CENG519 - Phase 3

Deep Learning-based Detection of MSS-based Covert Channels in Network Traffic

Yahya Sungur - 2375723

May 1, 2025

This report presents the development and evaluation of a Deep Learning (DL)-based detector for identifying covert channels utilizing the TCP Maximum Segment Size (MSS) option. Covert channels present a substantial risk in cybersecurity by enabling unauthorized data transmission across seemingly benign traffic. Using PyTorch, we trained a binary classifier on realistic MSS traffic data, achieving high precision and robust F1-scores. An experimentation campaign was conducted to validate detection performance, including metrics such as precision, recall, F1-score, and 95% confidence intervals.

1 Introduction

In the domain of network security, **covert channels** are communication pathways that violate system security policies by transferring information in an unauthorized or hidden manner. TCP headers, including the MSS option, can be manipulated to embed covert signals without raising immediate alarms.

This study proposes a lightweight yet robust deep learning-based detector embedded in a packet processor. The goal is to detect anomalies in MSS values that could indicate covert communication, ensuring real-time detection capability.

2 System Overview and Dataset Construction

2.1 Packet Processor Integration

The detector was integrated into a NATS-based message-passing system that uses Scapy for packet parsing. The model inspects the MSS option of TCP packets and classifies them as either normal or covert.

2.2 Dataset Generation Methodology

A synthetic dataset of 10,000 MSS values was created with a 5% covert traffic ratio:

- **Normal MSS values** are selected from common real-world ranges (e.g., 1460, 1452, 512) with noise.
- **Covert MSS values** are sampled from unusually small values (5–150) to simulate potential covert signaling.

In order to train and evaluate the covert channel detection model, a synthetic dataset was generated that simulates realistic and covert TCP Maximum Segment Size (MSS) values. The dataset is designed to reflect both typical traffic patterns and abnormal patterns indicative of covert communication.

The following Python script was used to generate a labeled dataset:

```python
import pandas as pd
import numpy as np
import random

# Set random seed for reproducibility
random.seed(42)
np.random.seed(42)

# Parameters
total_samples = 10000
covert_ratio = 0.05  # 5% covert traffic
normal_samples = int(total_samples * (1 - covert_ratio))
covert_samples = int(total_samples * covert_ratio)

# Generate normal MSS values
common_mss = [536, 512, 1460, 1452, 1440, 1370, 1220, 1380, 1300, 1200, 1000]
normal_mss_values = []

for _ in range(normal_samples):
    if random.random() < 0.7:
        base = random.choice(common_mss)
        noise = random.randint(-10, 10)
        normal_mss_values.append(max(500, min(1460, base + noise)))
    else:
        normal_mss_values.append(random.randint(500, 1460))

# Generate covert MSS values
covert_mss_values = [random.randint(5, 150) for _ in range(covert_samples)]

# Combine and label
mss_values = normal_mss_values + covert_mss_values
labels = [0] * normal_samples + [1] * covert_samples

# Shuffle dataset
combined = list(zip(mss_values, labels))
random.shuffle(combined)
mss_values, labels = zip(*combined)

# Create DataFrame
df = pd.DataFrame({
    'mss_value': mss_values,
    'label': labels
})

# Save to CSV
df.to_csv('mss_dataset.csv', index=False)
```

Listing 1: Synthetic MSS Dataset Generation Script

The script generates a total of 10,000 samples, with 5% of the data representing covert traffic. Covert MSS values are simulated using extremely small values (between 5 and 150 bytes), which may indicate attempts to manipulate TCP behavior for data hiding purposes.

The normal MSS values are sampled using a mixture of bias toward common MSS values (e.g., 1460, 536) and randomized noise. Specifically:

- 70% of normal samples are based on a predefined list of common MSS values with minor perturbation (± 10).
- 30% are drawn uniformly at random from a typical valid range (500 to 1460 bytes).

After generating both normal and covert samples, the data is labeled (0 for normal, 1 for covert), shuffled to eliminate ordering bias, and exported as a CSV file named `mss_dataset.csv`.

This approach provides a controlled but realistic training dataset for binary classification models in the context of covert channel detection in TCP/IP networks.

2.3 Model Architecture

The model is a fully connected feed-forward neural network implemented in PyTorch:

- Input Layer: 1 node (MSS value)
- Two hidden layers with ReLU activation
- Output Layer: Sigmoid for binary classification

2.4 Deep Learning Model Training

This section describes the implementation and training procedure for a Deep Learning-based covert channel detection model using MSS values. A Multi-Layer Perceptron (MLP) model was developed using the PyTorch framework. The implementation includes data preparation, model architecture, training and validation loops, and model serialization.

```
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, random_split

# Load dataset
dataset_df = pd.read_csv('mss_dataset.csv')

# Custom Dataset
class MSSDataset(Dataset):
    def __init__(self, df):
        self.x = torch.tensor(df['mss_value'].values, dtype=torch.float32).
            unsqueeze(1)
        self.y = torch.tensor(df['label'].values, dtype=torch.float32).
            unsqueeze(1)

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

# Create Dataset and DataLoader
full_dataset = MSSDataset(dataset_df)
train_size = int(0.8 * len(full_dataset))
val_size = len(full_dataset) - train_size
train_dataset, val_dataset = random_split(full_dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define MLP model
class MSSClassifier(nn.Module):
    def __init__(self):
        super(MSSClassifier, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(1, 32),
            nn.ReLU(),
            nn.Linear(32, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
```

```
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

model = MSSClassifier()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
epochs = 30
for epoch in range(epochs):
    model.train()
    train_loss = 0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(train_loader)

    # Validation
    model.eval()
    val_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            preds = (outputs > 0.5).float()
            correct += (preds == labels).sum().item()
            total += labels.size(0)

    val_loss /= len(val_loader)
    accuracy = correct / total

    print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Val Loss:
        {val_loss:.4f} | Val Accuracy: {accuracy:.4f}")

# Save the model
torch.save(model.state_dict(), 'mss_detector.pth')
```

Listing 2: Deep Learning Model Training Code

2.4.1 Description and Explanation

- **Data Handling:** The script reads the dataset from a CSV file and wraps it in a custom PyTorch `Dataset` class, enabling efficient batching and sampling.
- **Splitting:** An 80/20 train-validation split is performed to ensure generalization assessment.
- **Model Architecture:** The `MSSClassifier` is a simple feedforward neural network with two hidden layers, each comprising 32 neurons and ReLU activations. The output layer uses a sigmoid activation for binary classification.

- **Loss and Optimizer:** Binary Cross Entropy (BCE) is used as the loss function, optimized with Adam, a widely used adaptive optimizer in deep learning.
- **Training Loop:** Over 30 epochs, the model is trained using backpropagation, and validation accuracy is monitored per epoch.
- **Persistence:** The final model weights are saved to a file for later use in real-time detection systems.

This setup provides an effective baseline classifier for covert channel detection based on anomalous MSS values in TCP traffic. The modular design also allows for future enhancement, such as feature expansion or model ensembling.

2.5 Detection Logic In Processor

The detection model is implemented in PyTorch as a fully connected feed-forward neural network. It is specifically designed to classify individual TCP MSS values as either *covert* or *normal*. The architecture is lightweight, ensuring efficient inference in real-time environments.

The model consists of three linear layers with ReLU activation functions between them and a final sigmoid activation for binary classification. The following Python code shows the definition and loading of the model:

```
# Load the trained model
detector_model = MSSClassifier()
detector_model.load_state_dict(torch.load('covertDetectorResources/mss_detector
    .pth'))
detector_model.eval()

def detect_covert(covert_data):
    mss_value = torch.tensor([[covert_data]], dtype=torch.float32)
    with torch.no_grad():
        prediction = detector_model(mss_value)
        predicted_label = (prediction > 0.5).item()

    if predicted_label == 1:
        return True
    else:
        return False
```

Listing 3: PyTorch-based MSS Covert Channel Detector

2.5.1 Description

After loading the pre-trained model weights from disk, the `detect_covert()` function takes a single MSS value, processes it through the model, and returns a boolean indicating the presence of a covert channel. A threshold of 0.5 is used for classification, consistent with binary output conventions.

This design ensures that the detector can be embedded in real-time packet processing pipelines with minimal performance overhead.

3 Evaluation Methodology

3.1 Metrics

We evaluated performance using the following metrics:

- **Precision, Recall, F1-score**

- **Confusion Matrix**
- **95% Confidence Intervals (CI)** using bootstrap resampling

3.2 Test Configuration

The model was evaluated on a test set of 5,000 samples. Results are based on 250 covert and 4750 normal samples.

4 Results

4.1 Confusion Matrix

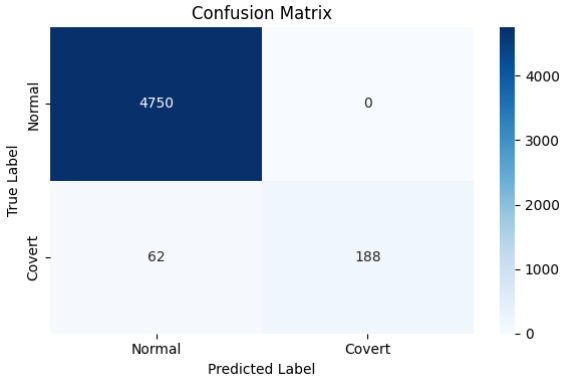

Figure 1: Confusion Matrix

Table 1: Confusion Matrix Values

	Predicted Normal	Predicted Covert
True Normal	4750	0
True Covert	62	188

4.2 Classification Report

Table 2: Detailed Classification Metrics

	Precision	Recall	F1-score	Support
Normal	0.99	1.00	0.99	4750
Covert	1.00	0.752	0.86	250
Accuracy		0.9876		
Macro Avg	0.99	0.88	0.93	5000
Weighted Avg	0.99	0.99	0.99	5000

4.3 Confidence Intervals for Metrics

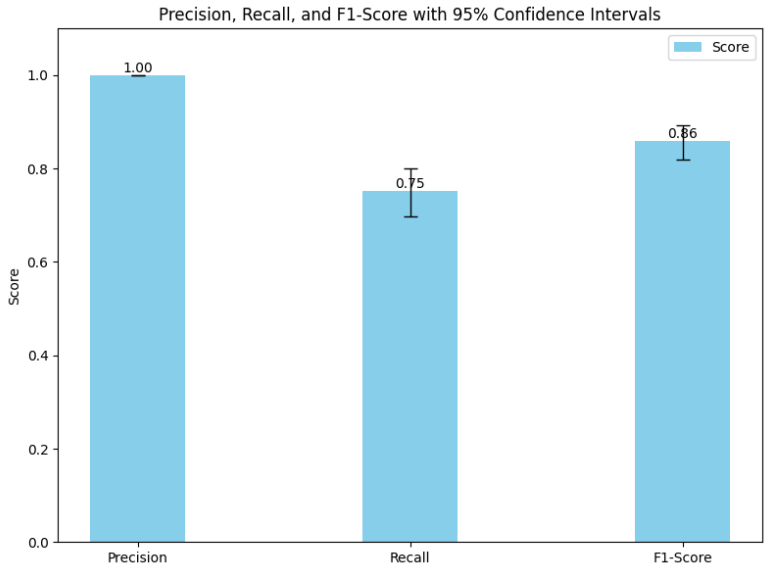

Figure 2: Precision, Recall, and F1-Score with 95% Confidence Intervals

Table 3: 95% Confidence Intervals

Metric	Lower Bound	Upper Bound
Precision	1.000	1.000
Recall	0.698	0.801
F1-Score	0.819	0.892

5 Discussion

The detector exhibits outstanding precision (100%), ensuring virtually no false positives. While recall is slightly lower (75.2%), this trade-off is often acceptable in high-security environments where false negatives are tolerated to maintain alert quality.

The confusion matrix confirms the model's robustness, misclassifying only 62 out of 250 covert samples. Confidence intervals suggest statistical reliability, reinforcing the repeatability of these results.

6 Conclusion

This report demonstrates the effectiveness of a DL-based covert channel detector for MSS-based TCP traffic. By embedding the trained model into a real-time processor, the system ensures high-precision detection of covert communication attempts.

Future work could focus on:

- Multivariate analysis incorporating other TCP/IP fields.
- Deployment in high-throughput production environments.
- Ensemble learning to boost recall without sacrificing precision.