TABLE OF CONTENTS

Serial No.	TITLE	PAGE NO.
1	**Introduction**	5-8
1.1	Overview	5
1.2	Motivation	7
2	**Software Requirement Specification**	9
2.1	Hardware Requirements	9
2.2	Software Requirements	9
3	**Design**	10-11
4	**Implementation**	12-13
5	**Results**	14
6	**Conclusion**	15-16
7	**References**	17

LIST OF FIGURES:

Serial No.	TITLE	PAGE NO.
1	**Working of the model**	13
2	**Log Monitoring**	18
3	**Attacker(Kali), Metasploitable(Victim), Ubuntu Server(Log server)**	18
4	**Working of Suricata model**	13
5	**Deployment of the NIDS Model**	18
6	**Source/destination address config.**	19
7	**Flowchart**	21

INTRODUCTION

1.1 Overview

With the rapid expansion of network infrastructure and an increasing volume of cyber threats, organizations face challenges in safeguarding sensitive information from unauthorized access and malicious attacks. A robust Network Intrusion Detection System (NIDS) is essential for identifying and mitigating these threats.

This project focuses on developing a **hybrid Network Intrusion Detection System (NIDS)** that combines the real-time packet analysis capabilities of **Suricata**, an open-source IDS/IPS engine, with the predictive power of **Machine Learning (ML)** algorithms. By leveraging Suricata's rule-based detection system and integrating it with ML-based anomaly detection, the proposed system aims to improve detection accuracy, reduce false positives, and adapt to evolving cyber threats.

The **Suricata component** of the system monitors network traffic, applying predefined rules to detect known attack patterns and generate alerts. In parallel, the **Machine Learning module** is trained on labeled network traffic datasets to identify anomalous behaviors that may indicate previously unknown or sophisticated attacks. The integration of these two approaches results in a hybrid system capable of addressing both signature-based and anomaly-based detection limitations.

This project also emphasizes the practical implementation of the system in a controlled network environment, showcasing its ability to detect a wide range of threats, such as Distributed Denial-of-Service (DDoS) attacks, malware infiltration, and unauthorized access attempts. Performance metrics, including detection rate, false-positive rate, and system efficiency, are evaluated to validate the effectiveness of the proposed hybrid solution.

By merging the strengths of Suricata and Machine Learning, this project contributes to the development of a more adaptive, efficient, and scalable intrusion detection system, addressing the growing need for proactive network security measures in modern computing environments.

Benefits:

- **Enhanced Detection Accuracy**

Combines Suricata's rule-based detection for known threats with Machine Learning's ability to identify anomalies, enabling comprehensive threat detection.

- **Reduced False Positives**

Machine Learning algorithms help differentiate between benign and malicious traffic, reducing the number of false alerts typically generated by signature-based systems.

- **Adaptability to Evolving Threats**

Machine Learning models can be trained on new datasets, enabling the system to adapt to emerging and zero-day attacks that rule-based systems might miss.

- **Real-Time Traffic Analysis**

Suricata's high-speed packet inspection ensures immediate detection and alerting for known threats, maintaining the system's responsiveness.

- **Scalability**

The hybrid approach can scale to handle large and complex network infrastructures, making it suitable for modern enterprise environments.

- **Cost-Effectiveness**

By integrating open-source tools like Suricata with customizable ML models, organizations can build a powerful IDS without incurring high software licensing costs.

- **Comprehensive Threat Coverage**

The combination of signature-based detection (Suricata) and anomaly-based detection (Machine Learning) provides coverage for a wider range of cyber threats.

- **Continuous Improvement**

Machine Learning models improve over time as they are retrained on updated traffic patterns and threat data, increasing the system's effectiveness.

- **Customizability**

The system allows for fine-tuning of Suricata rules and Machine Learning algorithms to meet the specific needs of different network environments.

- **Efficient Resource Utilization**

The hybrid system balances the workload between Suricata's lightweight rule-based processing and the ML model's computational power, optimizing overall performance.

- **Improved Incident Response**

By providing detailed alerts and insights, the system aids in faster and more effective responses to security incidents.

- **Future-Proof Security**

The hybrid model ensures the system remains effective as new attack techniques and tools emerge, addressing long-term network security challenges.

Challenges

- **Data Quality and Availability**

  Obtaining high-quality, labeled datasets for training Machine Learning models can be difficult, especially for detecting new and rare attack patterns.

  Ensuring that training data is representative of real-world traffic is critical but challenging.

- **High Computational Requirements**

  Machine Learning models, especially deep learning algorithms, can be resource-intensive, requiring significant computational power and memory.

  Real-time traffic analysis by Suricata combined with Machine Learning processing can strain system resources.

- **Integration Complexity**

  Combining Suricata's rule-based engine with a Machine Learning module requires seamless integration, which can be technically challenging.

  Ensuring smooth communication between the two components without bottlenecks is critical.

- **Model Training and Maintenance**

  ML models need regular retraining with updated datasets to remain effective, which can be time-consuming and require domain expertise.

  Improperly tuned models may lead to overfitting or underfitting, reducing detection accuracy.

- **False Positives and Negatives**

  While the hybrid approach reduces false positives, achieving an optimal balance between sensitivity and specificity remains challenging.

  Complex attack patterns or sophisticated adversarial techniques may still evade detection.

- **Real-Time Processing Overhead**

  The need to process high volumes of network traffic in real-time can create performance bottlenecks, especially in large-scale networks.

  Implementing efficient algorithms and load balancing is necessary to maintain speed and reliability.

- **Rule Management in Suricata**

  Maintaining and updating Suricata rules to keep up with new threats requires regular monitoring and expertise.

  Excessive or poorly written rules can lead to performance degradation.

- **Deployment in Diverse Network Environments**

  Adapting the system to different network architectures and traffic patterns can be challenging.

  Variations in network configurations and protocols may require extensive customization.

- **Security of the System Itself**

  The hybrid system must be protected against attacks targeting the IDS itself, such as evasion techniques or adversarial attacks on the Machine Learning model.

- **Resource Allocation and Cost**

  While open-source tools like Suricata reduce costs, deploying and maintaining a hybrid system with ML capabilities may still require significant investment in hardware and expertise.

- **Limited Explainability**

  Machine Learning models, especially complex ones like deep learning, can act as a "black box," making it difficult to interpret their decisions.

  This lack of explainability can hinder troubleshooting and trust in the system.

- **Scalability Challenges**

  Scaling the system to handle increasing network traffic and new attack vectors requires careful design and resource planning.

- **Dependence on Expertise**

  Implementing, maintaining, and tuning the system requires skilled personnel with expertise in cybersecurity, Machine Learning, and network engineering.

Motivation

In the digital era, organizations increasingly rely on networked systems to manage critical operations and store sensitive information. However, the growing dependence on these systems has led to a corresponding increase in sophisticated cyber threats, such as ransomware, advanced persistent threats (APTs), Distributed Denial-of-Service (DDoS) attacks, and zero-day exploits.

Traditional Network Intrusion Detection Systems (NIDS), primarily relying on rule-based methods, are effective in detecting known threats but often fail to identify novel or evolving attack patterns. Furthermore, these systems can generate a high number of false positives, overwhelming security teams and reducing overall efficiency. Anomaly-based approaches, such as those driven by Machine Learning, provide a promising solution by identifying deviations from normal traffic patterns. However, they face challenges such as computational overhead and difficulties in real-time implementation.

This project is motivated by the need for an enhanced and comprehensive solution that combines the best of both worlds: the proven capabilities of rule-based detection (using Suricata) and the adaptability of Machine Learning-based anomaly detection. A hybrid system leveraging these technologies offers several advantages:

1. **Proactive Defense:**
    - Detect both known and unknown threats, addressing limitations of purely rule-based systems.
2. **Reduction in False Alerts:**
    - Improve detection accuracy and reduce false positives, making the system more reliable and manageable for security teams.
3. **Adaptability to Emerging Threats:**
    - Leverage Machine Learning to dynamically learn and adapt to new attack patterns, ensuring the system remains effective against zero-day attacks.
4. **Real-Time Threat Mitigation:**

- Use Suricata's fast packet inspection and alerting capabilities for immediate action, while Machine Learning models analyze complex patterns.

5. **Cost-Effective Security:**

   - Combine open-source tools and tailored algorithms to create a solution accessible to organizations with limited resources.

The hybrid NIDS system aligns with the increasing demand for scalable, adaptive, and efficient cybersecurity solutions in modern network environments. It addresses the challenges of traditional methods while paving the way for future advancements in network security. This project is inspired by the vision of enhancing organizational resilience against cyber threats and contributing to the ongoing evolution of intelligent intrusion detection systems.

Hardware Requirements

1. **Host Machine Specifications**:
   - **Processor (CPU)**:
     - Multi-core processor with virtualization support (e.g., Intel VT-x or AMD-V).
     - Recommended: Quad-core or higher for running multiple VMs.
   - **Memory (RAM)**:
     - Minimum: 8 GB
     - Recommended: 16 GB or more (especially if running multiple VMs or heavy traffic simulations).
   - **Storage**:
     - Minimum: 50 GB of free disk space.
     - Recommended: SSD with at least 100 GB for better performance.
   - **Network Interface Card (NIC)**:
     - At least one NIC.
     - For advanced simulations, multiple NICs can be used (e.g., for mirroring traffic).
   - **Graphics**:
     - Not a critical requirement unless you use GUI tools extensively.
   - **Power Supply**:
     - Ensure a stable power supply for uninterrupted simulations.

2. **VM Network Configuration**:
   - **Virtual Switch**:
     - Required for simulating network traffic between VMs.
   - **Network Modes**:
     - Use **Bridged** or **NAT** for external traffic simulation.
     - Use **Host-Only** for isolated traffic simulation.

Software Requirements

1. Virtualization Software

- **Purpose**: Create and manage virtual machines.
- **Options**:
  - VMware Workstation/Player
  - Oracle VirtualBox (free and widely supported)
  - Microsoft Hyper-V (built into Windows Pro/Enterprise)
  - Linux KVM (Kernel-based Virtual Machine)

2. Operating System for VMs

- **Preferred OS**:
  - Lightweight Linux distributions to optimize resource usage:
    - **Ubuntu Server** (20.04 LTS or later)
    - **CentOS**/AlmaLinux/Rocky Linux
    - **Debian**
  - For specific use cases, Windows Server can also be used.

3. Intrusion Detection System Software

- **Options**:

- **Suricata**: Signature-based and anomaly detection, real-time traffic monitoring.
- **Snort**: Widely used open-source IDS/IPS.
- Bro/Zeek: For advanced network analysis and monitoring.

4. Supporting Tools

- **Testing and Simulation Tools**:
  - **Metasploit Framework**: For penetration testing and attack simulations.
  - **Hydra**: For brute force attack testing.
  - **Nmap**: For network scanning and attack simulation

Example VM Setup

1. **VM 1 (IDS and SIEM Host)**:
   - OS: Ubuntu Server
   - Software: Suricata ,Wazuh, Logstash, Elasticsearch, Kibana
   - Resources: 2 CPUs, 8GB RAM, 30 GB disk space
2. **VM 2 (Attacker)**:
   - OS: Kali Linux
   - Tools: Metasploit, Hydra, Nmap
   - Resources: 2 CPUs, 2 GB RAM, 20 GB disk space
3. **VM 3 (Victim)**:
   - OS: Windows 10 or Linux
   - Purpose: Simulate legitimate user activity.
   - Resources: 2 CPUs, 2 GB RAM, 20 GB disk space

3. DESIGN

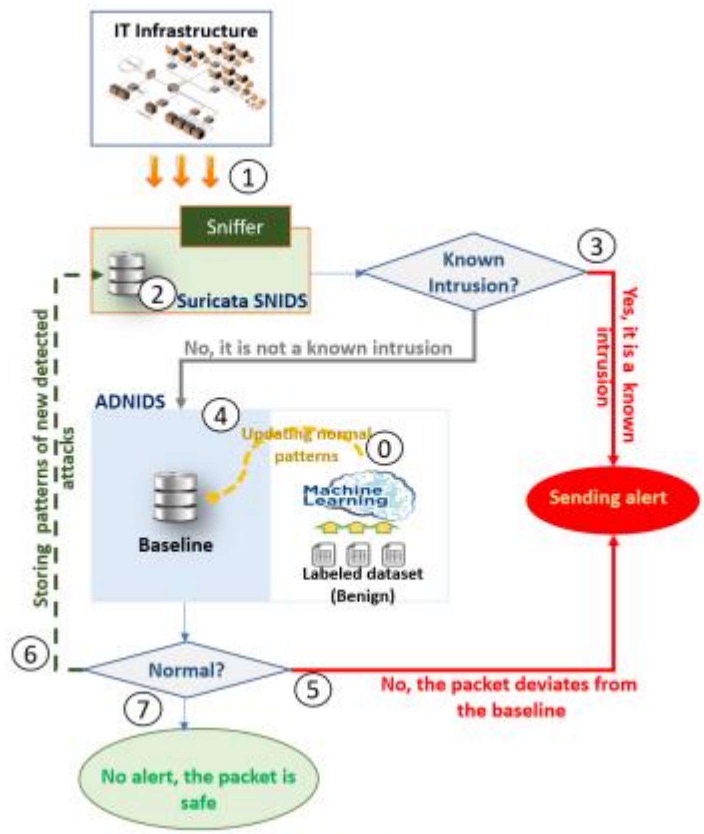


Fig. 3. The operating principle of the proposed NIDS

Fig. 1 Working of the model

4. IMPLEMENTATION

Here's an updated implementation for Installing and Configuring Suricata IDS on Ubuntu using the details from your PDF and replacing Wazuh with new tools for integration and monitoring, such as Elasticsearch, Kibana, and Filebeat. This setup ensures effective log aggregation, visualization, and monitoring.

Step 1: Install Suricata on Ubuntu

Install Suricata IDS on the Ubuntu endpoint using the following commands:

sudo add-apt-repository ppa:oisf/suricata-stable

sudo apt-get update

sudo apt-get install suricata -y

- **Purpose: This installs the latest stable version of Suricata, a high-performance IDS/IPS.**

Step 2: Install the Emerging Threat Rule Set

The Emerging Threats ruleset enhances Suricata by providing prebuilt signatures for detecting threats.

cd /tmp/

curl -LO https://rules.emergingthreats.net/open/suricata-6.0.8/emerging.rules.tar.gz

sudo tar -xvzf emerging.rules.tar.gz

sudo mv rules/*.rules /etc/suricata/rules/

sudo chmod 640 /etc/suricata/rules/*.rules

- **Why: These rules help detect common attack patterns and enhance Suricata's detection capabilities.**

Step 3: Update Suricata Configuration

Edit the Suricata YAML configuration file to define network interfaces, paths, and log settings.

sudo nano /etc/suricata/suricata.yaml

Update the following fields:

HOME_NET: "<YOUR_NETWORK_IP>"

EXTERNAL_NET: "any"

default-rule-path: /etc/suricata/rules

rule-files:

- "*.rules"

stats:

enabled: yes

af-packet:

- interface: eth0

- **HOME_NET: Replace <YOUR_NETWORK_IP> with your local network range (e.g., 192.168.1.0/24).**
- **af-packet: Set the interface (eth0 or similar) to monitor traffic.**

Step 4: Restart Suricata Service

Apply the configuration changes by restarting the Suricata service:

sudo systemctl restart suricata

Step 5: Integrate Suricata with Elasticsearch, Kibana, and Filebeat

Instead of using Wazuh, we'll integrate Filebeat to forward Suricata logs to Elasticsearch for indexing and use Kibana for visualization.

A. Install the Elastic Stack

1. **Add the Elastic GPG key:**
2. **curl -fsSL https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -**
3. **Add the Elastic repository:**
4. **echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-7.x.list**
5. **sudo apt-get update**
6. **Install Elasticsearch, Kibana, and Filebeat:**
7. **sudo apt-get install elasticsearch kibana filebeat -y**

B. Configure Filebeat

1. **Edit Filebeat configuration:**
2. **sudo nano /etc/filebeat/filebeat.yml**
3. **Add the following configuration for Suricata logs:**
4. **filebeat.inputs:**
5. **- type: filestream**

6. **enabled: true**
7. **paths:**
8. **- /var/log/suricata/eve.json**
9. **json.keys_under_root: true**
10. **json.add_error_key: true**
11.
12. **setup.template.settings:**
13. **index.number_of_shards: 1**
14.
15. **output.elasticsearch:**
16. **hosts: ["http://localhost:9200"]**
17. **Enable the Suricata Filebeat module:**
18. **sudo filebeat modules enable suricata**
19. **Load the index template and dashboards for Kibana:**
20. **sudo filebeat setup**

C. Start Filebeat

Start and enable the Filebeat service to send logs to Elasticsearch:

sudo systemctl start filebeat

sudo systemctl enable filebeat

Step 6: Access and Visualize Logs in Kibana

1. **Start Elasticsearch and Kibana:**
2. **sudo systemctl start elasticsearch**
3. **sudo systemctl start kibana**
4. **Open Kibana in your web browser at http://<server-ip>:5601.**
5. **Go to Discover to view Suricata logs or use prebuilt dashboards under the Filebeat Suricata module.**

Step 7: Monitor Alerts in Real-Time

- **Use Kibana dashboards to monitor:**
  - **Alerts and events generated by Suricata.**
  - **Network activity patterns based on Suricata's eve.json logs.**
- **Optionally, configure alerts in Kibana to notify administrators of critical events.**

RESULTS

Testing

To test our Suricata IDS against abnormal traffic. We will initiate Nmap SYN scan from Kali Linux to our Ubuntu server(running Wazuh + Suricata IDS). This can be accomplished using the below steps.

Step1: Launch SYN Scan

Access your Kali Linux and type Nmap SYN Scan(-sS) as shown below

$ nmap -sS -Pn 192.168.29.172

Step2: Check the output

With the output shown below, you can see the Status as Open, meaning the TCP port 22 is opened on the server side.

nmap -sS -Pn 192.168.29.246

Starting Nmap 7.94 (https://nmap.org) at 2023-12-11 00:33 IST

Nmap scan report for 192.168.29.246

Host is up (0.0030s latency).

Not shown: 999 closed tcp ports (reset)

PORT   STATE SERVICE

22/tcp open ssh

Nmap done: 1 IP address (1 host up) scanned in 1.50 seconds

Visualize the Alert

To view the security alerts, navigate to Security alerts module and then select agent.

You can apply filter ***rule.groups:suricata***

Fig 2. Log Monitoring

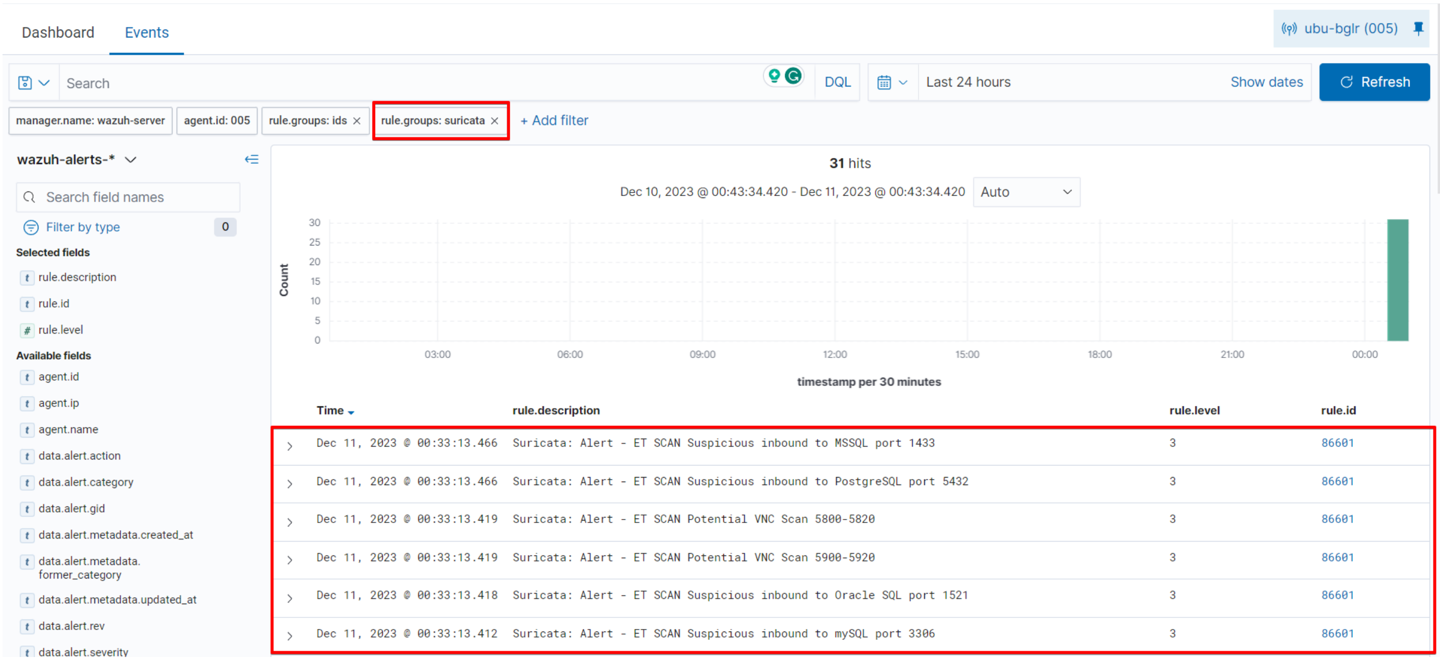
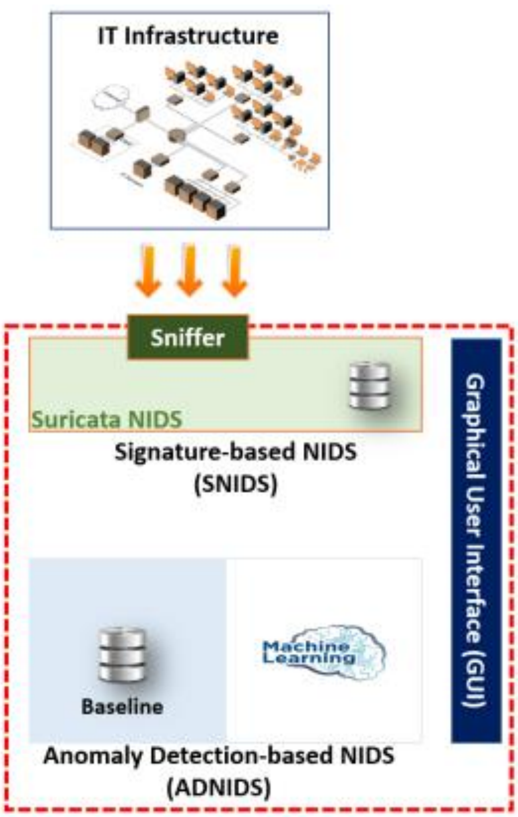


Fig3. Working of suricata and ml



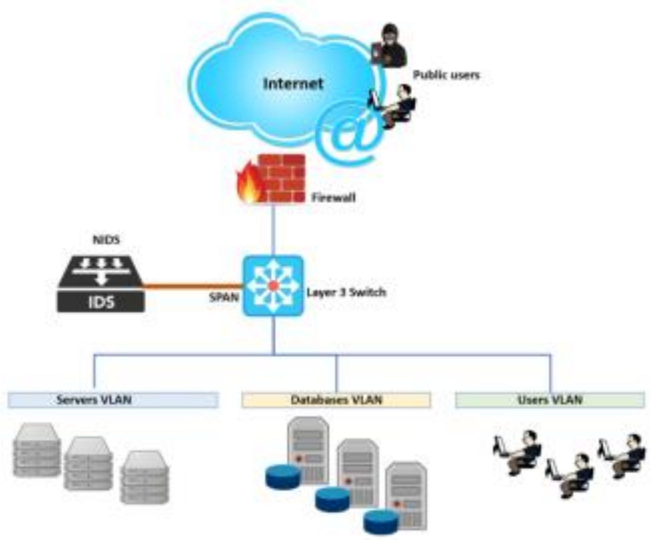
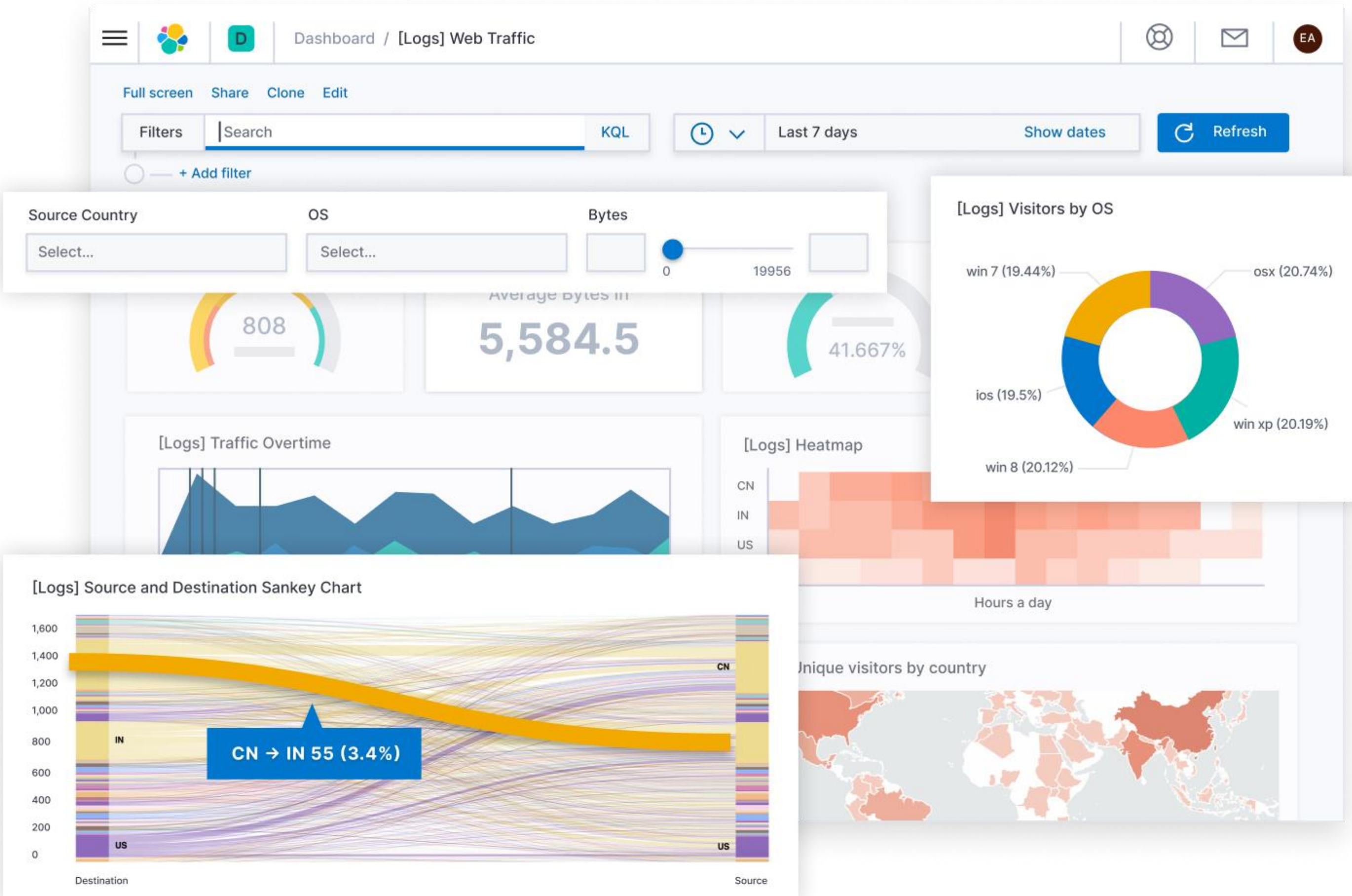


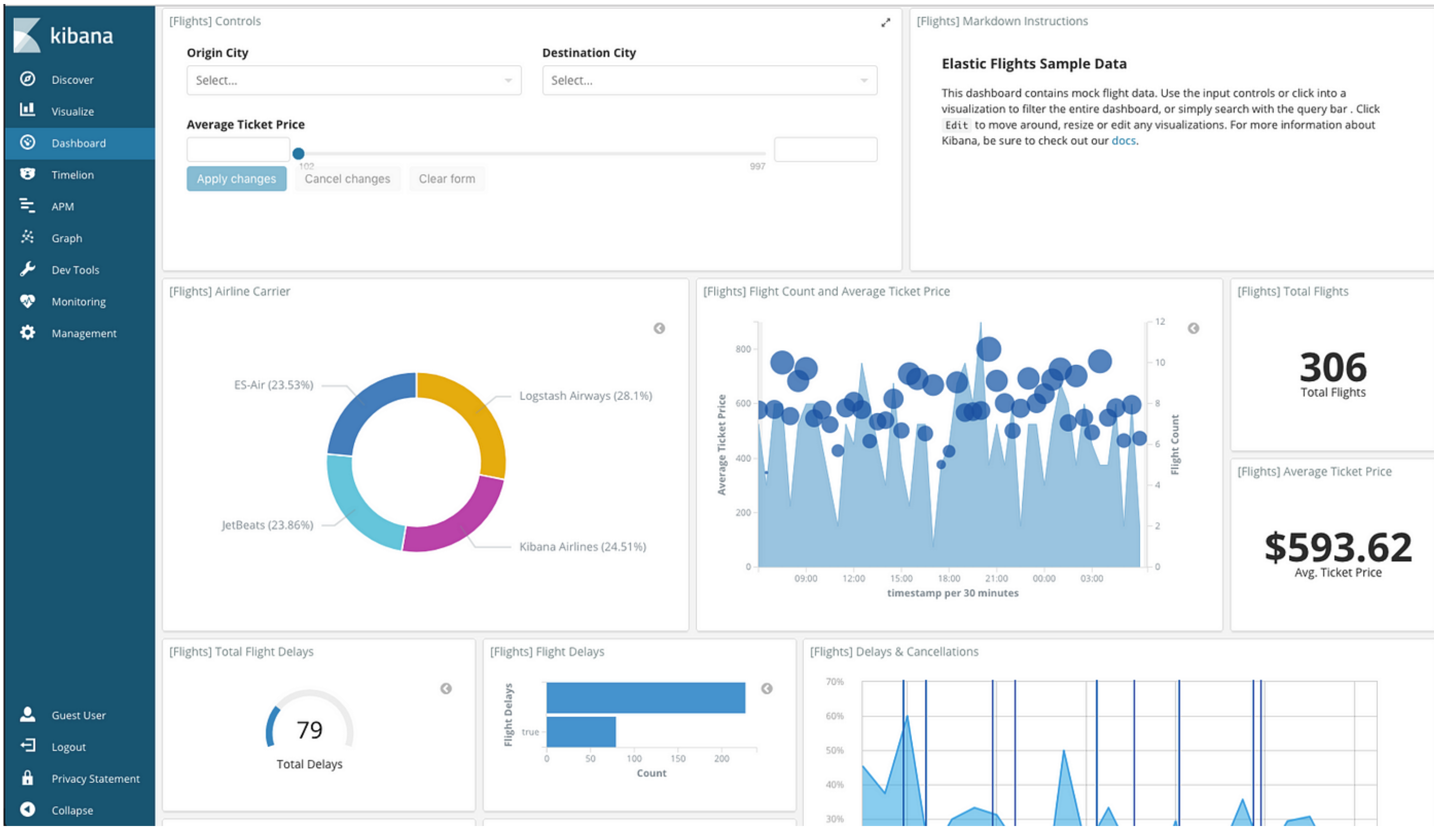
Fig. 4. Deployment mode of the proposed model

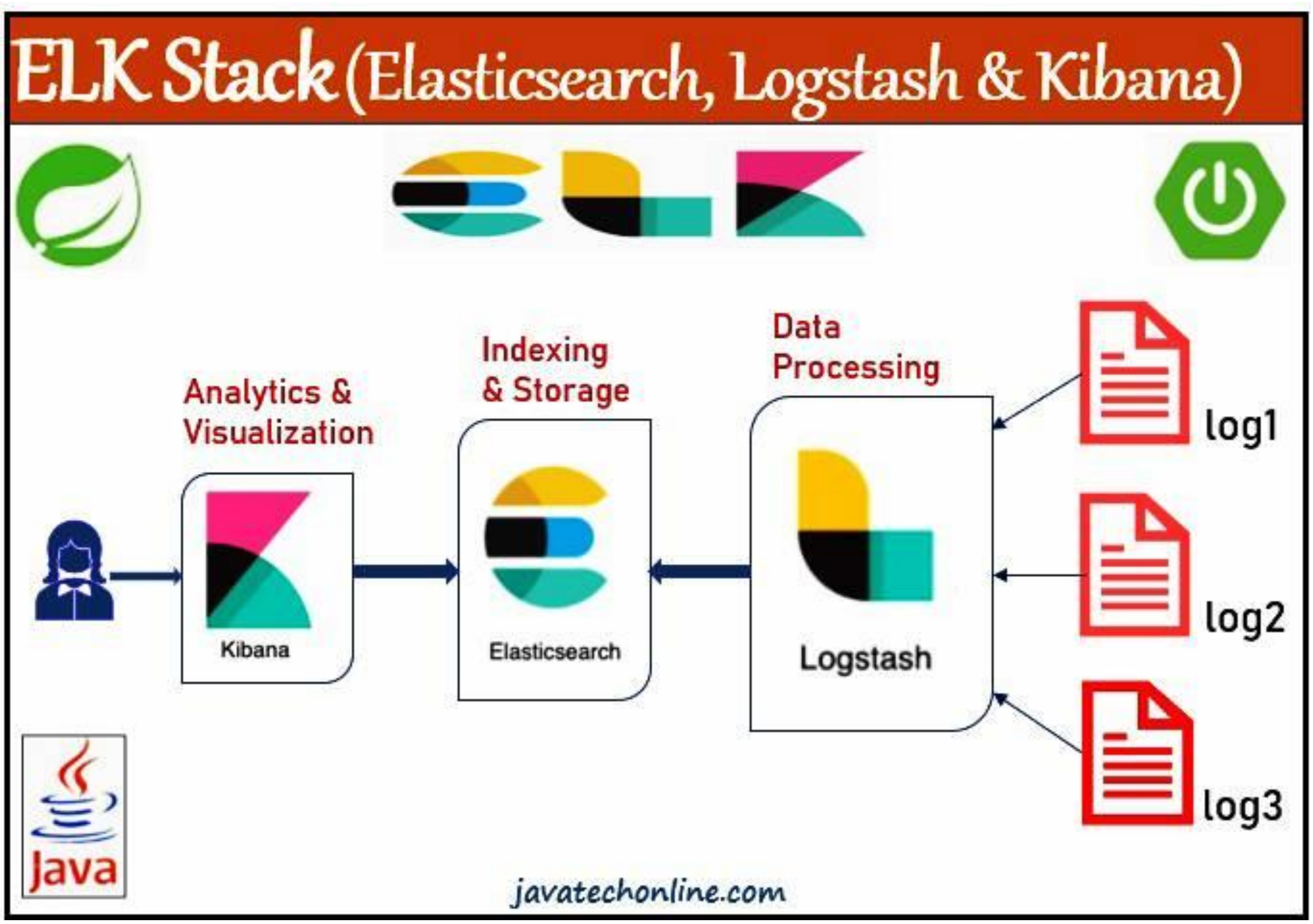
Table 2. The retained attributes of the CICIDS2017 dataset

Selected features	Selected features	Selected features
Destination Port	Flow IAT Mean	FIN Flag Count
Flow Duration	Flow IAT Min	PSH Flag Count
Total Fwd Packets	Fwd IAT Min	ACK Flag Count
Total Length of Fwd Packets	Bwd IAT Total	URG Flag Count
Fwd Packet Length Max	Bwd IAT Mean	Down/Up Ratio
Fwd Packet Length Min	Bwd IAT Std	Init_Win_bytes_forward
Fwd Packet Length Mean	Fwd PSH Flags	Init_Win_bytes_backward
Bwd Packet Length Max	Fwd Header Length	Active Mean
Bwd Packet Length Min	Bwd Header Length	Active Std
Flow Bytes/s	Bwd Packets/s	Idle Std
Flow Packets/s	Min Packet Length	Label

Fig. 5 Elasticsearch interface




kibana
Discover
Visualize
Dashboard
Timelion
APM
Graph
Dev Tools
Monitoring
Management
Guest User
Logout
Privacy Statement
Collapse
[Flights] Controls
Origin City
Destination City
Average Ticket Price
Apply changes
Cancel changes
Clear form
[Flights] Markdown Instructions
Elastic Flights Sample Data
This dashboard contains mock flight data. Use the input controls or click into a visualization to filter the entire dashboard, or simply search with the query bar . Click Edit to move around, resize or edit any visualizations. For more information about Kibana, be sure to check out our docs.
[Flights] Airline Carrier
ES-Air (23.53%)
Logstash Airways (28.1%)
JetBeats (23.86%)
Kibana Airlines (24.51%)
[Flights] Flight Count and Average Ticket Price
Average Ticket Price
Flight Count
timestamp per 30 minutes
[Flights] Total Flights
306
Total Flights
[Flights] Average Ticket Price
$593.62
Avg. Ticket Price
[Flights] Total Flight Delays
79
Total Delays
[Flights] Flight Delays
Flight Delays
Count
[Flights] Delays & Cancellations


ELK Stack (Elasticsearch, Logstash & Kibana)
Analytics & Visualization
Indexing & Storage
Data Processing
Kibana
Elasticsearch
Logstash
log1
log2
log3
Java
javatechonline.com

ML Code implementation:

```
import numpy as np
import pandas as pd
import seaborn as sns
import missingno as msno
sns.set(style='darkgrid')
import matplotlib.pyplot as plt
import pickle

data1 = pd.read_csv('/content/Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv')
data2 = pd.read_csv('/content/Tuesday-WorkingHours.pcap_ISCX.csv')
data3 = pd.read_csv('/content/Wednesday-workingHours.pcap_ISCX.csv')
data4 = pd.read_csv('/content/Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv')
data5 = pd.read_csv('/content/Thursday-WorkingHours-Afternoon-
Infilteration.pcap_ISCX.csv')
data6 = pd.read_csv('/content/Friday-WorkingHours-Morning.pcap_ISCX.csv')
data7 = pd.read_csv('/content/Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv')
data8 = pd.read_csv('/content/Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv')

data_list = [data1, data2, data3, data4, data5, data6, data7, data8]

data = pd.concat(data_list)
rows, cols = data.shape



# Deleting dataframes after concating to save memory
for d in data_list: del d
```

```
# Renaming the columns by removing leading/trailing whitespace
col_names = {col: col.strip() for col in data.columns}
data.rename(columns = col_names, inplace = True)
data.columns

dups = data[data.duplicated()]
print(f'Number of duplicates: {len(dups)}')

data.drop_duplicates(inplace = True)
data.shape
missing_val = data.isna().sum()
print(missing_val.loc[missing_val > 0])

# Checking for infinity values
numeric_cols = data.select_dtypes(include = np.number).columns
inf_count = np.isinf(data[numeric_cols]).sum()

# Replacing any infinite values (positive or negative) with NaN (not a number)
data.replace([np.inf, -np.inf], np.nan, inplace = True)
missing = data.isna().sum()
print(missing.loc[missing > 0])

# Calculating missing value percentage in the dataset
mis_per = (missing / len(data)) * 100
mis_table = pd.concat([missing, mis_per.round(2)], axis = 1)
mis_table = mis_table.rename(columns = {0 : 'Missing Values', 1 : 'Percentage of Total
Values'})
```

```
print(mis_table.loc[mis_per > 0])

med_flow_bytes = data['Flow Bytes/s'].median()
med_flow_packets = data['Flow Packets/s'].median()



# Filling missing values with median
data['Flow Bytes/s'].fillna(med_flow_bytes, inplace = True)
data['Flow Packets/s'].fillna(med_flow_packets, inplace = True)


data['Label'].unique()
# Creating a dictionary that maps each label to its attack type
attack_map = {
    'BENIGN': 'BENIGN',
    'DDoS': 'DDoS',
    'DoS Hulk': 'DoS',
    'DoS GoldenEye': 'DoS',
    'DoS slowloris': 'DoS',
    'DoS Slowhttptest': 'DoS',
    'PortScan': 'Port Scan',
    'FTP-Patator': 'Brute Force',
    'SSH-Patator': 'Brute Force',
    'Bot': 'Bot',
    'Web Attack   Brute Force': 'Web Attack',
    'Web Attack   XSS': 'Web Attack',
    'Web Attack   Sql Injection': 'Web Attack',
```

```
    'Infiltration': 'Infiltration',
    'Heartbleed': 'Heartbleed'
}
# Creating a new column 'Attack Type' in the DataFrame based on the attack_map dictionary
data['Attack Type'] = data['Label'].map(attack_map)
data['Attack Type'].value_counts()

data.drop('Label', axis = 1, inplace = True)

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
data['Attack Number'] = le.fit_transform(data['Attack Type'])
print(data['Attack Number'].unique())
# Printing corresponding attack type for each encoded value
encoded_values = data['Attack Number'].unique()
for val in sorted(encoded_values):
    print(f"{val}: {le.inverse_transform([val])[0]}")

corr = data.corr(numeric_only = True).round(2)
corr.style.background_gradient(cmap = 'coolwarm', axis = None).format(precision = 2)

# Positive correlation features for 'Attack Number'
pos_corr_features = corr['Attack Number'][(corr['Attack Number'] > 0) & (corr['Attack
Number'] < 1)].index.tolist()

print("Features with positive correlation with 'Attack Number':\n")
for i, feature in enumerate(pos_corr_features, start = 1):
    corr_value = corr.loc[feature, 'Attack Number']
```

```
    print('{:<3} {:<24} :{}'.format(f'{i}.', feature, corr_value))

print(f'Number of considerable important features: {len(pos_corr_features)}')

# Checking for columns with zero standard deviation (the blank squares in the heatmap)
std = data.std(numeric_only = True)
zero_std_cols = std[std == 0].index.tolist()
zero_std_cols
 # Data sampling for data analysis
sample_size = int(0.2 * len(data)) # 20% of the original size
sampled_data = data.sample(n = sample_size, replace = False, random_state = 0)
sampled_data.shape

data.drop('Attack Number', axis = 1, inplace = True)

# Identifying outliers
numeric_data = sampled_data.select_dtypes(include = ['float', 'int'])
q1 = numeric_data.quantile(0.25)
q3 = numeric_data.quantile(0.75)
iqr = q3 - q1
outlier = (numeric_data < (q1 - 1.5 * iqr)) | (numeric_data > (q3 + 1.5 * iqr))
outlier_count = outlier.sum()
outlier_percentage = round(outlier.mean() * 100, 2)
outlier_stats = pd.concat([outlier_count, outlier_percentage], axis = 1)
outlier_stats.columns = ['Outlier Count', 'Outlier Percentage']

print(outlier_stats)
```

```
# Identifying outliers based on attack type
outlier_counts = {}
for i in numeric_data:
    for attack_type in sampled_data['Attack Type'].unique():
        attack_data = sampled_data[i][sampled_data['Attack Type'] == attack_type]
        q1, q3 = np.percentile(attack_data, [25, 75])
        iqr = q3 - q1
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr
        num_outliers = ((attack_data < lower_bound) | (attack_data > upper_bound)).sum()
        outlier_percent = num_outliers / len(attack_data) * 100
        outlier_counts[(i, attack_type)] = (num_outliers, outlier_percent)

for i in numeric_data:
  print(f'Feature: {i}')
  for attack_type in sampled_data['Attack Type'].unique():
    num_outliers, outlier_percent = outlier_counts[(i, attack_type)]
    print(f'- {attack_type}: {num_outliers} ({outlier_percent:.2f}%)')
  print()

data.groupby('Attack Type').first()

old_memory_usage = data.memory_usage().sum() / 1024 ** 2
print(f'Initial memory usage: {old_memory_usage:.2f} MB')
for col in data.columns:
    col_type = data[col].dtype
    if col_type != object:
        c_min = data[col].min()
```

```
        c_max = data[col].max()
        # Downcasting float64 to float32
        if str(col_type).find('float') >= 0 and c_min > np.finfo(np.float32).min and c_max <
np.finfo(np.float32).max:
            data[col] = data[col].astype(np.float32)

        # Downcasting int64 to int32
        elif str(col_type).find('int') >= 0 and c_min > np.iinfo(np.int32).min and c_max <
np.iinfo(np.int32).max:
            data[col] = data[col].astype(np.int32)

data.info()

# Dropping columns with only one unique value
num_unique = data.nunique()
one_variable = num_unique[num_unique == 1]
not_one_variable = num_unique[num_unique > 1].index

dropped_cols = one_variable.index
data = data[not_one_variable]

# Standardizing the dataset
from sklearn.preprocessing import StandardScaler

features = data.drop('Attack Type', axis = 1)
attacks = data['Attack Type']

scaler = StandardScaler()
```

```
scaled_features = scaler.fit_transform(features)
with open('scaler.pkl', 'wb') as scaler_file:
    pickle.dump(scaler, scaler_file)

from sklearn.decomposition import IncrementalPCA

size = len(features.columns) // 2
ipca = IncrementalPCA(n_components = size, batch_size = 500)
for batch in np.array_split(scaled_features, len(features) // 500):
    ipca.partial_fit(batch)



transformed_features = ipca.transform(scaled_features)
new_data = pd.DataFrame(transformed_features, columns = [f'PC{i+1}' for i in range(size)])
new_data['Attack Type'] = attacks.values



# Creating a balanced dataset for Binary Classification
normal_traffic = new_data.loc[new_data['Attack Type'] == 'BENIGN']
intrusions = new_data.loc[new_data['Attack Type'] != 'BENIGN']

normal_traffic = normal_traffic.sample(n = len(intrusions), replace = False)

ids_data = pd.concat([intrusions, normal_traffic])
ids_data['Attack Type'] = np.where((ids_data['Attack Type'] == 'BENIGN'), 0, 1)
bc_data = ids_data.sample(n = 15000)

# Splitting the data into features (X) and target (y)
```

```
from sklearn.model_selection import train_test_split

X_bc = bc_data.drop('Attack Type', axis = 1)
y_bc = bc_data['Attack Type']

X_train_bc, X_test_bc, y_train_bc, y_test_bc = train_test_split(X_bc, y_bc, test_size = 0.25,
random_state = 0)

new_data['Attack Type'].value_counts()

class_counts = new_data['Attack Type'].value_counts()
selected_classes = class_counts[class_counts > 1950]
class_names = selected_classes.index
selected = new_data[new_data['Attack Type'].isin(class_names)]

dfs = []
for name in class_names:
  df = selected[selected['Attack Type'] == name]
  if len(df) > 2500:
    df = df.sample(n = 5000, random_state = 0)

  dfs.append(df)

df = pd.concat(dfs, ignore_index = True)
df['Attack Type'].value_counts()


from imblearn.over_sampling import SMOTE
```

```
X = df.drop('Attack Type', axis=1)
y = df['Attack Type']

smote = SMOTE(sampling_strategy='auto', random_state=0)
X_upsampled, y_upsampled = smote.fit_resample(X, y)

blnc_data = pd.DataFrame(X_upsampled)
blnc_data['Attack Type'] = y_upsampled
blnc_data = blnc_data.sample(frac=1)

blnc_data['Attack Type'].value_counts()

features = blnc_data.drop('Attack Type', axis = 1)
labels = blnc_data['Attack Type']

X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size = 0.25, random_state
= 0)

from sklearn.tree import DecisionTreeClassifier
# For cross validation
from sklearn.model_selection import cross_val_score
dt1 = DecisionTreeClassifier(max_depth = 6)
dt1.fit(X_train, y_train)

cv_dt1 = cross_val_score(dt1, X_train, y_train, cv = 5)
print('Decision Tree Model 1')
print(f'\nCross-validation scores:', ', '.join(map(str, cv_dt1)))
```

```
print(f'\nMean cross-validation score: {cv_dt1.mean():.2f}')

dt2 = DecisionTreeClassifier(max_depth = 8)
dt2.fit(X_train, y_train)

cv_dt2 = cross_val_score(dt2, X_train, y_train, cv = 5)
print('Decision Tree Model 2')
print(f'\nCross-validation scores:', ', '.join(map(str, cv_dt2)))
print(f'\nMean cross-validation score: {cv_dt2.mean():.2f}')

# Sampling and shuffling the balanced dataset
blnc_data = blnc_data.sample(frac=1.0, random_state=0).reset_index(drop=True)

# Splitting into features (X) and target (y)
X = blnc_data.drop('Attack Type', axis=1)
y = blnc_data['Attack Type']

# Splitting the data into train and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42,
stratify=y)

# Building a Classification Model (e.g., Random Forest)
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

clf = RandomForestClassifier(n_estimators=100, random_state=0)
```

```
clf.fit(X_train, y_train)
with open('random_forest_model.pkl', 'wb') as model_file:
    pickle.dump(clf, model_file)

# Predictions and evaluation
y_pred = clf.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

6. CONCLUSION

The proposed Suricata and Machine Learning-based Hybrid Network Intrusion Detection System (NIDS) demonstrates a robust approach to modern cybersecurity challenges. By integrating Suricata's signature-based detection with the anomaly-detection capabilities of Machine Learning algorithms, the system effectively addresses the limitations of traditional NIDS.

This hybrid model offers enhanced detection accuracy, reduced false positives, and the ability to adapt to emerging threats. The real-time traffic analysis provided by Suricata ensures timely alerts for known attacks, while the Machine Learning module enables proactive identification of previously unseen or sophisticated intrusion patterns.

The system has been designed to strike a balance between computational efficiency and accuracy, making it suitable for deployment in diverse network environments. Furthermore, its scalability, cost-effectiveness, and ability to evolve through regular updates to rules and models ensure its long-term viability as a security solution.

While this implementation has its challenges, such as integration complexity and the need for high-quality training data, the results indicate that a hybrid approach is a promising direction for advancing network security. Future work could explore enhancements like integrating deep learning techniques, improving explainability, and optimizing system performance in large-scale networks.

In conclusion, this project contributes to the development of intelligent, adaptive, and scalable intrusion detection systems, empowering organizations to strengthen their defense mechanisms against the ever-evolving landscape of cyber threats.

7. REFERENCES:

Research Papers and Articles:

1. IoT Security Challenges and Solutions for DDoS Attacks
   Author(s): S. Raza, T. Voigt, M. Jutvik
   Source: *IEEE Internet of Things Journal*
   URL: IEEE IoT Journal

2. Machine Learning for DDoS Attack Detection in IoT Networks: A Survey
   Author(s): H. HaddadPajouh, A. Dehghantanha, R. Khayami
   Source: *Computer Networks*
   URL: ScienceDirect

3. Efficient Machine Learning Techniques for Detecting DDoS Attacks in IoT Environments
   Author(s): A. Gupta, N. Gupta
   Source: *Journal of Network and Computer Applications*
   URL: Elsevier

4. A Lightweight DDoS Detection Framework for IoT Networks Using Machine Learning
   Author(s): A. Moustafa, E. Sitnikova
   Source: *Future Generation Computer Systems*
   URL: ScienceDirect

5. IoT in Healthcare: Machine Learning-Based DDoS Attack Detection
   Author(s): B. Farhan, S. Madani
   Source: *Sensors*
   URL: MDPI Sensors