FINAL PROJECT REPORT:

CS – 501: CLOUD SYSTEMS

BY

FARZANUDDIN SYED

MASTER'S

IN

COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY

VANCOUVER,

WA, USA.

Git Hub Repository: https://github.com/SyedFarzanuddin/ML-Based-Threat-Detection-in-Cloud-Systems

Machine Learning for Threat Detection in Cloud Systems:

1. Problem Motivation:

The growth of cloud computing has revolutionized modern business operations by providing **unmatched scalability, flexibility, and innovation**. However, with the increasing reliance on cloud platforms comes a rise in cyber-attacks targeting these environments. Common threats such as **Distributed Denial of Service (DDoS)** attacks, **worms**, and **reconnaissance activities** pose significant risks. The dynamic and complex nature of cloud ecosystems creates unique challenges for traditional security methods, which often fail to detect evolving and sophisticated attack patterns.

A Real-World Scenario

On June 19, 2024, a significant global incident highlighted the vulnerabilities of cloud systems when Windows systems crashed worldwide due to a misconfigured file being deployed within a cloud environment. This error triggered cascading failures, disrupting dependent systems and services on a large scale.

This incident illustrates the critical need for proactive threat detection and mitigation strategies to ensure the resilience of cloud systems and prevent such large-scale disruptions.

Proposed Approach

To address these challenges, we developed a system inspired by real-world scenarios. Using the **UNSW-NB15 networking dataset**, which contains a variety of network traffic, including malicious activities, we implemented an **ML-based threat detection solution**.

Our approach replicates a cloud environment by processing this data on platforms like **Amazon Web Services (AWS)**. This ensures the system is not only scalable but also adaptable to real-world settings.

Key Objectives:

- Detect threats in **real-time**.
- Predict potential attacks **proactively**.

- Strengthen cloud security against **emerging cyber threats**.

2. Design Goals or Performance Questions:

1. How Accurate Is It?

- Does the system achieve high precision and recall in detecting threats?
- Can it identify hidden or less obvious attacks effectively?

2. How Fast Can It Respond?

- Is the system capable of real-time alerting to mitigate attacks immediately?
- What is the average detection time for a threat?

3. Can It Handle Growth? If implemented on a large scale?

- Does the system scale effectively with increased cloud traffic or data volume?
- Can it operate seamlessly across multiple cloud platforms?

4. How Reliable Is It?

- What are the chances of false alarms or missed threats, and how do they impact performance?
- Does the system remain effective under varying network conditions?

5. Can It Learn and Improve?

- How well can it adapt to detect new attack types?
- Does it support regular updates to keep pace with emerging threats?

6. Is It Easy to Use?

- Can the system integrate with existing security tools?
- Is it intuitive enough for security teams to operate and understand?

3. Design Architecture:

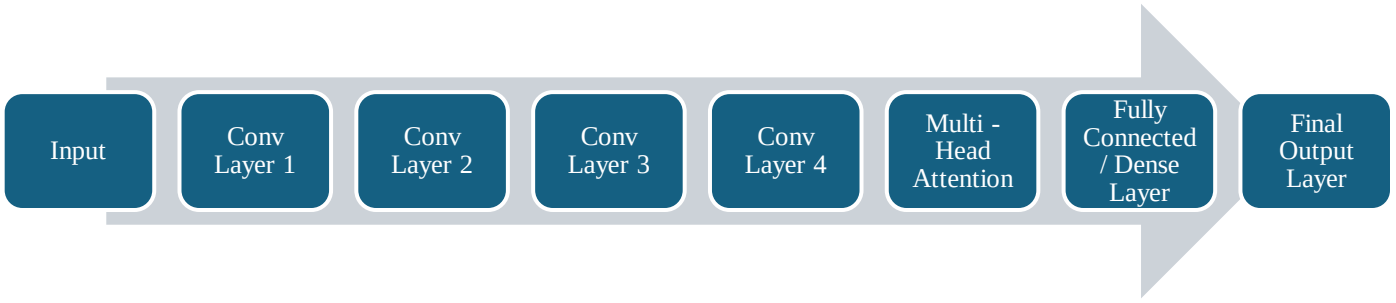

Fig: 1 – CNN with Multi-Head Attention Transformer

This architecture integrates **Convolutional Neural Networks (CNNs)** and a **Multi-Head Attention mechanism** to leverage their combined strengths for tasks like **sequence classification** and **multi-label prediction**. It effectively processes complex sequential data by using CNNs for feature extraction and attention mechanisms to capture long-range dependencies.

1. MultiHeadAttention Class

The **MultiHeadAttention** class is designed to help the model focus on different parts of the input sequence simultaneously. This allows it to learn relationships between distant positions within the data.

Key Features

- **Input Parameters**:
    - input_dim: Specifies the size of input features.
    - num_heads: Number of attention heads (default = 8).
    - dropout: Regularization rate to prevent overfitting.
- **Linear Layers**:
    - A shared linear layer projects the input into Queries (Q), Keys (K), and Values (V).
    - Another layer processes the attended output into the final result.

- **Attention Mechanism**:
    - Normalizes the input using **Layer Normalization**.
    - Split Q, K, and V across multiple heads for parallel attention.
    - Applies **scaled dot-product attention**, followed by **softmax**, **dropout**, and weighted multiplication with values.
- **Residual Connections**:
    - Adds the attention output back to the input, helping retain original information and improving gradient flow.

2. CNNWithAttention Class

This class blends CNNs with multi-head attention to enhance representation learning, particularly for multi-label tasks. The CNN layers extract rich, hierarchical features, while the attention mechanism highlights the most important aspects of the data.

Key Features

- **Convolutional Layers**:
    - Four layers extract features using **5×5 kernels**, **stride of 2**, and **padding of 2**.
    - Each layer uses **ReLU activation**, **batch normalization**, and **max-pooling**.
    - Filter sizes grow progressively (32, 64, 128, 256) to capture increasingly complex patterns.
- **Multi-Head Attention**:
    - Refines the features extracted by the CNN, focusing on relevant parts of the sequence.
- **Fully Connected Layers**:
    - After flattening, the features pass through:
        - A layer that reduces dimensionality to 128.

        - An output layer with **sigmoid activation** to predict probabilities for each label.

- **Dropout**:
    - Applied after convolutional and fully connected layers (rate: 0.4–0.6) to prevent overfitting.

How It Works

1. The input sequence first passes through the CNN layers to extract hierarchical features.
2. The attention mechanism processes these features, highlighting key relationships across the sequence.
3. The refined features are flattened and sent through fully connected layers to produce predictions.
4. The final output uses a sigmoid activation to assign probabilities to each label.

4. Detailed Description

To develop and evaluate our threat detection system, we utilized the **UNSW-NB15 dataset**, a comprehensive collection of network traffic data designed to simulate real-world scenarios. This dataset is particularly suited for testing intrusion detection systems, as it includes a diverse mix of both normal and malicious network activities.

Dataset Overview

The **UNSW-NB15 dataset** provides a wealth of information about network behavior, with **49 features** representing various aspects of network activity, including protocol usage, connection states, and payload size. It also includes a **binary label feature** indicating whether a record represents normal traffic or a malicious attack.

S.No.	Type of attributes	Name of attributes	Sequence No.
1	Flow	Script, Sport, Dstip. Dsport, Proto	1-5
2	Basic	State, Dur, Sbytes, Dbytes, Sttl, Dttl, Sloss, Dloss, Service, Sload, Dload, Spkts, Dpkts	6-18
3	Content	Swin, Dwin, Stepb, Dtcpb, Smeansz, Dmeansz, trans_depth, res_bdy_len	19-26
4	Time	Sjit, Djit, Stime , Ltime, Sintpkt, Dintpkt, Tcprtt, Synack, Ackdat	27-35
5	General Purpose	is_sm_ips_ports, ct_state_ttl, ct_flw_http_mthd,is_ftp_login, ct_ftp_cmd	36-40
6	Connection	ct_srv_src , ct_srv_dst , ct_dst_ltm, ct_src_ ltm ,ct_src_dport_ltm,ct_dst_sport_ltm,ct_dst_src_ltm	41-47
7	Labelled	attack_cat, Label	48-49

Fig: 2 Features / Attributes in UNSW-NB15

Category	Training Set	Testing set
Normal	56,000	37,000
Analysis	2,000	677
Backdoor	1,746	583
DOS	12,264	4,089
Exploits	33,393	11,132
Fuzzing	18,184	6,062
Generic	40,000	18,871
Reconnaissance	10,491	3,496
Shellcode	1,133	378
Worms	130	44
Total Records	175,341	82,332

Fig: 3 Target Features / Attributes in UNSW-NB15

Preprocessing

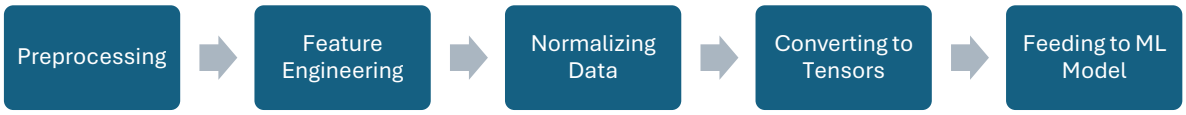

Fig: 4 Data Preprocessing

Steps

To ensure optimal performance of the machine learning models, the raw data from the **UNSW-NB15 dataset** underwent a series of preprocessing steps. These steps addressed issues such as class imbalances and inconsistencies in formatting, preparing the dataset for efficient training and accurate predictions.

1. Label Encoding and One-Hot Encoding

- **Categorical Variables**: Features like *protocol type*, *service*, and *flag* were **label encoded** to convert textual categories into numerical values.
- **Attack Categories**: To handle multi-class labels (e.g., different types of attacks), **one-hot encoding** was applied. This ensured that the model could process and differentiate between various attack types effectively.

2. Feature Engineering with SMOTE

- SMOTE was used to balance the dataset by generating synthetic samples for the minority class, improving detection of rare attacks.

3. Data Normalization

- The data was standardized using a Standard Scaler, ensuring each feature had a mean of 0 and a standard deviation of 1 to prevent any feature from dominating the learning process.

4. Converting Data to Tensors:

- The dataset was converted to tensors for compatibility with deep learning models, enabling efficient computation and GPU utilization during training.

Working with Amazon Web Services

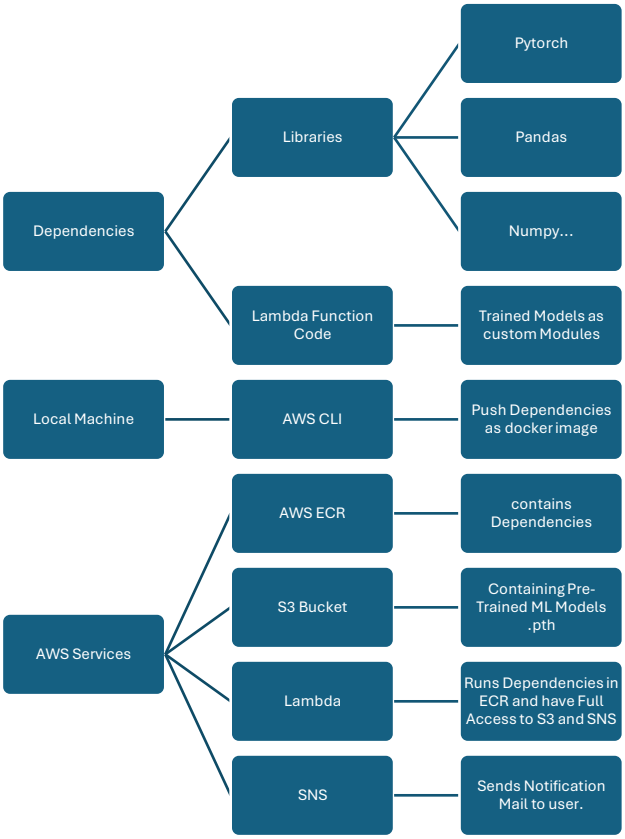

Fig: 5 AWS Setup

To implement our solution effectively, we leveraged several AWS services and tools to create a scalable, efficient, and automated environment. Here's an overview of how we used AWS:

Boto3 (Python SDK)

We used **Boto3**, AWS's SDK for Python, to interact programmatically with various AWS services. This allowed us to automate tasks such as managing resources, triggering actions, and handling workflows directly through Python scripts.

AWS Elastic Container Registry (ECR)

For containerization, we relied on **AWS ECR** to store, manage, and deploy Docker images. This service ensured seamless integration with other AWS services, providing a secure and scalable way to manage our application containers.

AWS Lambda

We used **AWS Lambda** to run our program in a serverless environment, eliminating the need for infrastructure management. Lambda automatically scaled to handle requests, making it efficient and cost-effective for running machine learning models. We also assigned **IAM roles** to Lambda, granting it access to other AWS services like **S3** for data storage and **SNS** for notifications. This setup allowed us to automate tasks and scale our system dynamically, paying only for to compute power used during execution.

Triggers: AWS S3 and SNS

- **AWS S3**: We used S3 buckets to store the Pre-Trained Models, input data and to trigger Lambda functions whenever new data was uploaded. This streamlined the workflow, enabling real-time processing.
- **AWS SNS (Simple Notification Service)**: SNS was employed to send notifications when critical events occurred, such as the detection of a potential threat. This ensured quick communication and response for security teams.

By combining these AWS services, we built a robust cloud-based solution that effectively handles networking data, scales with demand, and integrates automation for real-time threat detection.

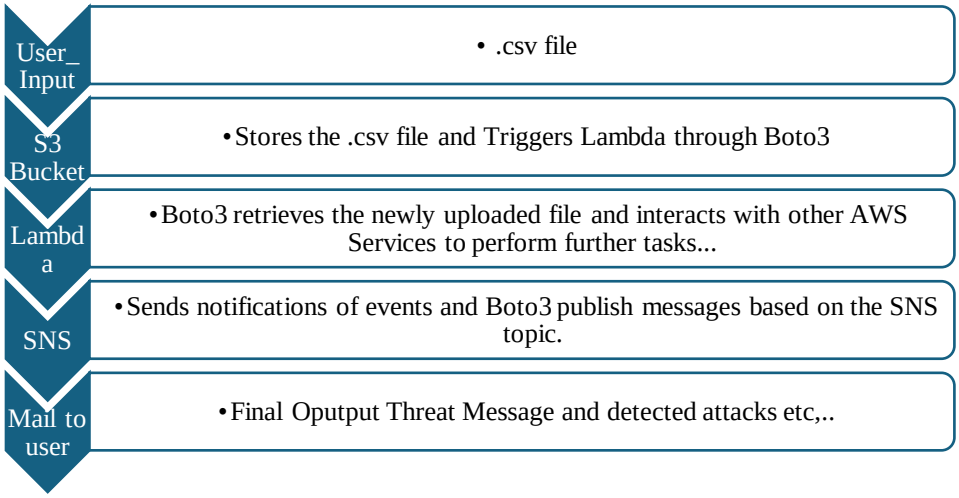

Fig: 6 AWS Working Setup

5. Difficulties:

1. **Training on Unbalanced Dataset**

   One of the primary challenges was training models on an unbalanced dataset. Despite numerous studies and approaches, balancing the dataset with oversampling was crucial. However, achieving a robust model without overfitting or underfitting proved to be difficult. I faced significant issues with overfitting during training, which resulted in less accurate predictions during testing. After experimenting with various techniques, I eventually managed to mitigate the overfitting problem by training custom-built neural network models, which provided better control over the performance which we will see in our further section.

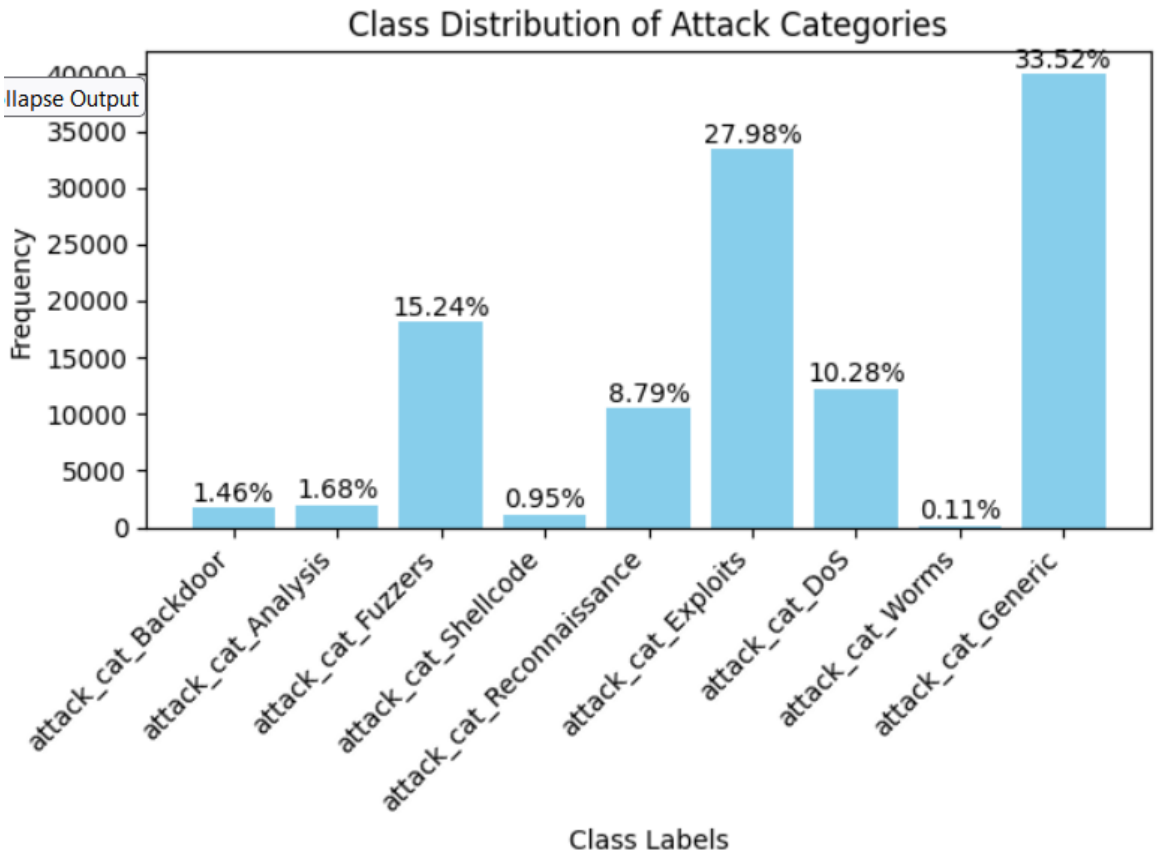

Fig: 7 Original Class Distribution

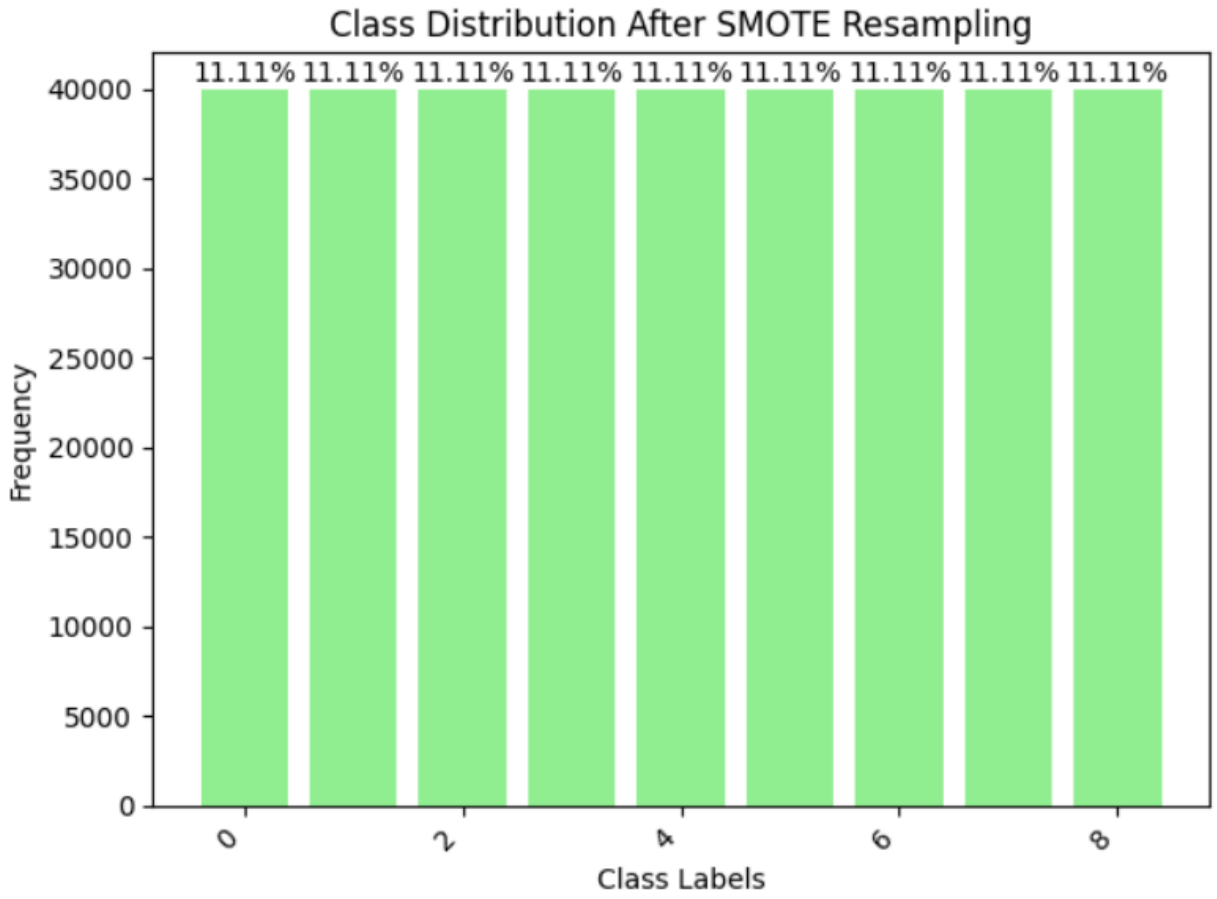

Fig: 8 Over Sampled Class Distribution

2. **Dependency Management on AWS Free Tier**
   Another challenge was managing the required dependencies within the constraints of the AWS free tier. Navigating the limitations of the free tier while ensuring the environment met the necessary specifications was both challenging and rewarding. This task pushed me to find creative solutions to optimize resource usage without exceeding the free tier limits.

6. Experimental Setup

Initially, we experimented by two types of models one with predicting label (0 or 1) and the other one predicting attack_cat with the UNSW-NB15 dataset using standard machine learning models such as:

- Random Forest Classifier (RF)
- Support Vector Classifier (SVC)
- Gradient Boosting Classifier (GBC)

These models were trained on the original, unbalanced dataset to predict binary labels (0 or 1) to establish a baseline for performance.

Model	Accuracy	Precision	Recall	F-1 Score
GBC (Training)	0.9467	0.9492	0.9274	0.9372
GBC (Testing)	0.8241	0.8750	0.8049	0.8101
RF (Training)	0.9982	0.9981	0.9977	0.9979
RF (Testing)	0.8684	0.8958	0.8551	0.8618
SVC (Training)	0.9374	0.9546	0.9040	0.9240
SVC (Testing)	0.8088	0.8663	0.7879	0.7917

Fig: 9 Model Performance on unbalanced dataset

Next, we applied SMOTE (Synthetic Minority Over-sampling Technique) to address the class imbalance in the dataset. After oversampling, we retrained the models and compared the results to evaluate the improvements. This step helped us understand the impact of balancing the dataset and gave us insights into which models with which approach performed better.

Model	Accuracy	Precision	Recall	F-1 Score
GBC (Training)	0.9473	0.9473	0.9473	0.9473
GBC (Testing)	0.8267	0.8739	0.8082	0.8136
RF (Training)	0.9980	0.9980	0.9980	0.9980
RF (Testing)	0.8910	0.9043	0.8820	0.8874
SVC (Training)	0.9400	0.9408	0.9400	0.9400
SVC (Testing)	0.7894	0.8264	0.7707	0.7738

Fig: 10 Model Performance after Over Sampling with SMOTE

With the clarity gained from these initial experiments, we proceeded to train Neural Networks with a more refined approach, ensuring better handling of the imbalanced data and aiming for improved model accuracy.

We carried out all these Experimentations in order to replicate a real world scenario on our AWS Free Tier Service.

7. Performance Evaluation:

As discussed in the previous section, we observed that models trained on oversampled data tended to fit more stably during training and produced consistent results during testing. However, the issue of overfitting still arose, even with the use of oversampling techniques.

To address this challenge, we trained and evaluated the oversampled dataset using our custom-built **Binary and Multi-Class Model: Convolutional Neural Network (CNN) with an Attention Mechanism Transformer Model**. This advanced model helped mitigate and reduce overfitting by refining the feature extraction process and capturing long-range dependencies within the data. As a result, it ensured more generalized performance during testing and ultimately improved the robustness of the model.

Model	Accuracy	Precision	Recall	F-1 Score
Binary Model (Training)	0.9365	0.9374	0.9365	0.9365
Binary Model (Testing)	0.8954	0.9050	0.8874	0.8925
Multi-Class Model (Training)	0.9485	0.9469	0.9485	0.9438
Multi-Class Model (Testing)	0.9481	0.9452	0.9481	0.9452

Fig: 11 Proposed Model Performance

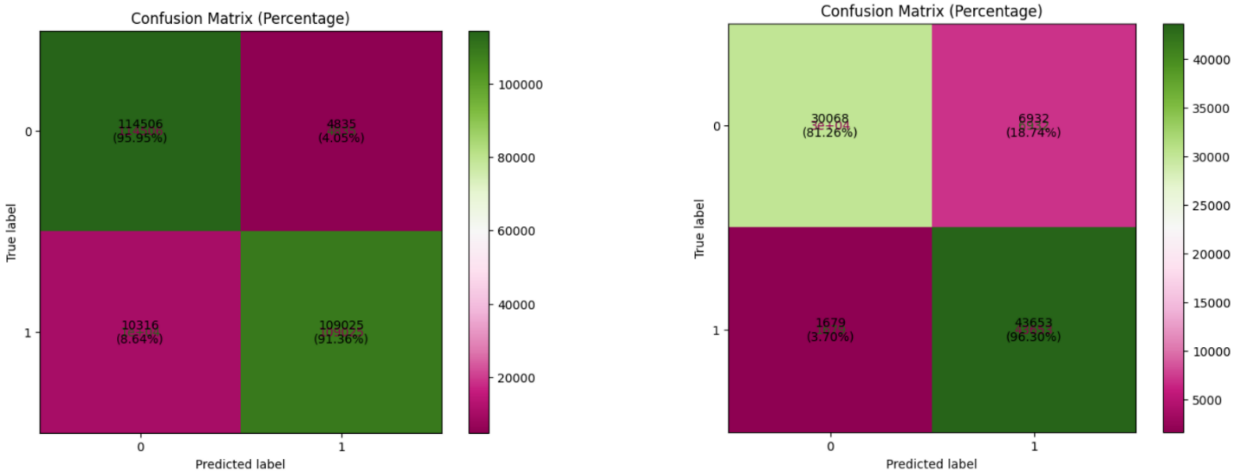

Fig: 12 Binary Model Performance on Training and Testing Confusion Matrix

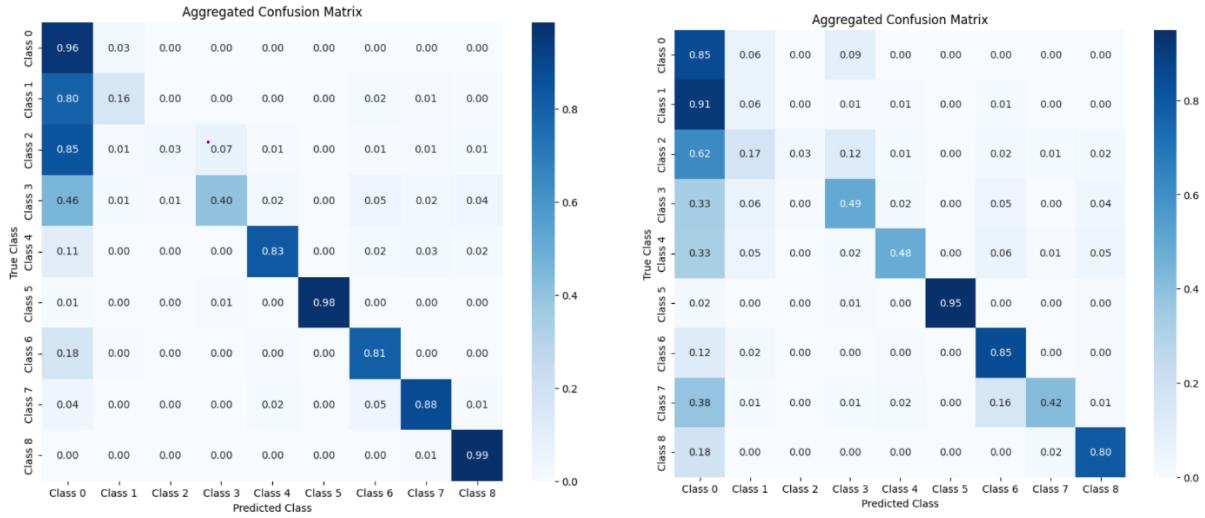

Fig: 13 Multi-Class Model Performance on Training and Testing Confusion Matrix

8. Conclusion:

In conclusion, our proposed models showed strong performance across metrics like accuracy, precision, recall, and F1 score. However, a closer look through the confusion matrix revealed that while the models generalized well overall, there were still areas for improvement. Overfitting remained a challenge, particularly when working with the oversampled dataset, but we managed to reduce prediction errors to about 0.4%–0.5% in the binary classification model. This marked a significant improvement, though there's still room to further refine the models and enhance their generalization capabilities.

Beyond model performance, the experiment successfully simulated a cloud-based environment using AWS. This setup demonstrated how anomalies could be effectively detected in real-world scenarios, showcasing the potential of integrating machine learning with cloud platforms for security applications. These results are promising, but they also highlight the need for ongoing work to optimize model performance and adaptability to evolving threats. This project has laid a solid foundation for future improvements and practical implementations in cloud security.

9. Future Work:

There is scope for further improvement in the machine learning models. Future efforts will focus on exploring additional models, refining existing ones, and enhancing generalization across various parameters to achieve even better performance and robustness.

10. Related Works:

1. Article Performance Evaluation of Deep Learning Based Network Intrusion Detection System across Multiple Balanced and Imbalanced Datasets Azizjon Meliboev 1 , Jumabek Alikhanov 2 and Wooseong Kim 1,

2. Addressing Imbalanced Data in Network Intrusion Detection: A Review and Survey Elham Abdullah Al-Qarni1, Ghadah Ahmad Al-Asmari2 Department of Computing and Information Technology, University of Bisha, Bisha, Saudi Arabia1 Agency for Planning and Digital Transformation, Ministry of Hajj and Umrah, Macca, Saudi Arabia2

3. Network Based Intrusion Detection Using the UNSW-NB15 Dataset Souhail Meftah1, Tajjeeddine Rachidi1 and Nasser Assem1 1 School of Science and Engineering, Al Akhawayn University in Ifrane, Ifrane 53000, Morocco Received 4 Nov. 2018, Revised 5 Jul. 2019, Accepted 15 Jul. 2019, Published 1 Sep. 2019

4. An Ensemble Intrusion Detection Technique Based on Proposed Statistical Flow Features for Protecting Network Traffic of Internet of Things Nour Moustafa , Member, IEEE, Benjamin Turnbull, Member, IEEE, and Kim-Kwang Raymond Choo, Senoir Member, IEEE.

5. Article Performance Evaluation of Deep Learning Based Network Intrusion Detection System across Multiple Balanced and Imbalanced Datasets Azizjon Meliboev 1 , Jumabek Alikhanov 2 and Wooseong Kim 1,

6. Unsw-Nb15 Dataset and Machine Learning Based Intrusion Detection Systems Avinash R. Sonule, Mukesh Kalla, Amit Jain, D. S. Chouhan

Git Hub Repository: https://github.com/SyedFarzanuddin/ML-Based-Threat-Detection-in-Cloud-Systems