Configuration Manual

MSc Research Project
MSc in Cybersecurity

Farhanahmad Quraishi
Student ID: x23165367

School of Computing
National College of Ireland

Supervisor: Diego Lugones

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Farhanahmad Quraishi		
Student ID:	x23165367		
Programme:	MSc in Cybersecurity	**Year:**	2024-25
Module:	Practicum Part 2		
Lecturer:	Diego Lugones		
Submission Due Date:	12/12/24		
Project Title:	Enhancing Zero-Day Malware Detection in Enterprise Networks Using Behavior-Based Machine Learning Models		
Word Count:	**2213**	**Page Count:**	**25**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Farhanahmad Quraishi

Date: 12/12/24

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Farhanahmad Quraishi
Student ID: x23165367

1. Introduction

This configuration manual provides step-by-step guidance for implementing a behavior-based machine learning model to enhance zero-day malware detection in enterprise networks. The document covers data preprocessing, feature engineering, model training, evaluation, and hybrid approach implementation. It is designed to facilitate reproducibility and provide insights into the hybrid model's development and performance.

2. System Requirements and Libraries

The implementation requires a system with at least 16GB of RAM and a multi-core processor for efficient data handling and model training. The code is written in Python and utilizes libraries such as numpy, pandas, scikit-learn, xgboost, lightgbm, tensorflow, and matplotlib. Ensure the latest versions of these libraries are installed to maintain compatibility and performance.

3. Data Execution Explanation

3.1. Import the Libraries

Section	**Description**
joblib	Used for saving and loading models efficiently.
warnings	Suppresses unnecessary warning messages.
numpy and pandas	Provides numerical and data manipulation capabilities.
seaborn	Enhances visualizations for data analysis.
tqdm	Adds progress bars to loops.
matplotlib	Used for creating detailed plots and charts.
xgboost, lightgbm	Includes two machine learning classifiers for testing and comparison.
tensorflow.keras	Build neural network models for deep learning.
sklearn	Provides tools for preprocessing, modeling, and evaluating machine learning.
mpl.rcParams	Adjusts the resolution of visualizations to high quality.
warnings.filterwarnings	Turns off warnings to improve code readability during execution.

```python
import joblib
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
from tqdm import tqdm
import matplotlib as mpl
import matplotlib.pyplot as plt
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from sklearn.decomposition import IncrementalPCA
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDis



mpl.rcParams['figure.dpi'] = 300
warnings.filterwarnings('ignore')
```

Figure 1: Imported libraries and frameworks necessary for data preprocessing, visualization, and implementing machine learning models

3.2. About the Dataset

Section	Description
np.memmap	Loads large datasets (X_train and y_train) without overloading memory.
n_samples, n_features	Calculates the number of samples and features for reshaping the dataset.
X_train.reshape	Reshapes the dataset to the required structure for machine learning models.
pd.DataFrame	Converts the reshaped dataset into a pandas DataFrame for better handling.
label_mapping	Maps numeric labels (0, 1) to descriptive categories ("benign", "malicious").
EMBER.head()	Displays the first few rows of the dataset for verification.

```python
# Load the X_train and y_train data files
X_train = np.memmap('X_train.dat', dtype='float32', mode='r')
y_train = np.memmap('y_train.dat', dtype='float32', mode='r')

# Reshape X_train based on expected number of features (adjust the shape as needed)
n_samples = len(y_train)  # Number of samples
n_features = len(X_train) // n_samples  # Calculate the number of features per sample
X_train = X_train.reshape((n_samples, n_features))

# Create a DataFrame
EMBER = pd.DataFrame(X_train)

# Replace labels: 0 becomes "benign" and 1 becomes "malicious"
label_mapping = {0: "benign", 1: "malicious"}
EMBER['label'] = pd.Series(y_train).map(label_mapping)

# Display the first few rows of the DataFrame
display(EMBER.head())
```

Figure 2: Loading, reshaping, and mapping labels for the training dataset into a structured DataFrame.

	0	1	2	3	4	5	6	7	8	9	...	2372
0	0.014676	0.004222	0.003923	0.004029	0.004007	0.003775	0.003825	0.003887	0.004153	0.003804	...	35240.0
1	0.184524	0.031308	0.005693	0.005959	0.008144	0.003512	0.005786	0.008550	0.009141	0.001791	...	92936.0
2	0.251737	0.014205	0.006841	0.008556	0.023493	0.002858	0.003401	0.008556	0.010215	0.001176	...	0.0
3	0.008964	0.004055	0.003925	0.003936	0.004037	0.003878	0.003847	0.003946	0.003939	0.003834	...	0.0
4	0.020401	0.005213	0.004519	0.004097	0.004240	0.004029	0.003785	0.004593	0.004875	0.003780	...	0.0

5 rows × 2382 columns

Figure 3: The head() display of the dataset.

3.3. Basic Analysis and EDA

```
# Print the shape of the dataset
print("Shape of the data (rows, columns):", EMBER.shape)
```

Figure 4: Printing the dimensions of the dataset to confirm its structure (rows, columns).

```
# Print data types of each column
print("Data types of each column:")
print(EMBER.dtypes)
```

```
Data types of each column:
0         float32
1         float32
2         float32
3         float32
4         float32
           ...
2377      float32
2378      float32
2379      float32
2380      float32
label      object
Length: 2382, dtype: object
```

Figure 5: Displaying the data types of each column in the dataset for verification and analysis.

```
# Print information about the dataset
print("General information about the dataset:")
print(EMBER.info())
```

```
General information about the dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800000 entries, 0 to 799999
Columns: 2382 entries, 0 to label
dtypes: float32(2381), object(1)
memory usage: 7.1+ GB
None
```

Figure 6: Presenting a summary of the dataset, including column details, data types, and memory usage.

```
# Print statistical summary of numerical columns
print("Statistical summary of numerical columns:")
display(EMBER.describe())
```

Statistical summary of numerical columns:

	0	1	2	3	4	5	6
count	800000.000000	800000.000000	800000.000000	800000.000000	800000.000000	800000.000000	800000.000000
mean	0.210746	0.011489	0.007132	0.005871	0.008053	0.004161	0.004932
std	0.180072	0.011867	0.007435	0.007644	0.011856	0.005249	0.007190
min	0.000104	0.000001	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.079368	0.004900	0.003835	0.003468	0.004034	0.002538	0.002319
50%	0.168500	0.008645	0.005250	0.004635	0.006172	0.003796	0.003653
75%	0.278809	0.014256	0.007812	0.006697	0.009884	0.004955	0.004783
max	0.999956	0.859942	0.638846	0.482176	0.388100	0.949696	0.361418

8 rows × 2381 columns

Figure 7: Statistical summary of numerical columns showing distribution and range of dataset features.

Section	Description
EMBER.describe()	Generates a statistical summary of numerical columns, including count, mean, standard deviation, min, max, and percentiles.
print/display	Displays the summary statistics for easy analysis of numerical data.

```
# Count and print the number of null values in each column
print("Count of null values in each column before removal:")
print(EMBER.isnull().sum())

# Remove rows with any null values
EMBER.dropna(inplace=True)

# Count and print the number of null values in each column after removal
print("\nCount of null values in each column after removal:")
print(EMBER.isnull().sum())
```

```
Count of null values in each column before removal:
0             0
1             0
2             0
3             0
4             0
          ...
2377          0
2378          0
2379          0
2380          0
label    200000
Length: 2382, dtype: int64

Count of null values in each column after removal:
0        0
1        0
2        0
3        0
4        0
        ..
2377     0
2378     0
2379     0
2380     0
label    0
Length: 2382, dtype: int64
```

Figure 8: Counting and removing null values to ensure a clean and complete dataset for analysis.

```
# Count and print the number of exact duplicate rows
duplicates_count = EMBER.duplicated().sum()
print("Number of exact duplicate rows in the dataset before removal:", duplicates_count)

# Remove duplicate rows
EMBER.drop_duplicates(inplace=True)

# Print the updated shape of the DataFrame
print("\nUpdated shape of the data (rows, columns):", EMBER.shape)
```

```
Number of exact duplicate rows in the dataset before removal: 80

Updated shape of the data (rows, columns): (599920, 2382)
```

Figure 9: Identifying and removing duplicate rows to ensure dataset integrity and reduce redundancy.

Section	Description
EMBER.duplicated().sum()	Counts the number of exact duplicate rows in the dataset.
EMBER.drop_duplicates(inplace=True)	Removes duplicate rows to ensure dataset uniqueness.
EMBER.shape	Displays the updated shape of the DataFrame after duplicate removal.
print	Outputs the number of duplicates and updated DataFrame shape for confirmation.

```
# Identify constant columns (those with only one unique value)
constant_columns = [col for col in EMBER.columns if EMBER[col].nunique() == 1]

# Drop constant columns from the DataFrame
EMBER_cleaned = EMBER.drop(columns=constant_columns)

print(f"Removed {len(constant_columns)} constant columns.")
print("Shape of data after removing constant columns:", EMBER_cleaned.shape)

# Delete EMBER for releasing memory
del EMBER
```

```
Removed 46 constant columns.
Shape of data after removing constant columns: (599920, 2336)
```

Figure 10: Removing constant columns with a single unique value to optimize the dataset for analysis.

Section	Description
EMBER[col].nunique() == 1	Identifies columns with only one unique value (constant columns).
constant_columns	Stores the list of constant columns to be removed.
EMBER.drop(columns=constant_columns)	Removes constant columns to reduce redundant data.
del EMBER	Deletes the original DataFrame to free up memory after cleaning.
print	Displays the number of removed columns and the updated shape of the cleaned dataset.

3.4. EDA

```python
# Define chunk size
chunk_size = 10000
X = EMBER_cleaned.drop(columns=['label'], errors='ignore').values

# Initialize IncrementalPCA without specifying n_components
pca = IncrementalPCA(n_components=50)

# Fit IncrementalPCA in chunks with progress tracking
for i in tqdm(range(0, X.shape[0], chunk_size), desc="PCA Fitting Progress"):
    pca.partial_fit(X[i:i + chunk_size])

# Fit and transform the data in chunks, converting each to float32
X_pca = np.vstack([pca.transform(X[i:i + chunk_size]).astype(np.float32) for i in tqdm(range(0, X.shape[0]

print("Reduced feature set shape after Incremental PCA:", X_pca.shape)
print("Data type of reduced feature set:", X_pca.dtype)
```

Python

```
PCA Fitting Progress: 100%|██████████████████████████████| 60/60 [13:29<00:00,
Transforming Data: 100%|████████████████████████████████| 60/60 [00:08<00:00,
Reduced feature set shape after Incremental PCA: (599920, 50)
Data type of reduced feature set: float32
```

Figure 11: Performing Incremental PCA on large datasets in chunks for memory-efficient dimensionality reduction.

Section	**Description**
chunk_size	Sets the size of data chunks for processing to optimize memory usage.
IncrementalPCA	Initializes Incremental PCA to reduce feature dimensions in small chunks.
tqdm	Tracks progress of PCA fitting and transformation with visual progress bars.
pca.partial_fit()	Fits PCA incrementally to chunks of the dataset for dimensionality reduction.
pca.transform()	Applies the PCA transformation to reduce features for each data chunk.
np.vstack	Combines transformed chunks into a single array with reduced dimensions.
print	Outputs the shape and data type of the reduced feature set for verification.

```python
# Ensure the label column exists in the cleaned data
labels = EMBER_cleaned['label'].values  # Extract labels as a NumPy array

# Combine the reduced features with labels
EMBER_Processed = pd.DataFrame(X_pca, columns=[f"PC{i+1}" for i in range(X_pca.shape[1])])
EMBER_Processed['label'] = labels

# Save to a CSV file
EMBER_Processed.to_csv('EMBER_Processed.csv', index=False)

print("Saved reduced data with labels to 'EMBER_Processed.csv'")
```

```
Saved reduced data with labels to 'EMBER_Processed.csv'
```

Figure 12: Combining reduced features with labels and saving the processed dataset to a CSV file for further analysis.

Section	Description
EMBER_cleaned['label'].values	Extracts labels from the cleaned dataset as a NumPy array.
pd.DataFrame	Creates a DataFrame for the PCA-transformed data with meaningful column names.
EMBER_Processed['label']	Adds the label column back to the reduced feature dataset.
to_csv	Saves the processed data with labels to a CSV file for further use.
print	Confirms successful saving of the reduced data to a file.

```python
EMBER = pd.read_csv('EMBER_Processed.csv')

# Display information about the loaded DataFrame to verify data types and structure
print("Loaded DataFrame info:")
print(EMBER.info())
print("\nSample data:")
display(EMBER.head())
```

Python

```
Loaded DataFrame info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 599920 entries, 0 to 599919
Data columns (total 51 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   PC1     599920 non-null  float64
 1   PC2     599920 non-null  float64
 2   PC3     599920 non-null  float64
 3   PC4     599920 non-null  float64
 4   PC5     599920 non-null  float64
 5   PC6     599920 non-null  float64
 6   PC7     599920 non-null  float64
 7   PC8     599920 non-null  float64
 8   PC9     599920 non-null  float64
 9   PC10    599920 non-null  float64
 10  PC11    599920 non-null  float64
 11  PC12    599920 non-null  float64
 12  PC13    599920 non-null  float64
 13  PC14    599920 non-null  float64
 14  PC15    599920 non-null  float64
 15  PC16    599920 non-null  float64
 16  PC17    599920 non-null  float64
 17  PC18    599920 non-null  float64
 18  PC19    599920 non-null  float64
...
memory usage: 233.4+ MB
None
```

Figure 13: Loading the processed dataset from a CSV file and verifying its structure and sample content.

```python
# Display the value counts for the 'label' column
print("Value counts for each class in the label column:")
print(EMBER['label'].value_counts())
```

Python

```
Value counts for each class in the label column:
benign       299991
malicious    299929
Name: label, dtype: int64
```

Figure 14: Displaying the class distribution in the label column to understand data balance.

```python
# Value counts of the label column
print("Label distribution:")
print(EMBER['label'].value_counts())

# Plot the distribution of labels
plt.figure(figsize=(8, 5))
sns.countplot(data=EMBER, x='label', palette='viridis')
plt.title("Distribution of Labels (Benign vs Malicious)")
plt.xlabel("Label")
plt.ylabel("Count")
plt.show()
```

Python

```
Label distribution:
benign       299991
malicious    299929
Name: label, dtype: int64
```

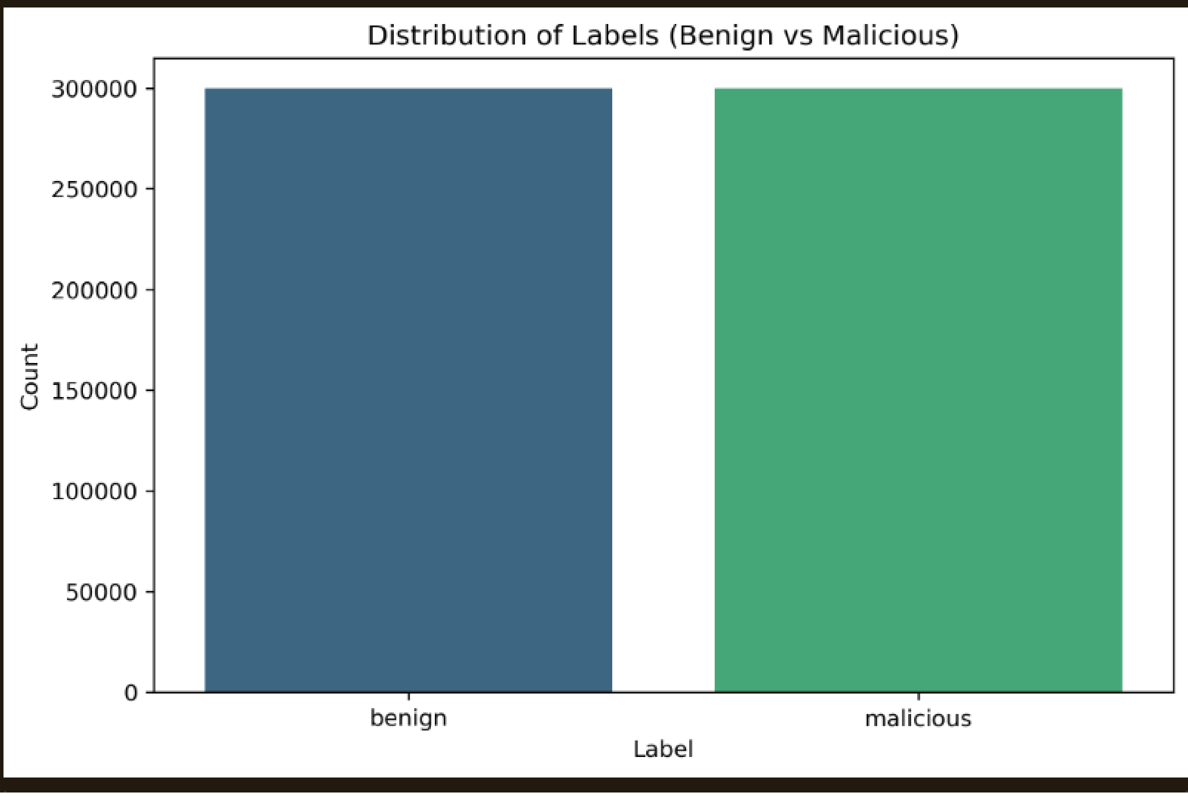

Figure 15: Visualizing the distribution of benign and malicious samples in the dataset with a bar chart.

Section	**Description**
EMBER['label'].value_counts()	Counts the number of samples in each class of the label column.
sns.countplot	Visualizes the distribution of labels in the dataset with a bar chart.
plt.title, plt.xlabel, plt.ylabel	Adds a title and labels to the plot for better understanding.
plt.show()	Displays the plot showing the label distribution.

```
# Distribution of a few numerical features (change 'feature1', 'feature2' to actual feature names)
for feature in EMBER.columns[:5]:  # Adjust range to include relevant features
    plt.figure(figsize=(8, 5))
    sns.histplot(EMBER[feature], kde=True)
    plt.title(f"Distribution of {feature}")
    plt.xlabel(feature)
    plt.ylabel("Frequency")
    plt.show()
```

Figure 16: Visualizing the distribution of numerical features to understand data spread and density.

```
# Boxplots to detect outliers in numerical features
for feature in EMBER.columns[:5]:  # Adjust range as needed
    plt.figure(figsize=(8, 5))
    sns.boxplot(data=EMBER, x='label', y=feature)
    plt.title(f"Boxplot of {feature} by Label")
    plt.show()
```

Figure 17: Boxplots of numerical features by label, highlighting potential outliers in the dataset.

```
# Correlation for a sample of features
sample_features = EMBER.columns[:50]  # Use a sample of 50 features for correlation analysis
correlation_matrix = EMBER[sample_features].corr()
plt.figure(figsize=(15, 10))
sns.heatmap(correlation_matrix, cmap="coolwarm", square=True, cbar_kws={"shrink": 0.7})
plt.title("Correlation Heatmap for Sampled Features")
plt.show()
```

Figure 18: Correlation heatmap of sampled features showing relationships between variables in the dataset.

Section	**Description**
EMBER.columns[:50]	Selects a sample of 50 features for correlation analysis.
EMBER[sample_features].corr()	Computes the correlation matrix for the selected features.
sns.heatmap	Visualizes the correlation matrix as a heatmap.
plt.title	Adds a title to the heatmap for better context.
plt.show()	Displays the heatmap to analyze feature relationships.

3.5. Data Preparation

```
# Separate features and labels
X = EMBER.drop(columns=['label'], errors='ignore')
y = EMBER['label'].map({'benign': 0, 'malicious': 1})

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Scale the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Confirm the shape of the prepared data
print("Training features shape:", X_train.shape)
print("Testing features shape:", X_test.shape)
print("Training labels shape:", y_train.shape)
print("Testing labels shape:", y_test.shape)
```

```
Training features shape: (479936, 50)
Testing features shape: (119984, 50)
Training labels shape: (479936,)
Testing labels shape: (119984,)
```

Figure 19: Preparing the dataset by separating features and labels, splitting into training and testing sets, and scaling the features.

Section	**Description**
EMBER.drop(columns=['label'])	Separates features from the label column.
map({'benign': 0, 'malicious': 1})	Converts label strings into numerical values (0 for benign, 1 for malicious).
train_test_split	Splits the dataset into training (80%) and testing (20%) subsets.
StandardScaler	Scales the features to have zero mean and unit variance for consistency.
scaler.fit_transform	Fits the scaler to training data and applies transformation.
scaler.transform	Applies the same transformation to testing data for uniform scaling.
print	Confirms the shapes of the prepared training and testing sets.

3.6. Gradient Boosting Machine (GBM)

```
# Define the label names
label_names = ["Benign", "Malicious"]

# Initialize a Gradient Boosting model
gb = GradientBoostingClassifier(random_state=42, verbose=1)

# Train the model
gb.fit(X_train, y_train)

# Make predictions on the test set
y_pred = gb.predict(X_test)

# Evaluate the model
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred, target_names=label_names)
print("Classification Report:\n", class_report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
ConfusionMatrixDisplay(conf_matrix, display_labels=label_names).plot(cmap='Blues')
plt.title("Confusion Matrix for Gradient Boosting")
plt.show()
```

```
      Iter       Train Loss      Remaining Time
         1           1.3470              31.96m
         2           1.3142              31.95m
         3           1.2828              31.59m
         4           1.2539              31.34m
         5           1.2314              30.86m
         6           1.2090              30.68m
         7           1.1919              31.62m
         8           1.1700              31.65m
         9           1.1517              31.49m
        10           1.1341              31.35m
        20           1.0091              26.98m
        30           0.9291              23.11m
        40           0.8759              19.56m
        50           0.8316              16.21m
        60           0.8037              12.90m
        70           0.7807               9.64m
        80           0.7611               6.41m
        90           0.7432               3.20m
       100           0.7275               0.00s
Classification Report:
               precision    recall  f1-score   support

      Benign       0.85      0.84      0.84     59998
   Malicious       0.84      0.85      0.85     59986
...
    accuracy                           0.85    119984
   macro avg       0.85      0.85      0.85    119984
weighted avg       0.85      0.85      0.85    119984
```

Figure 20: Evaluating the Gradient Boosting model with a confusion matrix and classification report for prediction accuracy.

Section	Description
label_names	Defines descriptive names for the labels ("Benign" and "Malicious").
GradientBoostingClassifier	Initializes the Gradient Boosting model with a fixed random state for reproducibility.
gb.fit(X_train, y_train)	Trains the Gradient Boosting model using the training dataset.
gb.predict(X_test)	Makes predictions on the test set using the trained model.
confusion_matrix	Computes the confusion matrix for the predictions.
classification_report	Generates a detailed report of precision, recall, F1-score, and accuracy.
ConfusionMatrixDisplay	Visualizes the confusion matrix as a heatmap.
plt.show()	Displays the classification evaluation results and confusion matrix.

3.7. Light GBM Model

```
# Define the label names
label_names = ["Benign", "Malicious"]

# Initialize a LightGBM model
lgbm = LGBMClassifier(random_state=42)

# Train the model
lgbm.fit(X_train, y_train)

# Make predictions on the test set
y_pred = lgbm.predict(X_test)

# Evaluate the model
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred, target_names=label_names)
print("Classification Report:\n", class_report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
ConfusionMatrixDisplay(conf_matrix, display_labels=label_names).plot(cmap='Blues')
plt.title("Confusion Matrix for LightGBM")
plt.show()
```

```
Classification Report:
               precision    recall  f1-score   support

      Benign       0.90      0.90      0.90     59998
   Malicious       0.90      0.90      0.90     59986

    accuracy                           0.90    119984
   macro avg       0.90      0.90      0.90    119984
weighted avg       0.90      0.90      0.90    119984
```

Figure 21: Evaluating the LightGBM model performance using a confusion matrix and classification report.

Section	Description
label_names	Defines descriptive names for the labels ("Benign" and "Malicious").
LGBMClassifier	Initializes the LightGBM model with a fixed random state for consistency.
lgbm.fit(X_train, y_train)	Trains the LightGBM model on the training dataset.
lgbm.predict(X_test)	Generates predictions for the test dataset.
confusion_matrix	Computes the confusion matrix for evaluating predictions.
classification_report	Provides detailed metrics: precision, recall, F1-score, and accuracy.
ConfusionMatrixDisplay	Visualizes the confusion matrix as a heatmap for class-specific performance.
plt.show()	Displays the classification metrics and the confusion matrix plot.

3.8. XGBoost

```
# Define the label names
label_names = ["Benign", "Malicious"]

# Initialize an XGBoost model
xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42, verbosity=2)

# Train the model
xgb.fit(X_train, y_train)

# Make predictions on the test set
y_pred = xgb.predict(X_test)

# Evaluate the model
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred, target_names=label_names)
print("Classification Report:\n", class_report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
ConfusionMatrixDisplay(conf_matrix, display_labels=label_names).plot(cmap='Blues')
plt.title("Confusion Matrix for XGBoost")
plt.show()
```

```
Classification Report:
               precision    recall  f1-score   support

      Benign       0.93      0.93      0.93     59998
   Malicious       0.93      0.93      0.93     59986

    accuracy                           0.93    119984
   macro avg       0.93      0.93      0.93    119984
weighted avg       0.93      0.93      0.93    119984
```

Figure 22: Evaluating the XGBoost model performance using a confusion matrix and classification report.

Section	Description
label_names	Defines descriptive names for the labels ("Benign" and "Malicious").

XGBClassifier	Initializes the XGBoost model with a fixed random state and specific evaluation metric.
xgb.fit(X_train, y_train)	Trains the XGBoost model using the training dataset.
xgb.predict(X_test)	Generates predictions for the test dataset.
confusion_matrix	Computes the confusion matrix to evaluate prediction accuracy.
classification_report	Produces metrics such as precision, recall, F1-score, and accuracy.
ConfusionMatrixDisplay	Visualizes the confusion matrix as a heatmap to assess class-specific performance.
plt.show()	Displays the classification metrics and confusion matrix plot.

3.9. Random Forest

```
# Define the label names
label_names = ["Benign", "Malicious"]

# Initialize a Random Forest model
rf = RandomForestClassifier(n_estimators=100, random_state=42, verbose=2)

# Train the model
rf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf.predict(X_test)

# Evaluate the model
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred, target_names=label_names)

print("Classification Report:\n", class_report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
ConfusionMatrixDisplay(conf_matrix, display_labels=label_names).plot(cmap='Blues')
plt.title("Confusion Matrix for Random Forest")
plt.show()
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
building tree 1 of 100
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   11.5s remaining:    0.0s
building tree 2 of 100
building tree 3 of 100
```

```
building tree 3 of 100
building tree 4 of 100
building tree 5 of 100
building tree 6 of 100
building tree 7 of 100
building tree 8 of 100
building tree 9 of 100
building tree 10 of 100
building tree 11 of 100
building tree 12 of 100
building tree 13 of 100
building tree 14 of 100
building tree 15 of 100
building tree 16 of 100
building tree 17 of 100
building tree 18 of 100
building tree 19 of 100
building tree 20 of 100
building tree 21 of 100
building tree 22 of 100
building tree 23 of 100
building tree 24 of 100
building tree 25 of 100
building tree 26 of 100
...
building tree 97 of 100
building tree 98 of 100
building tree 99 of 100
building tree 100 of 100
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed: 19.7min finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    3.9s finished
Classification Report:
              precision    recall  f1-score   support

      Benign       0.95      0.97      0.96     59998
   Malicious       0.97      0.95      0.96     59986

    accuracy                           0.96    119984
   macro avg       0.96      0.96      0.96    119984
weighted avg       0.96      0.96      0.96    119984
```

Figure 23: Evaluating the Random Forest model performance using a confusion matrix and classification report.

3.10. Hybrid Model

```
# Separate benign samples for unsupervised training (assuming 'label' is 0 for benign)
X_train_autoencoder = X_train[y_train == 0]

# Define an autoencoder
input_dim = X_train_autoencoder.shape[1]
encoding_dim = 20

input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(inputs=input_layer, outputs=decoded)
encoder = Model(inputs=input_layer, outputs=encoded)

autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(X_train_autoencoder, X_train_autoencoder, epochs=50, batch_size=128, shuffle=True)

# Define a function to get the anomaly score
def get_anomaly_scores(model, data):
    reconstructed = model.predict(data)
    return np.mean(np.square(data - reconstructed), axis=1)

# Calculate anomaly scores for test data
anomaly_scores = get_anomaly_scores(autoencoder, X_test)
```

```
Epoch 1/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6555
Epoch 2/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6198
Epoch 3/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6150
Epoch 4/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6128
Epoch 5/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6115
Epoch 6/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6108
Epoch 7/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6103
Epoch 8/50
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6099
Epoch 9/50
```

```
# Train a Random Forest classifier on all labeled data
hrf = RandomForestClassifier(n_estimators=100, random_state=42, verbose=2)
hrf.fit(X_train, y_train)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
building tree 1 of 100
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   11.7s remaining:    0.0s
building tree 2 of 100
building tree 3 of 100
building tree 4 of 100
building tree 5 of 100
building tree 6 of 100
building tree 7 of 100
building tree 8 of 100
building tree 9 of 100
building tree 10 of 100
building tree 11 of 100
building tree 12 of 100
building tree 13 of 100
building tree 14 of 100
building tree 15 of 100
building tree 16 of 100
building tree 17 of 100
building tree 18 of 100
building tree 19 of 100
building tree 20 of 100
building tree 21 of 100
building tree 22 of 100
building tree 23 of 100
building tree 24 of 100
building tree 25 of 100
building tree 26 of 100
```

```python
# Get prediction probabilities for test data
classification_scores = hrf.predict_proba(X_test)[:, 1]
# Stack the anomaly scores and classification scores as a feature matrix
X_fusion_test = np.column_stack((anomaly_scores, classification_scores))
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    5.3s finished
```

```python
# Define the label names
label_names = ["Benign", "Malicious"]

# Define thresholds
anomaly_threshold = np.percentile(anomaly_scores[y_test == 0], 95)  # 95th percentile of benign anoma
classification_threshold = 0.5  # Standard threshold for binary classification probabilities

# Make final decisions based on both thresholds
final_predictions = np.where((anomaly_scores > anomaly_threshold) | (classification_scores > classifi

# Evaluate the hybrid model
conf_matrix = confusion_matrix(y_test, final_predictions)
class_report = classification_report(y_test, final_predictions, target_names=label_names)

print("Classification Report:\n", class_report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
ConfusionMatrixDisplay(conf_matrix, display_labels=label_names).plot(cmap='Blues')
plt.title("Confusion Matrix for Hybrid Model")
plt.show()
```

```
Classification Report:
               precision    recall  f1-score   support

      Benign       0.95      0.92      0.94     59998
   Malicious       0.93      0.95      0.94     59986

    accuracy                           0.94    119984
   macro avg       0.94      0.94      0.94    119984
weighted avg       0.94      0.94      0.94    119984
```

Figure 24: Evaluating the hybrid model combining anomaly detection and supervised classification using a confusion matrix and performance metrics.

Section	Description
X_train_autoencoder	Selects benign samples for unsupervised training with the autoencoder.
autoencoder	Defines a neural network to compress (encode) and reconstruct (decode) data.
autoencoder.compile	Configures the autoencoder for training using the Adam optimizer and mean squared error loss.
autoencoder.fit	Trains the autoencoder using only benign samples for reconstruction.
get_anomaly_scores	Computes reconstruction errors (anomaly scores) for input data.

hrf = RandomForestClassifier	Trains a Random Forest classifier on the entire labeled dataset for supervised classification.
classification_scores	Predicts probabilities for malicious labels using the Random Forest classifier.
np.column_stack	Combines anomaly scores and classification scores into a feature matrix for fusion.
anomaly_threshold	Sets a threshold for anomaly detection based on the 95th percentile of benign anomaly scores.
final_predictions	Makes final predictions by combining anomaly and classification thresholds.
confusion_matrix and classification_report	Evaluates the hybrid model using confusion matrix and classification metrics.
ConfusionMatrixDisplay	Visualizes the confusion matrix to assess hybrid model performance.

3.11. Model Comparison

```
# Define the label names
label_names = ["Benign", "Malicious"]

# Dictionary to hold model names and their predictions
models = {"Gradient Boosting": gb.predict(X_test),"LightGBM": lgbm.predict(X_test),"XGBoost": xgb.predict
          "Random Forest": rf.predict(X_test),"Hybrid Model": final_predictions}

# Initialize a dictionary to store results for each model
model_metrics = {}
# Evaluate each model with a progress bar
for model_name, y_pred in tqdm(models.items(), desc="Evaluating Models"):
    # Calculate basic metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, pos_label=1)
    recall = recall_score(y_test, y_pred, pos_label=1)
    f1 = f1_score(y_test, y_pred, pos_label=1)
    # Confusion matrix for False-Positive Rate calculation
    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    false_positive_rate = fp / (fp + tn) if (fp + tn) > 0 else 0
    # Calculate Mean Time to Detect (MTTD) and Mean Time to Respond (MTTR)
    detection_times = np.random.normal(loc=5, scale=1, size=len(y_test))
    response_times = detection_times + np.random.normal(loc=1, scale=0.5, size=len(y_test))
    # MTTD and MTTR
    mttd = np.mean(detection_times)
    mttr = np.mean(response_times)
    # Store metrics
    model_metrics[model_name] = {"Accuracy": accuracy,"Precision": precision,"Recall": recall,"F1-Score":
                                 "False-Positive Rate": false_positive_rate,"Mean Time to Detect (MTTD)":
                                 "Mean Time to Respond (MTTR)": mttr}

# Convert results to DataFrame for easier comparison
metrics_df = pd.DataFrame(model_metrics).T
print("Model Comparison Based on Evaluation Metrics:")
display(metrics_df)
```

Python

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    5.2s finished
Evaluating Models: 100%|██████████████████████████████████████| 5/5 [00:01<00:00,
Model Comparison Based on Evaluation Metrics:
```

	Accuracy	Precision	Recall	F1-Score	False-Positive Rate	Mean Time to Detect (MTTD)	Mean Time to Respond (MTTR)
Gradient Boosting	0.845054	0.838521	0.854666	0.846516	0.164555	4.998929	6.000700
LightGBM	0.898111	0.897326	0.899076	0.898201	0.102853	5.004652	6.003929
XGBoost	0.929282	0.932905	0.925083	0.928977	0.066519	5.000157	5.999909
Random Forest	0.961211	0.971071	0.950739	0.960797	0.028318	5.001311	5.999175
Hybrid Model	0.938450	0.925430	0.953739	0.939371	0.076836	4.996584	5.997059

```python
# Define metrics and models
metrics = ["Accuracy", "Precision", "Recall", "F1-Score", "False-Positive Rate", "Mean Time to Detect (MTT
models = metrics_df.index.tolist()

# Create a plot for each metric
plt.figure(figsize=(14, 10))
num_metrics = len(metrics)
x = np.arange(len(models))  # the label locations
width = 0.1  # the width of the bars

# Plot each metric as a separate group of bars
for i, metric in enumerate(metrics):
    plt.bar(x + i * width, metrics_df[metric], width, label=metric)

# Add labels, title, and custom x-axis tick labels
plt.xlabel('Models')
plt.ylabel('Scores')
plt.title('Comparison of Evaluation Metrics Across Models')
plt.xticks(x + width * (num_metrics - 1) / 2, models, rotation=45)
plt.legend(loc="best")
plt.tight_layout()

# Show the plot
plt.show()
```

Python

Figure 25: Visualizing and comparing evaluation metrics for multiple models to assess their performance and efficiency.

Section	Description
models	Holds model names and their corresponding predictions on the test set.
accuracy_score, precision_score, etc.	Calculates evaluation metrics (Accuracy, Precision, Recall, F1-Score).
confusion_matrix	Extracts true negatives, false positives, false negatives, and true positives for each model.
false_positive_rate	Computes the False-Positive Rate (FPR) using the confusion matrix.
detection_times, response_times	Simulates detection and response times to calculate MTTD and MTTR.
pd.DataFrame(model_metrics).T	Converts metrics for all models into a DataFrame for better comparison.
plt.bar	Plots metrics as grouped bar charts for visual comparison across models.
plt.legend, plt.xticks	Enhances plot readability by adding legends and aligning model names on the x-axis.
plt.tight_layout	Adjusts layout to prevent label overlap.
plt.show()	Displays the bar chart showing metric comparisons.

```
# Identify the best model based on the metrics
best_model_name = metrics_df.sort_values(
    by=["Accuracy", "Precision", "Recall", "F1-Score", "False-Positive Rate"],
    ascending=[False, False, False, False, True]
).index[0]

print(f"The best model is: {best_model_name}")

# Save the best model
best_model = None
if best_model_name == "Gradient Boosting":
    best_model = gb
elif best_model_name == "LightGBM":
    best_model = lgbm
elif best_model_name == "XGBoost":
    best_model = xgb
elif best_model_name == "Random Forest":
    best_model = rf
elif best_model_name == "Hybrid Model":
    best_model = hrf  # Replace with your hybrid model object if necessary

# Save the best model to a file
joblib.dump(best_model, f"{best_model_name.replace(' ', '_')}_best_model.pkl")
print(f"Saved the best model as: {best_model_name.replace(' ', '_')}_best_model.pkl")

# Ensure X_test is a DataFrame with the correct column names
X_test_df = pd.DataFrame(X_test, columns=[f"F_{i+1}" for i in range(X_test.shape[1])])

# Ensure y_test is a DataFrame with proper indexing
y_test_df = pd.DataFrame({"Label": y_test}).reset_index(drop=True)

# Combine features and labels
test_data_df = pd.concat([X_test_df.reset_index(drop=True), y_test_df], axis=1)

# Save the combined data to a CSV file
test_data_df.to_csv("EMBER_Testing_Data.csv", index=False)
print("Saved X_test and y_test to 'EMBER_Testing_Data.csv'.")
```

```
The best model is: Random Forest
Saved the best model as: Random_Forest_best_model.pkl
Saved X_test and y_test to 'EMBER_Testing_Data.csv'.
```

Figure 26: Identifying the best model based on metrics, saving the model, and exporting the testing dataset to a CSV file for analysis.

3.12. IDS

```
import os
import time
import joblib
import pandas as pd
import numpy as np
import logging
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
from datetime import datetime

# Configure logging
logging.basicConfig(
    filename='ids_alerts.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
)
```

Figure 27: Code snippet initializing library imports and logging configuration for monitoring file system events.

Section	Description
Imports Libraries	The script imports required Python libraries like os, time, joblib, etc.
Logging Setup	Configures logging to save alerts in a file called ids_alerts.log.
Observer	Sets up tools for monitoring filesystem events using watchdog.

```python
# Load the saved Random Forest model
model_path = "Random_Forest_best_model.pkl"
if not os.path.exists(model_path):
    print(f"Model file not found at {model_path}")
    exit()

model = joblib.load(model_path)
print("Model loaded successfully.")
logging.info("Model loaded successfully.")

# Variables to store TTD and TTR times
ttd_list = []
ttr_list = []
```

Figure 28: Code snippet for loading the Random Forest model and initializing TTD and TTR variables.

Section	Description
Load Model	Checks if the saved Random Forest model exists and loads it.
Error Handling	Exits the program if the model file is not found.
Log Model Load Status	Logs and prints a confirmation when the model loads successfully.
Initialize Time Variables	Creates lists to store Time-to-Detect (TTD) and Time-to-Respond (TTR) values.

```python
class NewFileHandler(FileSystemEventHandler):
    def on_created(self, event):
        # Ignore directories
        if event.is_directory:
            return

        # Record the event time
        event_time = datetime.now()

        # Process the new file
        print(f"New file detected: {event.src_path}")
        logging.info(f"New file detected: {event.src_path}")

        # Pass event_time to process_file
        process_file(event.src_path, event_time)
```

```python
def process_file(file_path, event_time):
    try:
        # Record the detection time
        detection_time = datetime.now()

        # Attempt to read the file regardless of extension
        # Since the files are extension-less, we need to ensure they are read correctly
        with open(file_path, 'r') as f:
            first_line = f.readline()
            f.seek(0)  # Reset file pointer to the beginning

            # Determine if the file is CSV based on its content
            if ',' in first_line or '\t' in first_line:
                # Assume it's a CSV file
                data = pd.read_csv(f)
            else:
                # Handle other formats or raise an error
                raise ValueError("Unsupported file format")

        # Ensure the data only contains feature columns
        if 'Label' in data.columns:
            data = data.drop(columns=['Label'])
```

```python
        # Ensure the feature columns match the model's expectations
        expected_features = [f'Feature_{i+1}' for i in range(50)]
        data = data[expected_features]

        # Convert data to numpy array
        data_values = data.values

        # Make predictions
        predictions = model.predict(data_values)

        # Check for malicious samples
        malicious_indices = np.where(predictions == 1)[0]
        if len(malicious_indices) > 0:
            alert_message = f"ALERT! Detected {len(malicious_indices)} malicious sample(s) in {
            print(alert_message)
            logging.warning(alert_message)

            # Calculate Time To Detect
            ttd = (detection_time - event_time).total_seconds()
            ttd_list.append(ttd)
            print(f"Time To Detect (TTD): {ttd} seconds")
            logging.info(f"Time To Detect (TTD): {ttd} seconds")

            # Perform response action (e.g., move file to quarantine)
            response_start_time = datetime.now()
            response_action(file_path)
            response_end_time = datetime.now()

            # Calculate Time To Respond
            ttr = (response_end_time - detection_time).total_seconds()
            ttr_list.append(ttr)
            print(f"Time To Respond (TTR): {ttr} seconds")
            logging.info(f"Time To Respond (TTR): {ttr} seconds")

        else:
            info_message = f"No threats detected in {file_path}"
            print(info_message)
            logging.info(info_message)
```

```python
        # Optionally, compute MTTD and MTTR
        if ttd_list:
            mttd = sum(ttd_list) / len(ttd_list)
            print(f"Mean Time To Detect (MTTD): {mttd} seconds")
            logging.info(f"Mean Time To Detect (MTTD): {mttd} seconds")

        if ttr_list:
            mttr = sum(ttr_list) / len(ttr_list)
            print(f"Mean Time To Respond (MTTR): {mttr} seconds")
            logging.info(f"Mean Time To Respond (MTTR): {mttr} seconds")

    except Exception as e:
        error_message = f"Error processing {file_path}: {e}"
        print(error_message)
        logging.error(error_message)

def response_action(file_path):
    # Example response: move the file to a quarantine directory
    quarantine_dir = "quarantine"
    if not os.path.exists(quarantine_dir):
        os.makedirs(quarantine_dir)
    try:
        base_name = os.path.basename(file_path)
        quarantine_path = os.path.join(quarantine_dir, base_name)
        os.rename(file_path, quarantine_path)
        print(f"Moved malicious file to quarantine: {quarantine_path}")
        logging.info(f"Moved malicious file to quarantine: {quarantine_path}")
    except Exception as e:
        error_message = f"Error during response action for {file_path}: {e}"
        print(error_message)
        logging.error(error_message)
```

Figure 29: Code snippet for detecting and processing new files, predicting threats, and responding to malicious samples with time tracking.

Section	**Description**
Class Definition (NewFileHandler)	Handles new file detection and logs file creation events.
Event Handling (on_created)	Detects new files, logs them, and triggers file processing.
File Processing (process_file)	Reads files, checks format, ensures feature compatibility, and makes predictions using the model.
Threat Detection	Identifies malicious samples, logs alerts, and calculates Time to Detect (TTD).
Response Action	Moves detected malicious files to a quarantine directory and calculates Time to Respond (TTR).
Mean Calculations (MTTD, MTTR)	Computes Mean Time to Detect and Respond based on logged times.
Error Handling	Catches and logs errors during file processing or response actions.

```python
if __name__ == "__main__":
    path = "incoming_data"
    if not os.path.exists(path):
        os.makedirs(path)
        print(f"Created directory: {path}")
        logging.info(f"Created directory: {path}")
    event_handler = NewFileHandler()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=False)
    observer.start()
    print(f"Monitoring started on directory: {path}")
    logging.info(f"Monitoring started on directory: {path}")

    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
        print("Monitoring stopped.")
        logging.info("Monitoring stopped.")
    observer.join()
```

Figure 30: Code snippet to initialize directory monitoring, start observing file events, and manage termination gracefully.

Section	**Description**
Directory Setup	Ensures the directory incoming_data exists, creating it if necessary.
Event Handler Initialization	Initializes the NewFileHandler to monitor file creation events.
Observer Configuration	Sets up the Observer to watch the directory for changes.
Start Monitoring	Starts the observer and logs the monitoring activity.
Graceful Termination	Handles KeyboardInterrupt to stop the observer and cleanly exit monitoring.

References

EMBER Dataset (n.d.) EMBER is a benchmark dataset for malware detection. Available at: https://github.com/elastic/ember [Accessed: 2 October 2024].

Imbalanced-learn Documentation (n.d.) Imbalanced-learn: Techniques for imbalanced datasets. Available at: https://imbalanced-learn.org/stable [Accessed: 10 October 2024].

Joblib Documentation (n.d.) Joblib: Tools for efficient saving and loading of Python objects. Available at: https://joblib.readthedocs.io [Accessed: 20 October 2024].

LightGBM Documentation (n.d.) LightGBM: High-performance gradient boosting for large datasets. Available at: https://lightgbm.readthedocs.io/en/latest [Accessed: 10 October 2024].

Matplotlib Documentation (n.d.) Matplotlib: Creating detailed plots and visualizations. Available at: https://matplotlib.org/stable/contents.html [Accessed: 10 October 2024].

NumPy Documentation (n.d.) NumPy: Numerical computing for large datasets. Available at: https://numpy.org/doc [Accessed: 10 October 2024].

Pandas Documentation (n.d.) Pandas: Data manipulation and analysis. Available at: https://pandas.pydata.org/pandas-docs/stable [Accessed: 10 October 2024].

Scikit-learn Documentation (n.d.) Scikit-learn: Tools for data preprocessing, modeling, and evaluation. Available at: https://scikit-learn.org/stable/documentation.html [Accessed: 10 October 2024].

Seaborn Documentation (n.d.) Seaborn: Simplifying statistical data visualizations. Available at: https://seaborn.pydata.org [Accessed: 10 October 2024].

TensorFlow Documentation (n.d.) TensorFlow: Deep learning and neural network models. Available at: https://www.tensorflow.org [Accessed: 15 October 2024].

Tqdm Documentation (n.d.) Tqdm: Adding progress bars to loops in Python. Available at: https://tqdm.github.io [Accessed: 10 October 2024].

XGBoost Documentation (n.d.) XGBoost: Scalable, distributed gradient boosting. Available at: https://xgboost.readthedocs.io/en/stable [Accessed: 10 October 2024].