Malware Detection System: Technical Report

1. Introduction

The purpose of this term project is to demonstrate your practical skills in implementing and deploying machine learning models for malware classification. The technical implementation of this project is comprised of three main tasks that need to be completed sequentially:

2. Building and Training the Model

For this task, a deep neural network based on the MalConv architecture was created and trained to classify Portable Executable (PE) files as malware or benign. The EMBER-2017 v2 dataset was utilized for training purposes. This dataset provides a comprehensive collection of features extracted from PE files, facilitating the training of effective malware classifiers.

This code snippet was used to create the model architecture:

```
# Create the MalConv model class
class MalConv(nn.Module):
    def __init__(self, input_length=2000000, embedding_dim=8, window_size=500, output_dim=1):
        super(MalConv, self).__init__()
        self.embed = nn.Embedding(256, embedding_dim, padding_idx=0)  # 256 unique bytes, embedding dimension
        self.conv1 = nn.Conv1d(in_channels=embedding_dim, out_channels=128, kernel_size=32, stride=32)
        self.conv2 = nn.Conv1d(in_channels=128, out_channels=128, kernel_size=32, stride=32)
        self.fc1 = nn.Linear(128, 128)
        self.fc2 = nn.Linear(128, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embed(x.clamp(min=0, max=255))  # Ensure indices are within the valid range
        x = x.transpose(1, 2)  # Conv1d expects (batch_size, channels, length)
        x = self.conv1(x)
        x = torch.relu(x)
        x = self.conv2(x)
        x = torch.relu(x)
        x = torch.squeeze(torch.max(x, dim=2)[0])  # Global max pooling
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
```

This is part of the code to train the model.

```
        # Validation step
        model.eval()  # Set model to evaluation mode
        val_loss = 0.0
        with torch.no_grad():
            for inputs, labels in sampled_val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs.squeeze(), labels.squeeze())
                val_loss += loss.item()
        print(f'Validation Loss: {val_loss/len(sampled_val_loader)}')

        # Save checkpoint every 5 epochs
        if (epoch + 1) % 5 == 0:
            checkpoint_path = os.path.join(save_dir, f'model_epoch_{epoch+1}.pt')
            torch.save(model.state_dict(), checkpoint_path)
            print(f'Model checkpoint saved to {checkpoint_path}')
```

3. Deploying the Model as a Cloud API

Amazon SageMaker was employed to deploy the trained model on the cloud and create an endpoint, or API, to enable other applications to utilize the model. Leveraging SageMaker simplified the deployment process, allowing for seamless integration of the model into cloud-based environments. The PE file and the model had different feature architecture, so the PE files were converted into vectorized feature data to match the model architecture.

This code snippet was used to create the model object and deploy the model into AWS Endpoints:

```python
import torch
import tarfile
import os
import boto3

# Define paths
model_path = 'model.pt'
compressed_model_path = 'model.tar.gz'
s3_bucket = 'vmalconv'
s3_key = 'vMalConv1-2/vMalConv1/model.pt'

# Download the PyTorch model from S3
s3 = boto3.client('s3')
s3.download_file(s3_bucket, os.path.join(s3_key, model_path), model_path)

# Create a tar.gz file containing the .pt file
with tarfile.open(compressed_model_path, 'w:gz') as tar:
    tar.add(model_path, arcname=os.path.basename(model_path))

# Upload the compressed file back to S3
s3.upload_file(compressed_model_path, s3_bucket, os.path.join(s3_key, compressed_model_path))

print("Model successfully compressed and uploaded to:", os.path.join(s3_bucket, s3_key, compressed_model_path))
```

4. Creating a Client Application

A web application was developed using Streamlit to serve as the client interface. Users can upload PE files through the application, which then converts them into feature vectors compatible with the MalConv/EMBER model. These feature vectors are then passed to the cloud API for classification, and the results (i.e., malware or benign labels, or probabilities) are displayed to the user.

This code snippet was used to create an API to upload PE files and classify them into malware or benign labels, or probabilities.

```
    # Make prediction
    prediction = malconv_model(feature_vector)
    return prediction.item()

def main():
    st.title('Malware Detection Web App')

    uploaded_file = st.file_uploader("Upload a PE file", type="exe")

    if uploaded_file is not None:
        # Extract features
        features = extract_features(uploaded_file)

        # Display uploaded file
        st.write('Uploaded PE file:', uploaded_file.name)

        # Predict
        prediction = predict_malware(features)

        # Display result
        if prediction > 0.5:
            st.write('Prediction: Malware (Probability:', prediction, ')')
        else:
            st.write('Prediction: Benign (Probability:', 1 - prediction, ')')

if __name__ == "__main__":
    main()
```

5. Performance Analysis

The performance of the deployed model can be assessed through various metrics, including accuracy, precision, recall, and F1 score. Additionally, the average latency of the classification process can be measured to evaluate the system's responsiveness. According to my model results:

Test Accuracy: 0.5077

Precision: 0.5068

Recall: 0.5950,

6. Conclusion

The overall view of the model's performance was not fully captured because I used a small sample of 30000, this may not fully capture how well the model performs on each class, especially in imbalanced datasets. Precision and Recall: These metrics complement each other. High precision indicates a low false positive rate, while high recall indicates low false negative rate, and this may mean that the Malware detected may be lower than the Benign.

The implementation and deployment of machine learning models for malware detection represents a critical aspect of cybersecurity. By following the outlined approach, a robust malware detection system can be developed and deployed, capable of effectively identifying and classifying malicious software. Continual monitoring and evaluation of the system's performance are essential to ensure its efficacy in detecting evolving threats.

7. References

Anderson, H. S., Kharkar, A., & Roth, P. (2018). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. Retrieved from https://github.com/endgameinc/ember

BSides San Francisco (2018). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. Retrieved from [Insert link to the talk]CAMLIS (2019). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. Retrieved from [Insert link to the talk]