

# AI Project

**Team:**

**Sara Atalla 2205094**

**Yehia Tarek 2205062**

**Shadwa Ahmed 2205026**

## 1. Explanation of the Attack and the Model Applied

The dataset represents network intrusion data containing features like protocol type, service type, bytes sent/received, packet count, and other characteristics. The target variable indicates whether the network traffic is malicious (attack) or normal (non-attack). Each row contains data points of network packets, with labels identifying normal or malicious activity.

### *Intrusion Detection System (IDS)*

The model implements a binary classification task using a custom neural network to classify incoming traffic as an "attack" or "non-attack". The primary steps in this model are:

#### 1. Preprocessing:

- a. **Shuffling and splitting:** Ensures unbiased data.
- b. **Encoding categorical features:** service, protocol\_type, and flag columns are encoded for machine learning compatibility.
- c. **Feature scaling:** Numerical features are standardized to ensure the neural network converges faster during training.
- d. **Data preparation:** The data is converted to PyTorch tensors for compatibility with the deep learning framework.

#### 2. Neural Network Architecture:

- a. A fully connected feed-forward network with three layers.
- b. Uses ReLU activations for intermediate layers and a Sigmoid function at the output for binary classification.
- c. Optimized with Adam and Binary Cross-Entropy Loss for gradient-based learning.

#### 3. Evaluation Metrics:

- a. The **accuracy** on both train and test sets is measured at each epoch to evaluate the model's predictive performance.
- b. Separate plots for **loss and accuracy** give visual insights into the learning process.

## 2. Mitigation

Mitigation strategies must aim to address the vulnerabilities exploited in detected attacks. For this project:

### 1. Detectable Attacks:

- a. Utilize the model's predictions to identify malicious patterns.
- b. Explore attack techniques classified using MITRE ATT&CK, if provided in the dataset.

### 2. Potential Mitigation Techniques:

- a. **Network Hardening:** Apply strict network rules, limit unnecessary services, and implement segmentation.
- b. **Behavioral Analysis:** Use logs from attacks to create behavioral patterns to flag suspicious activities proactively.
- c. **Signature-based Updates:** Regularly update IDS/IPS signatures against known vulnerabilities.

### 3. Real-World Application:

- a. Integrate the model into a broader intrusion detection system pipeline for real-time monitoring.
- b. For flagged events, automatically initiate scripts to isolate affected hosts or reroute traffic.
- c. Use the MITRE ATT&CK classifications to trace exploit paths for remediation and patching.

## **MITRE ATT&CK Analysis for Intrusion Detection (BONUS)**

### **1. Overview of Intrusion Techniques**

Intrusion techniques encompass methods adversaries employ to gain unauthorized access to systems, escalate privileges, and maintain persistence. Understanding these techniques is crucial for developing effective IDS models.

### **2. Relevant MITRE ATT&CK Techniques**

Based on dataset and IDS focus, the following MITRE ATT&CK techniques are pertinent:

- **T1071 - Application Layer Protocol:** Adversaries may use application layer protocols (e.g., HTTP, FTP) to communicate, potentially evading detection by blending with legitimate traffic.
- **T1070 - Indicator Removal on Host:** Techniques such as clearing logs or deleting artifacts can hinder detection efforts.
- **T1105 - Remote File Copy:** Transferring files to or from a compromised system can facilitate further exploitation.

### **3. Associated Adversary Groups**

common groups associated with intrusion techniques :

- **APT28 (Fancy Bear):** Known for leveraging application layer protocols for command and control.
- **APT29 (Cozy Bear):** Utilizes remote file copying to exfiltrate data.

## 4. Mitigation Strategies

To counteract these intrusion techniques, consider the following mitigations:

- **Application Layer Protocol Monitoring:** Implement network intrusion detection and prevention systems (IDS/IPS) that use network signatures to identify traffic for specific adversary malware.
- **Indicator Removal Prevention:** Employ host-based IDS/IPS to monitor and alert on suspicious activities, such as log clearing or unauthorized file modifications.
- **File Transfer Monitoring:** Utilize IDS/IPS to detect and alert on unauthorized file transfers, especially those involving sensitive data.

## 5. Conclusion

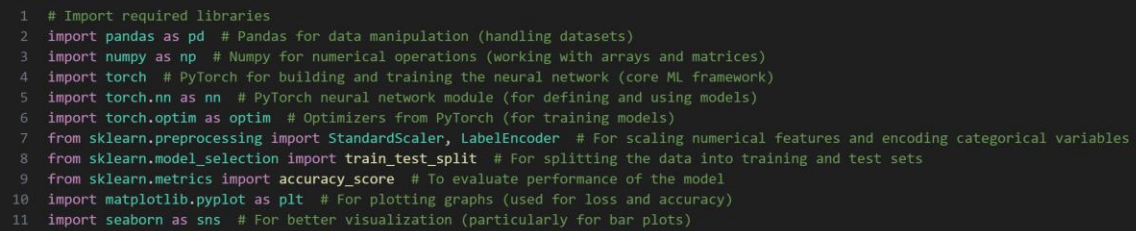
Integrating the MITRE ATT&CK framework into my IDS development process enhances the system's capability to detect and mitigate sophisticated intrusion techniques. By aligning detection strategies with known adversary behaviors, you can improve the effectiveness of your intrusion detection efforts.

### References

- **MITRE ATT&CK Framework:** [\[OBJ\]](#)
- **Network Intrusion Prevention, Mitigation M1031 - MITRE ATT&CK®:** [\[OBJ\]](#)

## 3. Code

### 1. Import Required Libraries

A terminal window with a dark background and light-colored text. It contains 11 lines of Python code for importing various libraries. The code is as follows:

```
1 # Import required libraries
2 import pandas as pd # Pandas for data manipulation (handling datasets)
3 import numpy as np # Numpy for numerical operations (working with arrays and matrices)
4 import torch # PyTorch for building and training the neural network (core ML framework)
5 import torch.nn as nn # PyTorch neural network module (for defining and using models)
6 import torch.optim as optim # Optimizers from PyTorch (for training models)
7 from sklearn.preprocessing import StandardScaler, LabelEncoder # For scaling numerical features and encoding categorical variables
8 from sklearn.model_selection import train_test_split # For splitting the data into training and test sets
9 from sklearn.metrics import accuracy_score # To evaluate performance of the model
10 import matplotlib.pyplot as plt # For plotting graphs (used for loss and accuracy)
11 import seaborn as sns # For better visualization (particularly for bar plots)
```

## 2. Load and Preprocess the Data

```
1 # Load the dataset containing intrusion detection data
2 df = pd.read_csv('intrusion_detection_data.csv') # Load the CSV file into a Pandas DataFrame
3 # Print the first few rows of the dataset to understand its structure
4 print(f"Sample data:\n{df.head()}") # Display the first five rows to preview the data
5
6 # Shuffle the data to avoid any bias caused by ordered rows
7 df = df.sample(frac=1, random_state=42).reset_index(drop=True) # Shuffle data randomly
8
```

## 3. Map Data to MITRE ATT&CK Techniques

```
1 # Define a function to map the features of the dataset to MITRE ATT&CK techniques
2 def map_to_mitre(row):
3     if row['attack'] == 1: # Check if the attack label is 1 (positive class for attack)
4         # Check different conditions for MITRE ATT&CK tactic and technique mappings
5         if row['protocol_type'] == 'udp' and row['service'] == 'ftp':
6             return "Tactic: Exfiltration, Technique: Exfiltration Over Alternative Protocol"
7         elif row['flag'] == 'FIN':
8             return "Tactic: Command and Control, Technique: Application Layer Protocol"
9         elif row['packet_loss'] > 0.01:
10            return "Tactic: Impact, Technique: Data Destruction"
11        elif row['packet_count'] > 1000 and row['total_bytes'] > 100000:
12            return "Tactic: Initial Access, Technique: Exploit Public-Facing Application"
13        else:
14            return "Tactic: Discovery, Technique: Network Service Scanning"
15    else:
16        return "Tactic: None, Technique: None" # If no attack, return no technique mapping
17
18 # Apply the function to each row of the dataset to create the 'mitre_attack' column
19 df['mitre_attack'] = df.apply(map_to_mitre, axis=1) # Map MITRE ATT&CK technique labels for all rows
20
```

## 4. Prepare Features and Split Data into Train/Test Sets

```
1 # Separate the features (X) and target variable (y)
2 X = df.drop(['attack', 'mitre_attack'], axis=1) # Drop attack and mitre_attack columns from features (X)
3 y = df['attack'] # Use the 'attack' column as the target variable (y)
4
5 # Split the dataset into train and test sets (30% for testing, 70% for training)
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42) # Split the data
7
8 # Preprocessing: Drop unnecessary columns that cannot be used by the model (like source_ip and destination_ip)
9 X_train = X_train.drop(columns=['source_ip', 'destination_ip'], errors='ignore') # Drop irrelevant columns
10 X_test = X_test.drop(columns=['source_ip', 'destination_ip'], errors='ignore') # Drop irrelevant columns
11
```

## 5. Preprocessing: Encoding and Scaling the Data

```
1 # Apply Label Encoding for categorical features (service, protocol_type, flag) to convert them into numeric format
2 label_encoders = {} # Dictionary to store label encoders for each column
3 categorical_columns = ['service', 'protocol_type', 'flag'] # List of categorical columns to encode
4 for col in categorical_columns:
5     le = LabelEncoder() # Initialize the label encoder
6     X_train[col] = le.fit_transform(X_train[col]) # Fit and transform the categorical feature on the training data
7     X_test[col] = le.transform(X_test[col]) # Transform the test data based on training data encoding
8     label_encoders[col] = le # Store the encoder for future use
9
10 # Standardize numerical features (scaling them to have mean=0 and variance=1)
11 scaler = StandardScaler() # Initialize the StandardScaler to scale numerical features
12 X_train = scaler.fit_transform(X_train) # Fit and transform the training set (scaling it)
13 X_test = scaler.transform(X_test) # Only transform the test set, using parameters from training data
```

## 6. Neural Network Model Definition

```
1 # Neural Network Model Definition
2 class IntrusionDetectionNN(nn.Module): # Define a custom neural network class inheriting from nn.Module
3     def __init__(self, input_size): # Constructor to initialize layers
4         super(IntrusionDetectionNN, self).__init__()
5         # Define a 3-layer fully connected neural network architecture
6         self.fc1 = nn.Linear(input_size, 64) # Input layer to hidden layer with 64 neurons
7         self.fc2 = nn.Linear(64, 32) # First hidden layer to second hidden layer with 32 neurons
8         self.fc3 = nn.Linear(32, 1) # Second hidden layer to output layer (binary output)
9         self.sigmoid = nn.Sigmoid() # Sigmoid function for binary classification
10
11     def forward(self, x):
12         # Define the forward pass using ReLU activations for hidden layers and Sigmoid for output
13         x = torch.relu(self.fc1(x)) # ReLU activation for first hidden layer
14         x = torch.relu(self.fc2(x)) # ReLU activation for second hidden layer
15         return self.sigmoid(self.fc3(x)) # Apply Sigmoid activation for final output
16
```

## 7. Model Training and Evaluation

```
1 # Initialize model with input size equal to the number of features
2 model = IntrusionDetectionNN(X_train.shape[1]) # Initialize model based on input size (number of features in X_train)
3 criterion = nn.BCELoss() # Binary Cross-Entropy loss for binary classification tasks
4 optimizer = optim.Adam(model.parameters(), lr=0.001) # Adam optimizer for weight updates, learning rate set to 0.001
5
6 # Training Loop
7 epochs = 20 # Number of epochs for training (iterations over the entire dataset)
8 batch_size = 32 # Batch size for training (number of samples processed in each batch)
9 train_losses, test_losses = [], [] # List to store loss values during training
10 train_accuracies, test_accuracies = [], [] # List to store accuracy values
11
```



## 8. Training Process

```
1 # Training the model
2 for epoch in range(epochs): # Loop through the specified number of epochs
3     model.train() # Set model to training mode (this enables dropout, batch normalization, etc.)
4     permutation = torch.randperm(X_train_tensor.size(0)) # Randomly shuffle the dataset for batch processing
5     epoch_train_loss = 0 # Initialize total training loss for this epoch
6
7     # Train in batches
8     for i in range(0, X_train_tensor.size(0), batch_size):
9         indices = permutation[i:i + batch_size] # Get indices for the current batch
10        batch_X, batch_y = X_train_tensor[indices], y_train_tensor[indices] # Extract the features and target for the batch
11
12        optimizer.zero_grad() # Reset gradients to zero for each batch
13        outputs = model(batch_X) # Perform a forward pass and get predictions
14        loss = criterion(outputs, batch_y) # Calculate loss (error between predictions and true values)
15        loss.backward() # Backpropagate the loss (compute gradients)
16        optimizer.step() # Perform an optimization step (adjust weights)
17
18        epoch_train_loss += loss.item() # Add the batch loss to the total loss for the epoch
19
20    # Store average training loss for the epoch
21    train_losses.append(epoch_train_loss / len(X_train_tensor))
22
23    # Calculate train accuracy
24    with torch.no_grad(): # Disable gradient tracking for evaluation (faster inference)
25        train_predictions = (model(X_train_tensor) > 0.5).float() # Make binary predictions (1 if probability > 0.5, else 0)
26        train_accuracy = accuracy_score(y_train_tensor.numpy(), train_predictions.numpy()) # Calculate accuracy
27        train_accuracies.append(train_accuracy)
28
29    # Evaluate on the test set
30    model.eval() # Set the model to evaluation mode
31    with torch.no_grad():
32        test_outputs = model(X_test_tensor) # Get predictions for the test set
33        test_loss = criterion(test_outputs, y_test_tensor) # Calculate test loss
34        test_losses.append(test_loss.item()) # Append test loss to the list
35        test_predictions = (test_outputs > 0.5).float() # Make binary predictions for test data
36        test_accuracy = accuracy_score(y_test_tensor.numpy(), test_predictions.numpy()) # Calculate test accuracy
37        test_accuracies.append(test_accuracy)
38
39    # Print epoch statistics
40    print(f"Epoch {epoch + 1}/{epochs}, Train Loss: {train_losses[-1]:.4f}, Test Loss: {test_losses[-1]:.4f}")
41    print(f"Train Accuracy: {train_accuracies[-1]:.4f}, Test Accuracy: {test_accuracies[-1]:.4f}")
42
```

## 9. Plot Loss and Accuracy over Epochs

```
1 # Plot Loss and Accuracy over epochs
2 plt.figure(figsize=(12, 6)) # Set up a figure for the plots
3
4 # Loss plot
5 plt.subplot(1, 2, 1) # Create the first subplot for loss
6 plt.plot(range(1, epochs + 1), train_losses, label='Train Loss', marker='o') # Plot train loss
7 plt.plot(range(1, epochs + 1), test_losses, label='Test Loss', marker='o') # Plot test loss
8 plt.xlabel('Epochs') # Label for the x-axis
9 plt.ylabel('Loss') # Label for the y-axis
10 plt.title('Train and Test Loss') # Title for the loss plot
11 plt.legend() # Display the legend
12
13 # Accuracy plot
14 plt.subplot(1, 2, 2) # Create the second subplot for accuracy
15 plt.plot(range(1, epochs + 1), train_accuracies, label='Train Accuracy', marker='o') # Plot train accuracy
16 plt.plot(range(1, epochs + 1), test_accuracies, label='Test Accuracy', marker='o') # Plot test accuracy
17 plt.xlabel('Epochs') # Label for the x-axis
18 plt.ylabel('Accuracy') # Label for the y-axis
19 plt.title('Train and Test Accuracy') # Title for the accuracy plot
20 plt.legend() # Display the legend
21
22 plt.tight_layout() # Adjust the layout to prevent overlap
23 plt.show() # Show the plots
24
25 # Visualize the distribution of MITRE ATT&CK techniques
26 mitre_counts = df['mitre_attack'].value_counts() # Count occurrences of each MITRE ATT&CK technique
27 plt.figure(figsize=(12, 8)) # Set the figure size for the bar plot
28 sns.barplot(x=mitre_counts.values, y=mitre_counts.index) # Plot a bar chart of technique frequencies
29 plt.title("MITRE ATT&CK Technique Distribution") # Set the plot title
30 plt.xlabel("Frequency") # Label for the x-axis
31 plt.ylabel("MITRE ATT&CK Techniques") # Label for the y-axis
32 plt.show() # Show the plot
```

## 10. Visualize the Distribution of MITRE ATT&CK Techniques

```
1  # Example input for real-time prediction
2  live_input = [
3      [300, 'udp', 'ftp', 'SF', 10000, 5000, 0, 0.01, 0, 2, 0, 0, 0, 0]
4  ] # Example of a single data sample with same structure as dataset
5
6  # Get MITRE ATT&CK tactic and technique prediction for the new input
7  mapped_tactic = map_to_mitre(live_input)
8  print(f"Predicted MITRE ATT&CK Technique: {mapped_tactic}")
```

***THANK YOU***