

# **DDOS Attack Detection Using Machine Learning**

## **Introduction**

In a constantly evolving landscape where malicious entities persistently seek new avenues to exploit individuals and businesses, detecting cyber attacks is crucial to maintaining integrity. Among the various threats, a distributed denial of service attack, or DDOS, stands out as a potential source of significant damage to both individuals and businesses alike. Numerous research papers have delved into more efficient methods of detecting these attacks, the ones we chose to cover involved detection using machine learning.

Machine learning, a branch of artificial intelligence, focuses on utilizing data to train models that progressively improve in accuracy with gradually increasing data input. The papers we selected had various methodologies, with some incorporating SNORT, an open-source intrusion detection system, while others leverage algorithms such as random forest and linear regression. Certain projects opt for a hybrid strategy, amalgamating these methods to minimize false positives and enhance overall accuracy.

The primary objective of our project was to train a machine learning program adept at successfully and accurately detecting DDOS attacks. To achieve this, we utilized logistic regression and support vector machines.

## **Literature Review**

Rana Abubakar et al.[1] proposed an optimized Distributed Denial of Service (DDoS) protection technique integrating with SNORT IPS. The approach involves rigorous monitoring of network traffic using supervised learning and comparison with datasets like KDDCUP99. The algorithm distinguishes benign and malicious traffic, implementing individual packet thresholds for accurate identification, addressing the challenge of varying threshold values in different networks.

Mazhar Javed Awan et al.[2] conducted research on real-time prediction of application layer DDoS attacks using machine learning. They employed Random Forest and Multi-Layer Perceptron models implemented with Scikit ML and Spark ML libraries. The study focused on detecting Denial of Service attacks and optimized model performance for faster predictions compared to existing methods. Regardless of the use of big data approaches, the models achieved a high mean accuracy of 99.5%.

H. Karthikeyan et al.[3] introduce an innovative method to detect DDoS flooding attacks in Intelligent and Innovative Transportation Systems (IITS), enhancing security. The approach incorporates reinforcement learning and is tailored to meet the security needs of ITS, overcoming limitations in current detection methods for advanced vehicular systems. The framework integrates the Q learning algorithm. To ensure a fully secure connection, future efforts should go beyond mitigating DDoS flooding attacks and address other potential threats.

Sanjeetha R et al.[4] introduces a model that dynamically calculates real-time threshold limits for applications transmitting data to a specific switch using machine learning. The model evaluates incoming application traffic to identify Distributed Denial of Service activity. Detected DDoS-related application types are selectively blocked, ensuring uninterrupted network traffic for legitimate applications. The dynamic threshold, adjusted based on current network conditions, enhances the efficiency of DDoS detection.

Shah et al.[5] compared SNORT IPS and Suricata IPS, finding that SNORT consumes fewer computational resources than Suricata. Despite Suricata handling more packets per second, both systems exhibit high false positive rates (FPR). SNORT triggered a 55.2% FPR compared to Suricata's 74.3% FPR with default rules. Due to Suricata's multi-threading capabilities, it requires more memory and CPU resources than SNORT, with Suricata's 4-core CPU utilization exceeding that of SNORT. Suricata also used an average of 3.8GB memory, surpassing SNORT's 600MB utilization at 10Gb of network resources.

Xuan et al.[6] proposed DeepDefense, a deep learning model for DDoS attack detection. They framed DDoS detection as a series of classification problems, transitioning from packet-based to a Windows-based approach. DeepDefense integrates Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), and fully connected layers of Artificial Neural Network (ANN). The use of LS-TM and GRU in RNN addresses scaling issues, resulting in improved feature learning, especially in longer historical contexts, and better generalization performance compared to random forest.

Zhang et al.[7] proposed an intrusion detection framework utilizing the Random Forest algorithm within Apache Spark's big data environment. The algorithm was applied to high-speed network data, showcasing higher accuracy and faster intrusion detection (0.01 seconds) compared to existing models, including a basic Random Forest model that took 1.10 seconds for intrusion detection. The study evaluated limited classification algorithms, a notable limitation.

Ahmad et al.[8] evaluated SVM, RF, and ELM models for network intrusion detection, employing an 80-20 train-test split. ELM demonstrated superior performance, with a 95.5% accuracy and precision on the full dataset, albeit with a slightly lower recall compared to SVM and RF.

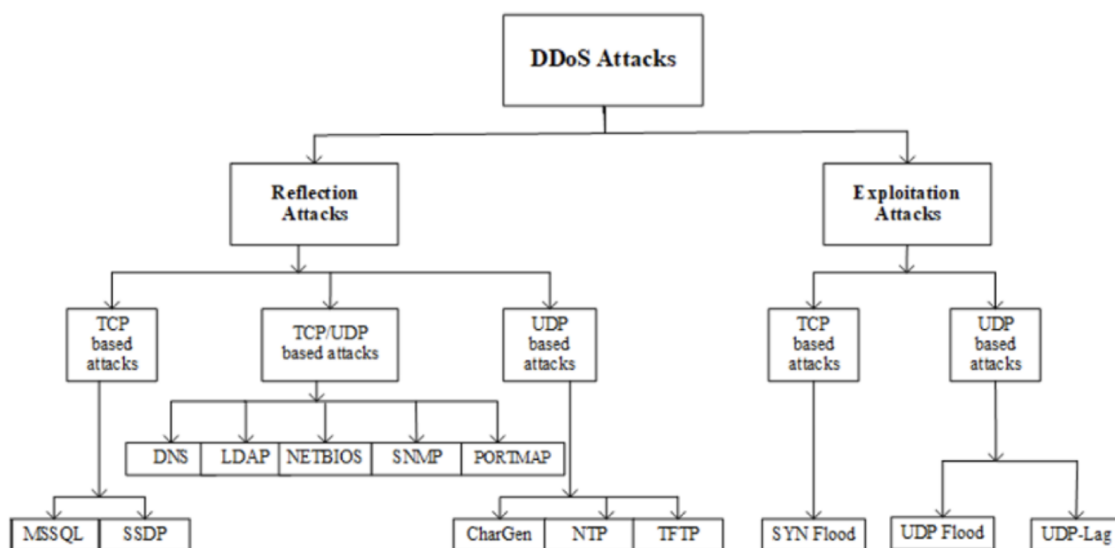
Dehkordi et al.[9] proposed a method for DDoS attack detection in Software Defined Networks (SDN) using various machine learning models. The approach, consisting of data collection,

entropy-based analysis, and classification, achieved high accuracy on the ISCX-SlowDDoS2016 dataset: Logistic algorithms (99.62%), J48 algorithm (99.87%), BayesNet algorithm (99.33%), Random Tree algorithm (99.8%), and REPTree algorithm (99.88%).

Liang et al.[10] employed a statistical method alongside machine learning techniques for attack detection. The statistical approach models normal and abnormal network behaviors using predetermined distributions and distance measurement techniques. In the machine learning phase, classifiers including K-Means, SVM, decision tree, Naive Bayes algorithm, and AI algorithm are applied.

## DataSet Description

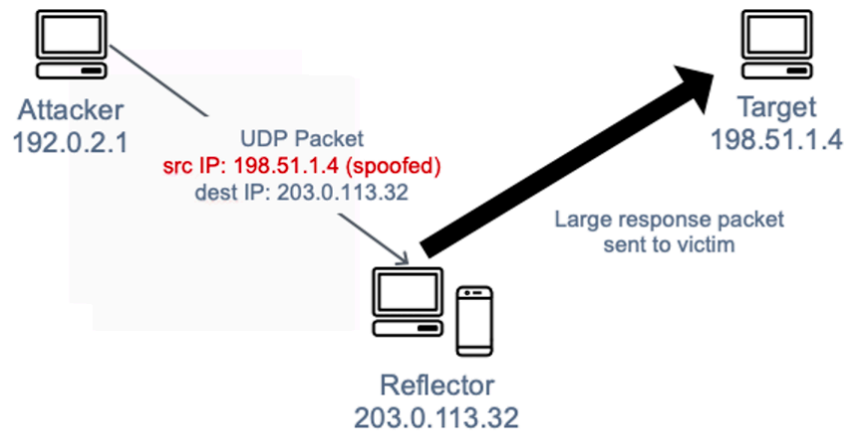
The CIC-DDoS2019 dataset, developed by the Canadian Institute for Cybersecurity, is a comprehensive collection designed to aid in the detection and analysis of Distributed Denial of Service (DDoS) attacks [11]. This dataset addresses the critical need for real-time DDoS attack detection with minimal computational overhead and supports the evaluation of new detection algorithms and techniques [12-13].



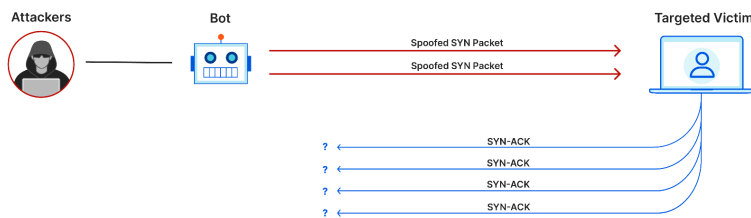
**Dataset Composition:** CICDDoS2019 encompasses a wide range of benign and common DDoS attacks, simulating realistic real-world data. It includes network traffic analysis results, with labeled flows indicating various parameters like timestamps, IP addresses, ports, protocols, and the type of attack.

**Taxonomy of DDoS Attacks:** The dataset identifies two primary types of DDoS attacks:

**1- Reflection-based attacks:** Reflection-based attacks hide the attacker's identity by using legitimate third-party components. In these attacks, the attacker sends packets to reflector servers, setting the source IP address to that of the target victim. This causes the reflector servers to send response packets back to the victim's IP address.



**2- Exploitation-based attacks:** exploitation-based attacks also conceal the attacker's identity but exploit legitimate components differently, often overwhelming the target with response packets. Exploitation-based attacks are a category of cyber attacks where the attacker exploits known vulnerabilities in systems or protocols to carry out a DDoS attack. A common example is the SYN flood attack, which takes advantage of the TCP handshake process to consume server resources, leading to service disruption.



**Types of Attacks Recorded:** A range of modern reflective DDoS attacks were executed and recorded in the dataset, including:

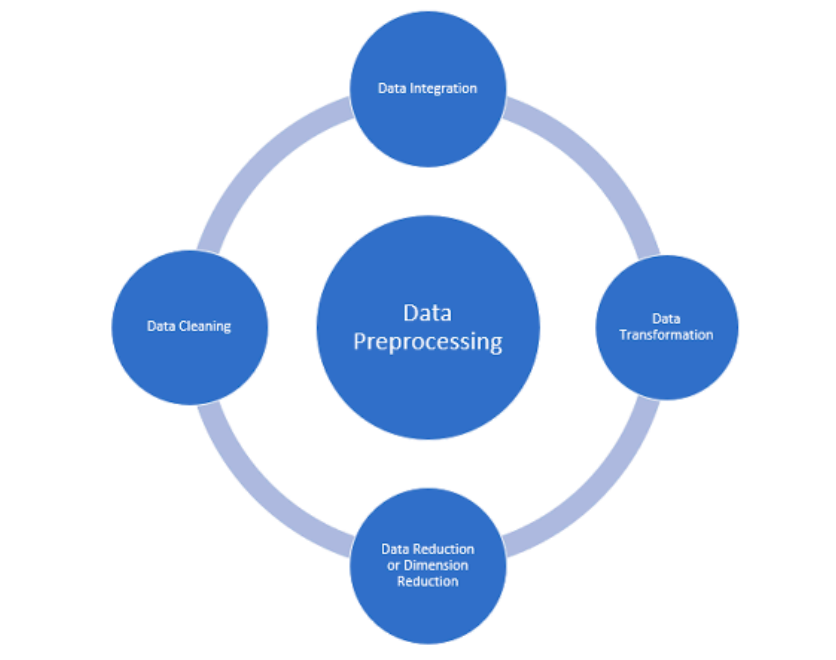
- a. **PortMap**
- b. **NetBIOS**
- c. **LDAP**
- d. **MSSQL**
- e. **UDP**
- f. **UDP-Lag**
- g. **SYN**
- h. **NTP**
- i. **DNS**
- j. **SNMP**

The dataset details the specific attacks executed on different days and the corresponding times .This dataset provides a valuable resource for researchers and cybersecurity professionals aiming to develop more effective methods for detecting and analyzing DDoS attacks.

**Licensing and Redistribution:** The CICDDoS2019 dataset can be redistributed and republished in any form, but any use or redistribution must include proper citation to the dataset and its associated research paper.

## Preprocessing

In the data preprocessing section, we explain a critical phase of enhancing the quality and usability of the CICDDOS2019 dataset for our DDoS detection model. This important stage involves a series of methods and operations aimed at addressing various challenges in raw data, such as handling missing values, encoding categorical variables, and normalizing feature scales. Furthermore, we explain feature selection methodologies, including correlation-based filtering and leveraging the Extra Trees Classifier to identify and retain the most informative features while removing redundancy. The preprocessing efforts are necessary for optimizing the performance of our machine learning model, promoting robustness, and ensuring that it can effectively find patterns of DDoS attacks within the network traffic data.



The preprocessing steps executed for performance optimization include:

1. Checking for Null Values
2. Checking for Duplicated Values

3. Checking Data Imbalance by Protocol
4. Balancing Data using Random Over Sampling
5. Balancing Data using SMOTE
6. Reordering Data
7. Label encoding
8. Min-Max Scaling
9. Correlation-based Feature Removal
10. Feature Selection using Extra Trees Classifier:

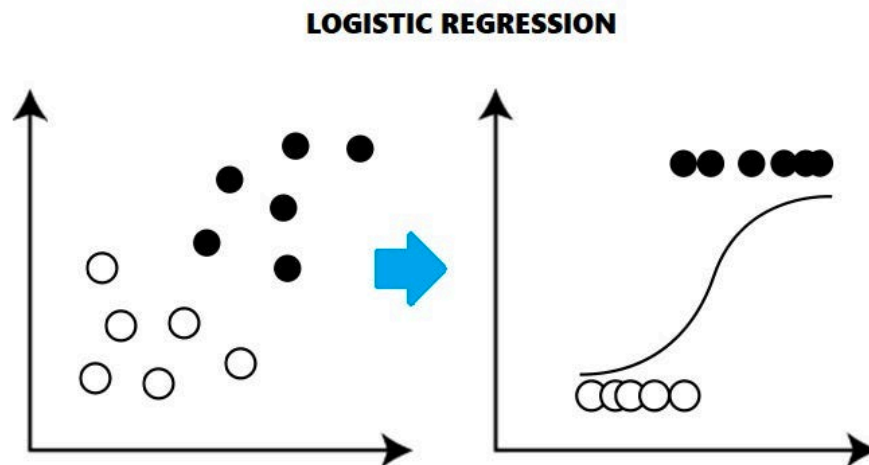
## **ML algorithms**

In recent years, the exponential growth of networked systems and the increasing dependence on digital infrastructures have made cybersecurity a paramount concern. Among all cyber threats, Distributed Denial of Service (DDoS) attacks pose a significant and persistent challenge to the availability and performance of online services. These attacks involve overwhelming a target system with a flood of traffic, rendering it inaccessible to legitimate users. As traditional methods struggle to cope with the scale and complexity of modern DDoS attacks, the integration of machine learning techniques emerges as a promising solution.

Machine learning, a subset of artificial intelligence, empowers systems to learn from data patterns and make predictions without explicit programming. In the context of cybersecurity, machine learning algorithms can analyze network traffic, detect anomalies, and discern malicious patterns indicative of a DDoS attack. This report explores the utilization of machine learning in the detection of DDoS attacks, delving into the intricacies of feature selection, model training, and evaluation.

### **Logistic Regression in DDoS Detection**

Logistic Regression is a widely-used classification algorithm that is particularly well-suited for binary classification tasks. In the context of DDoS detection, Logistic Regression serves as a powerful tool for distinguishing between normal and malicious network traffic. This chapter provides an overview of the key aspects of Logistic Regression in the application of DDoS detection, covering training the model, cost calculation, model evaluation, and the application of Logistic Regression in predicting and identifying DDoS attacks.



In Logistic Regression, the model is trained to learn the relationship between input features and the binary target variable, which, in the case of DDoS detection, represents whether a network instance is an attack or not. The training process involves adjusting the model parameters to minimize the logistic loss, a function that quantifies the difference between predicted and actual class labels.

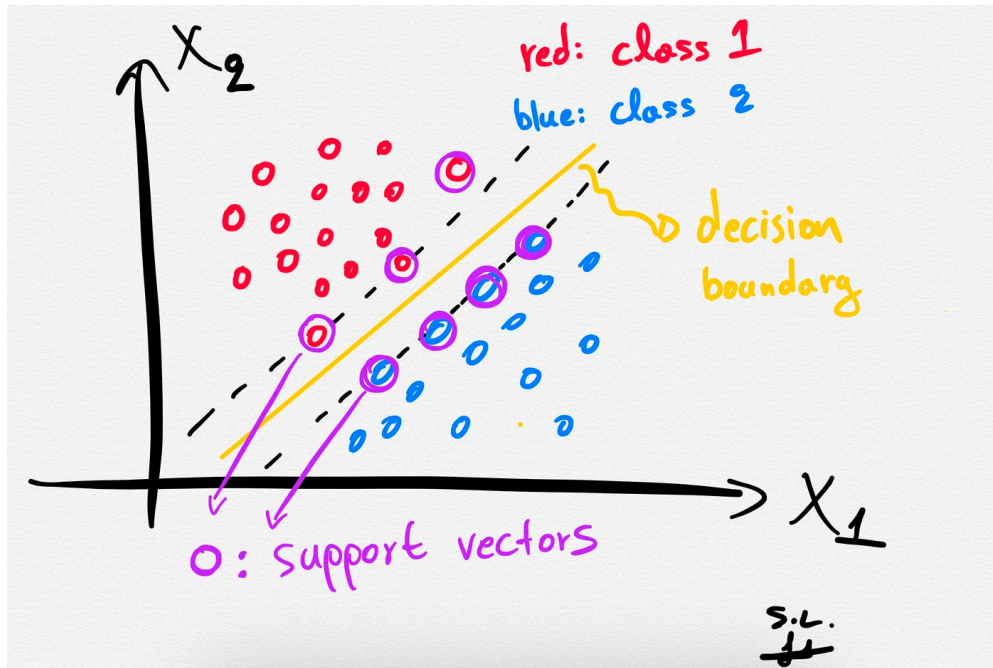
Evaluation of the Logistic Regression model is crucial for assessing its performance on unseen data. Common metrics include accuracy, precision, recall, and the F1 score. In the context of DDoS detection, these metrics provide insights into the model's ability to correctly identify attacks while minimizing false positives.

Logistic Regression finds application in DDoS detection by leveraging its ability to model complex relationships between network features and attack instances. It provides an interpretable framework for understanding the influence of each feature on the likelihood of an attack, aiding in the identification of key indicators.

## Support Vector Machines (SVM)

Support Vector Machines (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It works by finding a hyperplane in a high-dimensional space that best separates data points of different classes. The "support vectors," which are the data points closest to the decision boundary, play a crucial role in defining the optimal hyperplane.

SVM is a versatile algorithm known for its effectiveness in high-dimensional spaces and its ability to handle non-linear decision boundaries. Its performance depends on appropriate kernel selection and parameter tuning. SVM has found applications in various domains, including image classification, bioinformatics, and text categorization.



The training process of Support Vector Machines (SVM) involves finding the optimal hyperplane that maximizes the margin between classes. The primary objective is to separate data points of different classes with the largest possible margin, achieved by identifying support vectors, which are data points crucial for defining the decision boundary. SVM can handle non-linear decision boundaries through kernel functions like polynomial or radial basis function (RBF). The training process is essentially an optimization problem that aims to determine the optimal weights for each feature and the bias term. The algorithm minimizes classification errors while considering a soft margin in real-world scenarios, controlled by the regularization parameter ( $C$ ). The Sequential Minimal Optimization (SMO) algorithm is often employed for training, especially with non-linear kernels, as it efficiently solves the optimization problem by optimizing pairs of weights iteratively. The choice of the kernel and appropriate parameter tuning is essential for the successful implementation of SVM in classification tasks.

**Sequential Minimal Optimization (SMO) Algorithm:** SVM training involves solving a convex optimization problem. The optimization aims to find the optimal weights (coefficients) for each feature and the bias term. The support vectors influence the decision function.

The Sequential Minimal Optimization (SMO) algorithm is an optimization technique specifically designed for solving the quadratic programming problem that arises in training Support Vector Machines (SVM). The objective of SVM training is to find the optimal hyperplane that separates data points of different classes while maximizing the margin.

Understanding the mathematics behind SMO is essential for implementing SVM efficiently, as it provides insights into how the algorithm optimizes the Lagrange multipliers to find the optimal hyperplane.



# Methodology

## Code description for CICDDOS2019

### Installing Packages

The initial part of the code includes a function `install_package` designed to simplify the installation process of Python packages. The function uses the `subprocess` module to execute the installation command via the command line. The function uses the `subprocess` module to call the Python Package Manager (pip) and install the specified package.

### Data Loading

- **`pd.read_csv("DataSets/cicddos2019_dataset.csv", usecols=lambda column: column not in ['Unnamed: 0'])`**: This line reads the dataset from a CSV file (`cicddos2019_dataset.csv`) using Pandas. The `usecols` parameter is utilized to exclude the 'Unnamed: 0' column from the DataFrame.
- **`df.drop(['Label'], axis=1, inplace=True)`**: The 'Label' column, which represents different types of DDOS attacks, is removed from the DataFrame. This decision is motivated by the project's focus on detecting whether an attack is present or not, which is captured by the 'Class' column. One can use this column to create a model that detect different types of attacks.

### Checking for Null Values:

**`df.isnull().sum()`**: This line utilizes the `isnull` method to create a boolean DataFrame where True indicates the presence of a null value. The `sum` method is then applied to count the number of null values in each column.

### Handling Duplicate Rows

The code segment handles duplicate rows within the dataset by first assessing their presence and subsequently removing them to enhance data quality. The function `drop_duplicates` is employed to eliminate duplicate entries directly from the original DataFrame, ensuring a streamlined dataset for subsequent analyses. The approach focuses on maintaining the integrity of the data by eradicating redundant information, promoting accuracy in downstream tasks.

### Balancing Data

In this section of the code, the balance of the dataset with respect to the 'Protocol' feature is assessed, and oversampling techniques are employed for class balancing. The initial print

statement displays the distribution of protocols using the Counter function, providing insights into the class distribution. To address class imbalance, the code utilizes the Random Over Sampler (ROS) method, implemented through the RandomoverSampler function. This method generates synthetic instances by duplicating data from the minority class. The choice of Random Over Sampling is explained in the context of time efficiency, as it is observed to be less time-consuming than the Synthetic Minority Over-sampling Technique (SMOTE) method, especially for larger datasets. While SMOTE is a powerful technique that creates synthetic instances along decision boundaries, its computational demands are significant for large datasets. Hence, for practical considerations, the decision to use Random Over Sampling is justified to achieve a balanced dataset within reasonable computational resources.

```
#checking if the data is balanced according to Protocols used
print (sorted(Counter(df['Protocol']).items()))

[49] ✓ 0.0s
... [(0, 1622), (6, 112422), (17, 304712)]

#the data is imbalanced
X=df
Y=df['Protocol']
X.drop(['Protocol'], axis=1,inplace=True)

[52] ✓ 0.1s

#balancing using oversampling
ros=RandomOverSampler(random_state=0)
X_resampled,y_resampled=ros.fit_resample(X,Y)
print (sorted(Counter(y_resampled).items()),y_resampled.shape)
## balancing using SMOTE
## Let's assume 'categorical_feature1' and 'categorical_feature2' are your categorical features.
#label_enc_1 = LabelEncoder()
#label_enc_2 = LabelEncoder()
#X['Label'] = label_enc_1.fit_transform(X['Label'])
#X['Class'] = label_enc_2.fit_transform(X['Class'])

## Specify which columns are categorical (after encoding). Assume they are the first two columns.
#categorical_features_indices = [X.columns.tolist().index('Class'),
#                                X.columns.tolist().index('Class')]

#smote_nc = SMOTENC(categorical_features=categorical_features_indices, random_state=42)
#X2_resampled, y2_resampled = smote_nc.fit_resample(X, Y)

#print("The result of SMOTE method:")
#print(sorted(Counter(y_resampled).items()), y_resampled.shape)

[53] ✓ 3.3s
... [(0, 304712), (6, 304712), (17, 304712)] (914136,)
```

## Encoding the class

In this segment, label encoding is applied to the 'Class' column of the balanced dataset (bf). The LabelEncoder function from the scikit-learn library is employed to transform categorical class labels into numerical representations. Specifically, the label encoder is instantiated and fitted to the 'Class' column using the fit\_transform method, replacing the categorical labels with corresponding numerical codes. This encoding facilitates the integration of categorical information into machine learning models, which often require numerical input. The resulting DataFrame is displayed to provide a glimpse of the transformed dataset. The application of label encoding in this context is essential for ensuring compatibility with classification algorithms and enabling effective model training on the balanced dataset.

## Data Scaling

In this section, feature scaling is performed on the balanced dataset (bf) using the Min-Max Scaling technique. The MinMaxScaler from scikit-learn is employed to scale the feature values between 0 and 1, ensuring uniformity and preventing the dominance of certain features due to their scale.

## Remove Correlated Features

A custom function named `correlation` is defined to identify and handle multicollinearity in the feature set. The function takes a dataset and a correlation threshold as parameters and returns a set of column names representing highly correlated features. The script then splits the scaled dataset into training and testing sets using the `train_test_split` function, with a test size of 30% and a random seed for reproducibility.

Following the data split, the script utilizes the `correlation` function to identify features that exhibit high correlation (greater than 0.8) within the training set (`X_train`). The identified correlated features are subsequently dropped from both the training and testing sets (`x_train` and `x_test`). This process is crucial for mitigating multicollinearity issues, ensuring that the input features are not overly redundant and preserving the independence of features during model training and evaluation.

```
def correlation(dataset, threshold):
    col_corr = set() # Set of all names of correlated columns
    corr_matrix = dataset.corr()
    for i in range(len(corr_matrix.columns)):
        for j in range(i):
            if abs(corr_matrix.iloc[i, j]) > threshold: # Note: corrected condition
                colname = corr_matrix.columns[i] # Name of the correlated column
                col_corr.add(colname)
    return col_corr

# Split the data
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=0)

# Identify the correlated features
ncorr_features = correlation(X_train, 0.8)

# Drop the correlated features
x_train = X_train.drop(ncorr_features, axis=1)
x_test = X_test.drop(ncorr_features, axis=1)
```

[227] ✓ 3.1s

## Feature Selection

In this section, feature selection is performed using the Extra Trees Classifier. An Extra Trees Classifier with 100 estimators is instantiated and fitted to the training set (`x_train` and `y_train`). The feature importances are then computed using the `feature_importances_` attribute of the classifier. The script prints the number of features in the training set and specifies the desired

number of features (n\_features) to be selected. In this instance, the variable n\_features is set to 5.

The feature importances are sorted, and the indices of the top features are obtained. The selected feature indices are printed, providing insights into which features are considered most important by the Extra Trees Classifier. Finally, the training and testing sets are updated to include only the selected features, facilitating a more focused and efficient model training process.

#### Important Features are:

- URG Flag Count: Number of packets with the URG flag set.
- Down/Up Ratio: Ratio of download to upload traffic.
- Fwd/Bwd Packet Length Min/Max: Minimum/Maximum size of packets in the forward/Backward direction.
- CWE Flag Count: Number of packets with the Common Weakness Enumeration (CWE) flag set.
- Init Fwd/ Bwd Win Bytes: Total number of bytes sent in the initial window in the forward/Backward direction.
- ACK Flag Count: Number of packets with the ACK flag set.
- Protocol: Network protocol used.
- Fwd PSH Flags: Number of packets with the PSH flag set in the forward direction.
- Flow Duration: Duration of the flow in microseconds.
- Bwd IAT Total: Total time between two packets sent in the backward direction.

```
▶ # print top 15 important features
top_15_indices = np.argsort(importances[::-1][:15])
top_15_features = x_train.columns[top_15_indices]
print("Top 15 Important Features:")
print(top_15_features)
```

[230] ✓ 0.0s

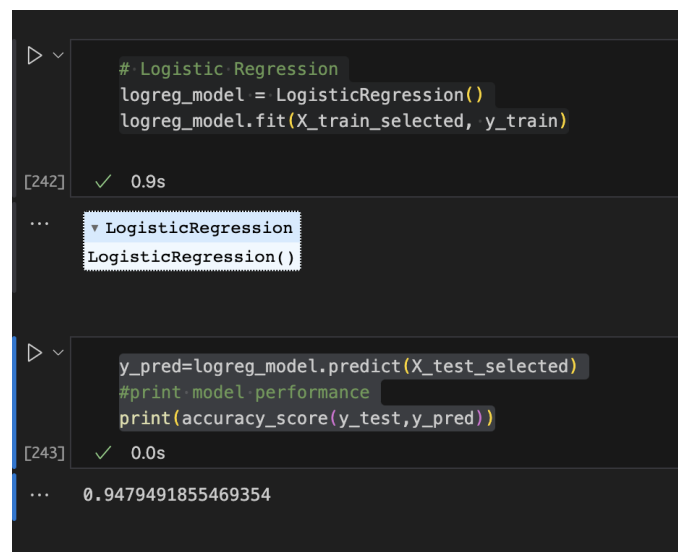
```
... Top 15 Important Features:
Index(['URG Flag Count', 'Fwd Packet Length Min', 'Fwd Packet Length Max',
      'Down/Up Ratio', 'Protocol', 'Bwd Packet Length Min', 'ACK Flag Count',
      'Bwd Packet Length Max', 'CWE Flag Count', 'Init Fwd Win Bytes',
      'Init Bwd Win Bytes', 'Fwd PSH Flags', 'Fwd Packet Length Std',
      'Fwd Packets Length Total', 'Flow Duration'],
      dtype='object')
```

## Logistic Regression

In this section, a Logistic Regression model is instantiated using the LogisticRegression function

from scikit-learn. The model is then trained on the selected features from the training set (X\_train\_selected and y\_train). Subsequently, the trained model is used to predict the target variable for the selected features in the testing set (X\_test\_selected). The script calculates and prints the accuracy score of the logistic regression model on the testing set using the accuracy\_score function.

The accuracy score provides an indication of the model's performance in correctly classifying instances in the testing set. It is a fundamental evaluation metric for classification models, representing the proportion of correctly predicted instances among the total instances. The printed accuracy score serves as a quantitative measure of the logistic regression model's effectiveness in making accurate predictions based on the selected features.



```

# Logistic Regression
logreg_model = LogisticRegression()
logreg_model.fit(X_train_selected, y_train)

[242] ✓ 0.9s

...
▼ LogisticRegression
LogisticRegression()

y_pred=logreg_model.predict(X_test_selected)
#print model performance
print(accuracy_score(y_test,y_pred))

[243] ✓ 0.0s

...
0.9479491855469354

```

## SVM

In this part of the code, a Support Vector Machine (SVM) classifier with a linear kernel is instantiated using the svm.SVC function from scikit-learn. The classifier is then trained on the selected features from the training set (X\_train\_selected and y\_train). Subsequently, the trained SVM model is used to predict the target variable for the selected features in the testing set (X\_test\_selected). The script calculates and prints the accuracy score of the SVM model on the testing set using the accuracy\_score function.

The accuracy score provides an assessment of the SVM model's performance in correctly classifying instances in the testing set. Additionally, the script prints the count of each class in the testing set to provide insights into the distribution of classes and aid in understanding the classification results. This information is valuable for assessing the model's effectiveness in handling different class instances and can be crucial in scenarios where class imbalance exists.

```
# Instantiate SVM classifier
# fit model
classifier=svm.SVC(kernel='linear',gamma='auto',C=2)
classifier.fit(X_train_selected,y_train)

[244] ✓ 4m 50.5s

... SVC
SVC(C=2, gamma='auto', kernel='linear')

y_predict=classifier.predict(X_test_selected)
#print model performances

print(accuracy_score(y_test,y_predict))

[245] ✓ 29.2s

... 0.9489150928816493

class_counts = pd.Series(y_test).value_counts()
print(class_counts)

[247] ✓ 0.0s

... Class
0.0    100011
1.0     29401
Name: count, dtype: int64
```

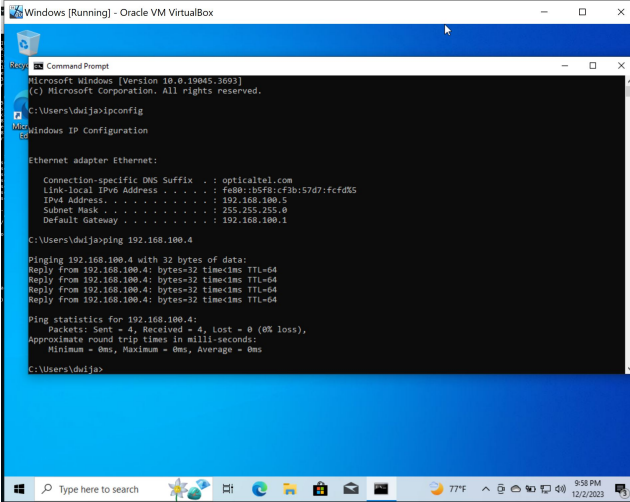
## Plot the confusion matrix for them

In this part of the code, confusion matrices for both the Logistic Regression (LR) and Support Vector Machine (SVM) models are generated and visualized using the `confusion_matrix` function from `scikit-learn`. The figure is divided into two subplots, with the left subplot representing the confusion matrix for the LR model, and the right subplot representing the confusion matrix for the SVM model. Each confusion matrix is displayed as a heatmap, with annotated values representing the counts of true positive, true negative, false positive, and false negative predictions.

Confusion matrices provide a comprehensive view of a classifier's performance by detailing the number of correct and incorrect predictions for each class. They are valuable tools for assessing the model's ability to correctly classify instances and identify potential areas for improvement. The generated visualizations enable a comparative analysis of the LR and SVM models in terms of their classification results.

# DDoS Attack Simulation

Two Virtual Machines, one running Kali Linux (attacker system) and another running Windows 10 (target system), were set up in Oracle VirtualBox. Both the virtual machines were placed in the same subnet, 192.168.100.0/24 (IP address of Kali Linux VM – 192.168.100.4, IP address of Windows VM – 192.168.100.5). Connectivity between the 2 systems was verified using the PING command. The command “nmap 192.168.100.5” was then executed on the attacker system (Kali Linux VM) to obtain a list of open ports on the target system (Windows VM). One of the open ports on the target system was 135/tcp, commonly associated with the Microsoft Remote Procedure Call (rpc) service. This open port was exploited to initiate a DDoS attack against the target machine, through the execution of the command "hping3 -S --flood --rand-source 192.168.100.5 -p 135". This command sent a high volume of SYN packets with random source IP addresses to port 135 of the target machine.



```
Windows [Running] - Oracle VM VirtualBox
Command Prompt
Microsoft Windows [Version 10.0.19045.3693]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dui>ipconfig

Windows IP Configuration

Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : opticaltel.com
   Link-local IPv6 Address . . . . . : fe80::b5f8:cfb3:57d7:fcfd%5
   IPv4 Address. . . . . : 192.168.100.5
   Subnet Mask . . . . . : 255.255.255.0
   Default Gateway . . . . . : 192.168.100.1

C:\Users\dui>ping 192.168.100.4

Pinging 192.168.100.4 with 32 bytes of data:
Reply from 192.168.100.4: bytes=32 time=1ms TTL=64
Reply from 192.168.100.4: bytes=32 time=1ms TTL=64
Reply from 192.168.100.4: bytes=32 time=1ms TTL=64
Reply from 192.168.100.4: bytes=32 time=1ms TTL=64

Ping statistics for 192.168.100.4:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\dui>
```

```
Applications ▾ Places ▾ Terminal ▾ Sun 03:04
root@kali: ~

File Edit View Search Terminal Help

root@kali:~# ifconfig
eth0: flags=16<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.100.4 netmask 255.255.255.0 broadcast 192.168.100.255
    inet6 fe80::e0b5:98ba:4ba9:e572 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:22:ae:63 txqueuelen 1000 (Ethernet)
    RX packets 868 bytes 1100192 (1.0 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 208 bytes 17992 (17.1 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

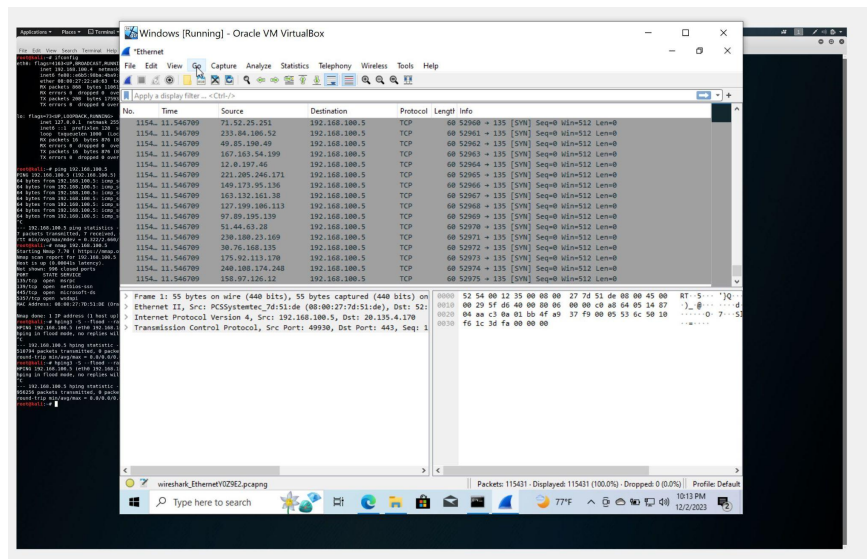
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (local loopback)
    RX packets 16 bytes 876 (876.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 16 bytes 876 (876.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@kali:~# ping 192.168.100.5
PING 192.168.100.5 (192.168.100.5): 56(88) bytes of data:
64 bytes from 192.168.100.5: icmp_seq=1 ttl=128 time=0.791 ms
64 bytes from 192.168.100.5: icmp_seq=2 ttl=128 time=0.398 ms
64 bytes from 192.168.100.5: icmp_seq=3 ttl=128 time=0.322 ms
64 bytes from 192.168.100.5: icmp_seq=4 ttl=128 time=0.744 ms
64 bytes from 192.168.100.5: icmp_seq=5 ttl=128 time=0.450 ms
64 bytes from 192.168.100.5: icmp_seq=6 ttl=128 time=0.156 ms
64 bytes from 192.168.100.5: icmp_seq=7 ttl=128 time=0.322 ms
^C
--- 192.168.100.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 852ms
rtt min/avg/max/mdev = 0.322/2.066/15.599/5.205 ms
root@kali:~# nmap 192.168.100.5
Starting Nmap 7.70 ( https://nmap.org ) at 2023-12-03 02:57 UTC
Nmap scan report for 192.168.100.5
Host is up (0.0004s latency).
Not shown: 996 closed ports
PORT      STATE SERVICE
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ss
5357/tcp  open  wsddapi
MAC Address: 08:00:27:70:51:DE (Oracle VM VirtualBox virtual NIC)

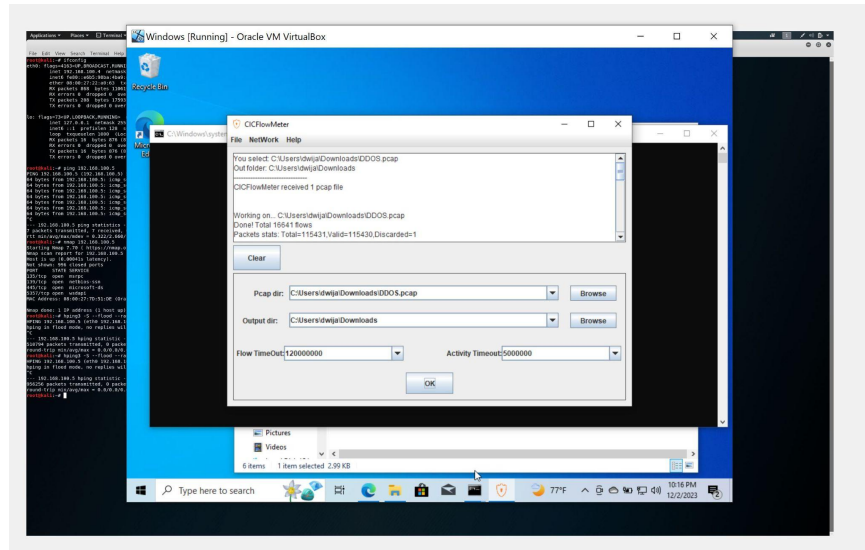
Nmap done: 1 IP address (1 host up) scanned in 3.27 seconds
root@kali:~# hping3 -S --flood --rand-source 192.168.100.5 -p 135
HPING 192.168.100.5 (eth0 192.168.100.5): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown.
--- 192.168.100.5 hping statistic ---
510794 packets transmitted, 0 packets received, 100% packet loss
round trip min/avg/max = 0.0/0.0/0.0 ms
root@kali:~#
```

## Packet Capture

On the target system, Wireshark was used to capture details of all the packets (both DDoS and benign) associated with IP address 192.168.100.5. The capture (output) was then saved as a PCAP file. Lastly, CICFlowMeter, a network traffic analyzer, was used to extract the necessary features from the PCAP file and subsequently export them to a CSV file.







## Code Description for Captured Data

### Imports and Setup:

Libraries like `joblib`, `pandas`, `numpy`, `sklearn.preprocessing`, `sklearn.svm`, and `sklearn.linear_model` are imported.

### Paths for four CSV files are defined.

The data contains 4 sets, 2 for benign and 2 for DDOS simulated attack. We merge one of the benign and one of the DDOS dataset and use it for training the models and merge two others to use for our test purpose. Two DataFrames (`df_test` and `df_train`) are created by concatenating these files.

### Data Loading and Preprocessing:

- CSV files are read into pandas DataFrames.
- A 'Class' column is added to these DataFrames to label the data.

### Feature Selection and Mapping:

- **A list of selected features is defined.** A dictionary for mapping feature names is created, and then applied to the selected features. This is because the name of the features are different in our simulated data and the CICDDOS-2019. In this section we used the important features that we captured using `ExtraTreeClassifier` in CICDDOS-2019 dataset to validate our results.

### Data Preparation for Training and Testing:

- The training DataFrame is shuffled.
- Target variables (y\_train and y\_test) and feature variables (x\_train and x\_test) are separated.
- The feature variables are scaled using MinMaxScaler.

### Logistic Regression Model:

- A Logistic Regression model is trained on the scaled data. The model is saved using joblib.dump and then loaded back.
- Predictions are made on the test set and accuracy is printed.

### Support Vector Machine (SVM) Model:

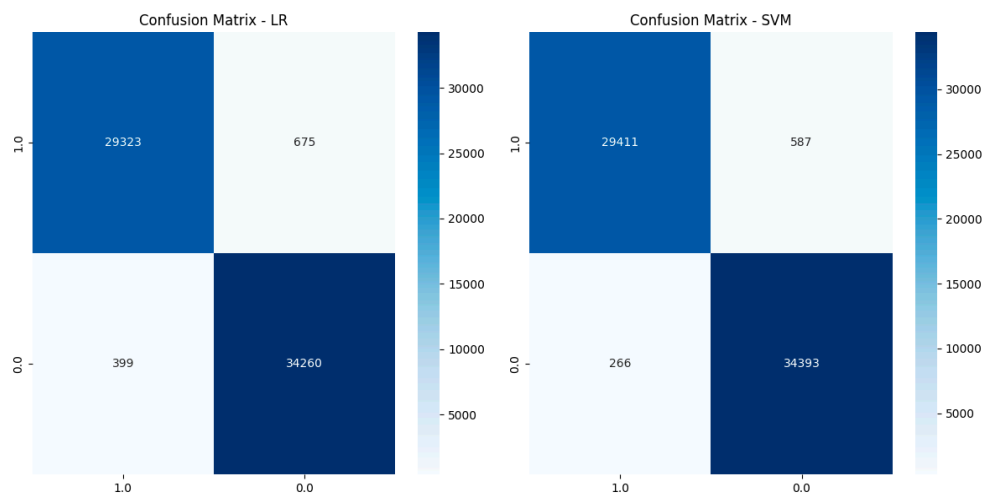
- An SVM classifier is instantiated, trained, and its performance is evaluated similarly to the Logistic Regression model.
- Saving Predictions and Analysis:
  - Predictions from the Logistic Regression model are added to df\_test.
  - The modified DataFrame is saved as a CSV file.
  - A count of classes in y\_test is printed.

### Visualization:

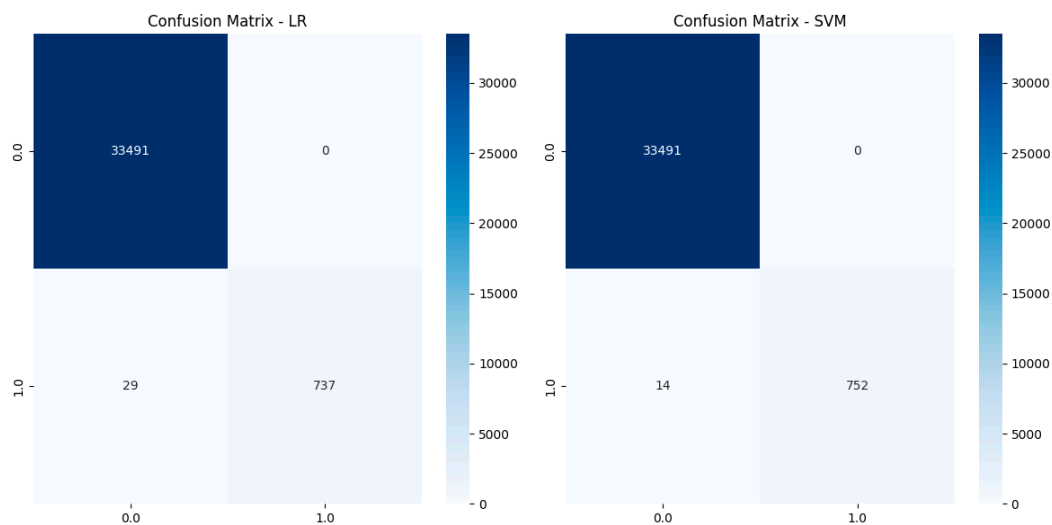
- Confusion matrices for both Logistic Regression and SVM models are plotted using Matplotlib and Seaborn.

## Results

The confusion matrices indicated a 97% accuracy rate for both Logistic Regression and SVM models focused on the CICDDOS2019 dataset while detecting Malicious DDoS attack traffic.



For the Logistic Regression and SVM models trained using data captured on the target machine, the confusion matrices indicated a 100% accuracy rate while detecting Malicious DDoS attack traffic.



## Conclusion and Future Works

Both the Logistic Regression and Support Vector Machine models demonstrated high accuracy in detecting DDOS attacks in the extensive CICDDOS-2019 dataset and a simulated dataset generated using hping3. Logistic Regression showed remarkable sensitivity and specificity with the structured CICDDOS-2019 dataset, while the SVM outperformed LR in the simulated dataset, particularly by minimizing false negatives, which is crucial for attack detection systems. The results indicate that while LR is well-suited to structured, large datasets, SVM may offer better generalization capabilities in diverse and less predictable environments.

For future work, we recommend testing the models on real-time virtual environments created using tools such as Mininet. This would greatly help in identifying and subsequently addressing any setbacks that the models may encounter when deployed in a real-world production environment. Additionally, we recommend the integration of security controls to prevent the possibility of model poisoning attacks against our models.

## References

- [1] Abubakar, R., Aldegheishem, A., Majeed, M. F., Mehmood, A., Maryam, H., Alrajeh, N. A., ... & Jawad, M. (2020). An effective mechanism to mitigate real-time DDoS attack. *IEEE Access*, 8, 126215-126227.
- [2] Awan, M. J., Farooq, U., Babar, H. M. A., Yasin, A., Nobanee, H., Hussain, M., ... & Zain, A. M. (2021). Real-time DDoS attack detection system using big data approach. *Sustainability*, 13(19), 10743.
- [3] Karthikeyan, H., & Usha, G. (2022). Real-time DDoS flooding attack detection in intelligent transportation systems. *Computers and Electrical Engineering*, 101, 107995.
- [4] AGARWAL, S. (2022). Real-time DDoS Detection and Mitigation in Software Defined Networks using Machine Learning Techniques.
- [5] Shah, S. A. R., & Issac, B. (2018). Performance comparison of intrusion detection systems and application of machine learning to Snort system. *Future Generation Computer Systems*, 80, 157-170.
- [6] Yuan, X., Li, C., & Li, X. (2017, May). DeepDefense: identifying DDoS attack via deep learning. In *2017 IEEE international conference on smart computing (SMARTCOMP)* (pp. 1-8). IEEE.
- [7] Zhang, H., Dai, S., Li, Y., & Zhang, W. (2018, November). Real-time distributed-random-forest-based network intrusion detection system using Apache spark. In *2018 IEEE 37th international performance computing and communications conference (IPCCC)* (pp. 1-7). IEEE.
- [8] Ahmad, I., Basher, M., Iqbal, M. J., & Rahim, A. (2018). Performance comparison of support vector machine, random forest, and extreme learning machine for intrusion detection. *IEEE access*, 6, 33789-33795.
- [9] Banitalebi Dehkordi, A., Soltanaghaei, M., & Boroujeni, F. Z. (2021). The DDoS attacks detection through machine learning and statistical methods in SDN. *The Journal of Supercomputing*, 77, 2383-2415.
- [10] Liang, X., & Znati, T. (2019). On the performance of intelligent techniques for intensive and stealthy DDos detection. *Computer Networks*, 164, 106906.
- [11] Canadian Institute for Cybersecurity. (2019). Licensing and Redistribution of CICDDoS2019 Dataset. Retrieved from <https://www.unb.ca/cic/datasets/ddos-2019.html>.
- [12] Akgun, D., Hizal, S., & Cavusoglu, U. (2022). A new DDoS attacks intrusion detection model based on deep learning for cybersecurity. *Computers & Security*, 118, 102748.
- [13] Doriguzzi-Corin, R., Millar, S., Scott-Hayward, S., Martinez-del-Rincon, J., & Siracusa, D. (2020). LUCID: A practical, lightweight deep learning solution for DDoS attack detection. *IEEE Transactions on Network and Service Management*, 17(2), 876-889.