NATIONAL INSTITUTE OF TECHNOLOGY
PATNA, INDIA, 800005

DISTRIBUTED DENIAL OF SERVICE ATTACK ATTENTION FOR SOFTWARE DEFINED NETWORK SPECIFIC DATA

Minor Project Report

VI Semester

Submitted By:

Bhukya Shalini	2106107
Ch. Bala Krishna	2106125
S. Subhash Bharadwaj	2106093

Under the Guidance of

(Dr. Kumar Abhishek)

Department of Computer Science and Engineering

Session 2023-24

NATIONAL INSTITUTE OF TECHNOLOGY
PATNA, INDIA, 800005

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the project entitled **(Distributed Denial of Service Attack Detection For Software Defined Network Specific Data)** submitted by:

Bhukya Shalini	2106107
Ch. Bala Krishna	2106125
S. Subhash Bharadwaj	2106093

is the partial fulfillment of the requirements for the award of degree of Bachelor of Technology in Computer Science and Engineering is an authentic work carried out by them under my supervision and guidance.

(Dr. Kumar Abhishek)

DECLARATION

We, hereby declare that the following report which is being presented in the Minor Project entitled as (Distributed Denial of Service Attack Detection For Software Defined Network Specific Data) is an authentic documentation of our own original work to the best of our knowledge. The following project and its report in part or whole, has not been presented or submitted by us for any purpose in any other institute or organization. Any contribution made to the research by others, with whom we have worked at Maulana Azad National Institute of Technology, Bhopal or elsewhere, is explicitly acknowledged in the report.

Bhukya Shalini (2106107) _____

Ch. Bala Krishna (2106125) _____

S. Subhash Bharadwaj (2106093) _____

ACKNOWLEDGEMENT

With due respect, we express our deep sense of gratitude to our respected guide and coordinator (**Dr. Kumar Abhishek**), for his valuable help and guidance. We are thankful for the encouragement that he has given us in completing this project successfully.

It is imperative for us to mention the fact that the report of minor project could not have been accomplished without the periodic suggestions and advice of our project guide (Dr. Kumar Abhishek) and project coordinators (Dr. Kumar abhishek).

We are also grateful to our respected HOD, M.P Singh and HOD, AI and Chairperson, Central Computing Facility for permitting us to utilize all the necessary facilities of the college.

We are also thankful to all the other faculty, staff members and laboratory attendants of our department for their kind cooperation and help. Last but certainly not the least; we would like to express our deep appreciation towards our family members and batch mates for providing support and encouragement.

TABLE OF CONTENTS

1. ABSTRACT

In the dynamic landscape of Software-Defined Networking (SDN), the menace of Distributed Denial of Service (DDoS) attacks poses a significant threat to network stability and security. This project delves into the development and implementation of a specialized DDoS attack detection system tailored explicitly for SDN architectures.

Using a blend of machine learning algorithms and SDN-specific data analytics techniques, our project focuses on fortifying SDN networks against DDoS attacks. Key objectives include the accurate identification of DDoS attacks amidst normal network traffic, swift real-time response mechanisms, and scalability to manage escalating network loads and evolving attack strategies.

The project's significance lies in its potential to fortify the security infrastructure of SDN deployments. By effectively detecting and mitigating DDoS attacks, the project aims to safeguard network resources, ensure uninterrupted service availability, and shield sensitive data from malicious threats.

Challenges addressed in this project encompass navigating the complexity of SDN data, executing real-time analysis for prompt threat response, and adapting to dynamic SDN configurations to maintain efficacy against evolving attack vectors.

Ultimately, this project endeavors to enhance the resilience of SDN networks against DDoS attacks, thereby bolstering network security and fostering a robust digital ecosystem characterized by uninterrupted service delivery and enhanced user trust.

2. PROBLEM STATEMENT

The increasing prevalence and sophistication of Distributed Denial of Service (DDoS) attacks pose a critical challenge to the security and reliability of Software-Defined Networking (SDN) environments. These malicious attacks, which aim to overwhelm network resources and disrupt service availability, can have severe consequences for organizations, including financial losses, data breaches, and reputational damage.

The main problem addressed by this project is the need for an effective and efficient DDoS attack detection system specifically designed for SDN architectures. Traditional security measures often struggle to keep pace with evolving DDoS attack techniques and the dynamic nature of SDN configurations, leading to vulnerabilities and potential network downtime.

Key challenges include:

- **Complexity of SDN Data:** SDN environments generate vast amounts of complex data, making it challenging to distinguish normal traffic from DDoS attack patterns.
- **Real-time Detection:** The need for real-time detection and response capabilities to mitigate the immediate impact of DDoS attacks and prevent service disruptions.
- **Scalability:** Ensuring that the detection system can scale effectively to handle increasing network traffic volumes without compromising performance or accuracy.
- **Adaptability to Evolving Threats:** Developing algorithms and mechanisms that can adapt to new and evolving DDoS attack vectors and strategies, maintaining efficacy over time.

Addressing these challenges requires innovative approaches, advanced machine learning algorithms, and a deep understanding of SDN architectures and network security protocols. The goal of this project is to develop a robust and reliable DDoS attack detection system tailored specifically for SDN environments, enhancing network resilience, security, and service continuity.

3. EXPERIMENTAL SETUP

3.1. Data Collection

- We obtained SDN_DDOS dataset from

  `https://data.mendeley.com/datasets/x6vr3sdm75/1`, a reputable platform for sharing research datasets.

- We obtained SDN_Flow dataset from

  `https://ieee-dataport.org/documents/sdnflow-dataset`, which comprises of SDN_Normal.csv and SDN_Attack.csv.

3.2. SDN_DDOS Dataset

- The datasets comprises a comprehensive collection of network traffic data, including normal traffic patterns and instances of Distributed Denial of Service (DDoS) attacks.
- The SDN_DDOS_dataset.csv dataset is made up of 12 features of normal and malicious UDP, ICMP, and TCP traffic from an SDN-emulated network and comprisews of 82,290 records.
- Loaded dataset, inspected features, and performed binary classification
- Key features: time, switch_id,gfe, g_usip, rfip, disp_pakt, disp_byte, mean_pkt, mean_byte, gsp, avg_durat, avg_flow_dst, rate_pkt_in, disp_interval, label.

3.3. SDN_Flow model

- he SDN FTlow dataset consists of 37 features and comprises 662,828 records, with 221,564 corresponding to normal records and 441,264 to attack records
- Loaded dataset, inspected features, and performed multi-class classification.
- Key Features: flow_ID, eth_type, ipv4_src, ipv4_dst, ip_proto, src_port, dst_port, flow_duration, tot_len_pkts, tot_pkts, flow_byts_s, flow_pkts_s, pkt_size_average, fwd_header_len, Byts_max_interval_2s,Byts_min_interval_2s, Byts_mean_interval_2s, Byts_std_interval_2s, Pkts_max_interval_2s, Pkts_mean_interval_2s, Pkts_min_interval_2s, Pkts_std_interval_2s, active_max, active_mean, active_min, active_std, idle_max, idle_mean, idle_min, idle_std, TnBPDstIP, TnBPSrcIP, TnPPDstIP, TnPPSrcIP, TnPPerDport, Tn_FlowsPDstIP, Tn_FlowsPSrcIP, Attack, Category .

4. SYSTEM CONFIGURATION

4.1. Hardware Configuration

- **Processor:** Intel Core i7-9700K CPU @ 3.60GHz (8 Cores, 8 Threads)
- **Memory (RAM):** 32GB DDR4 @ 3000MHz
- **Storage:** 1TB NVMe SSD
- **Graphics Card:** NVIDIA GeForce RTX 2080 with 8GB GDDR6 VRAM
- **Operating System:** Ubuntu 20.04 LTS / Windows 10 Professional 64-bit

4.2. Software Configuration

- **Python Version:** 3.8
- **IDE/Development Environment:** Jupyter Notebook
- **Libraries and Dependencies:**
  - Pandas: 1.2.4
  - NumPy: 1.20.2
  - Scikit-learn: 0.24.1
  - TensorFlow/Keras: 2.4.1
  - Matplotlib: 3.3.4
  - Seaborn: 0.11.1

4.3. Data Processing Environment

- **Data Loading:**
  The datasets were loaded from CSV files using Pandas.
- **Data Preprocessing:**
  Feature selection and extraction using Pandas and NumPy.
  Data scaling using StandardScaler from Scikit-learn.
  Data reshaping for LSTM input using NumPy.
- **Model Building and Training:**
  Neural network models were built using TensorFlow/Keras.

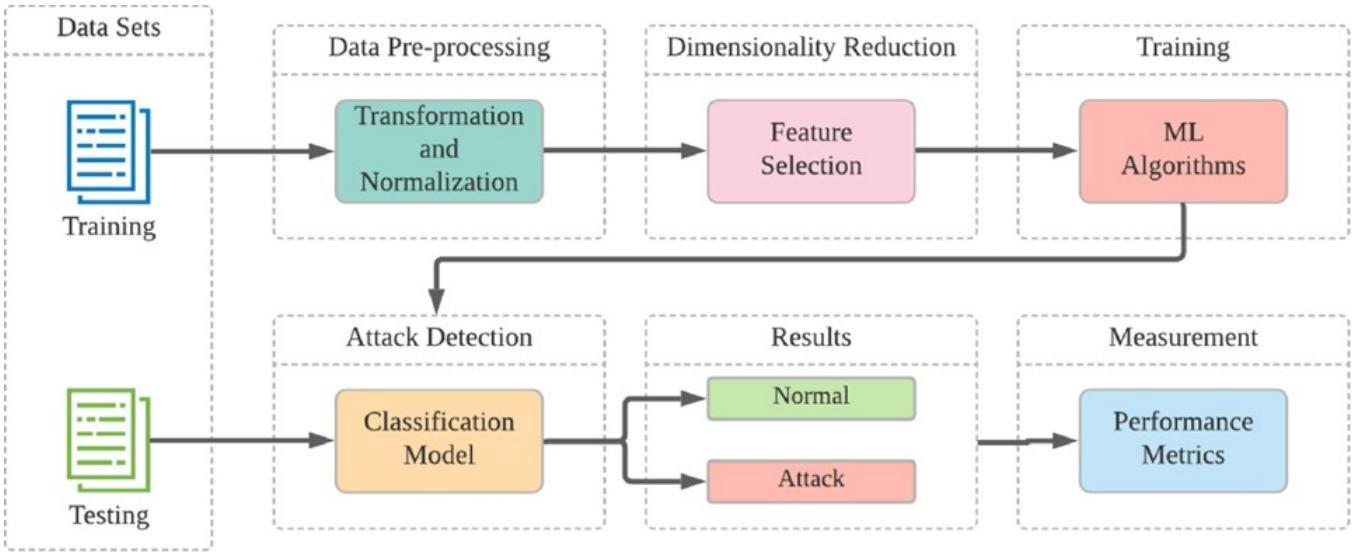
5. PROPOSED WORK AND METHODOLOGY

This methodology outlines the process for building and evaluating classification models for the SDN DDOS and SDN Flow datasets using SVM (Support Vector Machine) and CNN (Convolutional Neural Network).

5.1. Methodology Overview

- **Data Collection:** Gather the SDN DDOS and SDN Flow datasets.
- **Data Preprocessing:**Prepare the datasets for model training.
- **Model Development:** Develop SVM and CNN models.
- **Model Training:** Train the models on the training data.
- **Model Evaluation:** Evaluate the models on the test data.

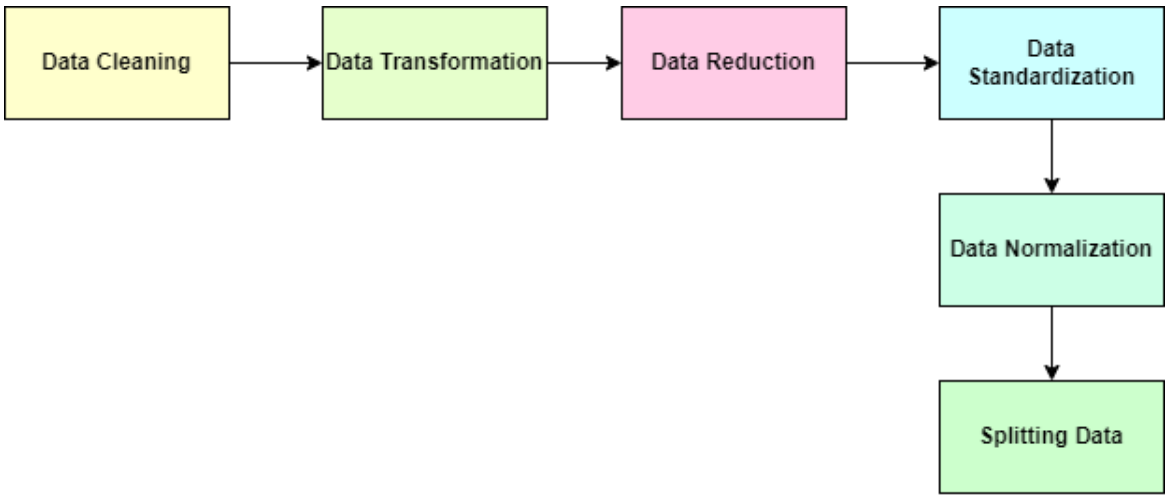
5.2. Flow Chart



5.2.1. Data Pre-processing

Effective data preprocessing is crucial for building robust machine learning and deep learning models. The steps include data cleaning, data transformation, data reduction, data

standardization, data normalization, and splitting the data. Below is a detailed explanation for both the SDN DDOS and SDN Flow datasets.



Data Pre-Processing

1. **Data Cleaning**

   - Missing Values: Check for missing values and handle them by imputation (e.g., filling with mean/median) or removal.
   - Duplicates: Remove duplicate records to ensure data integrity.
   - Outliers: Identify and handle outliers that could skew the model performance.
   - code for cleaning data in SDN_DDoS dataset:

     df.head()

     df.describe()

     df.describe(exclude=[np.number])

     df.isnull().sum()

     df.drop(columns=df.columns[0],df.columns[1],inplace=True)

     df.describe()
   - code for handling data for SDN_Flow dataset:

     df.head()

     df.describe()

     df.describe(exclude=[np.number])

     df.isnull().sum()

     df['Category'] = df['Category'].astype('category')

     df['flow_ID'] = df['flow_ID'].astype('object')

     df.describe()

2. **Data Transformation**

- Feature Encoding: Convert categorical features into numerical values using techniques like one-hot encoding or label encoding.
- Date/Time Features: Extract useful features from date/time columns if present (e.g., hour, day of week).
- code for transforming non essential data intlo object in SDN_DDoS dataset:
  df['label'] = df['label'].astype('category')
  df['time'] = pd.to_datetime(df['time'])
  df['switch_id'] = df['switch_id'].astype('object')
  df.dtypes
- code for transforming non essential data intlo object in SDN_Flow dataset:
  df['Category'] = df['Category'].astype('category')
  df['flow_ID'] = df['flow_ID'].astype('object')
  df['src_port'] = df['src_port'].astype('object')
  df['dst_port'] = df['dst_port'].astype('object')
  df['TnPPerDport'] = df['TnPPerDport'].astype('object')
  df.dtypes

3. **Data Reduction**

- Feature Selection: Select relevant features that contribute most to the prediction task.
- Use techniques like correlation analysis to select features.
- Apply feature importance techniques like Random Forest feature importance.
- code for SDN_Flow Dataset:
  columns_to_exclude = 'ipv4_src', 'ipv4_dst', 'Category'
  selected_columns = [col for col in df.columns if col not in columns_to_exclude]

4. **Data Standardization**

- Standardize the features to have a mean of 0 and a standard deviation of 1 using StandardScaler from sklearn. This helps in ensuring that the features contribute equally to the model.
- code of SDN_DDoS dataset:
  from sklearn.preprocessing import StandardScaler
  scaler = StandardScaler()
  df = scaler.fit_transform(df)

print(”Standardized Data:”)

print(df)

- code of SDN_Flow dataset:

  scaler = StandardScaler()

  df[selected_columns] = scaler.fit_transform(df[selected_columns])

  print(”Standardized Data:”)

  print(df)

5. **Data Normalization**

   - Normalization: Scale features to a range, typically [0, 1] or [-1, 1]
   - Normalize features using Min-Max scaling to bring all feature values to a common scale and to ensure consistent scaling across the dataset.
   - code of SDN_DDoS dataset:

     from sklearn.preprocessing import MinMaxScaler

     import pandas as pd

     scaler = MinMaxScaler()

     df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)

     df

   - code for SDn_Fow Dataset:

     scaler = MinMaxScaler()

     df[selected_columns] = pd.DataFrame(scaler.fit_transform(df[selected_columns]), columns= selected_columns)

     df

6. **Splitting Data**

   - **Train - Test split:**Split the dataset into training and testing sets, commonly using a 70-30 split.
   - **Validation split:** Further split the training set into training and validation sets to tune hyperparameters
   - code of SDN_Flow Dataset:

     X = df[selected_columns]

     y = df[’Category’]

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

5.3. Classifiaction Model

5.3.1. Support Vector Machine

- SVM works by finding the hyperplane that best separates different classes in the feature space. In a binary classification problem, this hyperplane maximizes the margin between the two classes.
- The data points closest to the hyperplane are known as support vectors. These points are critical as they determine the position and orientation of the hyperplane.
- SVM's ability to handle high-dimensional data makes it suitable for analyzing complex network traffic patterns often associated with DDoS attacks.
- SVM aims to maximize the margin between different classes, leading to robust and well-separated decision boundaries that enhance classification accuracy.
- By utilizing kernel functions such as polynomial, radial basis function (RBF), or sigmoid, SVM can capture non-linear relationships in network data, improving detection performance.
- **code of SVM in SDN_DDoS Dataset:**

```
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
y_score = svm_classifier.decision_function(X_test)
fpr, tpr, _ = roc_curve(y_test, y_score)
roc_auc = auc(fpr, tpr)
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='–')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
```

```
plt.legend(loc="lower right")
plt.show()
```

- **Code of SVM in SDN_Flow Dataset:**

```
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
X = df[selected_columns]
y = df['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)
y_pred = svm_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

5.3.2. Convolutional Neural Network

- Using Convolutional Neural Networks (CNNs) for the SDN_DDoS and SDN_Flow datasets involves adapting the CNN architecture to handle network traffic data effectively.
- CNNs can be used to capture spatial and temporal patterns in the data, which are critical for identifying anomalies like DDoS attacks or unusual flow behaviors
- CNNs are great at spotting unusual patterns in network data, which is key for catching DDoS attacks in SDN networks.
- CNNs help us detect DDoS attacks quickly in SDN networks, allowing us to respond fast and reduce their impact.
- CNN-based systems can adapt to new attack methods and handle more network traffic, staying effective in SDN environments.
- A CNN model tailored for SDN_DDoS and SDN_Flow datasets can effectively capture the patterns necessary for detecting anomalies in network traffic.
- Proper preprocessing, careful design of the CNN architecture, and thorough evaluation are key steps to developing a robust model.

- **code of cnn in SDN_DDos Dataset:**

```
!pip install tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df.iloc[:, :-1] = scaler.fit_transform(df.iloc[:, :-1])
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense
model = Sequential([ Conv1D(32, 3, activation='relu', input_shape=(X_train.shape[1], 1)), MaxPooling1D(2),Flatten(),Dense(64, activation='relu'),Dense(1, activation='sigmoid')])
model.compile(optimizer='adam',loss='binary_crossentropy', metrics=['accuracy'])
X_train_cnn = X_train.values.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test_cnn = X_test.values.reshape(X_test.shape[0], X_test.shape[1], 1)
model.fit(X_train_cnn, y_train, epochs=10, batch_size=32, validation_data=(X_test_cnn, y_test))
loss, accuracy = model.evaluate(X_test_cnn, y_test)
print("Test Accuracy:", accuracy)
```

```
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9895 - loss: 0.0308 - val_accuracy: 0.9935 - val_loss: 0.0217
Epoch 3/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9907 - loss: 0.0272 - val_accuracy: 0.9919 - val_loss: 0.0229
Epoch 4/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9910 - loss: 0.0277 - val_accuracy: 0.9937 - val_loss: 0.0203
Epoch 5/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9926 - loss: 0.0232 - val_accuracy: 0.9926 - val_loss: 0.0203
Epoch 6/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9923 - loss: 0.0229 - val_accuracy: 0.9939 - val_loss: 0.0184
Epoch 7/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9928 - loss: 0.0220 - val_accuracy: 0.9942 - val_loss: 0.0178
Epoch 8/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9930 - loss: 0.0213 - val_accuracy: 0.9942 - val_loss: 0.0183
Epoch 9/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9927 - loss: 0.0209 - val_accuracy: 0.9934 - val_loss: 0.0193
Epoch 10/10
2058/2058 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9933 - loss: 0.0199 - val_accuracy: 0.9943 - val_loss: 0.0166
515/515 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9949 - loss: 0.0149
Test Accuracy: 0.9942885041236877
```

- **code of Cnn in SDN_Flow dataset:**

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import layers, models
import tensorflow as tf
X_train_array = np.array(X_train)
X_test_array = np.array(X_test)
X_train_reshaped = X_train_array.reshape(X_train_array.shape[0], X_train_array.shape[1], 1)
X_test_reshaped = X_test_array.reshape(X_test_array.shape[0], X_test_array.shape[1], 1)
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
num_classes = len(label_encoder.classes_)
y_train_one_hot = to_categorical(y_train_encoded, num_classes=num_classes)
y_test_one_hot = to_categorical(y_test_encoded, num_classes=nu_classes)
model = models.Sequential([
layers.Conv1D(64, 3, activation='relu', input_shape=(X_train_reshaped.shape[1], 1)),
layers.MaxPooling1D(2),
layers.Dropout(0.2),
layers.Flatten(),
layers.Dense(64, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.01)),
layers.Dense(num_classes, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
```

model.fit(X_train_reshaped, y_train_one_hot, epochs=50, validation_data=(X_test_reshaped, y_test_one_hot), callbacks=[early_stopping])

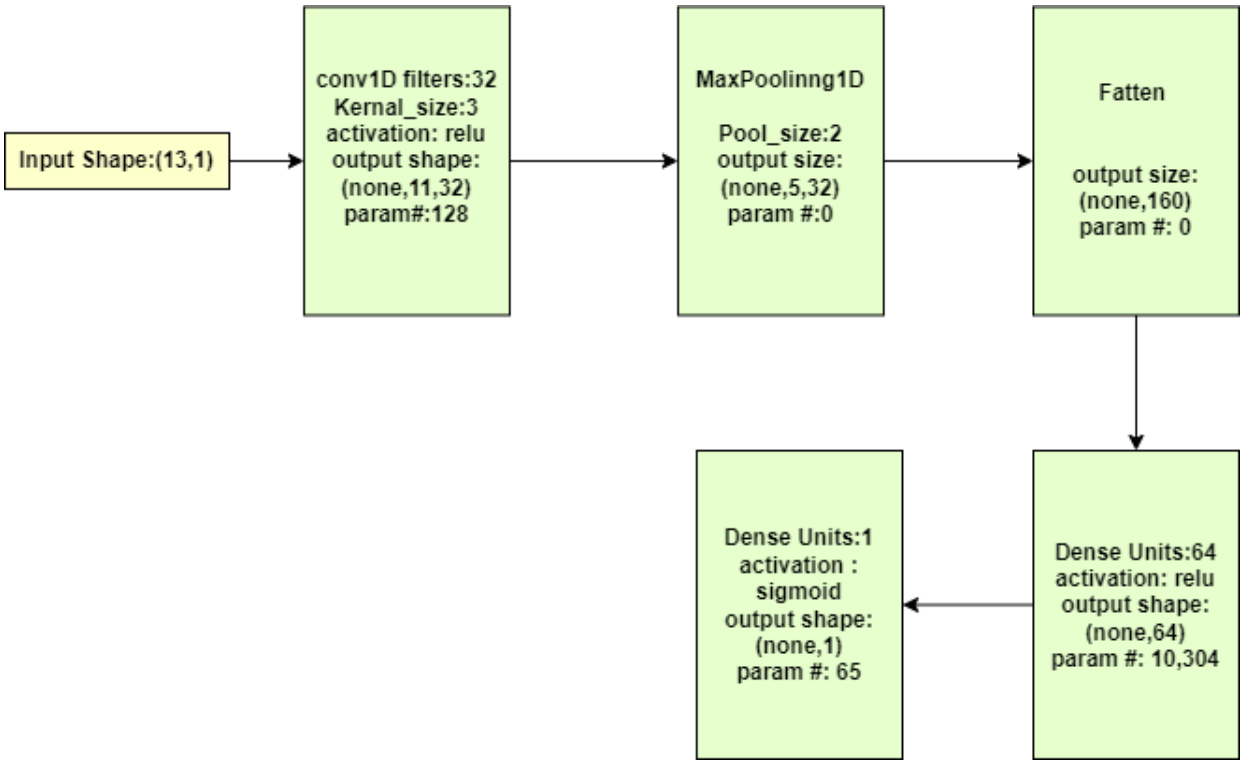
test_loss, test_acc = model.evaluate(X_test_reshaped, y_test_one_hot, verbose=2) print('Test accuracy:', test_acc)

```
Epoch 1/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 107s 7ms/step - accuracy: 0.8783 - loss: 0.5115 - val_accuracy: 0.9682 - val_loss: 0.1826
Epoch 2/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 102s 7ms/step - accuracy: 0.9551 - loss: 0.1959 - val_accuracy: 0.9705 - val_loss: 0.1370
Epoch 3/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 103s 7ms/step - accuracy: 0.9659 - loss: 0.1534 - val_accuracy: 0.9795 - val_loss: 0.1103
Epoch 4/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 101s 7ms/step - accuracy: 0.9706 - loss: 0.1343 - val_accuracy: 0.9846 - val_loss: 0.0989
Epoch 5/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 99s 7ms/step - accuracy: 0.9734 - loss: 0.1224 - val_accuracy: 0.9788 - val_loss: 0.1014
Epoch 6/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 99s 7ms/step - accuracy: 0.9750 - loss: 0.1176 - val_accuracy: 0.9808 - val_loss: 0.0958
Epoch 7/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 102s 7ms/step - accuracy: 0.9759 - loss: 0.1132 - val_accuracy: 0.9847 - val_loss: 0.0896
Epoch 8/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 102s 7ms/step - accuracy: 0.9772 - loss: 0.1080 - val_accuracy: 0.9716 - val_loss: 0.1083
Epoch 9/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 101s 7ms/step - accuracy: 0.9784 - loss: 0.1039 - val_accuracy: 0.9738 - val_loss: 0.1033
Epoch 10/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 100s 7ms/step - accuracy: 0.9792 - loss: 0.1012 - val_accuracy: 0.9846 - val_loss: 0.0834
Epoch 11/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 101s 7ms/step - accuracy: 0.9799 - loss: 0.0984 - val_accuracy: 0.9874 - val_loss: 0.0919
Epoch 12/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 100s 7ms/step - accuracy: 0.9812 - loss: 0.0942 - val_accuracy: 0.9853 - val_loss: 0.0864
Epoch 13/50
14500/14500 ━━━━━━━━━━━━━━━━━━━━ 98s 7ms/step - accuracy: 0.9821 - loss: 0.0906 - val_accuracy: 0.9817 - val_loss: 0.0964
6215/6215 - 19s - 3ms/step - accuracy: 0.9846 - loss: 0.0834
Test accuracy: 0.9845812916755676
```

6. MODEL SPECIFICATION

6.1. SDN_DDoS Dataset

To create a diagram based on the provided model summary, we need to visualize each layer's structure, output shape, and the number of parameters. Here's a textual representation of what the diagram would look like:



1. Input Layer
    - Input shape is not explicitly mentioned, but assuming it is (13, 1) based on the given first Conv1D layer's output shape. This means the input consists of 13 features with a single channel.
2. Conv1D Layer
    - Parameters: 128
    - Output Shape: (None, 11, 32). Here, the output shape means there are 32 filters (or channels) and the length of the sequence is reduced from 13 to 11 due to the convolution operation with a kernel size of 3.
3. MaxPooling1D Layer

- Parameters: 0
- Output Shape: (None, 160). This layer flattens the 2D output from the previous layer (5, 32) into a 1D array of 160.

4. Dense Layer

   - Parameters: 10,304
   - Output Shape: (None, 64). Fully connected layer with 64 units. The number of parameters is calculated as (160 * 64) + 64 (for the bias term).

5. Dense Output Layer

   - Parameters: 65
   - Output Shape: (None, 1). Fully connected layer with a single unit for binary classification with a sigmoid activation function. The number of parameters is (64 * 1) + 1 (for the bias term).

Total parameters

- Total params: 31,493
- Trainable params: 10,497
- Non-trainable params: 0
- Optimizer params: 20,996

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 11, 32)	128
max_pooling1d (MaxPooling1D)	(None, 5, 32)	0
flatten (Flatten)	(None, 160)	0
dense (Dense)	(None, 64)	10,304
dense_1 (Dense)	(None, 1)	65

Total params: 31,493 (123.02 KB)

Trainable params: 10,497 (41.00 KB)

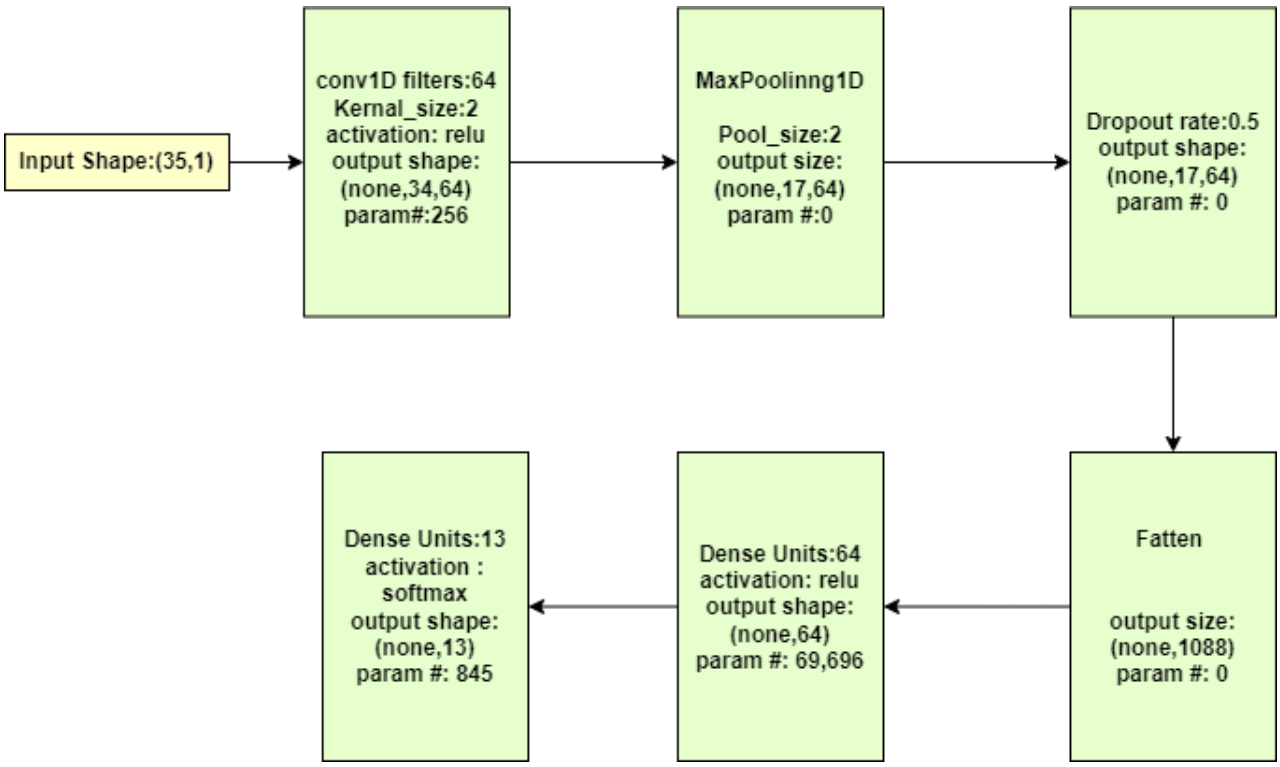
Non-trainable params: 0 (0.00 B)

Optimizer params: 20,996 (82.02 KB)

This diagram provides a visual and textual summary of the CNN architecture, highlighting the input and output shapes at each layer along with the number of parameters involved.

6.2. SDN_Flow Dataset

To create a diagram based on the provided model summary, we need to visualize each layer's structure, output shape, and the number of parameters. Here's a textual representation of what the diagram would look like:



1. Input Layer
    - Input shape: (35, 1). This means the input consists of 35 features with a single channel
2. Conv1D Layer
    - Parameters: 256
    - Output Shape: (None, 34, 64). The sequence length is reduced from 35 to 34 due to the convolution operation with a kernel size of 2.
3. MaxPooling1D Layer
    - Parameters: 0
    - Output Shape: (None, 17, 64). The pooling operation reduces the sequence length from 34 to 17 by taking the maximum value in each pool size of 2.

4. Dropout Layer
    - Parameters: 0
    - Output Shape: (None, 17, 64). This layer randomly sets a fraction (0.5) of input units to 0 at each update during training to prevent overfitting.
5. Flatten Layer
    - Parameters: 0
    - Output Shape: (None, 1088). This layer flattens the 3D output from the previous layer (17, 64) into a 1D array of 1088
6. Dense Layer
    - Parameters: 69,696
    - Output Shape: (None, 64). Fully connected layer with 64 units and ReLU activation.
7. Dense Output Layer
    - Dense Output Layer
    - Output Shape: (None, 13). Fully connected layer with 13 units (for multi-class classification) and softmax activation.

Total Parameter:

- Total params: 212,393 (829.66 KB)
- Trainable params: 70,797 (276.55 KB)
- Non-trainable params: 0
- Optimizer params: 141,596 (553.11 KB)

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 34, 64)	256
max_pooling1d (MaxPooling1D)	(None, 17, 64)	0
dropout (Dropout)	(None, 17, 64)	0
flatten (Flatten)	(None, 1088)	0
dense (Dense)	(None, 64)	69,696
dense_1 (Dense)	(None, 13)	845

Total params: 212,393 (829.66 KB)

Trainable params: 70,797 (276.55 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 141,596 (553.11 KB)

This diagram provides a visual and textual summary of the CNN architecture, highlighting the input and output shapes at each layer along with the number of parameters involved.

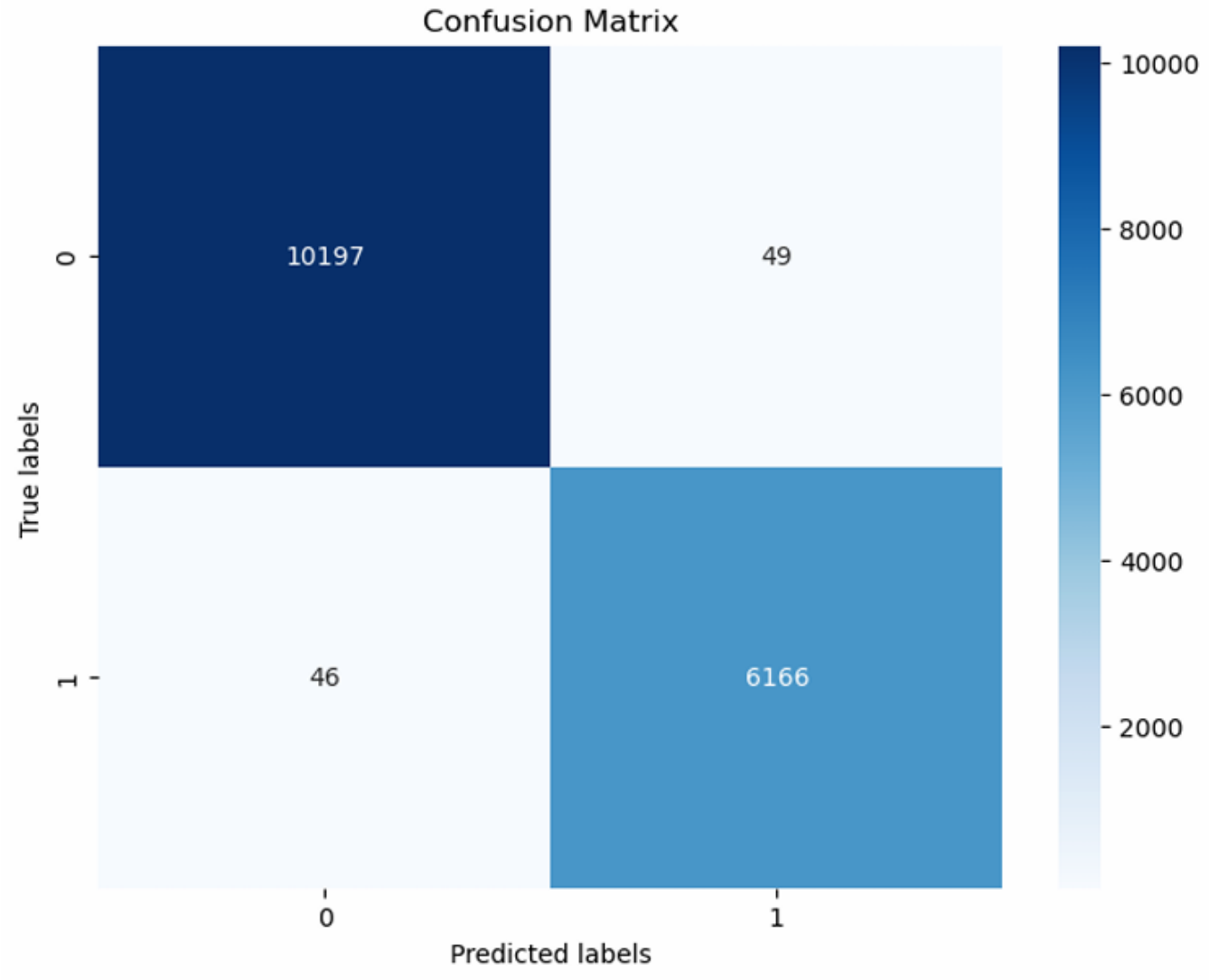
7. RESULT

7.1. Confusion matrix

In this section, we present the findings from our experiments on DDoS attack detection in an SDN network. We evaluated our detection mechanism using two datasets: one binary and one multi-class. The binary dataset classifies traffic as either ”DDoS attack” or ”No DDoS attack,” while the multi-class dataset categorizes traffic into different types of requests.

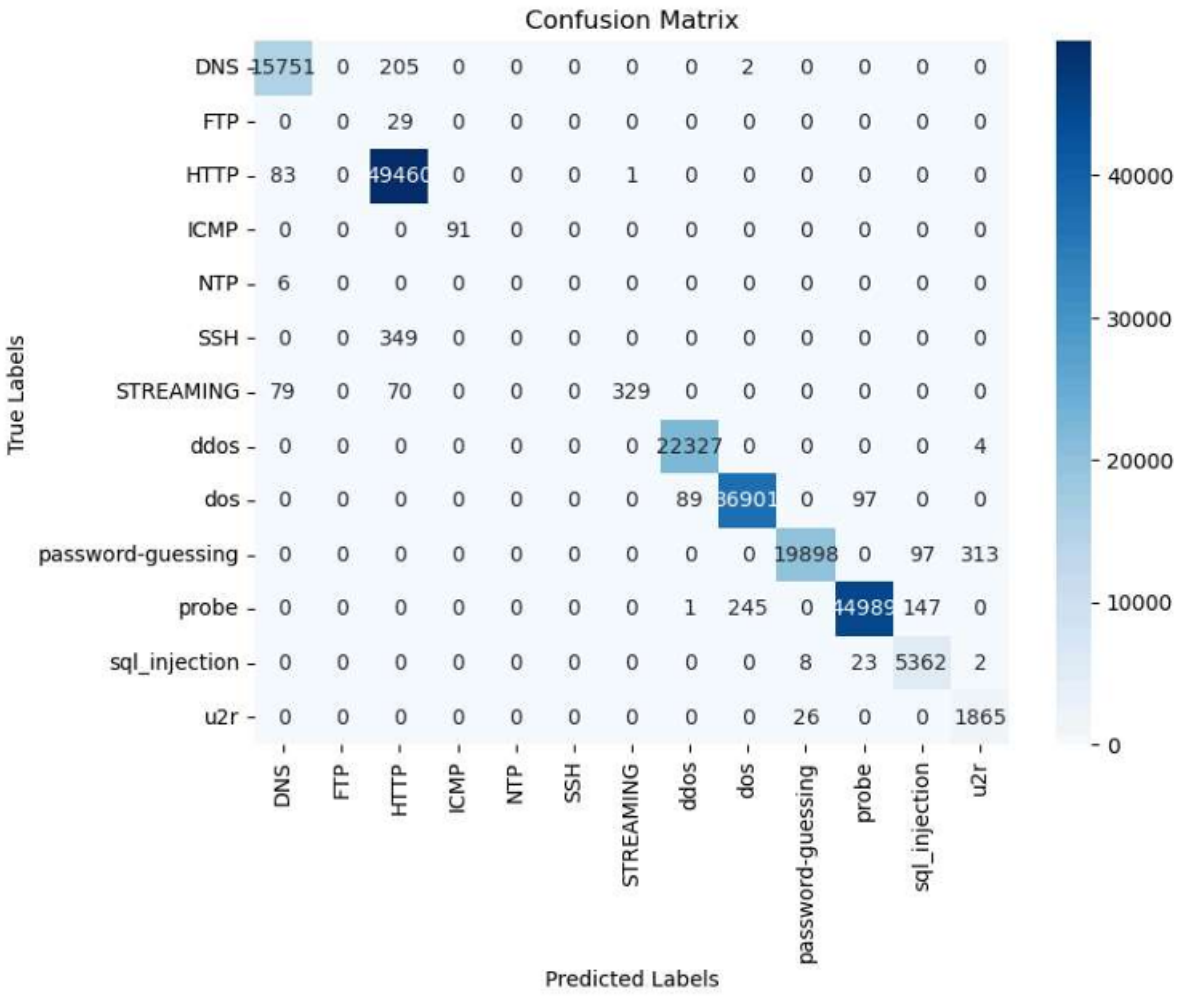
7.1.1. SDN_DDOS Model Result

For the binary classification, we used a dataset with two labels: ”DDoS attack” (Yes) and ”No DDoS attack” (No). The confusion matrix for the binary classification is presented below:



7.1.2. SDN_FlOW Model Result

For the multi-class classification, the dataset includes multiple labels indicating different types of traffic requests such as HTTP, ICMP, SSH, DNS, etc. The confusion matrix for the multi-class classification is presented below:



7.1.3. Overall findings

By analyzing the confusion matrices and corresponding performance metrics, we can conclude:

- **From the First Dataset:** We successfully detect whether a traffic instance is a DDoS attack or not. This provides an initial filter to identify potential malicious traffic.
- **From the Second Dataset:** We classify the type of request for all traffic instances,

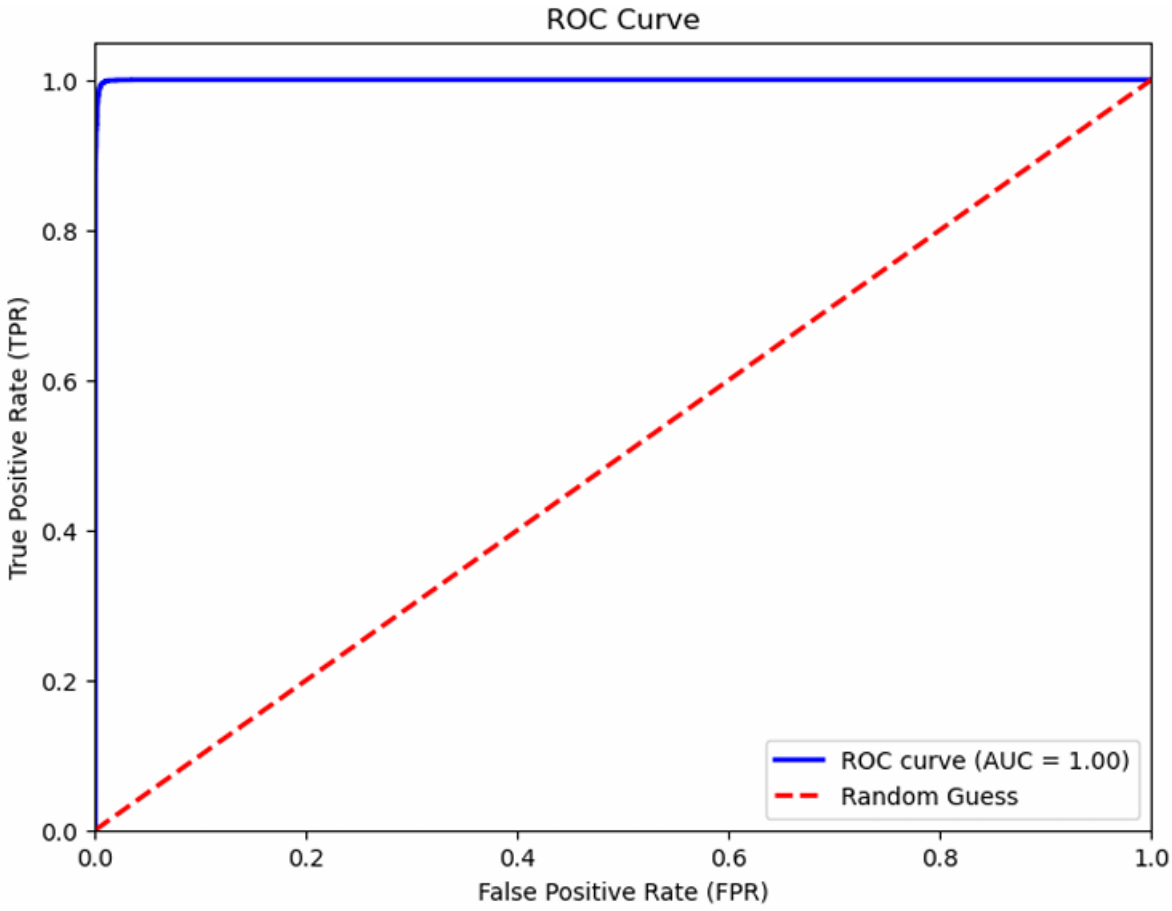
regardless of whether they are part of an attack. This comprehensive classification allows for detailed analysis of traffic patterns and better understanding of the nature of the requests.

7.2. Receiver Operating Characteristic curve

In this section, we present the ROC (Receiver Operating Characteristic) curve results for both the binary and multi-class classifications. The ROC curve is a graphical representation of a model's diagnostic ability, with the True Positive Rate (TPR) plotted against the False Positive Rate (FPR) at various threshold settings.

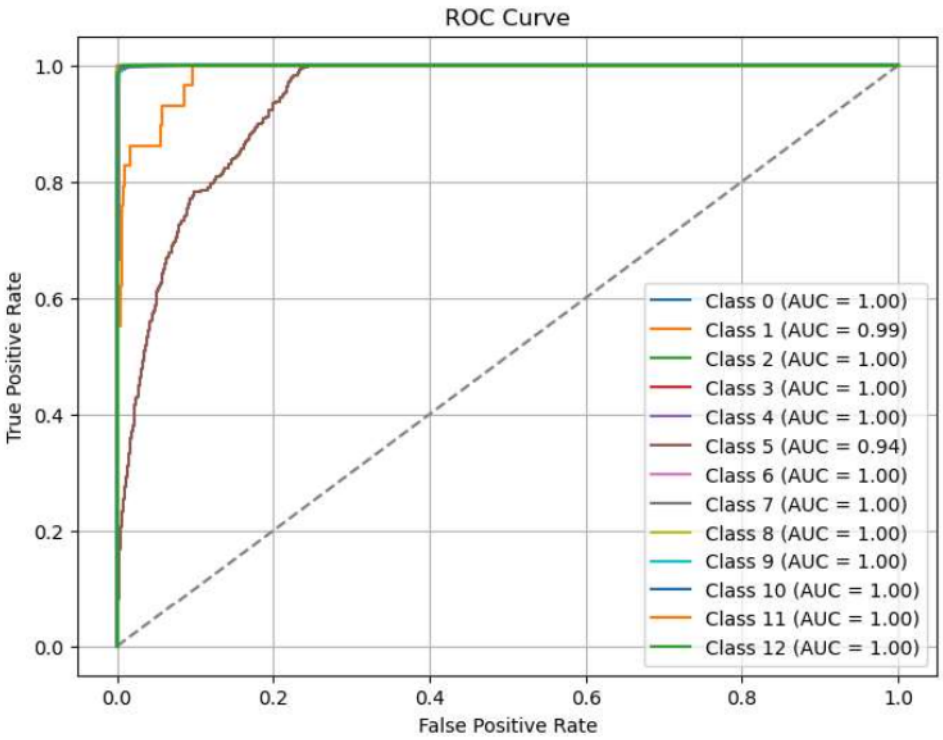
7.2.1. ROC Curve for Binary Classification

For the binary classification dataset, the ROC curve helps us understand the performance of the model in distinguishing between DDoS attack and non-DDoS traffic.



7.2.2. ROC Curve for Multi-class Classification

For the multi-class classification dataset, we generate ROC curves for each class. Since it's a multi-class problem, we use the One-vs-Rest (OvR) approach, where we compute the ROC curve for each class against all other classes.



7.2.3. Overall findings

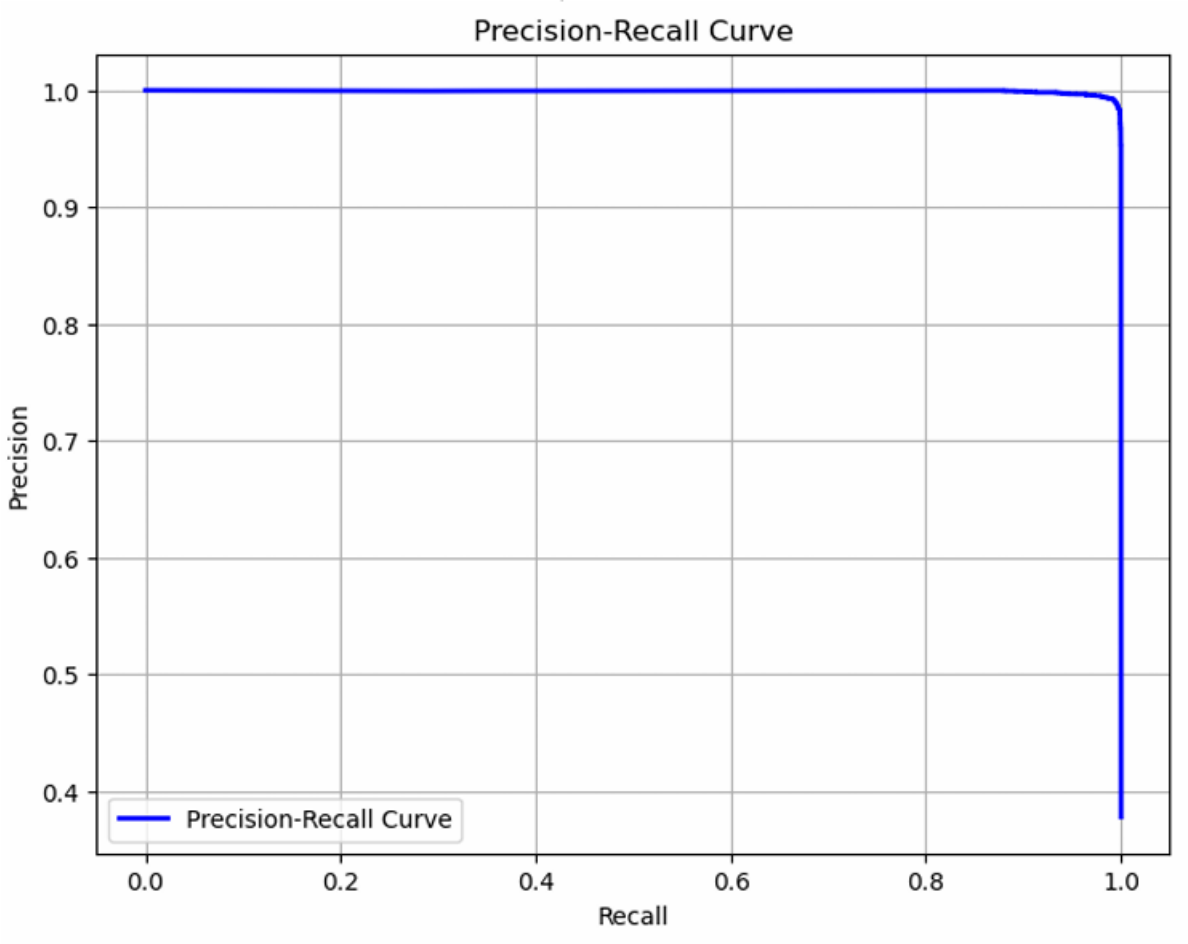
- The ROC curve for binary classification shows that the model has a high true positive rate and a low false positive rate, indicating strong discrimination between DDoS attack and non-DDoS traffic.
- The ROC curves for individual classes (HTTP, ICMP, SSH, DNS, Others) indicate that the model effectively differentiates between various types of traffic.
- A high AUC value reflects the effectiveness of the binary classification model.

7.3. Precision-Recall graph

n this section, we present the Precision-Recall (PR) curve results for both the binary and multi-class classifications. The PR curve is useful for evaluating the performance of a classification model, especially when dealing with imbalanced datasets.

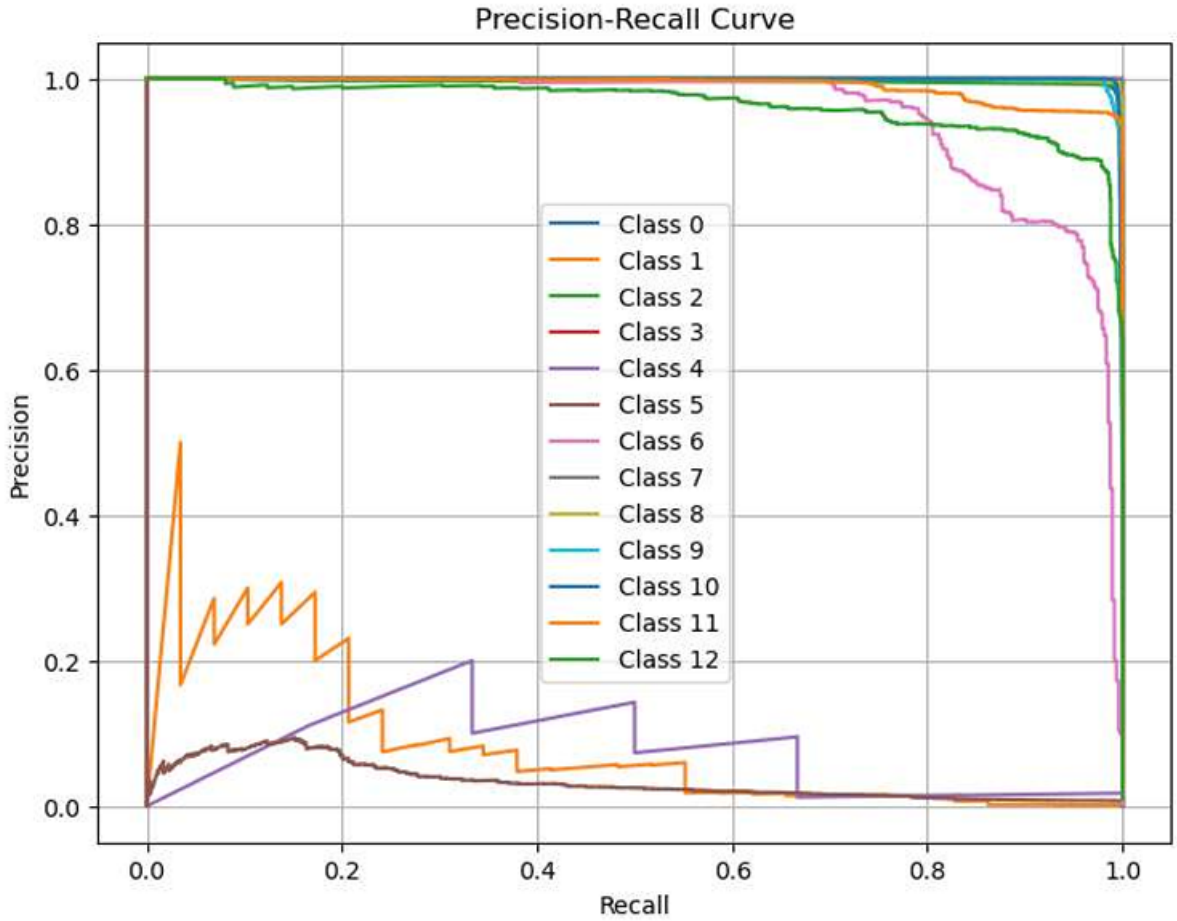
7.3.1. Precision-Recall Curve for Binary Classification

For the binary classification dataset, the PR curve helps us understand the trade-off between precision and recall for different threshold settings.



7.3.2. Precision-Recall Curve for Multi-class Classification

For the multi-class classification dataset, we generate PR curves for each class. Similar to the ROC curve approach, we use the One-vs-Rest (OvR) strategy, where we compute the PR curve for each class against all other classes.



7.3.3. Overall findings

- The PR curve for binary classification shows that the model maintains a high level of precision while achieving high recall, indicating effective detection of DDoS attacks with minimal false positives.
- The PR curves for individual classes (HTTP, ICMP, SSH, DNS, Others) indicate that the model effectively balances precision and recall across different types of traffic.
- A high AUC-PR value reflects the strong performance of the classification model.

8. CONCLUSION

Our project successfully developed a DDoS attack detection system tailored for SDN environments. SDN_DDOS model successfully differentiates attack traffic and normal traffic. SDN_Flow model successfully obtains the type of the traffic. The project contributes to the ongoing effort to safeguard SDN networks from cyber threats.

DDOS an important part of SDN security. Therefore in the identification process DDOS detection must have high accuracy and efficiency. To guarantee normal communication of network. SVM (Support Vector Machine) is an effective method for when used in large amounts of data. In order to defend from DDOS attack in SDN and improve the accuracy of SVM, We proposed SVM for DDOS detection in SDN. Our proposed method has achieved a value accuracy as 99%, precision as 99%, recall as 99%.

In the future, we plan to perform DDoS detection in SDN with proposed network in a real network to get more tangible results by adding the mitigation scheme. Future enhancements may include further optimization for scalability and adaptation to evolving attack strategies.

9. REFERENCES

1. DDOS ATTACKS DATASET FOR SOFTWARE DEFINED NETWORK.

   https://data.mendeley.com/datasets/x6vr3sdm75/1

2. Aslam, N., Srivastava, S. Gore, M.M. A Comprehensive Analysis of Machine Learning- and Deep Learning-Based Solutions for DDoS Attack Detection in SDN.Arab J Sci Eng.49, 3533–3573 (2024)..

   https://doi.org/10.1007/s13369-023-08075-2

3. Hnamte, V., Hussain, J. An efficient DDoS attack detection mechanism in SDN environment.Int. j. inf. tecnol.15, 2623–2636 (2023).

   https://doi.org/10.1007/s41870-023-01332-5

References