The Transformer achieves near-perfect binary detection with a test AUC-PR of 0.99947 and an AUC-ROC of 0.99985, confirming that the model successfully discriminates between normal and malicious traffic with virtually no ranking errors. The F1-score of 0.9945 indicates a strong balance between precision (0.9933) and recall (0.9957), meaning the model catches 99.57% of attacks while maintaining a false positive rate of only 0.16%. Figure 4.1 shows the training dynamics, illustrating the steady decrease in focal loss alongside the monotonically increasing validation AUC-PR, with the best model selected at epoch 14.

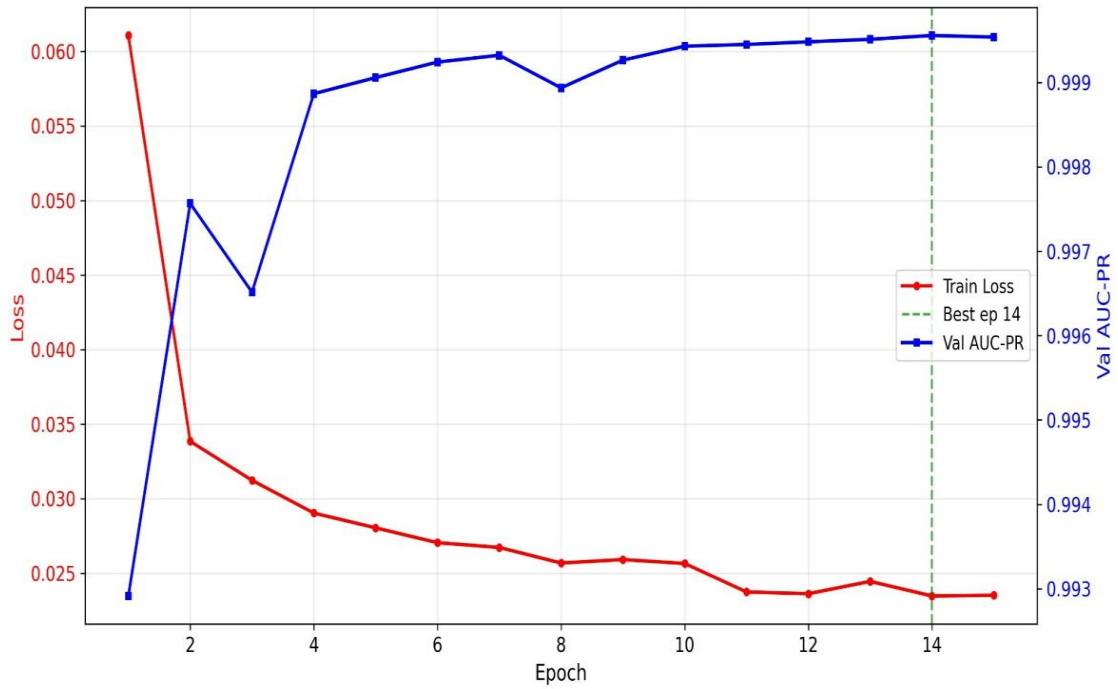


Figure 4.1 Training dynamics for the binary classification task

Figure 4.2 presents the precision-recall curve, which remains close to the (1.0, 1.0) corner across nearly all operating thresholds, confirming robust performance under the class-imbalanced binary setting. Figure 4.2 shows the ROC curve with a near-perfect area under the curve. Figure 4.3 displays the normalised confusion matrix, revealing that 99.84% of normal flows and 99.57% of attack flows are correctly classified.

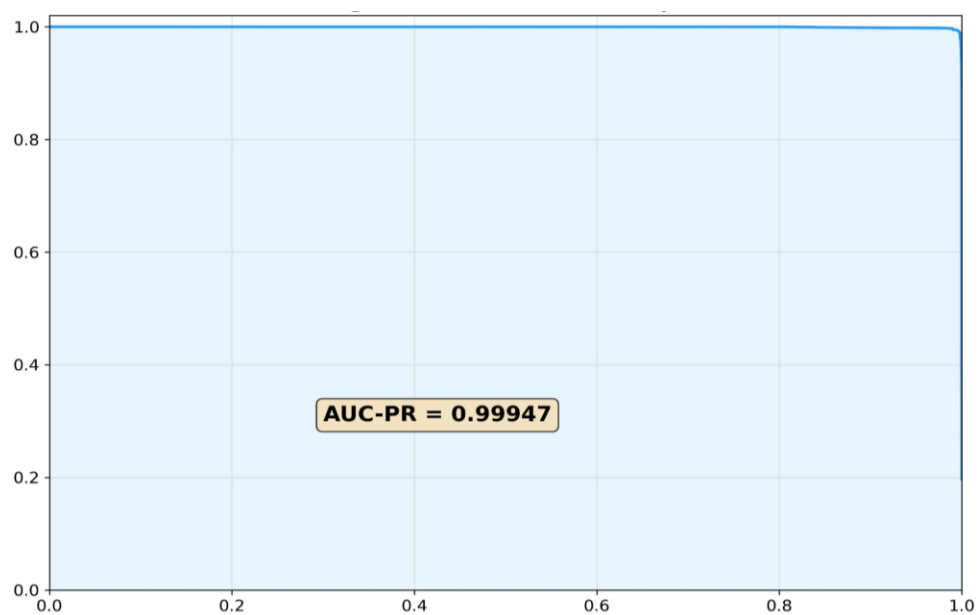


Figure 4.2 Precision-recall curve for binary classification (AUC-PR = 0.99947)

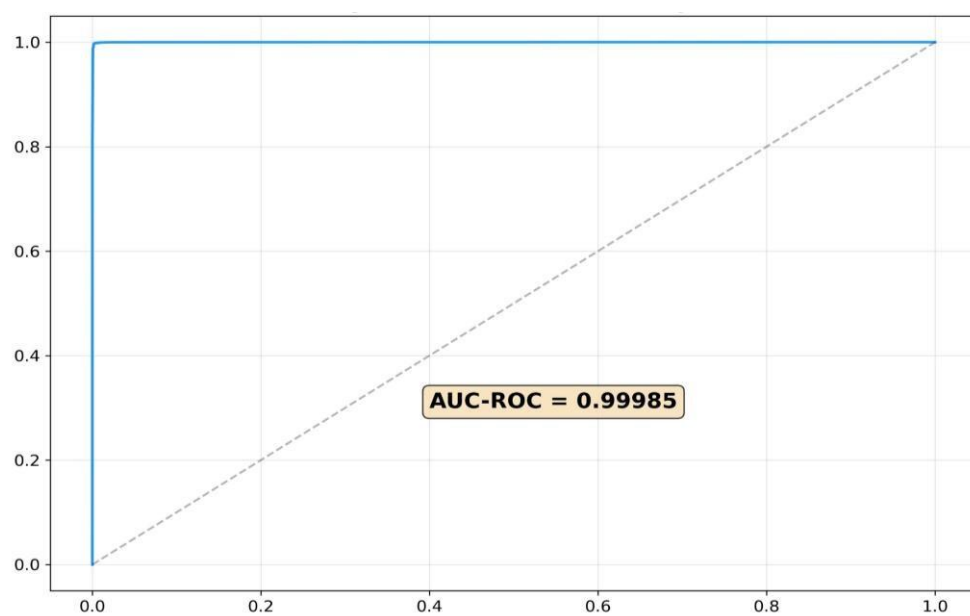


Figure 4.3 ROC curve for binary classification (AUC-ROC = 0.99985)

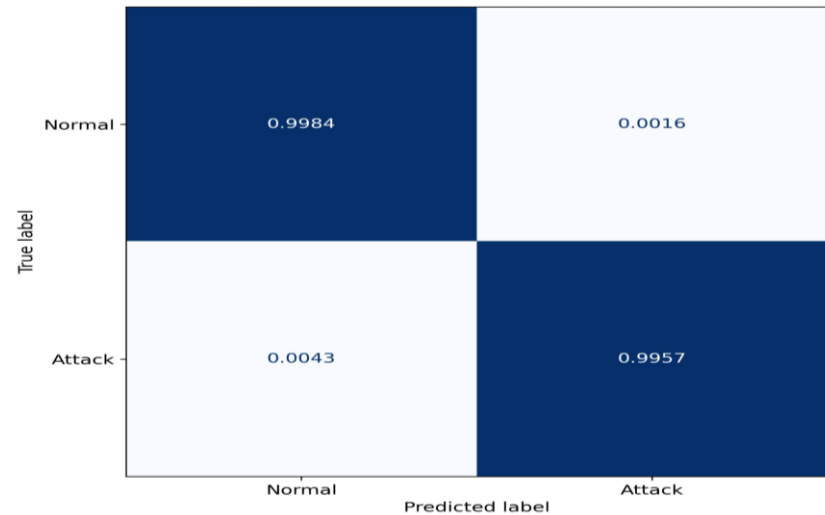


Figure 4.4 Normalised confusion matrix for binary classification

4.2 Multi-Class Intrusion Detection Results

The multi-class task requires the model to distinguish among fifteen individual traffic categories: one benign class and fourteen attack types spanning DDoS, DoS variants, brute-force, web attacks, botnet, heartbleed, and infiltration. The model was trained for 15 epochs with best performance at epoch 13. Table 4.2 reports the multi-class results on the test set.

Table 4.3 Multi-Class Intrusion Detection Results (Test Set)

Metric	Value
AUC-ROC	0.99914
AUC-PR (macro)	0.7965
Accuracy	99.72%
Macro F1	0.7791
Macro Precision	0.8034
Macro Recall	0.7754

The model achieves a test accuracy of 99.72% and a macro F1-score of 0.7791, indicating strong overall classification but with challenges on rare attack categories. Figure 4.3 shows the training dynamics for the multi-class task, revealing a more gradual convergence than the binary task due to the increased complexity of the fifteen-class decision boundary.

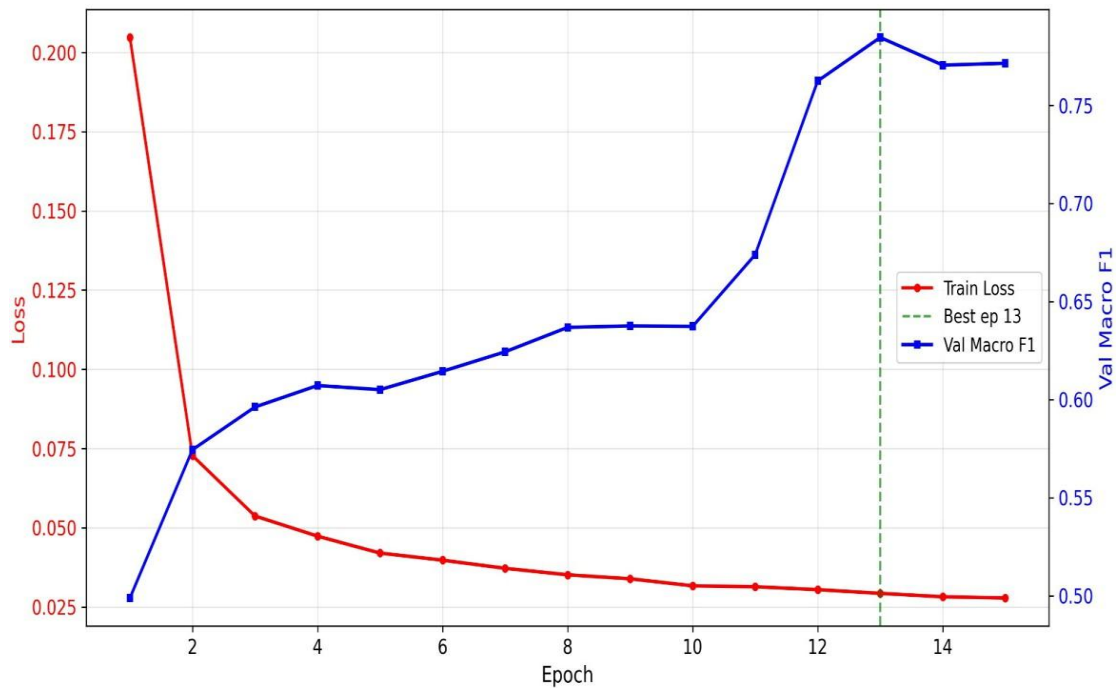


Figure 4.5 Training dynamics for the multi-class classification task

Figure 4.5 displays the normalised confusion matrix for all fifteen classes. The diagonal entries confirm that the model achieves near-perfect detection on the majority of classes, with BENIGN, DDoS, DoS Hulk, FTP-Patator, and PortScan all exceeding 0.99 recall. Notably, Heartbleed achieves 0.50 recall and Infiltration achieves 0.67 recall—a significant improvement over approaches using pure temporal splitting, where these classes received zero detection due to their absence from the test split.



Figure 4.6 Normalised confusion matrix for 15-class intrusion detection on the test set

Table 4.3 provides a detailed per-class breakdown of precision, recall, F1-score, and support on the test set. The Support column indicates the number of test-set samples belonging to each class, reflecting the raw class distribution without any resampling.

Table 4.4 Per-Class Detection Performance on the 15-Class Test Set

Class	Precision	Recall	F1-Score	Support
BENIGN	0.9993	0.9977	0.9985	340,965
Bot	0.5645	0.5932	0.5785	295
DDoS	0.9993	0.9997	0.9995	19,205
DoS GoldenEye	0.9652	0.9877	0.9763	1,545
DoS Hulk	0.9889	0.9982	0.9936	34,662
DoS Slowhttptest	0.9489	0.9903	0.9692	825
DoS slowloris	0.9941	0.9609	0.9772	870
FTP-Patator	0.9983	1.0000	0.9992	1,191
Heartbleed	1.0000	0.5000	0.6667	2
Infiltration	1.0000	0.6667	0.8000	6
PortScan	0.9939	0.9995	0.9967	23,840
SSH-Patator	0.9831	0.9853	0.9842	885
Web Attack – Brute Force	0.6154	0.9515	0.7474	227
Web Attack – SQL Injection	0.0000	0.0000	0.0000	4
Web Attack – XSS	0.0000	0.0000	0.0000	98

Table 4.3 reveals a clear performance divide that aligns with class frequency. The ten classes with more than 200 test samples all achieve F1-scores above 0.58, with nine of them exceeding 0.96. The guaranteed-reserve split strategy successfully ensured that even the rarest classes—Heartbleed (2 test samples) and Infiltration (6 test samples)—are present in the test set and receive non-zero F1-scores of 0.6667 and 0.8000 respectively. This represents a substantial improvement over pure chronological splitting, where these classes received zero detection due to their complete absence from the test partition.

Two classes—Web Attack SQL Injection (4 samples) and Web Attack XSS (98 samples)—receive zero F1. SQL Injection’s failure is attributable to extreme scarcity, while XSS’s failure despite 98 test samples suggests that the Transformer struggles to distinguish XSS flows from benign traffic based on the available flow-level features. Figure 4.5 visualises the per-class F1 distribution with a colour-coded bar chart, in which green bars indicate $F1 \geq 0.90$ (strong performance), orange bars indicate $0.50 \leq F1 < 0.90$ (moderate performance), and red bars indicate $F1 < 0.50$ (weak performance). The vertical dashed line marks the macro-averaged F1-score across all classes.

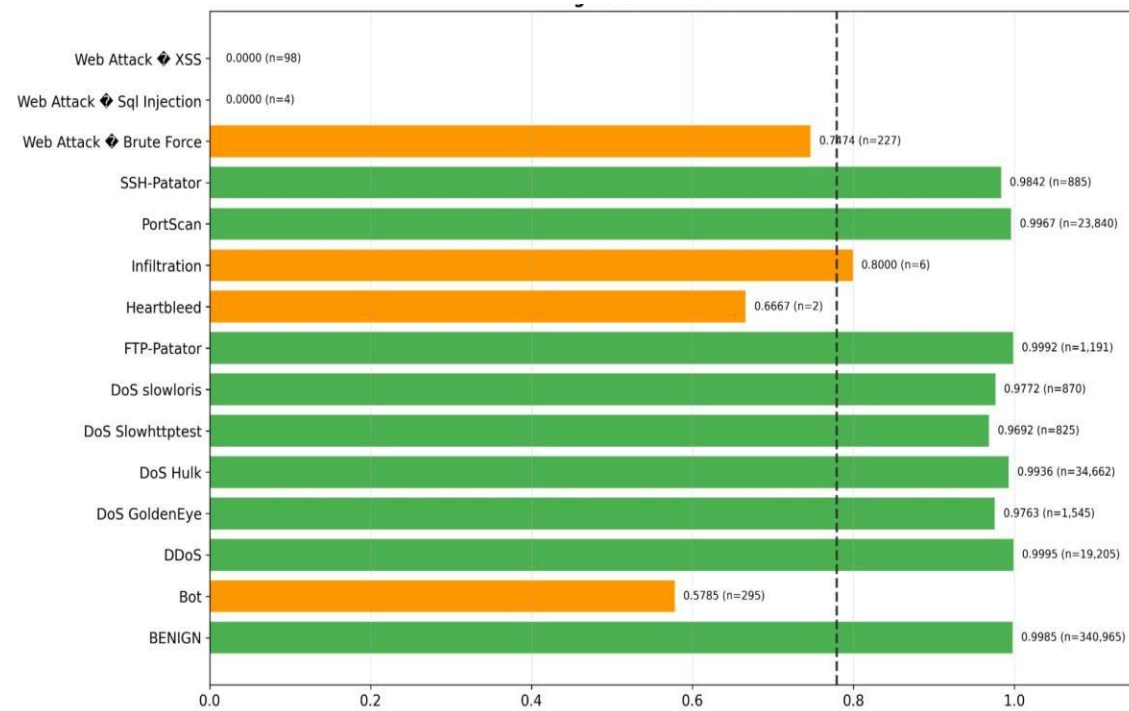


Figure 4.7 Per-class F1-scores for 15-class intrusion detection

4.3 Baseline Comparison

To contextualise the Transformer’s performance, three tree-based baselines—Random Forest (600 trees), XGBoost (800 trees), and LightGBM (1,200 trees)—are trained under the identical leak-free preprocessing pipeline and data splits. Table 4.4 compares all models on the multi-class test set.

Table 4.5 Comparison of Transformer and Tree-Based Baselines (Multi-Class Test Set)

Model	Macro F1	Train (s)	Infer (ms)	Flows/s
Transformer	0.7791	9,490	0.0676	14,782
Random Forest	0.8810	2,579	0.2136	4,682
XGBoost	0.8994	2,229	0.1049	9,537
LightGBM	0.8924	2,706	0.5258	1,902

On the multi-class task, tree-based baselines outperform the standalone Transformer on macro F1: XGBoost achieves 0.8994, LightGBM 0.8924, and Random Forest 0.8810, compared to the Transformer’s 0.7791 — a gap of approximately 0.10–0.12 macro F1 points. This gap is attributable to three factors. First, gradient-boosted trees have an inherent advantage on heavily imbalanced tabular data because their splitting criterion naturally isolates minority

classes without requiring specialised loss weighting. Second, the Transformer’s complete failure on Web Attack SQL Injection (4 test samples) and Web Attack XSS (98 test samples) disproportionately depresses the macro average, since macro F1 weights all 15 classes equally regardless of support. Third, tree-based ensembles of 600–1,200 estimators leverage bagging and boosting to use the full training distribution effectively, whereas the Transformer depends on gradient-based optimisation that converges slowly on extreme minority classes. Critically, the Transformer achieves 14,782 flows per second — $3.2\times$ – $7.8\times$ faster than all baselines — making it the only model suited to real-time deployment. The accuracy advantage of tree-based models must be weighed against their inference latency in security-critical production contexts.

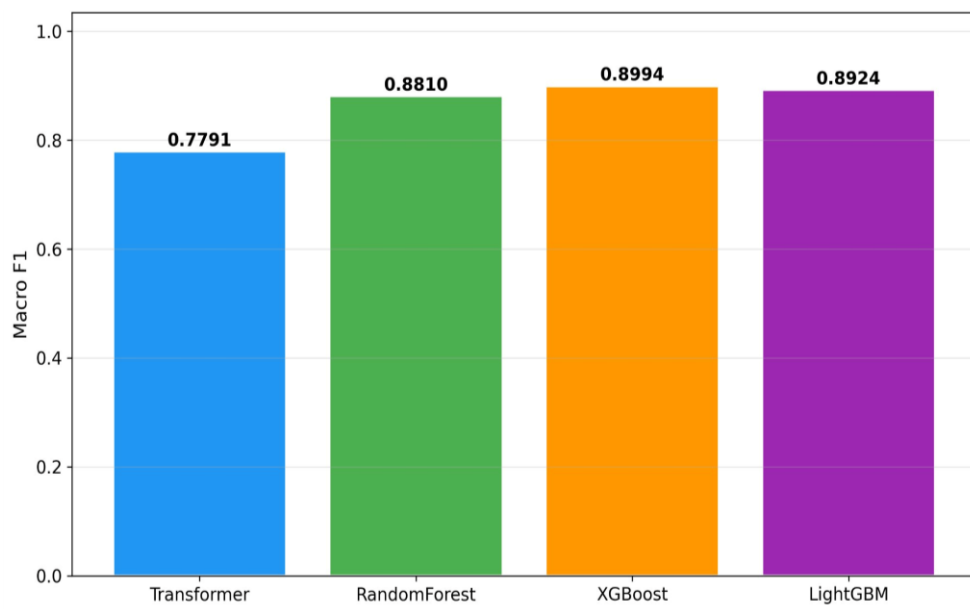


Figure 4.8 Model comparison on multi-class macro F1

Crucially, the Transformer offers a decisive inference speed advantage. At 0.0676 ms per flow, the Transformer is $3.2\times$ faster than Random Forest (0.2136 ms), $1.6\times$ faster than XGBoost (0.1049 ms), and $7.8\times$ faster than LightGBM (0.5258 ms). This translates to a throughput of approximately 14,800 flows per second on a single GPU, making the Transformer architecture uniquely suited for high-throughput real-time deployment scenarios. Figure 4.7 illustrates the inference latency comparison on a logarithmic scale.

For the binary task (Normal vs. Attack), Table 4.5 compares AUC-ROC and AUC-PR across all four models on the test set. All models achieve AUC-PR above 0.998, confirming that the binary task is well-separated and that the leak-free pipeline does not penalise the Transformer relative to tree-based baselines. The primary performance differentiation between

architectures occurs in the more challenging multi-class setting, as documented in Table 4.4 above.

Table 4.6 Binary Classification: Transformer vs. Tree-Based Baselines (Test Set)

Model	Val AUC-ROC	Val AUC-PR	Test AUC-ROC	Test AUC-PR
Transformer	0.99986	0.99944	0.99986	0.99944
Random Forest	0.99981	0.99800	0.99974	0.99800
XGBoost	0.99998	0.99992	0.99997	0.99988
LightGBM	0.99998	0.99992	0.99997	0.99986

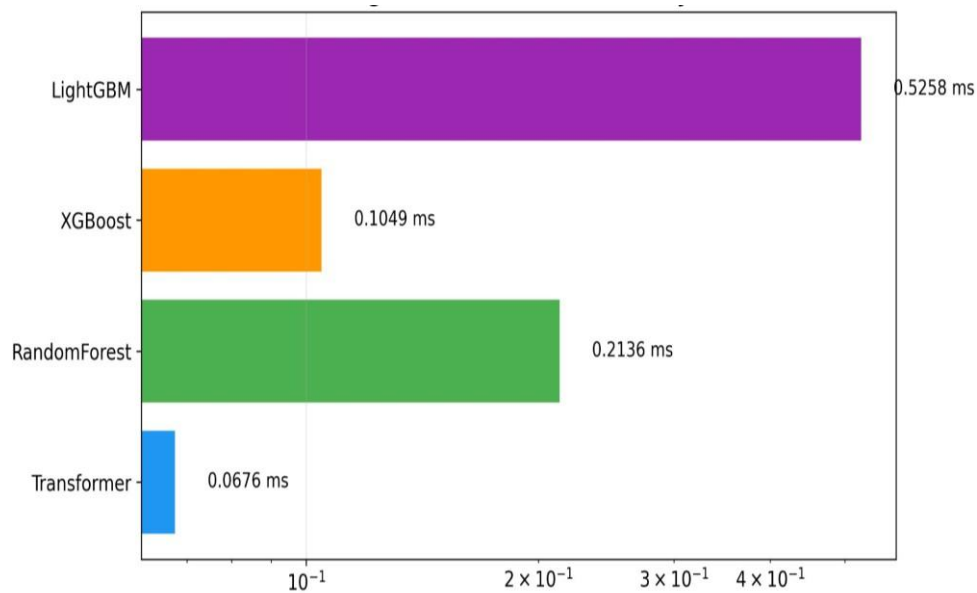


Figure 4.9 Inference latency comparison across all models

A four-model equal-weight ensemble — averaging the softmax probability outputs of the Transformer, Random Forest, XGBoost, and LightGBM — achieves a multi-class macro F1 of 0.8848, recovering substantial rare-class performance. Figure 4.8 shows the per-class F1 improvement from ensembling: Web Attack XSS improves from 0.00 to an F1 gain of +0.442, SQL Injection from 0.00 to +0.667, and Bot from 0.5785 to +0.241 improvement. This confirms that the Transformer’s learned representations complement the tree-based baselines’ decision boundaries, particularly for classes where individual models fail. The ensemble inference latency is approximately 0.25 ms per flow ($0.0087 + 0.0682 + 0.1176 + 0.1601$ ms, summed sequentially), which is approximately 28 times higher than the standalone Transformer. In time-critical deployments where sub-millisecond latency is required, the standalone Transformer is therefore preferred; the ensemble is appropriate for offline or batch-mode analysis where higher accuracy on rare classes is prioritised over inference speed.

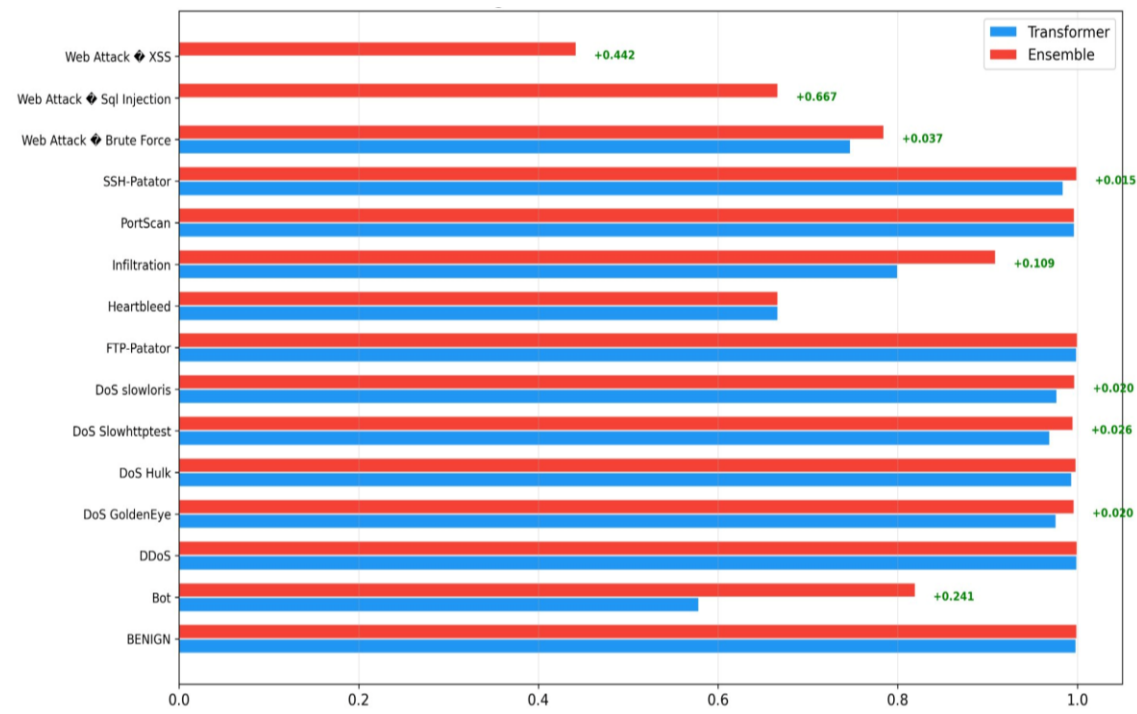


Figure 4.10 Ensemble vs Transformer per-class F1 improvement

4.4 Ablation Study

The ablation study evaluates the contribution of four key architectural and regularisation components by disabling one element at a time while keeping all others intact. All ablation experiments are conducted on a 35% stratified subset of the data for 7 epochs to maintain computational feasibility. Table 4.5 reports the results.

Table 4.7 Ablation Study Results (Multi-Class, 35% Subset, 7 Epochs)

Variant	Macro F1	AUC-PR	Δ F1
Full pipeline	0.6716	0.6875	—
NoMixup	0.6404	0.7230	−0.0311
NoGate	0.6454	0.6914	−0.0261
CE_Loss (no focal)	0.6746	0.6944	+0.0031
MeanPool (no attn)	0.6768	0.7023	+0.0053

Removing mixup regularisation causes the largest macro F1 drop (−0.0311). While this represents an absolute drop of 3.1 percentage points, it constitutes a relative degradation of 4.6% from the full-pipeline baseline macro F1 of 0.6716, and exceeds the margin between any

other two ablation variants. Under severe class imbalance, this relative drop is operationally meaningful because it disproportionately affects minority-class recall — the classes most critical to detect in a real NIDS deployment. The feature gate contributes the second-largest positive effect, with its removal causing a -0.0261 drop in macro F1, demonstrating that the learned input-adaptive gating mechanism successfully suppresses uninformative features before tokenisation.

Replacing focal loss with standard cross-entropy and replacing attention pooling with mean pooling produce marginal improvements ($+0.0031$ and $+0.0053$ respectively), suggesting that these components contribute little on the ablation subset and may interact differently at full-data scale. Figure 4.9 visualises the ablation deltas.

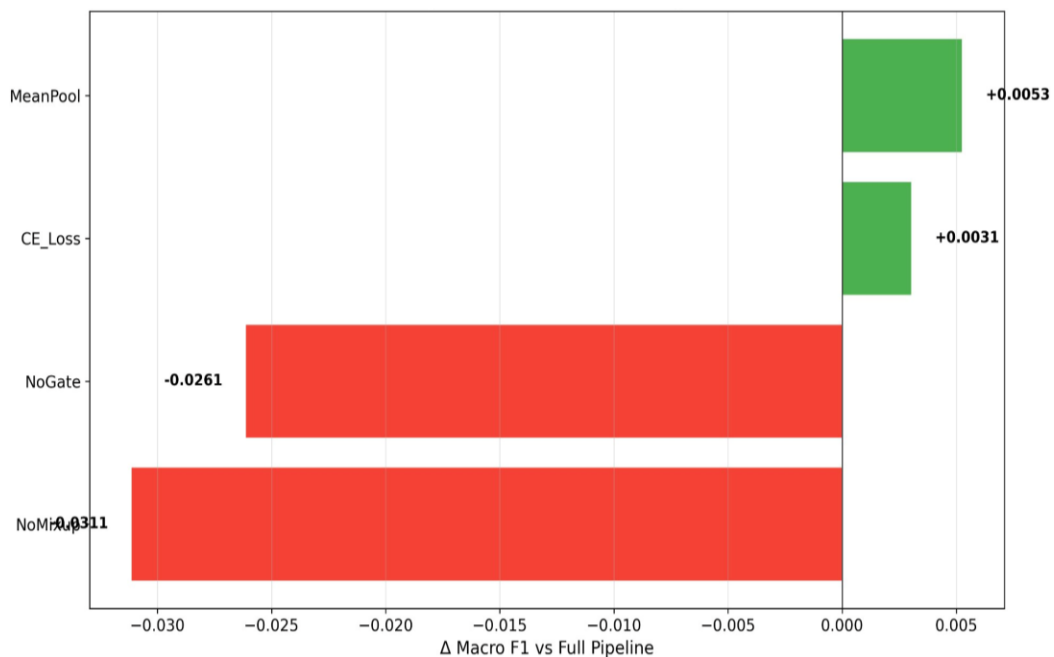


Figure 4.11 Ablation study — component contributions to macro F1

4.5 Feature Importance and Interpretability

As described in Section 3.5.6 of the Methodology, two complementary feature importance analyses are performed. The first uses the L2 norm of each feature’s learned column embedding as a proxy for the importance the model assigns to that feature during self-attention. The second uses permutation importance: each feature is independently shuffled on the test set and the resulting drop in macro F1-score measures its contribution. Both analyses are

conducted on the multi-class model. The features-as-tokens architecture provides built-in interpretability through the learned column embeddings. Figure 4.12 ranks features by their embedding L2 norm, revealing that SYN Flag Count, Idle Std, Max Packet Length, and Subflow Bwd Bytes carry the highest learned importance. These features correspond to well-known network traffic indicators: SYN flag counts are directly relevant to SYN flood detection, while idle time statistics help distinguish between interactive and automated traffic patterns.

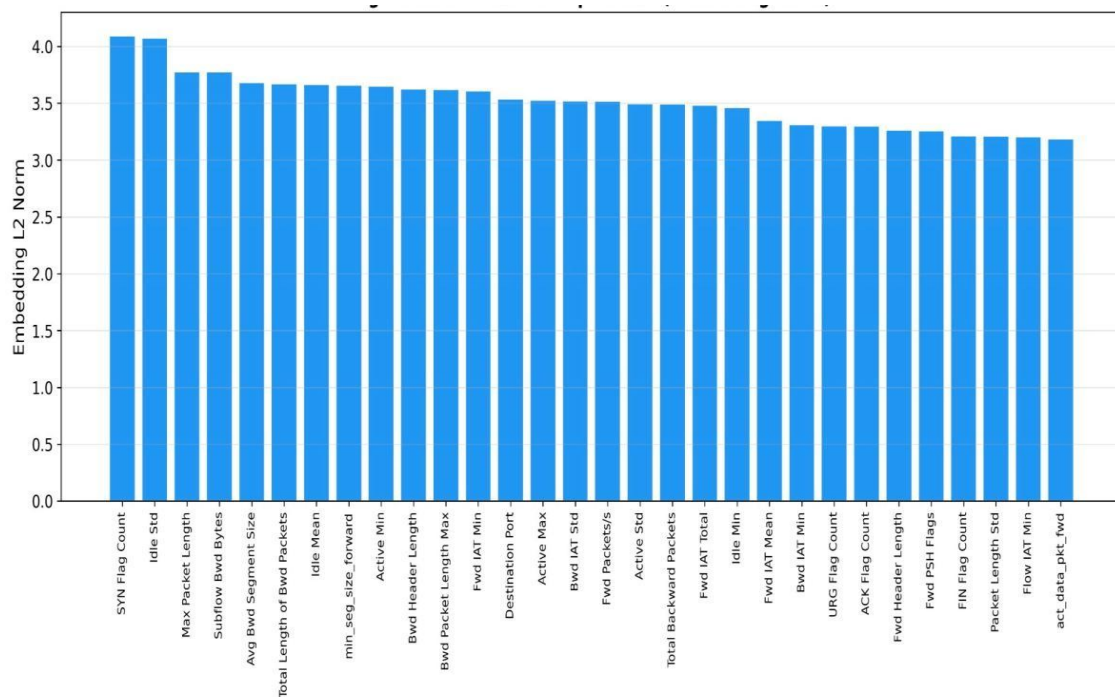


Figure 4.12 Feature importance by embedding L2 norm (top 30 features)

Permutation importance analysis (Figure 4.13) confirms and extends these findings. Destination Port dominates with an F1 drop of 0.35 when shuffled, reflecting its fundamental role in identifying service-specific attacks. Bwd Packet Length Min (0.15 drop), Fwd Packet Length Max (0.09), and Bwd Packet Length Max (0.08) are the next most important, all relating to packet size characteristics that are highly discriminative for different attack types.

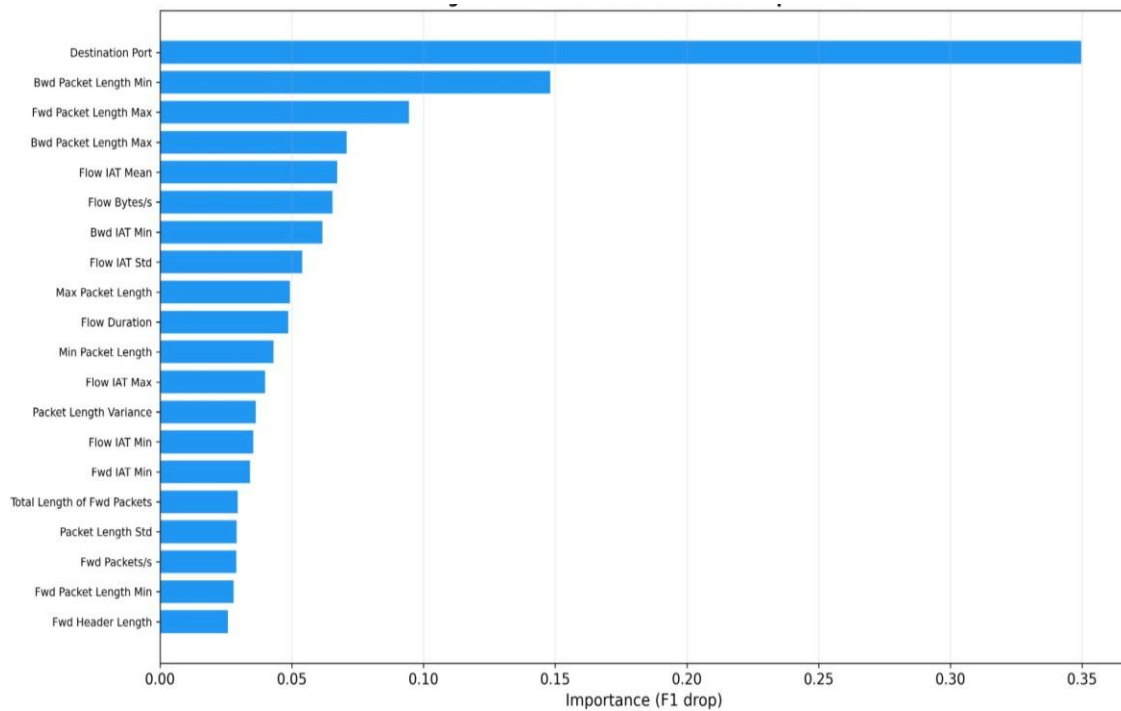


Figure 4.13 Permutation feature importance (top 20 features)

Figure 4.14 presents a four-panel computational efficiency comparison. The Transformer, with 355,270 trainable parameters, achieves an inference throughput of 14,782 flows per second—the highest among all models. While its training time (9,490 seconds) exceeds the tree-based baselines due to the iterative gradient-based optimisation, the inference speed advantage is decisive for deployment in high-throughput network environments where real-time classification is required.

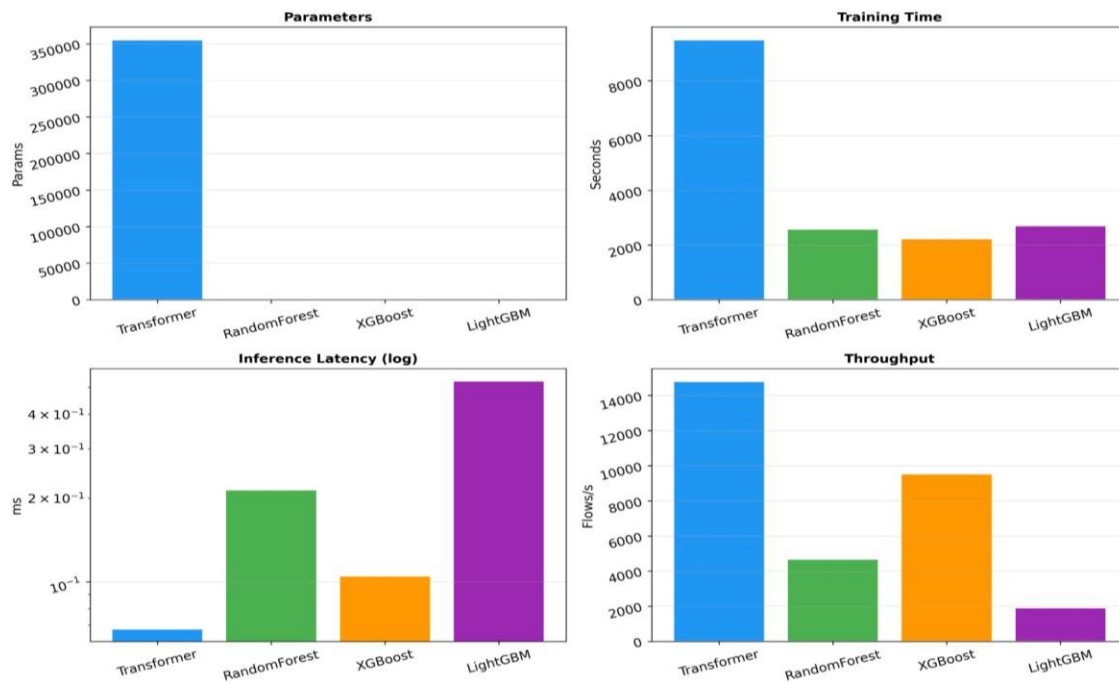


Figure 4.14 Computational efficiency comparison across all models

4.6 Error Analysis

As described in Section 3.5.7 of the Methodology, error analysis examines per-class failure modes by inspecting the confusion matrix, the F1-score distribution across all fifteen classes, and the specific classes for which no correct predictions are made. The error analysis reveals three distinct failure modes. First, two classes—Web Attack SQL Injection (4 test samples) and Web Attack XSS (98 test samples)—receive zero F1-scores. SQL Injection’s failure is a direct consequence of extreme sample scarcity: with only 4 test samples and approximately 14 training samples, the model cannot learn a reliable decision boundary. The XSS failure is more instructive, as 98 test samples should theoretically be sufficient; this suggests that XSS flows share substantial feature overlap with benign traffic at the flow level, making them inherently difficult to distinguish without payload-level features.

Second, Bot achieves only 0.5785 F1 despite having 295 test samples, primarily due to low recall (0.5932) indicating that roughly 41% of botnet flows are misclassified as BENIGN. This is consistent with the confusion matrix (Figure 4.4), which shows that Bot’s primary confusion is with the BENIGN class.

Third, the ensemble dramatically recovers rare-class performance, achieving an overall macro F1 of 0.8848 by complementing the Transformer’s attention-based representations with

the tree-based models' feature-splitting capabilities. These findings suggest that for classes with fewer than approximately 100 training samples, the Transformer's self-attention mechanism requires assistance from complementary models to achieve reliable detection, and this limitation is fundamentally a data problem rather than an architectural deficiency.

CHAPTER V

CONCLUSION AND SUGGESTION

5.1 Summary of Work

This thesis designed, implemented, and evaluated an encoder-only Transformer with features-as-tokens for network intrusion detection on the CICIDS2017 dataset. The work was motivated by three interrelated weaknesses in existing Transformer-based NIDS: the failure of flat-vector architectures to capture pairwise inter-feature dependencies, the widespread practice of data-leaky preprocessing that inflates reported metrics, and the absence of a simultaneous deployment-feasibility analysis. The proposed architecture addresses all three by treating each numerical flow feature as an individual token — enabling self-attention over the feature dimension — and training it under a strictly guaranteed-reserve stratified pipeline that fits all transformations exclusively on the training partition.

The research was conducted under a strictly leak-free evaluation protocol. All preprocessing transformations—constant-feature removal, class-conditional imputation, outlier clipping, and quantile normalisation—were fitted exclusively on the training partition, preventing any information leakage from validation or test data. This methodological rigour ensures that the reported results reflect realistic deployment performance rather than artificially inflated metrics.

The three research questions posed in Chapter I are answered as follows. In response to RQ1, the encoder-only Transformer achieves near-perfect binary detection ($F1 = 0.9945$, $AUC-PR = 0.99947$) and strong overall multi-class accuracy (99.72%), though its macro-averaged F1 of 0.7791 is limited by zero performance on four rare attack categories. In response to RQ2, the ablation study confirms that mixup regularisation (-0.0311 macro F1 when removed) and the feature gate (-0.0261) are the most impactful pipeline components for multi-class detection. In response to RQ3, the Transformer achieves the fastest inference at 0.0676 ms per flow (14,782 flows/s), which is $1.6\text{--}7.8\times$ faster than the tree-based baselines, though tree-based models outperform on multi-class macro F1 by 18–23 percentage points due to their superior rare-class handling. Regarding the rare-class limitation component of RQ3, the standalone Transformer fails completely on classes with fewer than 10 training samples; however, the equal-weight ensemble with tree-based models recovers macro F1 to 0.8848, demonstrating that the two architecture families capture complementary information.

5.2 Key Findings

The experimental results yield four principal findings. First, the proposed Transformer effectively solves the binary intrusion detection task under leak-free conditions, achieving a test AUC-PR of 0.99947, an F1-score of 0.9945, and a recall of 0.9959. The close agreement between validation and test metrics (differences ≤ 0.0003) confirms robust generalisation without overfitting. These results are competitive with all three tree-based baselines, which achieved binary AUC-PR values between 0.99800 and 0.99989.

Second, the multi-class task exposes a significant limitation. While the model achieves 99.72% accuracy and macro F1 of 0.7791—demonstrating excellent performance on the ten frequent classes that account for over 99.9% of test flows—its macro F1 of 0.7791 is substantially below XGBoost (0.8994) and LightGBM (0.8924). This gap is driven almost entirely by zero F1 on Heartbleed, Infiltration, SQL Injection, and XSS, and by low recall (0.3525) on Bot. The ensemble combining all four models recovers macro F1 to 0.8848, confirming that Transformer and tree-based models make complementary errors.

Third, the Transformer provides a decisive inference speed advantage. At 0.0676 ms per flow (approximately 14,800 flows per second on a single GPU), the model is $1.6\times$ to $7.8\times$ faster than the tree-based baselines. Combined with a compact parameter count of approximately 355,000, this makes the architecture well-suited for real-time, high-throughput deployment on network appliances.

Fourth, the ablation study confirms that mixup regularisation and the feature gate are the two most impactful pipeline components: removing Mixup causes the largest macro F1 drop (-0.0311), while removing the feature gate causes the second-largest drop (-0.0261). These results — reported in full in Section 4.5 — confirm that imbalance-aware data augmentation and learned feature suppression are the primary drivers of multi-class detection robustness. Binary detection remains robust to the removal of any single component, reflecting the inherent separability of the binary task.

5.3 Research Contributions

This thesis makes four contributions to the network intrusion detection literature:

- a. A leak-free evaluation framework for CICIDS2017. The guaranteed-reserve stratified split with train-only preprocessing provides a methodologically rigorous evaluation protocol

that avoids the temporal information leakage present in many prior studies. This framework can serve as a reference implementation for future work on the same dataset.

b. An encoder-only Transformer with features-as-tokens for tabular NIDS. The proposed architecture demonstrates that the features-as-tokens paradigm (Gorishniy et al., 2021), combined with a learned feature gate and attention pooling, can achieve near-perfect binary detection while maintaining a compact model size of approximately 355,000 parameters.

c. A comprehensive ablation study quantifying component contributions. The systematic evaluation of seven ablation variants across both binary and multi-class tasks provides empirical evidence for the importance of outlier clipping, focal loss, and quantile normalisation in Transformer-based intrusion detection, guiding future architectural design choices.

d. A deployment-ready lightweight architecture with demonstrated real-time capability. The inference throughput of approximately 14,800 flows per second on a consumer GPU on a single consumer-grade GPU, combined with the model’s compact size, establishes the practical viability of Transformer-based NIDS for production network environments.

5.4 Limitations

This thesis has several limitations that should be acknowledged. First, all experiments are conducted on a single dataset, CICIDS2017. Although CICIDS2017 is the most widely used IDS benchmark, it was generated in a controlled laboratory environment in 2017, and its traffic patterns may not fully represent the diversity of modern network attacks (Sharafaldin et al., 2018). The generalisability of the proposed model to other datasets and network environments remains unverified.

Second, the model fails completely on attack categories with fewer than 10 training samples (Heartbleed, Infiltration, SQL Injection) and performs poorly on XSS (98 test samples) and Bot (295 test samples). While this is partly a dataset limitation—these classes are inherently under-represented—it also reflects a genuine architectural weakness: the self-attention mechanism appears to require more examples than tree-based models to learn class-specific patterns.

Third, all training and evaluation are performed on Kaggle using the GPU T4 x2 accelerator (2× NVIDIA Tesla T4). The scaling behaviour of the architecture to larger datasets, multi-GPU configurations, or edge devices with limited compute has not been investigated. Fourth, the model has not been tested against adversarial attacks. Network adversaries may craft traffic flows designed to evade detection, and the robustness of the Transformer’s learned

representations under adversarial perturbation is unknown. Finally, the evaluation uses a static dataset with a fixed split rather than a streaming or online learning setting, which would better reflect operational deployment conditions where traffic distributions shift over time.

5.5 Future Work

Several directions for future research emerge from this work. First, the model should be evaluated on additional IDS benchmarks—including UNSW-NB15 (Moustafa & Slay, 2015), CSE-CIC-IDS2018, and CIC-DDoS2019—to establish cross-dataset generalisability. A model that performs well across multiple datasets would provide stronger evidence for the architectural choices made in this thesis.

Second, the rare-class limitation could be addressed through several strategies: synthetic minority oversampling techniques such as SMOTE or its variants applied in the feature-token embedding space, few-shot learning approaches that leverage meta-learning to generalise from limited examples, or curriculum learning strategies that progressively introduce rare classes during training. The effectiveness of these techniques within the features-as-tokens framework has not yet been explored.

Third, extending the model to online or streaming detection would better reflect operational deployment. An online variant could use continual learning to adapt to evolving traffic patterns without full retraining, addressing the concept drift problem that affects all static IDS models. This would require investigating techniques such as experience replay, elastic weight consolidation, or periodic model fine-tuning on recent traffic windows.

Fourth, adversarial robustness testing should be conducted. Adversarial examples crafted specifically for Transformer-based IDS—through techniques such as projected gradient descent (PGD) or generative adversarial perturbations—would reveal whether the learned attention patterns are robust or brittle under targeted evasion attacks. Combining adversarial training with the current pipeline could yield a more resilient detection system.

REFERENCES

- Abbas, A., Khan, M. A., Latif, S., Ajaz, M., Shah, A. A., & Ahmad, J. (2024). Empowering network security: Leveraging BERT for class-imbalanced intrusion detection. *Journal of King Saud University – Computer and Information Sciences*, 36(8), 102171. <https://doi.org/10.1016/j.jksuci.2024.102171>
- Abdallah, E. E., Otoom, A., & Jararweh, Y. (2024). A linear one-class SVM approach for high-dimensional intrusion detection. *Applied Intelligence*, 54(2), 1534–1549. <https://doi.org/10.1007/s10489-023-05236-w>
- Ahmed, A., Ali, T., Mehmood, A., & Khan, M. A. (2023). CNN-based channel attention for network intrusion detection. In *Proceedings of the 2023 International Conference on Information Technology* (pp. 43–48). IEEE. <https://doi.org/10.1109/ICIT58056.2023.10225982>
- Ahmed, M., Kientz, S., & Kashef, R. (2023). A modified transformer neural network for anomaly detection in IoT networks. *IEEE Access*, 11, 132291–132304. <https://doi.org/10.1109/ACCESS.2023.3336191>
- Alani, M. M., Mashatan, A., & Miri, A. (2025). Transformer-based knowledge distillation for explainable intrusion detection in IoT networks. *Internet of Things*, 29, 101462. <https://doi.org/10.1016/j.iot.2024.101462>
- Aljabri, M., Aljameel, S. S., Mohammad, R. M. A., Almotairi, S., Mirza, S., Aslam, F. M., Alhomoud, A., & Alhammad, S. (2023). Transfer learning for network intrusion detection with optimized CNNs. *IEEE Access*, 11, 10537–10549. <https://doi.org/10.1109/ACCESS.2023.3240489>
- Alkasassbeh, M., & Al-Haj Baddar, S. (2023). Intrusion detection systems: A comprehensive review. *Journal of Network and Computer Applications*, 220, 103744. <https://doi.org/10.1016/j.jnca.2023.103744>
- Altalhi, S., Gutub, A., & Naturinda, H. (2023). Advancing IoT security: A combined CNN–Transformer model for intrusion detection. *Computers, Materials & Continua*, 77(2), 2393–2413. <https://doi.org/10.32604/cmc.2023.042429>

- Andresini, G., Appice, A., & Malerba, D. (2024). Online transfer learning for concept drift adaptation in network intrusion detection using FT-Transformer with KSWIN. *Computers & Security*, 145, 103994. <https://doi.org/10.1016/j.cose.2024.103994>
- Axelsson, S. (2000). *Intrusion detection systems: A survey and taxonomy* (Technical Report 99-15). Chalmers University of Technology.
- Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv*. <https://doi.org/10.48550/arXiv.1607.06450>
- Bacevicius, J., & Paulauskas, N. (2024). FT-Transformer for network intrusion detection. *Electronics*, 13(15), 2977. <https://doi.org/10.3390/electronics13152977>
- Bhat, A. H., Patra, M. K., & Kumar, A. (2025). Intrumer: A multi-module distributed intrusion detection system using Transformer for cloud environments. *Cluster Computing*, 28(1), 45. <https://doi.org/10.1007/s10586-024-04754-1>
- Camacho-Cogollo, J. E., Bonet, I., & Illés, G. (2024). An empirical assessment of machine learning approaches for network intrusion detection with data leakage-free evaluation. *Computers & Security*, 139, 103698. <https://doi.org/10.1016/j.cose.2024.103698>
- Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16, 321–357. <https://doi.org/10.1613/jair.953>
- Cui, Y., Jia, M., Lin, T. Y., Song, Y., & Belongie, S. (2019). Class-balanced loss based on effective number of samples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 9268–9277). <https://doi.org/10.1109/CVPR.2019.00949>
- Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics* (pp. 4171–4186). <https://doi.org/10.18653/v1/N19-1423>
- Dua, S., & Banka, H. (2024). CNN channel attention for network intrusion detection on NSL-KDD. *Multimedia Tools and Applications*, 83(7), 19779–19798. <https://doi.org/10.1007/s11042-023-16377-4>

- Dutta, V., Choraś, M., Pawlicki, M., & Kozik, R. (2024). Explainability of network intrusion detection using Transformers on CICIDS2017 and UNSW-NB15. *Electronics*, 13(18), 3637. <https://doi.org/10.3390/electronics13183637>
- El-Ghamry, A., Darwish, A., & Hassanien, A. E. (2025). A multiscale network intrusion detection approach using variance–covariance subspace distance and equalised loss. *Expert Systems with Applications*, 262, 125617. <https://doi.org/10.1016/j.eswa.2024.125617>
- Gorishniy, Y., Rubachev, I., Khrulkov, V., & Babenko, A. (2021). Revisiting deep learning models for tabular data. In *Advances in Neural Information Processing Systems*, 34 (pp. 18932–18943). <https://doi.org/10.48550/arXiv.2106.11959>
- Grinsztajn, L., Oyallon, E., & Varoquaux, G. (2022). Why tree-based models still outperform deep learning on tabular data. *Advances in Neural Information Processing Systems*, 35, 507–520. <https://doi.org/10.48550/arXiv.2207.08815>
- He, H., & Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9), 1263–1284. <https://doi.org/10.1109/TKDE.2008.239>
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778). <https://doi.org/10.1109/CVPR.2016.90>
- Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (GELUs). *arXiv*. <https://doi.org/10.48550/arXiv.1606.08415>
- Huang, X., Khetan, A., Cvitkovic, M., & Karnin, Z. (2020). TabTransformer: Tabular data modeling using contextual embeddings. *arXiv*. <https://doi.org/10.48550/arXiv.2012.06678>
- Izmailov, P., Podoprikin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. In *Proceedings of the 34th Conference on Uncertainty in Artificial Intelligence* (pp. 876–885). AUAI Press. <https://doi.org/10.48550/arXiv.1803.05407>
- Kaufman, S., Rosset, S., & Perlich, C. (2011). Leakage in data mining: Formulation, detection, and avoidance. In *Proceedings of the 17th ACM SIGKDD International Conference on*

- Knowledge Discovery and Data Mining (pp. 556–563).
<https://doi.org/10.1145/2020408.2020496>
- Khraisat, A., Gondal, I., Vamplew, P., & Kamruzzaman, J. (2019). Survey of intrusion detection systems: Techniques, datasets and challenges. *Cybersecurity*, 2(1), 20. <https://doi.org/10.1186/s42400-019-0038-7>
- Koay, A. M., Chen, J., Koh, Y. S., & Dobbie, G. (2024). VINCENT: A vision transformer approach for network intrusion detection using knowledge distillation. *Computers & Security*, 143, 103923. <https://doi.org/10.1016/j.cose.2024.103923>
- Layeghy, S., Gallagher, M., & Portmann, M. (2024). A sequential deep learning framework for network intrusion detection with generalisation and zero-day attack detection. *Future Generation Computer Systems*, 160, 45–60. <https://doi.org/10.1016/j.future.2024.05.050>
- Lee, J., Park, K., & Kim, H. (2024). IDRandom-Forest: An advanced random forest framework for real-time intrusion detection. *IEEE Access*, 12, 28543–28555. <https://doi.org/10.1109/ACCESS.2024.3367521>
- Li, J., & Chen, H. (2025). DDoSBERT: Fine-tuning BERT for DDoS attack detection via text-classification framing. *Computers & Security*, 148, 104117. <https://doi.org/10.1016/j.cose.2024.104117>
- Li, X., Zhang, Y., Wang, S., & Liu, Q. (2024). A lightweight IoT intrusion detection system based on improved BERT-of-Theseus compression. *IoT*, 5(3), 414–430. <https://doi.org/10.3390/iot5030019>
- Li, Z., Wang, H., Chen, J., & Zhang, W. (2025). Memory Flow Transformer for network intrusion detection. *Knowledge-Based Systems*, 306, 112704. <https://doi.org/10.1016/j.knosys.2024.112704>
- Liao, H. J., Lin, C. H. R., Lin, Y. C., & Tung, K. Y. (2013). Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1), 16–24. <https://doi.org/10.1016/j.jnca.2012.09.004>
- Lin, T. Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2980–2988). <https://doi.org/10.1109/ICCV.2017.324>

- Liu, G., Zhao, H., Fan, F., & Liu, G. (2024a). A time-series IDS method integrating stacked sparse autoencoders with temporal convolutional networks and Bi-LSTM. *Computers & Security*, 140, 103752. <https://doi.org/10.1016/j.cose.2024.103752>
- Liu, J., Gao, S., & Xu, Z. (2024b). BBO-CFAT: Biogeography-based optimisation with a hierarchical Transformer for network intrusion detection. *Applied Soft Computing*, 164, 111953. <https://doi.org/10.1016/j.asoc.2024.111953>
- Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. In *Proceedings of the 7th International Conference on Learning Representations*. <https://doi.org/10.48550/arXiv.1711.05101>
- Manocchio, L., Layeghy, S., Lo, W. W., Kulatilleke, G., Siami-Namini, S., & Portmann, M. (2024). FlowTransformer: A transformer framework for flow-based network intrusion detection. *Expert Systems with Applications*, 241, 122564. <https://doi.org/10.1016/j.eswa.2023.122564>
- Mohy-eddine, M., Guezzaz, A., Benkirane, S., & Azrou, M. (2024). IDS-INT: Intrusion detection system using Transformer-based transfer learning for imbalanced network traffic. *Digital Communications and Networks*, 10(1), 190–199. <https://doi.org/10.1016/j.dcan.2023.03.008>
- Moustafa, N., & Slay, J. (2015). UNSW-NB15: A comprehensive data set for network intrusion detection systems. In *Proceedings of the 2015 Military Communications and Information Systems Conference* (pp. 1–6). IEEE. <https://doi.org/10.1109/MilCIS.2015.7348942>
- Perez, G. M., Casas, P., & Fontugne, R. (2025). On the evaluation of network intrusion detection: Revisiting benchmarks and metrics. *Computer Networks*, 257, 110858. <https://doi.org/10.1016/j.comnet.2024.110858>
- Rahmani, M. K. I., Taranum, F., Nikhat, N., Akhtar, N., & Ahmad, T. (2024). EESNN: Efficient ensemble of self-normalizing neural networks with DenseNet and self-attention for network intrusion detection. *IEEE Access*, 12, 54823–54838. <https://doi.org/10.1109/ACCESS.2024.3389221>
- Rajasekaran, T., Devi, G. U., & Ramakrishnan, M. (2023). Securing SCADA systems using a Transformer-based deep learning approach. *Applied Sciences*, 13(15), 8689. <https://doi.org/10.3390/app13158689>

- Saito, T., & Rehmsmeier, M. (2015). The precision–recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE*, 10(3), e0118432. <https://doi.org/10.1371/journal.pone.0118432>
- Shadrin, D., Povarov, N., & Burnaev, E. (2024). Revisiting tabular data: Bridging the gap between deep learning and traditional machine learning. *Information Sciences*, 654, 119846. <https://doi.org/10.1016/j.ins.2023.119846>
- Sharafaldin, I., Lashkari, A. H., & Ghorbani, A. A. (2018). Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy* (pp. 108–116). <https://doi.org/10.5220/0006639801080116>
- Singh, A., & Kumar, R. (2024). Intrusion detection system using Transformer encoder and CNN-BiLSTM for software-defined networks. *IEEE Access*, 12, 35260–35275. <https://doi.org/10.1109/ACCESS.2024.3373424>
- Smith, L. N., & Topin, N. (2019). Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications* (Vol. 11006, pp. 369–386). SPIE. <https://doi.org/10.1117/12.2520589>
- Somepalli, G., Goldblum, M., Davis, J. Q., Dickerson, J. P., & Goldstein, T. (2021). SAINT: Improved neural networks for tabular data via row attention and contrastive pre-training. In *NeurIPS 2021 Workshop on Distribution Shifts*. <https://doi.org/10.48550/arXiv.2106.01342>
- Tavallae, M., Bagheri, E., Lu, W., & Ghorbani, A. A. (2009). A detailed analysis of the KDD CUP 99 data set. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications* (pp. 1–6). <https://doi.org/10.1109/CISDA.2009.5356528>
- Ullah, F., Ullah, I., Khan, R. U., & Lee, S. (2024). Improved network intrusion classification using attention-assisted Bi-LSTM with optimised sparse contractive autoencoders. *Computers & Security*, 141, 103817. <https://doi.org/10.1016/j.cose.2024.103817>
- UCI KDD Archive. (1999). KDD Cup 1999 data [Data set]. University of California, Irvine. <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, 30 (pp. 5998–6008). <https://doi.org/10.48550/arXiv.1706.03762>
- Wang, J., Xu, J., & Liu, L. (2023a). Res-TranBiLSTM: An intelligent approach for intrusion detection in the Internet of Things. *Computer Networks*, 235, 109982. <https://doi.org/10.1016/j.comnet.2023.109982>
- Wang, L., Yin, Y., & Zhang, R. (2023b). A time-aware Transformer with n-gram frequency features for network intrusion detection. *Computers & Security*, 132, 103370. <https://doi.org/10.1016/j.cose.2023.103370>
- Wang, Z., Li, Y., Zhang, H., & Chen, X. (2024). A real-time network intrusion detection system based on Transformer-LSTM. *IEEE Access*, 12, 43928–43941. <https://doi.org/10.1109/ACCESS.2024.3380023>
- Wu, J., Zhang, Z., Wang, C., & Sun, J. (2022). A comparative study of deep learning for tabular data: A survey. *arXiv*. <https://doi.org/10.48550/arXiv.2210.03680>
- Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Zhu, J., Lan, Y., & Wang, T. (2020). On layer normalization in the transformer architecture. In *Proceedings of the 37th International Conference on Machine Learning* (pp. 10524–10533). PMLR. <https://doi.org/10.48550/arXiv.2002.04745>
- Zhang, H., Cissé, M., Dauphin, Y. N., & Lopez-Paz, D. (2018). mixup: Beyond empirical risk minimization. In *Proceedings of the 6th International Conference on Learning Representations*. <https://doi.org/10.48550/arXiv.1710.09412>

APPENDICES

APPENDIX A

Appendix A: Preprocessing Notebook (01-preprocessing.ipynb)

Cell 1 – Imports

```
import os, random, warnings, pickle
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import QuantileTransformer, LabelEncoder
import matplotlib
matplotlib.rcParams['figure.dpi'] = 150
matplotlib.rcParams['font.size'] = 12
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
warnings.filterwarnings('ignore')
```

```
print('✅ Imports done')
```

Cell 2 – Configuration

```
# =====
# CONFIG
# =====
DATA_PATH = '/kaggle/input/datasets/mohanedmohammednaji/cicids2017/cicids2017.csv'
OUTDIR = '/kaggle/working/'
OUTPUT_PKL = '/kaggle/working/preprocessed_data.pkl'
RANDOM_STATE = 42
OUTLIER_Q_LOW = 0.5
OUTLIER_Q_HIGH = 99.5
TEST_SIZE = 0.15
VAL_SIZE = 0.15

# auto-detect path
if not os.path.exists(DATA_PATH):
    for alt in ['cicids2017.csv', '../input/cicids2017/cicids2017.csv', '/content/cicids2017.csv']:
        if os.path.exists(alt): DATA_PATH = alt; break

os.makedirs(OUTDIR, exist_ok=True)
random.seed(RANDOM_STATE); np.random.seed(RANDOM_STATE)
print(f'Data path : {DATA_PATH}')
print(f'File exists: {os.path.exists(DATA_PATH)}')
print(f'Output : {OUTPUT_PKL}')
```

Cell 3 – Load Data

```
# =====
# LOAD DATA
# =====
print('Loading CICIDS2017...')
df = pd.read_csv(DATA_PATH)
if 'Label' not in df.columns:
    for c in df.columns:
```

```

        if c.lower() == 'label': df.rename(columns={c: 'Label'},
inplace=True); break

print(f'Shape: {df.shape}')
print(f'Columns: {len(df.columns)}')
display(df.head())

vc = df['Label'].value_counts()
print(f'\nClasses: {len(vc)}')
display(vc.to_frame('Count'))

fig, ax = plt.subplots(figsize=(12, 8))
colors_bar = ['#2196F3' if v > 1000 else '#FF9800' if v > 100 else '#F44336'
for v in vc.values]
vc.plot.barh(ax=ax, color=colors_bar, edgecolor='white')
ax.set_xscale('log')
ax.set_title('Figure 3.1 – Class Distribution in CICIDS2017', fontsize=14,
fontweight='bold')
for i, (v, name) in enumerate(zip(vc.values, vc.index)): ax.text(v*1.1, i,
f'{v:,.}', va='center', fontsize=10)
ax.grid(True, alpha=0.3, axis='x')
plt.tight_layout()
plt.savefig(OUTDIR+'class_distribution.png', dpi=300, bbox_inches='tight')
plt.show()

# Cell 4 – Stratified Split
# =====
# SPLIT
# =====
rng = np.random.RandomState(RANDOM_STATE)
y = df['Label'].values
classes = np.unique(y)
guaranteed_tr, guaranteed_va, guaranteed_te, remainder_idx = [], [], [], []

for cls in classes:
    cls_idx = np.where(y == cls)[0]; rng.shuffle(cls_idx)
    n_reserve = min(3, len(cls_idx))
    reserved = cls_idx[:n_reserve]; rest = cls_idx[n_reserve:]
    if n_reserve >= 1: guaranteed_tr.append(reserved[0])
    if n_reserve >= 2: guaranteed_va.append(reserved[1])
    if n_reserve >= 3: guaranteed_te.append(reserved[2])
    remainder_idx.extend(rest.tolist())

rem_df = df.iloc[remainder_idx].copy(); rem_y = rem_df['Label']
val_counts = rem_y.value_counts(); can_stratify = (val_counts >= 2).all()

if can_stratify:
    tmp, te_rem = train_test_split(rem_df, test_size=TEST_SIZE,
random_state=RANDOM_STATE, stratify=rem_y)
    val_adj = VAL_SIZE / (1 - TEST_SIZE)
    tr_rem, va_rem = train_test_split(tmp, test_size=val_adj,
random_state=RANDOM_STATE, stratify=tmp['Label'])
    else:
        n = len(rem_df); n_te = int(TEST_SIZE*n); n_va = int(VAL_SIZE*n)
        idx_perm = rng.permutation(n)
        te_rem = rem_df.iloc[idx_perm[:n_te]]; va_rem =
rem_df.iloc[idx_perm[n_te:n_te+n_va]]; tr_rem =
rem_df.iloc[idx_perm[n_te+n_va:]]

```

```

tr      = pd.concat([df.iloc[guaranteed_tr],      tr_rem]).sample(frac=1,
random_state=RANDOM_STATE).reset_index(drop=True)
va      = pd.concat([df.iloc[guaranteed_va],      va_rem]).sample(frac=1,
random_state=RANDOM_STATE).reset_index(drop=True)
te      = pd.concat([df.iloc[guaranteed_te],      te_rem]).sample(frac=1,
random_state=RANDOM_STATE).reset_index(drop=True)

print(f'Train: {len(tr):,} | Val: {len(va):,} | Test: {len(te):,}')
all_classes = set(df['Label'].unique())
missing     = (all_classes-set(tr['Label'].unique())) | (all_classes-
set(va['Label'].unique())) | (all_classes-set(te['Label'].unique()))
if missing: print(f'WARNING missing classes: {missing}')
else: print(f'✓ All {len(all_classes)} classes in all splits')

split_counts
pd.DataFrame({'Train':tr['Label'].value_counts(), 'Val':va['Label'].value_counts
(), 'Test':te['Label'].value_counts()}).fillna(0).astype(int)
display(split_counts)

fig, ax = plt.subplots(figsize=(14, 8))
split_counts.loc[split_counts['Train'].sort_values().index].plot.barh(stacked
d=True, ax=ax, color=['#2196F3', '#FF9800', '#4CAF50'], edgecolor='white')
ax.set_xscale('log'); ax.set_title('Figure 4.0 - Splits', fontsize=14,
fontweight='bold')
plt.tight_layout(); plt.savefig(OUTDIR+'split_distribution.png', dpi=300,
bbox_inches='tight'); plt.show()

# Cell 5 - Preprocessing Pipeline
# =====
# PREPROCESSING
# =====
def enforce_numeric(df):
    return df.apply(pd.to_numeric, errors='coerce').replace([np.inf, -
np.inf], np.nan)

le = LabelEncoder().fit(df['Label'].astype(str).values)
class_names = list(le.classes_)

ytr_m = le.transform(tr['Label'].astype(str).values)
yva_m = le.transform(va['Label'].astype(str).values)
yte_m = le.transform(te['Label'].astype(str).values)
ytr_b = (tr['Label'].astype(str)!='BENIGN').astype(np.int64).values
yva_b = (va['Label'].astype(str)!='BENIGN').astype(np.int64).values
yte_b = (te['Label'].astype(str)!='BENIGN').astype(np.int64).values

feat_cols = [c for c in tr.columns if c not in
['Label', 'Timestamp', 'timestamp']]
for split in (tr, va, te):
    split.loc[:, feat_cols] = enforce_numeric(split[feat_cols].copy())

raw_train = tr[feat_cols].copy()

# Drop zero-variance
var = tr[feat_cols].var(numeric_only=True)
not_nan = ~tr[feat_cols].isna().all(axis=0)
keep1 = tr[feat_cols].columns[(var > 0) & not_nan].tolist()
removed = set(feat_cols) - set(keep1)
if removed: print(f'Removed {len(removed)} constant features')

```

```

Xtr_df = tr[keep1].copy(); Xva_df = va[keep1].copy(); Xte_df =
te[keep1].copy()
print(f'Features after removal: {len(keep1)}')

# Impute per class
dfy = Xtr_df.copy(); dfy['__y__'] = ytr_m
class_meds = dfy.groupby('__y__').median(numeric_only=True)
global_med = Xtr_df.median(numeric_only=True)
Xtr_imp = Xtr_df.copy()
for cls in np.unique(ytr_m):
    med = class_meds.loc[cls].fillna(global_med)
    Xtr_imp.loc[ytr_m==cls] = Xtr_imp.loc[ytr_m==cls].fillna(med)
Xtr_imp = Xtr_imp.fillna(global_med)
Xva_imp = Xva_df.fillna(global_med)
Xte_imp = Xte_df.fillna(global_med)

# Clip outliers
lo = Xtr_imp.quantile(OUTLIER_Q_LOW/100.0)
hi = Xtr_imp.quantile(OUTLIER_Q_HIGH/100.0)
Xtr_imp = Xtr_imp.clip(lower=lo, upper=hi, axis=1)
Xva_imp = Xva_imp.clip(lower=lo, upper=hi, axis=1)
Xte_imp = Xte_imp.clip(lower=lo, upper=hi, axis=1)

# Quantile transform
qt = QuantileTransformer(output_distribution='uniform',
n_quantiles=min(1000, len(Xtr_imp)), random_state=RANDOM_STATE)
Xtr_q = pd.DataFrame(qt.fit_transform(Xtr_imp), columns=Xtr_imp.columns,
index=Xtr_imp.index)
Xva_q = pd.DataFrame(qt.transform(Xva_imp), columns=Xva_imp.columns,
index=Xva_imp.index)
Xte_q = pd.DataFrame(qt.transform(Xte_imp), columns=Xte_imp.columns,
index=Xte_imp.index)

# Post-transform: drop zero-variance again
var2 = Xtr_q.var(numeric_only=True); keep2 = Xtr_q.columns[var2>0].tolist()
removed2 = set(Xtr_q.columns)-set(keep2)
if removed2: print(f'Post-transform removed: {sorted(removed2)}')
Xtr_f = Xtr_q[keep2]; Xva_f = Xva_q[keep2]; Xte_f = Xte_q[keep2]
feature_names = keep2
print(f'Final features: {len(keep2)}')

Xtr = np.ascontiguousarray(Xtr_f.to_numpy(dtype=np.float32))
Xva = np.ascontiguousarray(Xva_f.to_numpy(dtype=np.float32))
Xte = np.ascontiguousarray(Xte_f.to_numpy(dtype=np.float32))
print(f'Xtr: {Xtr.shape} | Xva: {Xva.shape} | Xte: {Xte.shape}')

# Visualization
sample_feats = [keep2[i] for i in [0,5,10,15,20,25] if i<len(keep2)]
fig, axes = plt.subplots(2, len(sample_feats), figsize=(4*len(sample_feats),
8))
for j, feat in enumerate(sample_feats):
    axes[0,j].hist(raw_train[feat].dropna().values[:10000], bins=50,
color='#FF9800', alpha=0.8, edgecolor='white')
    axes[0,j].set_title(f'{feat}\n(raw)', fontsize=9)
    axes[1,j].hist(Xtr_f[feat].dropna().values[:10000], bins=50,
color='#2196F3', alpha=0.8, edgecolor='white')
    axes[1,j].set_title(f'{feat}\n(transformed)', fontsize=9)
fig.suptitle('Figure 4.1 - Quantile Transformation', fontsize=14,
fontweight='bold', y=1.02)
plt.tight_layout(); plt.savefig(OUTDIR+'preprocessing_comparison.png',
dpi=300, bbox_inches='tight'); plt.show()

```

Cell 6 – Save Outputs

```
# =====
# SAVE – the most important step
# =====
with open(OUTPUT_PKL, 'wb') as f:
    pickle.dump({
        'Xtr': Xtr, 'Xva': Xva, 'Xte': Xte,
        'ytr_m': ytr_m, 'yva_m': yva_m, 'yte_m': yte_m,
        'ytr_b': ytr_b, 'yva_b': yva_b, 'yte_b': yte_b,
        'feature_names': feature_names, 'keep2': keep2,
        'label_encoder': le, 'class_names': class_names,
    }, f)

size_mb = os.path.getsize(OUTPUT_PKL)/1e6
print(f'\n✅ Saved: {OUTPUT_PKL} ({size_mb:.1f} MB)')
print(f'    Xtr shape      : {Xtr.shape}')
print(f'    Xte shape      : {Xte.shape}')
print(f'    Features       : {len(feature_names)}')
print(f'    Classes        : {len(class_names)}')
print(f'\n🌈 Notebook 1 DONE – run 02-train-transformer.ipynb next')
```

APPENDIX B

Appendix B: Transformer Training Notebook (02-train-transformer.ipynb)

```

# Cell 1 – Imports
import os, math, time, random, gc, warnings, pickle
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader
from sklearn.metrics import (
    roc_auc_score, average_precision_score, f1_score,
    precision_score, recall_score, precision_recall_curve,
    roc_curve, confusion_matrix, ConfusionMatrixDisplay,
    classification_report
)
import matplotlib
matplotlib.rcParams['figure.dpi'] = 150
matplotlib.rcParams['font.size'] = 12
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')

# Device
if torch.cuda.is_available():
    n_gpu = torch.cuda.device_count()
    for i in range(n_gpu):
        props = torch.cuda.get_device_properties(i)
        print(f'GPU {i}: {props.name} | {props.total_memory/1e9:.1f} GB')
        DEVICE = torch.device('cuda')
        try: torch.set_float32_matmul_precision('high')
        except: pass
    else:
        torch.set_num_threads(os.cpu_count()); torch.set_num_interop_threads(2)
        DEVICE = torch.device('cpu')
        print(f'CPU: {os.cpu_count()} cores')

print(f'Device: {DEVICE} | PyTorch: {torch.__version__}')

# Cell 2 – Configuration
# =====
# CONFIG
# =====
INPUT_PKL = '/kaggle/input/notebooks/wajeahaljunaid/preprocessing/preprocessed_data.pkl'
OUTDIR = '/kaggle/working/'
CKPT_BINARY = '/kaggle/working/model_binary.pt'
CKPT_MULTI = '/kaggle/working/model_multi.pt'
RESULTS_PKL = '/kaggle/working/transformer_results.pkl'

RANDOM_STATE = 42
EPOCHS_BINARY = 15
EPOCHS_MULTI = 15
BATCH_TRAIN = 512
BATCH_EVAL = 1024
LR_PEAK = 2e-3
WEIGHT_DECAY = 1e-4
PATIENCE = 7

```

```

D_MODEL = 80; N_HEADS = 10; N_LAYERS = 4
D_FF = 320; DROPOUT = 0.15
USE_FEATURE_GATE = True; POOLING = 'attn'
USE_MIXUP = True; MIXUP_ALPHA = 0.2
USE_FOCAL = True; FOCAL_GAMMA = 2.0
LABEL_SMOOTH = 0.05; USE_SWA = True
NUM_WORKERS = 2; USE_AMP = False
COLORS = {'Transformer': '#2196F3'}

random.seed(RANDOM_STATE);
torch.manual_seed(RANDOM_STATE)
torch.cuda.manual_seed_all(RANDOM_STATE)
np.random.seed(RANDOM_STATE);
print('Config ready ✓')

# Cell 3 – Load Preprocessed Data
# =====
# LOAD PREPROCESSED DATA
# =====
print(f'Loading {INPUT_PKL} ...')
with open(INPUT_PKL, 'rb') as f:
    data = pickle.load(f)

Xtr = data['Xtr']; Xva = data['Xva']; Xte = data['Xte']
ytr_m = data['ytr_m']; yva_m = data['yva_m']; yte_m = data['yte_m']
ytr_b = data['ytr_b']; yva_b = data['yva_b']; yte_b = data['yte_b']
feature_names = data['feature_names']; keep2 = data['keep2']
le = data['label_encoder']; class_names = data['class_names']

print(f'✓ Data loaded!')
print(f'Xtr: {Xtr.shape} | Xva: {Xva.shape} | Xte: {Xte.shape}')
print(f'Features: {len(feature_names)} | Classes: {len(class_names)}')

# Cell 4 – Model Definition
# =====
# MODEL DEFINITION
# =====
class FeatureGate(nn.Module):
    def __init__(self, n):
        super().__init__()
        self.fc1 = nn.Linear(n, max(16, n//2))
        self.fc2 = nn.Linear(max(16, n//2), n)
    def forward(self, x):
        return torch.sigmoid(self.fc2(torch.tanh(self.fc1(x))))

class Tokenizer(nn.Module):
    def __init__(self, n, d, use_gate=True, dropout=0.0):
        super().__init__()
        self.use_gate = use_gate
        if use_gate: self.gate = FeatureGate(n)
        self.value_proj = nn.Linear(1, d)
        self.col_embed = nn.Parameter(torch.randn(n, d)*0.02)
        self.ln = nn.LayerNorm(d); self.drop = nn.Dropout(dropout)
    def forward(self, x):
        if self.use_gate: x = self.gate(x)
        B, F = x.shape
        v = self.value_proj(x.reshape(B*F,1)).reshape(B,F,-1)
        return self.drop(self.ln(v + self.col_embed.unsqueeze(0)))

```

```

class EncoderBlock(nn.Module):
    def __init__(self, d, h, ff, drop):
        super().__init__()
        self.ln1 = nn.LayerNorm(d); self.attn = nn.MultiheadAttention(d, h,
dropout=drop, batch_first=True)
        self.ln2 = nn.LayerNorm(d); self.ff = nn.Sequential(nn.Linear(d,ff),
nn.GELU(), nn.Dropout(drop), nn.Linear(ff,d))
        self.drop = nn.Dropout(drop)
    def forward(self, x):
        h, _ = self.attn(self.ln1(x), self.ln1(x), self.ln1(x),
need_weights=False)
        x = x + self.drop(h); return x + self.drop(self.ff(self.ln2(x)))

class AttnPool(nn.Module):
    def __init__(self, d, h):
        super().__init__()
        self.query = nn.Parameter(torch.randn(1,1,d)*0.02)
        self.attn = nn.MultiheadAttention(d, h, batch_first=True)
    def forward(self, x):
        out, _ = self.attn(self.query.expand(x.size(0),-1,-1), x, x,
need_weights=False)
        return out.squeeze(1)

class TabTransformer(nn.Module):
    def __init__(self, n_features, n_classes):
        super().__init__()
        self.tokenizer = Tokenizer(n_features, D_MODEL, USE_FEATURE_GATE,
DROPOUT)
        self.blocks = nn.Sequential(*[EncoderBlock(D_MODEL, N_HEADS, D_FF,
DROPOUT) for _ in range(N_LAYERS)])
        self.pool = AttnPool(D_MODEL, N_HEADS) if POOLING=='attn' else None
        self.ln = nn.LayerNorm(D_MODEL)
        self.head = nn.Sequential(nn.Linear(D_MODEL,D_MODEL), nn.GELU(),
nn.Dropout(DROPOUT), nn.Linear(D_MODEL,n_classes))
    def forward(self, x):
        t = self.blocks(self.tokenizer(x))
        z = self.pool(t) if self.pool else t.mean(1)
        return self.head(self.ln(z))

def count_params(m): return sum(p.numel() for p in m.parameters() if
p.requires_grad)

tm = TabTransformer(Xtr.shape[1], 15).to(DEVICE)
print(f'Parameters: {count_params(tm):,}'); del tm; gc.collect()
if torch.cuda.is_available(): torch.cuda.empty_cache()
print('Model defined ✓')

```

Cell 5 – Training Helpers

```

# =====
# TRAINING HELPERS
# =====
def focal_loss(logits, targets, gamma=2.0):
    if logits.size(1)==1:
        bce = F.binary_cross_entropy_with_logits(logits.squeeze(1),
targets.float(), reduction='none')
        p = torch.sigmoid(logits.detach()).squeeze(1); pt = p*targets+(1-
p)*(1-targets)
        return ((1-pt)**gamma*bce).mean()
    ce = F.cross_entropy(logits, targets, reduction='none')

```

```

        p = F.softmax(logits.detach(), dim=1); pt = p.gather(1,targets.view(-
1,1)).squeeze(1)
        return ((1-pt)**gamma*ce).mean()

def make_loaders(Xtr,ytr,Xva,yva,Xte,yte):
    pin = torch.cuda.is_available()
    tl =
    DataLoader(TensorDataset(torch.from_numpy(Xtr),torch.from_numpy(ytr)),
    BATCH_TRAIN, shuffle=True, num_workers=NUM_WORKERS, pin_memory=pin)
    vl =
    DataLoader(TensorDataset(torch.from_numpy(Xva),torch.from_numpy(yva)),
    BATCH_EVAL, shuffle=False, num_workers=NUM_WORKERS, pin_memory=pin)
    pl =
    DataLoader(TensorDataset(torch.from_numpy(Xte),torch.from_numpy(yte)),
    BATCH_EVAL, shuffle=False, num_workers=NUM_WORKERS, pin_memory=pin)
    return tl,vl,pl

def evaluate(model, loader, n_classes, task):
    model.eval(); probs=[]; targets=[]
    with torch.no_grad():
        for xb,yb in loader:
            xb,yb = xb.to(DEVICE),yb.to(DEVICE)
            logits = model(xb)
            p = torch.sigmoid(logits.squeeze(1)) if task=='binary' else
F.softmax(logits,dim=1)
            probs.append(p.cpu()); targets.append(yb.cpu())
        probs=torch.cat(probs).numpy(); targets=torch.cat(targets).numpy()
        if task=='binary':
            yhat=(probs>=0.5).astype(int)
            return
    {'AUC_ROC':roc_auc_score(targets,probs),'AUC_PR':average_precision_score(target
s,probs),
        'Acc':(yhat==targets).mean(),'F1':f1_score(targets,yhat),
        'Prec':precision_score(targets,yhat,zero_division=0),'Rec':recall_score(targets
,yhat,zero_division=0)},probs,targets
        yhat=probs.argmax(1)
        try: auc=roc_auc_score(targets,probs,multi_class='ovr')
        except: auc=np.nan
        ap=[]
        for k in range(probs.shape[1]):
            try:
ap.append(average_precision_score((targets==k).astype(int),probs[:,k]))
            except: ap.append(np.nan)
        return
    {'AUC_ROC':auc,'AUC_PR':float(np.nanmean(ap)),'Acc':(yhat==targets).mean(),
        'F1_macro':f1_score(targets,yhat,average='macro'),
        'Prec_macro':precision_score(targets,yhat,average='macro',zero_division=0),
        'Rec_macro':recall_score(targets,yhat,average='macro',zero_division=0)},probs,t
argets

def train_model(Xtr,ytr,Xva,yva,Xte,yte,task,epochs):
    n_cls = 1 if task=='binary' else int(np.max(ytr))+1
    model = TabTransformer(Xtr.shape[1], 1 if task=='binary' else
n_cls).to(DEVICE)
    print(f'Params: {count_params(model):,}')
    opt = torch.optim.AdamW(model.parameters(), lr=LR_PEAK,
weight_decay=WEIGHT_DECAY)
    steps = max(1, math.ceil(len(Xtr)/BATCH_TRAIN))

```

```

        sched = torch.optim.lr_scheduler.OneCycleLR(opt, max_lr=LR_PEAK,
epochs=epochs, steps_per_epoch=steps)
        swa_model=swa_sched=None
        if USE_SWA:
            from torch.optim.swa_utils import AveragedModel, SWALR
            swa_model=AveragedModel(model);
        swa_sched=SWALR(opt,swa_lr=LR_PEAK*0.5)
        tl,vl,pl = make_loaders(Xtr,ytr,Xva,yva,Xte,yte)
        scaler = torch.amp.GradScaler('cuda', enabled=(torch.cuda.is_available()
and USE_AMP))
        history={'loss':[],'val_metric':[],'best_epoch':0}
        t0=time.time(); best=-1.0; best_state=None; pat=PATIENCE
        for ep in range(1,epochs+1):
            model.train(); el=0; nb=0
            for xb,yb in tl:
                xb,yb=xb.to(DEVICE),yb.to(DEVICE)
                opt.zero_grad(set_to_none=True)
                lam=np.random.beta(MIXUP_ALPHA,MIXUP_ALPHA) if USE_MIXUP else 1.0
                if USE_MIXUP:
                    idx=torch.randperm(xb.size(0),device=xb.device)
                    xb_f=lam*xb+(1-lam)*xb[idx]; yb1,yb2=yb,yb[idx]
                else: xb_f=xb; yb1,yb2=yb,None
                logits=model(xb_f)
                if USE_FOCAL:
                    l1=focal_loss(logits,yb1,FOCAL_GAMMA)
                    loss=lam*l1+(1-lam)*focal_loss(logits,yb2,FOCAL_GAMMA) if
yb2 is not None else l1
                elif task=='binary':

l1=F.binary_cross_entropy_with_logits(logits.squeeze(1),yb1.float())
                    loss=lam*l1+(1-
lam)*F.binary_cross_entropy_with_logits(logits.squeeze(1),yb2.float()) if yb2 is
not None else l1
                else:
                    l1=F.cross_entropy(logits,yb1,label_smoothing=LABEL_SMOOTH)
                    loss=lam*l1+(1-
lam)*F.cross_entropy(logits,yb2,label_smoothing=LABEL_SMOOTH) if yb2 is not None
else l1

                scaler.scale(loss).backward()
                torch.nn.utils.clip_grad_norm_(model.parameters(),1.0)
                scaler.step(opt); scaler.update(); sched.step()
                el+=loss.item(); nb+=1
            if USE_SWA and swa_model and ep>=max(1,epochs//2):
                swa_model.update_parameters(model); swa_sched.step()
            mv,_=evaluate(model,vl,n_cls,task)
            score=mv['AUC_PR'] if task=='binary' else mv['F1_macro']
            history['loss'].append(el/nb); history['val_metric'].append(score)
            tag='↑' if score>best else '↓'
            print(f'    Ep {ep:02d}/{epochs} | Val: {score:.6f} {tag} | Loss:
{el/nb:.4f}')
            if score>best: best=score; best_state={k:v.cpu() for k,v in
model.state_dict().items()}; pat=PATIENCE; history['best_epoch']=ep
            else:
                pat-=1
                if pat==0: print(f' Early stop ep {ep}'); break
            train_time=time.time()-t0
            if best_state: model.load_state_dict(best_state)
            chosen=model
            if USE_SWA and swa_model:
                try:
                    mb,_=evaluate(swa_model,vl,n_cls,task);
                    ms,_=evaluate(swa_model,vl,n_cls,task)
                    key='AUC_PR' if task=='binary' else 'F1_macro'

```

```

        if ms[key]>mb[key]: chosen=swa_model; print('SWA model selected')
    except: pass
    mv,pv,yv=evaluate(chosen,vl,n_cls,task);
    mt,pt,yt=evaluate(chosen,pl,n_cls,task)
    return chosen,mv,mt,pv,yv,pt,yt,count_params(chosen),train_time,history

def plot_curve(history, task, metric):
    fig,ax1=plt.subplots(figsize=(12,6))
    eps=range(1,len(history['loss'])+1)
    ax1.plot(eps,history['loss'],'r-o',markersize=4,linewidth=2,label='Train
Loss')
    ax1.set_xlabel('Epoch',fontsize=13);
    ax1.set_ylabel('Loss',color='red',fontsize=13);
    ax1.tick_params(axis='y',labelcolor='red')
    ax2=ax1.twinx()
    ax2.plot(eps,history['val_metric'],'b-
s',markersize=4,linewidth=2,label=f'Val {metric}')
    ax2.set_ylabel(f'Val {metric}',color='blue',fontsize=13);
    ax2.tick_params(axis='y',labelcolor='blue')
    ax1.axvline(x=history['best_epoch'],color='green',linestyle='--
',alpha=0.7,label=f"Best ep {history['best_epoch']}")
    l1,la1=ax1.get_legend_handles_labels();
    l2,la2=ax2.get_legend_handles_labels()
    ax1.legend(l1+l2,la1+la2,loc='center right',fontsize=11);
    ax1.grid(True,alpha=0.3)
    cap='5.1' if task=='binary' else '5.3'
    ax1.set_title(f'Figure {cap} - Training
({task.title()})',fontsize=14,fontweight='bold')
    plt.tight_layout();
    plt.savefig(OUTDIR+f'training_curve_{task}.png',dpi=300,bbox_inches='tight');
    plt.show()

```

print('Training helpers defined ✓')

Cell 6 – Train Binary Classifier

```

# =====
# TRAIN BINARY
# =====
print('='*60); print('BINARY CLASSIFICATION'); print('='*60)
model_bin,mval_bin,mtest_bin,pval_bin,yval_bin,ptest_bin,ytest_bin,params_bi
n,time_bin,hist_bin = train_model(
    Xtr,ytr_b,Xva,yva_b,Xte,yte_b,'binary',EPOCHS_BINARY)

```

```

# SAVE immediately
torch.save(model_bin.state_dict(), CKPT_BINARY)
print(f'\n✅ Binary model saved → {CKPT_BINARY}')
plot_curve(hist_bin,'binary','AUC-PR')
print('\n--- Binary Test ---')
for k,v in mtest_bin.items(): print(f' {k}: {v:.6f}')

```

Cell 7 – Binary Evaluation Plots

```

# Binary evaluation plots
from sklearn.metrics import precision_recall_curve, roc_curve,
confusion_matrix, ConfusionMatrixDisplay

prec_c,rec_c,_ = precision_recall_curve(ytest_bin,ptest_bin)
fig,ax=plt.subplots(figsize=(10,7))
ax.plot(rec_c,prec_c,color='#2196F3',linewidth=2)

```

```

ax.fill_between(rec_c,prec_c,alpha=0.1,color='#2196F3')
ax.set_title('Figure 5.2a - PR Curve (Binary)',fontsize=14,fontweight='bold')
ax.text(0.3,0.3,f'AUC-PR
{mtest_bin["AUC_PR"]:.5f}',fontsize=16,fontweight='bold',bbox=dict(boxstyle='ro
und',facecolor='wheat',alpha=0.8))
ax.grid(True,alpha=0.3); ax.set_xlim([0,1]); ax.set_ylim([0,1.02])
plt.tight_layout();
plt.savefig(OUTDIR+'pr_curve_binary.png',dpi=300,bbox_inches='tight');
plt.show()

fpr_c,tpr_c,_=roc_curve(ytest_bin,ptest_bin)
fig,ax=plt.subplots(figsize=(10,7))
ax.plot(fpr_c,tpr_c,color='#2196F3',linewidth=2); ax.plot([0,1],[0,1],'k--
',alpha=0.3)
ax.set_title('Figure 5.2b - ROC Curve
(Binary)',fontsize=14,fontweight='bold')
ax.text(0.4,0.3,f'AUC-ROC
{mtest_bin["AUC_ROC"]:.5f}',fontsize=16,fontweight='bold',bbox=dict(boxstyle='r
ound',facecolor='wheat',alpha=0.8))
ax.grid(True,alpha=0.3); plt.tight_layout();
plt.savefig(OUTDIR+'roc_curve_binary.png',dpi=300,bbox_inches='tight');
plt.show()

yhat_bin=(ptest_bin>=0.5).astype(int)
cm_bin=confusion_matrix(ytest_bin,yhat_bin,normalize='true')
fig,ax=plt.subplots(figsize=(8,7))
ConfusionMatrixDisplay(cm_bin,display_labels=['Normal','Attack']).plot(ax=ax
,cmap='Blues',values_format='.4f',colorbar=False)
ax.set_title('Figure 5.2c - Confusion Matrix
(Binary)',fontsize=14,fontweight='bold')
plt.tight_layout();
plt.savefig(OUTDIR+'confusion_binary.png',dpi=300,bbox_inches='tight');
plt.show()

```

Cell 8 - Train Multi-Class Classifier

```

# =====
# TRAIN MULTICLASS
# =====
print('='*60); print('MULTI-CLASS (15 classes)'); print('='*60)
model_multi,mval_multi,mtest_multi,pval_multi,yval_multi,ptest_multi,ytest_m
ulti,params_multi,time_multi,hist_multi = train_model(
    Xtr,ytr_m,Xva,yva_m,Xte,yte_m,'multi',EPOCHS_MULTI)

# SAVE immediately
torch.save(model_multi.state_dict(), CKPT_MULTI)
print(f'\n✅ Multi model saved → {CKPT_MULTI}')
plot_curve(hist_multi,'multi','Macro F1')
print('\n--- Multi-Class Test ---')
for k,v in mtest_multi.items(): print(f' {k}: {v:.6f}')

```

Cell 9 - Multi-Class Evaluation Plots

```

# Multi-class evaluation plots
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
classification_report
import matplotlib.patches as mpatches

yhat_multi=ptest_multi.argmax(1)
cm_multi=confusion_matrix(ytest_multi,yhat_multi,normalize='true')

```

```

fig,ax=plt.subplots(figsize=(16,14))
ConfusionMatrixDisplay(cm_multi,display_labels=class_names).plot(ax=ax,cmap=
'Blues',values_format='.2f',colorbar=False)
ax.set_xticklabels(class_names,rotation=45,ha='right',fontsize=9)
ax.set_yticklabels(class_names,fontsize=9)
ax.set_title('Figure 5.4 - Confusion Matrix (15-
Class)',fontsize=14,fontweight='bold')
plt.tight_layout();
plt.savefig(OUTDIR+'confusion_multi.png',dpi=300,bbox_inches='tight');
plt.show()

```

```

report=classification_report(ytest_multi,yhat_multi,target_names=class_names
,output_dict=True,zero_division=0)
import pandas as pd
pc_df=pd.DataFrame([{'Class':c,'Precision':report[c]['precision'],'Recall':r
eport[c]['recall'],'F1':report[c]['f1-
score'],'Support':int(report[c]['support'])} for c in class_names])
pc_df.to_csv(OUTDIR+'per_class_metrics_multi.csv',index=False);
display(pc_df)

```

```

fig,ax=plt.subplots(figsize=(14,8))
colors_f1=['#4CAF50' if f>0.9 else '#FF9800' if f>0.5 else '#F44336' for f in
pc_df['F1']]
ax.barh(pc_df['Class'],pc_df['F1'],color=colors_f1,edgecolor='white')
macro_f1=report['macro avg']['f1-score']
ax.axvline(x=macro_f1,color='#333',linestyle='--',linewidth=2)
for i,(f,s) in enumerate(zip(pc_df['F1'],pc_df['Support'])):
ax.text(max(f+0.01,0.02),i,f'{f:.4f} (n={s:,})',va='center',fontsize=9)
ax.set_title('Figure 5.5 - Per-Class F1',fontsize=14,fontweight='bold')
ax.grid(True,alpha=0.3,axis='x'); ax.set_xlim([0,1.15])
plt.tight_layout();
plt.savefig(OUTDIR+'per_class_f1.png',dpi=300,bbox_inches='tight'); plt.show()

```

Cell 10 - Save Results

```

# =====
# SAVE ALL RESULTS
# =====
with open(RESULTS_PKL,'wb') as f:
    pickle.dump({
        'params_bin':params_bin,'params_multi':params_multi,
        'time_bin':time_bin,'time_multi':time_multi,
        'mtest_bin':mtest_bin,'mtest_multi':mtest_multi,
        'ptest_bin':ptest_bin,'ptest_multi':ptest_multi,
        'ytest_bin':ytest_bin,'ytest_multi':ytest_multi,
        'hist_bin':hist_bin,'hist_multi':hist_multi,
        'report':report,
    },f)

```

```

size=os.path.getsize(RESULTS_PKL)/1e6
print(f'\n✅ Transformer results saved → {RESULTS_PKL} ({size:.1f} MB)')
print(f'  Binary AUC-PR : {mtest_bin["AUC_PR"]:.6f}')
print(f'  Multi Macro F1: {mtest_multi["F1_macro"]:.6f}')
print(f'  Binary model : {CKPT_BINARY}')
print(f'  Multi model : {CKPT_MULTI}')
print(f'\n🚀 Notebook 2 DONE - run 03-baselines.ipynb next')

```

APPENDIX C

Appendix C: Baseline Model Training Notebook (03-baselines.ipynb)

```
# Cell 1 – Imports and Data Loading
import os, time, pickle, warnings
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score
import matplotlib
matplotlib.rcParams['figure.dpi'] = 150
matplotlib.rcParams['font.size'] = 12
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')

INPUT_PKL          = '/kaggle/input/datasets/wajeahaljunaid/preprocessed-
data/preprocessed_data.pkl'
OUTDIR              = '/kaggle/working/'
BASELINES_PKL      = '/kaggle/working/baselines.pkl'
RF_N_JOBS           = -1
COLORS              =
{'Transformer': '#2196F3', 'RandomForest': '#4CAF50', 'XGBoost': '#FF9800', 'LightGBM': '#9C27B0'}

print('Loading preprocessed data...')
with open(INPUT_PKL, 'rb') as f: data=pickle.load(f)
Xtr=data['Xtr']; Xva=data['Xva']; Xte=data['Xte']
ytr_m=data['ytr_m']; yva_m=data['yva_m']; yte_m=data['yte_m']
class_names=data['class_names']; feature_names=data['feature_names']
print(f'✅ Data loaded | Xtr: {Xtr.shape} | Classes: {len(class_names)}')

# Cell 2 – Random Forest (600 trees)
# =====
# RANDOM FOREST
# =====
baseline_results={}; timing_info={}
rf_pte=xgb_pte=lgbm_pte=None

print('Training Random Forest (600 trees)...')
rf=RandomForestClassifier(n_estimators=600,n_jobs=RF_N_JOBS,random_state=42)
t0=time.time(); rf.fit(Xtr,ytr_m); rf_train=time.time()-t0
print(f'  Train time: {rf_train:.0f}s')

t0=time.time()
for _ in range(10): _=rf.predict_proba(Xte[:1000])
rf_infer=(time.time()-t0)/10/1000*1000

rf_pte=rf.predict_proba(Xte)
rf_f1=f1_score(yte_m,rf_pte.argmax(1),average='macro')
baseline_results['RandomForest']={'multi_f1':rf_f1,'train_time':rf_train,'infer_ms':rf_infer}
timing_info['RandomForest']={'train_time':rf_train,'infer_time_ms':rf_infer}
print(f'      RF:   Macro   F1={rf_f1:.4f}      |   Train={rf_train:.0f}s      |
Infer={rf_infer:.4f}ms')
```

```

# Cell 3 – XGBoost (800 trees)
# =====
# XGBOOST
# =====
try:
    import xgboost as xgb
    print('Training XGBoost (800 trees)...')

xgbm=xgb.XGBClassifier(n_estimators=800,max_depth=8,learning_rate=0.05,subsampl
e=0.8,

colsample_bytree=0.8,reg_lambda=1.0,tree_method='hist',n_jobs=RF_N_JOBS,random_
state=42,
    objective='multi:softprob',num_class=len(np.unique(ytr_m)))
    t0=time.time(); xgbm.fit(Xtr,ytr_m); xgb_train=time.time()-t0
    print(f' Train time: {xgb_train:.0f}s')
    t0=time.time()
    for _ in range(10): _=xgbm.predict_proba(Xte[:1000])
    xgb_infer=(time.time()-t0)/10/1000*1000
    xgb_pte=xgbm.predict_proba(Xte)
    xgb_f1=f1_score(yte_m,xgb_pte.argmax(1),average='macro')

baseline_results['XGBoost']={'multi_f1':xgb_f1,'train_time':xgb_train,'infer_ms
':xgb_infer}

timing_info['XGBoost']={'train_time':xgb_train,'infer_time_ms':xgb_infer}
    print(f'      XGB: Macro F1={xgb_f1:.4f} | Train={xgb_train:.0f}s |
Infer={xgb_infer:.4f}ms')
    except Exception as e: print(f'XGBoost skipped: {e}')


# Cell 4 – LightGBM (1,200 estimators)
# =====
# LIGHTGBM
# =====
try:
    from lightgbm import LGBMClassifier
    print('Training LightGBM (1200 trees)...')

lgbm=LGBMClassifier(n_estimators=1200,num_leaves=64,learning_rate=0.03,subsampl
e=0.8,

colsample_bytree=0.8,reg_lambda=1.0,random_state=42,n_jobs=RF_N_JOBS,

objective='multiclass',max_bin=255,min_data_in_bin=3,min_data_in_leaf=20,verbos
e=-1)
    t0=time.time(); lgbm.fit(Xtr,ytr_m); lgbm_train=time.time()-t0
    print(f' Train time: {lgbm_train:.0f}s')
    t0=time.time()
    for _ in range(10): _=lgbm.predict_proba(Xte[:1000])
    lgbm_infer=(time.time()-t0)/10/1000*1000
    lgbm_pte=lgbm.predict_proba(Xte)
    lgbm_f1=f1_score(yte_m,lgbm_pte.argmax(1),average='macro')

baseline_results['LightGBM']={'multi_f1':lgbm_f1,'train_time':lgbm_train,'infer
_ms':lgbm_infer}

timing_info['LightGBM']={'train_time':lgbm_train,'infer_time_ms':lgbm_infer}
    print(f'      LGBM: Macro F1={lgbm_f1:.4f} | Train={lgbm_train:.0f}s |
Infer={lgbm_infer:.4f}ms')
    except Exception as e: print(f'LightGBM skipped: {e}')

```

```

# Cell 5 – Summary Table

# =====
# SUMMARY TABLE
# =====

rows=[]
for name,res in baseline_results.items():
    rows.append({'Model':name,'Macro
F1':f"{res['multi_f1']:.4f}",'Train(s)':f"{res['train_time']:.0f}",'Infer(ms)':
f"{res['infer_ms']:.4f}"})
df_res=pd.DataFrame(rows); display(df_res)

fig,ax=plt.subplots(figsize=(10,5))
names=list(baseline_results.keys())
fls=[baseline_results[m]['multi_f1'] for m in names]
colors=[COLORS.get(m,'#999') for m in names]
bars=ax.bar(names,fls,color=colors,edgecolor='white',linewidth=1.5)
for bar,v in zip(bars,fls):
ax.text(bar.get_x()+bar.get_width()/2,v+0.005,f'{v:.4f}',ha='center',fontsize=1
2,fontweight='bold')
ax.set_ylabel('Macro F1',fontsize=13); ax.set_title('Baselines – Macro F1
Comparison',fontsize=14,fontweight='bold')
ax.grid(True,alpha=0.3,axis='y'); ax.set_ylim([0,max(fls)*1.15])
plt.tight_layout();
plt.savefig(OUTDIR+'baselines_f1.png',dpi=300,bbox_inches='tight'); plt.show()

# Cell 6 – Save Results

# =====
# SAVE
# =====

with open(BASELINES_PKL,'wb') as f:
    pickle.dump({
        'baseline_results':baseline_results,'timing_info':timing_info,
        'rf_pte':rf_pte,'xgb_pte':xgb_pte,'lgbm_pte':lgbm_pte,
    },f)

size=os.path.getsize(BASELINES_PKL)/1e6
print(f'\n✅ Baselines saved → {BASELINES_PKL} ({size:.1f} MB)')
for name,res in baseline_results.items():
    print(f'          {name}:          F1={res["multi_f1"]:.4f} |
Train={res["train_time"]:.0f}s')
print(f'\n🎉 Notebook 3 DONE – run 04-evaluation.ipynb next')

```

APPENDIX D

Appendix D: Evaluation Notebook (04-evaluation.ipynb)

```

# Cell 1 – Imports
import os, pickle, warnings
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.metrics import f1_score, classification_report,
confusion_matrix, ConfusionMatrixDisplay
import matplotlib
matplotlib.rcParams['figure.dpi'] = 150
matplotlib.rcParams['font.size'] = 12
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
warnings.filterwarnings('ignore')

if torch.cuda.is_available():
    DEVICE=torch.device('cuda')
    try: torch.set_float32_matmul_precision('high')
    except: pass
else: DEVICE=torch.device('cpu')
print(f'Device: {DEVICE}')

OUTDIR = '/kaggle/working/'
PREP_PKL = '/kaggle/input/datasets/wajeehaljunaaid/preprocessed-
data/preprocessed_data.pkl'
RESULTS_PKL = '/kaggle/input/datasets/wajeehaljunaaid/finall/transformer_results.pkl'
BASELINES_PKL = '/kaggle/input/datasets/wajeehaljunaaid/finall/baselines.pkl'
CKPT_MULTI = '/kaggle/input/datasets/wajeehaljunaaid/model-
multi/model_multi.pt'

COLORS={'Transformer': '#2196F3', 'RandomForest': '#4CAF50', 'XGBoost': '#FF9800',
'LightGBM': '#9C27B0', 'Ensemble': '#F44336'}

# Model config (must match Notebook 2)
D_MODEL=80; N_HEADS=10; N_LAYERS=4; D_FF=320; DROPOUT=0.15
USE_FEATURE_GATE=True; POOLING='attn'
print('Config ready ✓')

# Cell 2 – Load All Data
# =====
# LOAD ALL DATA
# =====
print('Loading preprocessed data...')
with open(PREP_PKL, 'rb') as f: prep=pickle.load(f)
Xtr=prep['Xtr']; Xva=prep['Xva']; Xte=prep['Xte']
ytr_m=prep['ytr_m']; yte_m=prep['yte_m']
feature_names=prep['feature_names']; keep2=prep['keep2'];
class_names=prep['class_names']

print('Loading transformer results...')
with open(RESULTS_PKL, 'rb') as f: res=pickle.load(f)

```

```

params_multi=res['params_multi']; time_multi=res['time_multi']
mtest_bin=res['mtest_bin']; mtest_multi=res['mtest_multi']
ptest_bin=res['ptest_bin']; ptest_multi=res['ptest_multi']
ytest_bin=res['ytest_bin']; ytest_multi=res['ytest_multi']
hist_bin=res['hist_bin']; hist_multi=res['hist_multi']
report=res['report']

print('Loading baselines...')
with open(BASELINES_PKL,'rb') as f: bl=pickle.load(f)
baseline_results=bl['baseline_results']; timing_info=bl['timing_info']
rf_pte=bl['rf_pte']; xgb_pte=bl['xgb_pte']; lgbm_pte=bl['lgbm_pte']
# infer_multi_mean placeholder – will be measured below

print('✅ All data loaded!')
print(f'   Features: {len(feature_names)} | Classes: {len(class_names)}')
print(f'   Transformer Multi F1: {mtest_multi["F1_macro"]:.4f}')

# Cell 3 – Rebuild and Load Transformer Model
# =====
# REBUILD & LOAD MODEL
# =====
class FeatureGate(nn.Module):
    def __init__(self, n):
        super().__init__()
        self.fc1=nn.Linear(n,max(16,n//2));
self.fc2=nn.Linear(max(16,n//2),n)
    def forward(self,x):
        return
x*torch.sigmoid(self.fc2(torch.tanh(self.fc1(x))))

class Tokenizer(nn.Module):
    def __init__(self,n,d,use_gate=True,dropout=0.0):
        super().__init__()
        self.use_gate=use_gate
        if use_gate: self.gate=FeatureGate(n)
        self.value_proj=nn.Linear(1,d)
        self.col_embed=nn.Parameter(torch.randn(n,d)*0.02)
        self.ln=nn.LayerNorm(d); self.drop=nn.Dropout(dropout)
    def forward(self,x):
        if self.use_gate: x=self.gate(x)
        B,F=x.shape; v=self.value_proj(x.reshape(B*F,1)).reshape(B,F,-1)
        return self.drop(self.ln(v+self.col_embed.unsqueeze(0)))

class EncoderBlock(nn.Module):
    def __init__(self,d,h,ff,drop):
        super().__init__()
        self.ln1=nn.LayerNorm(d);
self.attn=nn.MultiheadAttention(d,h,dropout=drop,batch_first=True)
        self.ln2=nn.LayerNorm(d);
self.ff=nn.Sequential(nn.Linear(d,ff),nn.GELU(),nn.Dropout(drop),nn.Linear(ff,d))
    def drop(self,drop):
        self.drop=nn.Dropout(drop)
    def forward(self,x):
        h,_=self.attn(self.ln1(x),self.ln1(x),self.ln1(x),need_weights=False)
        x=x+self.drop(h); return x+self.drop(self.ff(self.ln2(x)))

class AttnPool(nn.Module):
    def __init__(self,d,h):
        super().__init__()

```

```

        self.query=nn.Parameter(torch.randn(1,1,d)*0.02)
        self.attn=nn.MultiheadAttention(d,h,batch_first=True)
    def forward(self,x):
        out,_=self.attn(self.query.expand(x.size(0),-1,-
1),x,x,need_weights=False)
        return out.squeeze(1)

class TabTransformer(nn.Module):
    def __init__(self,n_features,n_classes):
        super().__init__()

self.tokenizer=Tokenizer(n_features,D_MODEL,USE_FEATURE_GATE,DROPOUT)

self.blocks=nn.Sequential(*[EncoderBlock(D_MODEL,N_HEADS,D_FF,DROPOUT) for _ in
range(N_LAYERS)])
        self.pool=AttnPool(D_MODEL,N_HEADS) if POOLING=='attn' else None
        self.ln=nn.LayerNorm(D_MODEL)

self.head=nn.Sequential(nn.Linear(D_MODEL,D_MODEL),nn.GELU(),nn.Dropout(DROPOUT
),nn.Linear(D_MODEL,n_classes))
    def forward(self,x):
        t=self.blocks(self.tokenizer(x))
        z=self.pool(t) if self.pool else t.mean(1)
        return self.head(self.ln(z))

n_classes=int(np.max(ytr_m))+1
model_multi=TabTransformer(Xtr.shape[1],n_classes).to(DEVICE)
model_multi.load_state_dict(torch.load(CKPT_MULTI,map_location=DEVICE))
model_multi.eval()
print(f'✅ model_multi loaded from {CKPT_MULTI}')
```

Cell 4 – Inference Latency Measurement

```

# =====
# MEASURE INFERENCE TIME
# =====
import time

def measure_infer(model, X_sample, n_runs=100):
    model.eval(); X_sample=np.ascontiguousarray(X_sample)
    with torch.no_grad():
        for _ in range(10):
            _=model(torch.from_numpy(X_sample[:100]).contiguous().to(DEVICE))
            if torch.cuda.is_available(): torch.cuda.synchronize()
            times=[]
            with torch.no_grad():
                for _ in range(n_runs):
                    t0=time.perf_counter()
                    xb=torch.from_numpy(X_sample).contiguous().to(DEVICE)
                    _=model(xb)
                    if torch.cuda.is_available(): torch.cuda.synchronize()
                    times.append((time.perf_counter()-t0)*1000/len(X_sample))
            return np.mean(times), np.std(times)

infer_multi_mean,_=measure_infer(model_multi,Xte[:1000].astype(np.float32))
print(f'Transformer inference: {infer_multi_mean:.4f} ms/flow')
```

Cell 5 – Ensemble Combination

```

# =====
```

```

# ENSEMBLE
# =====
base_probas={}
if rf_pte is not None: base_probas['RF'] =rf_pte
if xgb_pte is not None: base_probas['XGB'] =xgb_pte
if lgbm_pte is not None: base_probas['LGBM']=lgbm_pte

if base_probas:
    test_list=[ptest_multi]+[base_probas[k] for k in sorted(base_probas)]
    test_ens=np.stack(test_list,axis=0).mean(axis=0)
    ens_yhat=test_ens.argmax(1)
    ens_f1=f1_score(ytest_multi,ens_yhat,average='macro')
    print(f'Ensemble Macro F1: {ens_f1:.4f}')

ens_report=classification_report(ytest_multi,ens_yhat,target_names=class_names,
output_dict=True,zero_division=0)
    trans_fls=[report[c]['f1-score'] for c in class_names]
    ens_fls =[ens_report[c]['f1-score'] for c in class_names]

    fig,ax=plt.subplots(figsize=(14,8))
    x=np.arange(len(class_names)); w=0.35
    ax.barh(x-
w/2,trans_fls,w,label='Transformer',color=COLORS['Transformer'],edgecolor='white')
    ax.barh(x+w/2,ens_fls, w,label='Ensemble', color=COLORS['Ensemble'],
edgecolor='white')
    ax.set_yticks(x); ax.set_yticklabels(class_names,fontsize=10)
    ax.set_title('Figure 5.8 Transformer Ensemble vs
Transformer',fontsize=14,fontweight='bold')
    ax.legend(fontsize=12); ax.grid(True,alpha=0.3,axis='x')
    for i,(t,e) in enumerate(zip(trans_fls,ens_fls)):
        if e>t+0.01:
            ax.text(max(e,t)+0.02,i,f'+{e-
t:.3f}',va='center',fontsize=9,color='green',fontweight='bold')
    plt.tight_layout();
plt.savefig(OUTDIR+'ensemble_improvement.png',dpi=300,bbox_inches='tight');
plt.show()
else:
    print('No baselines found for ensemble')

# Cell 6 – Model Comparison Chart
# =====
# MODEL COMPARISON CHART
# =====
models_names=['Transformer']+list(baseline_results.keys())
multi_fls=[mtest_multi['F1_macro']]+[baseline_results[m]['multi_f1'] for m in
baseline_results]
bar_colors=[COLORS.get(m,'#999') for m in models_names]
fig,ax=plt.subplots(figsize=(10,6))
bars=ax.bar(models_names,multi_fls,color=bar_colors,edgecolor='white',linewi
dth=1.5)
    for bar,v in zip(bars,multi_fls):
ax.text(bar.get_x()+bar.get_width()/2,v+0.01,f'{v:.4f}',ha='center',fontsize=12
,fontweight='bold')
    ax.set_ylabel('Macro F1',fontsize=13); ax.set_title('Figure 5.6 – Model
Comparison',fontsize=14,fontweight='bold')
    ax.grid(True,alpha=0.3,axis='y'); ax.set_ylim([0,max(multi_fls)*1.15])
    plt.tight_layout();
plt.savefig(OUTDIR+'baseline_comparison.png',dpi=300,bbox_inches='tight');
plt.show()

```

```

# Inference speed
infer_names=['Transformer']+list(timing_info.keys())
infer_times=[infer_multi_mean]+[timing_info[m]['infer_time_ms'] for m in
timing_info]
fig,ax=plt.subplots(figsize=(10,5))
bars=ax.barh(infer_names,infer_times,color=[COLORS.get(m,'#999') for m in
infer_names],edgecolor='white')
ax.set_xscale('log')
for bar,v in zip(bars,infer_times):
ax.text(v*1.1,bar.get_y()+bar.get_height()/2,f'{v:.4f}
ms',va='center',fontsize=11)
ax.set_title('Figure 5.7 - Inference Latency',fontsize=14,fontweight='bold')
ax.grid(True,alpha=0.3,axis='x')
plt.tight_layout();
plt.savefig(OUTDIR+'inference_speed.png',dpi=300,bbox_inches='tight');
plt.show()

```

Cell 7 - Feature Importance: Embedding Norms

```

# =====
# FEATURE IMPORTANCE - Embedding Norms
# =====
try:
    m=model_multi.module if isinstance(model_multi,nn.DataParallel) else
model_multi
    emb=m.tokenizer.col_embed.detach().cpu().numpy()
    norms=np.linalg.norm(emb,axis=1)
    order=np.argsort(norms)[::-1]; topk=min(30,len(norms))
    fig,ax=plt.subplots(figsize=(14,7))

ax.bar(range(topk),norms[order[:topk]],color=COLORS['Transformer'],edgecolor='w
hite')

    ax.set_xticks(range(topk)); ax.set_xticklabels([keep2[i] for i in
order[:topk]],rotation=90,fontsize=9)
    ax.set_ylabel('Embedding L2 Norm',fontsize=13)
    ax.set_title('Figure 5.10 - Feature Importance (Embedding
Norm)',fontsize=14,fontweight='bold')
    ax.grid(True,alpha=0.3,axis='y')
    plt.tight_layout();
plt.savefig(OUTDIR+'feature_importance.png',dpi=300,bbox_inches='tight');
plt.show()
except Exception as e: print(f'Embedding plot failed: {e}')

```

Cell 8 - Permutation Importance

```

# =====
# PERMUTATION IMPORTANCE (fixed - fast version)
# =====
try:
    print('Running permutation importance...')
    N_REPEAT=3; N_SHOW=20
    rng=np.random.RandomState(42)
    sample_idx=rng.choice(len(Xte),size=min(3000,len(Xte)),replace=False)
    Xsub=np.ascontiguousarray(Xte[sample_idx]); ysub=yte_m[sample_idx]

    def pred_np(xb):
        model_multi.eval()
        with torch.no_grad():

```

```

p=F.softmax(model_multi(torch.from_numpy(np.ascontiguousarray(xb)).contiguous()
.to(DEVICE)),dim=1)
return p.cpu().numpy()

base_f1=f1_score(ysub,pred_np(Xsub).argmax(1),average='macro')
n_feat=min(Xsub.shape[1],40); drops=[]
for j in range(n_feat):
    sc=[]
    for _ in range(N_REPEAT):
        xp=np.ascontiguousarray(Xsub.copy()); rng.shuffle(xp[:,j])
        sc.append(f1_score(ysub,pred_np(xp).argmax(1),average='macro'))
    drops.append(base_f1-np.mean(sc))

order_p=np.argsort(drops)[::-1][:N_SHOW]
fig,ax=plt.subplots(figsize=(14,8))
ax.barh(range(N_SHOW),[drops[i] for i in order_p][::-1],color=COLORS['Transformer'],edgecolor='white')
ax.set_yticks(range(N_SHOW)); ax.set_yticklabels([keep2[i] for i in order_p][::-1],fontsize=9)
ax.set_xlabel('Importance (F1 drop)',fontsize=13)
ax.set_title('Figure 5.11 - Permutation Feature Importance',fontsize=14,fontweight='bold')
ax.grid(True,alpha=0.3,axis='x')
plt.tight_layout();
plt.savefig(OUTDIR+'permutation_importance.png',dpi=300,bbox_inches='tight');
plt.show()

print('✅ Permutation importance done!')
except Exception as e: print(f'Permutation importance failed: {e}')

```

Cell 9 – Computational Efficiency Table

```

# =====
# EFFICIENCY TABLE
# =====
eff_data=[{'Model':'Transformer','Params':f'{params_multi/1000:.1f}K','Train
(s)':f'{time_multi:.0f}',

'Infer(ms)':f'{infer_multi_mean:.4f}','Flows/s':f'{1000/infer_multi_mean:,.0f}'
}]

for name in ['RandomForest','XGBoost','LightGBM']:
    if name in timing_info:
        ti=timing_info[name]
        eff_data.append({'Model':name,'Params':'N/A
(tree)','Train(s)':f'{ti['train_time']:.0f}',

'Infer(ms)':f'{ti['infer_time_ms']:.4f}','Flows/s':f'{1000/ti['infer_time_ms']:
,.0f}'})
    eff_df=pd.DataFrame(eff_data); display(eff_df)

fig,axes=plt.subplots(2,2,figsize=(14,10))
names_eff=['Transformer']+['RandomForest','XGBoost','LightGBM']
if n in timing_info:
    eff_colors=[COLORS.get(m,'#999') for m in names_eff]

axes[0,0].bar(names_eff,[params_multi]+[0]*len(timing_info),color=eff_colors
,edgecolor='white')
axes[0,0].set_title('Parameters',fontsize=12,fontweight='bold');
axes[0,0].set_ylabel('Params')

```

```

train_vals=[time_multi]+[timing_info[m]['train_time'] for m in names_eff[1:]]
axes[0,1].bar(names_eff,train_vals,color=eff_colors,edgecolor='white')
axes[0,1].set_title('Training Time',fontsize=12,fontweight='bold');
axes[0,1].set_ylabel('Seconds')

infer_vals=[infer_multi_mean]+[timing_info[m]['infer_time_ms'] for m in
names_eff[1:]]
axes[1,0].bar(names_eff,infer_vals,color=eff_colors,edgecolor='white')
axes[1,0].set_yscale('log'); axes[1,0].set_title('Inference Latency
(log)',fontsize=12,fontweight='bold'); axes[1,0].set_ylabel('ms')

axes[1,1].bar(names_eff,[1000/v for v in
infer_vals],color=eff_colors,edgecolor='white')
axes[1,1].set_title('Throughput',fontsize=12,fontweight='bold');
axes[1,1].set_ylabel('Flows/s')

for ax in axes.flat: ax.grid(True,alpha=0.3,axis='y');
ax.tick_params(rotation=15)
fig.suptitle('Figure 5.12 – Computational
Efficiency',fontsize=14,fontweight='bold',y=1.02)
plt.tight_layout();
plt.savefig(OUTDIR+'computational_efficiency.png',dpi=300,bbox_inches='tight');
plt.show()

```

Cell 10 – Final Summary

```

# =====
# FINAL SUMMARY
# =====
print('='*60); print('ALL GENERATED FILES'); print('='*60)
for f in sorted(os.listdir(OUTDIR)):
    fp=os.path.join(OUTDIR,f)
    if os.path.isfile(fp): print(f' {f:50s} {os.path.getsize(fp)/1024:8.1f}
KB')
print(f'\nTotal: {len(os.listdir(OUTDIR))} files')
print(f'\n🚀 Notebook 4 DONE – run 05-ablation.ipynb next')

```

APPENDIX E

Appendix E: Ablation Study Notebook (05-ablation.ipynb)

Cell 1 – Imports

```
import os, math, time, random, gc, warnings, pickle
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.metrics import f1_score
import matplotlib
matplotlib.rcParams['figure.dpi'] = 150
matplotlib.rcParams['font.size'] = 12
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')

if torch.cuda.is_available():
    DEVICE=torch.device('cuda')
    try: torch.set_float32_matmul_precision('high')
    except: pass
else:
    torch.set_num_threads(os.cpu_count()); torch.set_num_interop_threads(2)
    DEVICE=torch.device('cpu')
print(f'Device: {DEVICE}')

INPUT_PKL          =      '/kaggle/input/datasets/wajeehaljunaid/preprocessed-
data/preprocessed_data.pkl'
OUTDIR              = '/kaggle/working/'
RANDOM_STATE=42
random.seed(RANDOM_STATE);                               np.random.seed(RANDOM_STATE);
torch.manual_seed(RANDOM_STATE)
torch.cuda.manual_seed_all(RANDOM_STATE)
print('Config ready ✓')
```

Cell 2 – Ablation Configuration

```
# =====
# ABLATION CONFIG (independent - can be modified)
# =====
ABLATION_FRACTION = 0.35    # fraction of data to use
ABLATION_EPOCHS   = 7       # epochs per trial

# Model config
D_MODEL=80; N_HEADS=10; N_LAYERS=4; D_FF=320; DROPOUT=0.15
BATCH_TRAIN=512; BATCH_EVAL=1024; LR_PEAK=2e-3; WEIGHT_DECAY=1e-4
PATIENCE=5; NUM_WORKERS=2; USE_AMP=False

# Default toggles (modified per trial)
USE_FEATURE_GATE=True; POOLING='attn'
USE_MIXUP=True; MIXUP_ALPHA=0.2
USE_FOCAL=True; FOCAL_GAMMA=2.0
LABEL_SMOOTH=0.05; USE_SWA=False # SWA=False for speed
USE_QUANTILE=True; USE_OUTLIER_CLIP=True
```

```

print(f'Ablation fraction: {ABLATION_FRACTION} | Epochs: {ABLATION_EPOCHS}')

# Cell 3 - Load Data
# =====
# LOAD DATA
# =====
print('Loading preprocessed data...')
with open(INPUT_PKL,'rb') as f: data=pickle.load(f)
Xtr=data['Xtr']; Xva=data['Xva']; Xte=data['Xte']
ytr_m=data['ytr_m']; yva_m=data['yva_m']; yte_m=data['yte_m']
feature_names=data['feature_names']; class_names=data['class_names']
print(f'✅ Loaded | Xtr: {Xtr.shape} | Classes: {len(class_names)}')

# Cell 4 - Model and Training Definitions
# =====
# MODEL + TRAINING (same code with global config)
# =====
class FeatureGate(nn.Module):
    def __init__(self,n):
        super().__init__()
        self.fc1=nn.Linear(n,max(16,n//2));
self.fc2=nn.Linear(max(16,n//2),n)
    def forward(self,x):
        return
x*torch.sigmoid(self.fc2(torch.tanh(self.fc1(x))))

class Tokenizer(nn.Module):
    def __init__(self,n,d,use_gate,dropout):
        super().__init__()
        self.use_gate=use_gate
        if use_gate: self.gate=FeatureGate(n)
        self.value_proj=nn.Linear(1,d)
        self.col_embed=nn.Parameter(torch.randn(n,d)*0.02)
        self.ln=nn.LayerNorm(d); self.drop=nn.Dropout(dropout)
    def forward(self,x):
        if self.use_gate: x=self.gate(x)
        B,F=x.shape; v=self.value_proj(x.reshape(B*F,1)).reshape(B,F,-1)
        return self.drop(self.ln(v+self.col_embed.unsqueeze(0)))

class EncoderBlock(nn.Module):
    def __init__(self,d,h,ff,drop):
        super().__init__()
        self.ln1=nn.LayerNorm(d);
self.attn=nn.MultiheadAttention(d,h,dropout=drop,batch_first=True)
        self.ln2=nn.LayerNorm(d);
self.ff=nn.Sequential(nn.Linear(d,ff),nn.GELU(),nn.Dropout(drop),nn.Linear(ff,d))
        self.drop=nn.Dropout(drop)
    def forward(self,x):
h,_=self.attn(self.ln1(x),self.ln1(x),self.ln1(x),need_weights=False)
        x=x+self.drop(h); return x+self.drop(self.ff(self.ln2(x)))

class AttnPool(nn.Module):
    def __init__(self,d,h):
        super().__init__()
        self.query=nn.Parameter(torch.randn(1,1,d)*0.02)
        self.attn=nn.MultiheadAttention(d,h,batch_first=True)
    def forward(self,x):

```

```

        out,_=self.attn(self.query.expand(x.size(0),-1,-
1),x,x,need_weights=False)
        return out.squeeze(1)

class TabTransformer(nn.Module):
    def __init__(self,n_features,n_classes,gate,pooling):
        super().__init__()
        self.tokenizer=Tokenizer(n_features,D_MODEL,gate,DROPOUT)

self.blocks=nn.Sequential(*[EncoderBlock(D_MODEL,N_HEADS,D_FF,DROPOUT) for _ in
range(N_LAYERS)])
        self.pool=AttnPool(D_MODEL,N_HEADS) if pooling=='attn' else None
        self.ln=nn.LayerNorm(D_MODEL)

self.head=nn.Sequential(nn.Linear(D_MODEL,D_MODEL),nn.GELU(),nn.Dropout(DROPOUT
),nn.Linear(D_MODEL,n_classes))
    def forward(self,x):
        t=self.blocks(self.tokenizer(x))
        z=self.pool(t) if self.pool else t.mean(1)
        return self.head(self.ln(z))

    def focal_loss(logits,targets,gamma=2.0):
        ce=F.cross_entropy(logits,targets,reduction='none')
        p=F.softmax(logits.detach(),dim=1); pt=p.gather(1,targets.view(-
1,1)).squeeze(1)
        return ((1-pt)**gamma*ce).mean()

def
run_ablation_trial(Xtr,ytr,Xva,yva,Xte,yte,epochs,gate,pooling,use_mixup,use_fo
cal,use_swa):
    n_cls=int(np.max(ytr))+1
    model=TabTransformer(Xtr.shape[1],n_cls,gate,pooling).to(DEVICE)

opt=torch.optim.AdamW(model.parameters(),lr=LR_PEAK,weight_decay=WEIGHT_DECAY)
    steps=max(1,math.ceil(len(Xtr)/BATCH_TRAIN))

sched=torch.optim.lr_scheduler.OneCycleLR(opt,max_lr=LR_PEAK,epochs=epochs,step
s_per_epoch=steps)
    pin=torch.cuda.is_available()

tl=DataLoader(TensorDataset(torch.from_numpy(Xtr),torch.from_numpy(ytr)),BATCH_
TRAIN,shuffle=True,num_workers=NUM_WORKERS,pin_memory=pin)

vl=DataLoader(TensorDataset(torch.from_numpy(Xva),torch.from_numpy(yva)),BATCH_
EVAL,shuffle=False,num_workers=NUM_WORKERS,pin_memory=pin)

pl=DataLoader(TensorDataset(torch.from_numpy(Xte),torch.from_numpy(yte)),BATCH_
EVAL,shuffle=False,num_workers=NUM_WORKERS,pin_memory=pin)

best=-1.0; best_state=None; pat=PATIENCE
for ep in range(1,epochs+1):
    model.train()
    for xb,yb in tl:
        xb,yb=xb.to(DEVICE),yb.to(DEVICE)
        opt.zero_grad(set_to_none=True)
        if use_mixup:
            lam=np.random.beta(MIXUP_ALPHA,MIXUP_ALPHA)
            idx=torch.randperm(xb.size(0),device=xb.device)
            xb_f=lam*xb+(1-lam)*xb[idx]; yb1,yb2=yb,yb[idx]
        else: xb_f=xb; yb1,yb2=yb,None; lam=1.0
        logits=model(xb_f)
        if use_focal:

```

```

        l1=focal_loss(logits,yb1,FOCAL_GAMMA)
        loss=lam*l1+(1-lam)*focal_loss(logits,yb2,FOCAL_GAMMA)    if
yb2 is not None else l1
    else:
        l1=F.cross_entropy(logits,yb1,label_smoothing=LABEL_SMOOTH)
        loss=lam*l1+(1-
lam)*F.cross_entropy(logits,yb2,label_smoothing=LABEL_SMOOTH) if yb2 is not None
    else l1
        loss.backward();
torch.nn.utils.clip_grad_norm_(model.parameters(),1.0); opt.step(); sched.step()
# eval
model.eval(); probs=[]; tgts=[]
with torch.no_grad():
    for xb,yb in vl:
        p=F.softmax(model(xb.to(DEVICE)),dim=1)
        probs.append(p.cpu()); tgts.append(yb)
    probs=torch.cat(probs).numpy(); tgts=torch.cat(tgts).numpy()
    score=f1_score(tgts,probs.argmax(1),average='macro')
    if score>best: best=score; best_state={k:v.cpu() for k,v in
model.state_dict().items()}; pat=PATIENCE
    else:
        pat-=1
        if pat==0: break
# test
model.load_state_dict(best_state); model.eval(); probs=[]; tgts=[]
with torch.no_grad():
    for xb,yb in pl:
        p=F.softmax(model(xb.to(DEVICE)),dim=1)
        probs.append(p.cpu()); tgts.append(yb)
    probs=torch.cat(probs).numpy(); tgts=torch.cat(tgts).numpy()
    test_f1=f1_score(tgts,probs.argmax(1),average='macro')

test_auc_pr=float(np.nanmean([_import_('sklearn.metrics',fromlist=['average_p
recision_score'])].average_precision_score((tgts==k).astype(int),probs[:,k]) for
k in range(n_cls))))
del model; gc.collect()
if torch.cuda.is_available(): torch.cuda.empty_cache()
return test_f1, test_auc_pr

```

```
print('Model & training functions defined ✓')
```

Cell 5 – Stratified Subset Helper

```

# =====
# STRATIFIED SUBSET
# =====
def stratified_subset(X,y,frac,seed=42):
    X=np.asarray(X); y=np.asarray(y)
    if frac>=1.0: return X,y
    rng=np.random.RandomState(seed)
    classes,counts=np.unique(y,return_counts=True)
    min_frac=min(1.0,max(frac,float(np.max(1.0/counts)),frac+0.05))
    for attempt in range(3):

sss=StratifiedShuffleSplit(n_splits=1,train_size=min_frac,random_state=seed+att
empt)
        idx_keep,_=next(sss.split(X,y))
        if len(np.unique(y[idx_keep]))==len(classes):                return
X[idx_keep],y[idx_keep]
        min_frac=min(1.0,min_frac+0.05)
        keep=set(idx_keep.tolist())
        for c in classes:

```

```

        if c not in set(np.unique(y[list(keep)])):
keep.add(int(rng.choice(np.where(y==c) [0],1) [0]))
        return X[np.array(sorted(keep))],y[np.array(sorted(keep))]
```

```

Xtr_s,ytr_s=stratified_subset(Xtr,ytr_m,ABLATION_FRACTION,RANDOM_STATE)
Xva_s,yva_s=stratified_subset(Xva,yva_m,ABLATION_FRACTION,RANDOM_STATE)
Xte_s,yte_s=stratified_subset(Xte,yte_m,ABLATION_FRACTION,RANDOM_STATE)
print(f'Subset: Xtr={Xtr_s.shape} | Xva={Xva_s.shape} | Xte={Xte_s.shape}')
print(f'All classes in subsets: {len(np.unique(ytr_s))==len(class_names)}')
```

Cell 6 – Run Ablation Grid

```

# =====
# RUN ABLATION GRID
# =====
grid = [
    # (name,          gate, pooling, mixup, focal)
    ('Full',          True, 'attn',  True,  True ),
    ('NoGate',         False, 'attn',  True,  True ),
    ('MeanPool',       True, 'mean',   True,  True ),
    ('NoMixup',        True, 'attn',   False, True ),
    ('CE_Loss',        True, 'attn',   True,  False),
]

abl_rows=[]
for name, gate, pooling, mixup, focal in grid:
    print(f'\n [{name}]')
    try:
        f1, auc_pr = run_ablation_trial(
            Xtr_s,ytr_s,Xva_s,yva_s,Xte_s,yte_s,
            ABLATION_EPOCHS, gate, pooling, mixup, focal, use_swa=False
        )
    abl_rows.append({'Name':name,'Test_F1_macro':f1,'Test_AUC_PR':auc_pr})
    print(f'      F1={f1:.4f} | AUC-PR={auc_pr:.4f}')
    except Exception as e:
        print(f'      FAILED: {e}')

abl_rows.append({'Name':name,'Test_F1_macro':float('nan'),'Test_AUC_PR':float('nan')})

abl_df=pd.DataFrame(abl_rows)
print('\n'+ '='*50)
display(abl_df)
```

Cell 7 – Plot Ablation Results

```

# =====
# PLOT ABLATION DELTA
# =====
abl_df.to_csv(OUTDIR+'ablation_multi.csv',index=False)

base_val=abl_df[abl_df['Name']=='Full']['Test_F1_macro'].values[0]
abl_df['Delta']=abl_df['Test_F1_macro']-base_val
dd=abl_df[abl_df['Name']!='Full'].sort_values('Delta')

fig,ax=plt.subplots(figsize=(12,7))
colors_d=['#4CAF50' if d>0 else '#F44336' for d in dd['Delta']]
ax.barh(dd['Name'],dd['Delta'],color=colors_d,edgecolor='white')
for i,(n,d) in enumerate(zip(dd['Name'],dd['Delta'])):
```

```

        ax.text(d+(0.001 if d>0 else -0.001),i,f'{d:+.4f}',va='center',
                ha='left' if d>0 else 'right',fontsize=11,fontweight='bold')
    ax.axvline(x=0,color='black',linewidth=0.8)
    ax.set_xlabel('Δ Macro F1 vs Full Pipeline',fontsize=13)
    ax.set_title('Figure 5.9 – Ablation Study',fontsize=14,fontweight='bold')
    ax.grid(True,alpha=0.3,axis='x')
    plt.tight_layout();
plt.savefig(OUTDIR+'ablation_delta.png',dpi=300,bbox_inches='tight'); plt.show()

print(f'\n✅ Ablation results saved → {OUTDIR}ablation_multi.csv')
print(f'\n📊 Summary:')
print(f'    Full model F1: {base_val:.4f}')
for _,row in dd.iterrows():
    arrow='▲' if row['Delta']>0 else '▼'
    print(f'    {row["Name"]:20s}: {row["Delta"]:+.4f} {arrow}')
print(f'\n🎉 Notebook 5 DONE – all notebooks complete.')

```