National College of Ireland

Configuration Manual

MSc Research Project
MSc in Cybersecurity

Ranjitha Raju
Student ID: x23307617

School of Computing
National College of Ireland

Supervisor: Vikas Sahni

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Ranjitha R

Student ID: 23307617

Programme: Msc Cyber Security **Year:** 2025

Module: Practicum

Supervisor: Vikas Sahni

Submission Due Date: 11/08/2025

Project Title: Threat Intelligence-Driven Machine Learning Framework for Predictive Ransomware Detection

Word Count: 1517 **Page Count** 10

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: Ranjitha R

Date: 11/08/2025

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

Introduction

System Requirements

- OS: Windows 11
- RAM: 8 GB (16 GB recommended)
- Disk: At least 10 GB free space for datasets and logs
- Python: Version 3.10
- Required toolkits: Zeek, Pandas, Scikit-learn, XGBoost, SMOTE (imblearn), RandomForest.

Dataset Configuration

- **CTU-13 Dataset**: Preprocessed flows required; placed under `/data/ctu13_flows.csv`
- **Threat Intelligence Feeds**:
  - IOC from **CISA JSON Feed** → stored as `iocs_cisa.json`
  - IOC from **Medusa Labs** → stored as `medusa_ioc_feed.csv`

Implementation

Loading and Previewing the CTU-13 Dataset

Step 1: The Python script is started by importing the pandas library and reading the CTU-13 .binetflow file using pd.read_csv(). The dataset, which contains labeled flow-level network data, is read from the file path /content/capture20110818 (1).binetflow, with low_memory=False to optimize memory use for mixed data types.

Step 2: After loading, the first few rows of the dataset are printed using head() to get a preview. Additionally, the script prints all column names and counts any missing (null) values using isnull().sum(), helping verify the integrity and completeness of the data before preprocessing.

```
import pandas as pd

# Load CTU-13 .binetflow file (flow-level labeled data)
ctu_df = pd.read_csv('/content/capture20110818 (1).binetflow', low_memory=False)

# Preview data
print("🔹 First 5 rows:")
print(ctu_df.head())

# Check column names and nulls
print("\n🔹 Columns:", ctu_df.columns.tolist())
print("\n🔹 Null values per column:\n", ctu_df.isnull().sum())
```

```
🔹 First 5 rows:
          StartTime       Dur Proto         SrcAddr  Sport    Dir  \
0  2011/08/18 10:21:46.633335    1.060248   tcp    93.45.239.29   1611    ->
1  2011/08/18 10:19:49.027650  279.349152   tcp  62.240.166.118   1031   <?>
2  2011/08/18 10:22:07.160628  166.390015   tcp  147.32.86.148  58067    ->
3  2011/08/18 10:26:02.052163    1.187083   tcp     147.32.3.51   3130    ->
4  2011/08/18 10:26:52.226748    0.980571   tcp  88.212.37.169   3134    ->

         DstAddr  Dport    State  sTos  dTos  TotPkts  TotBytes  SrcBytes  \
0   147.32.84.118   6881     S_RA   0.0   0.0        4       252       132
1   147.32.84.229  13363  SRPA_PA   0.0   0.0       15      1318       955
2  66.235.132.232     80    SR_SA   0.0   0.0        3       212       134
3    147.32.84.46  10010     S_RA   0.0   0.0        4       244       124
4   147.32.84.118   6881     S_RA   0.0   0.0        4       244       124

                          Label
```

Extracting and Cleaning IOCs from STIX JSON

Step 3: The script defines a function extract_iocs_fixed() that parses a STIX JSON object to extract three types of Indicators of Compromise (IOCs): IP addresses, domain names, and file hashes. It does so by using regex patterns to match the indicator fields across all objects in the STIX JSON structure.

Step 4: After extraction, the script ensures that each type of IOC is unique by converting each list into a set and back into a list. Finally, the function is applied to a variable called medusa_iocs, and the cleaned IOCs are printed in a dictionary format showing categorized threat intelligence data.

```
import re

def extract_iocs_fixed(stix_json):
    indicators = stix_json.get('objects', [])
    ioc_list = {'ip': [], 'domain': [], 'hash': []}

    for obj in indicators:
        if obj.get('type') == 'indicator':
            pattern = obj.get('pattern', '')
            # Extract IPs
            ip_match = re.findall(r"\[ipv4-addr:value *= *'([^']+)'", pattern)
            ioc_list['ip'].extend(ip_match)
            # Extract Domains
            domain_match = re.findall(r"\[domain-name:value *= *'([^']+)'", pattern)
            ioc_list['domain'].extend(domain_match)
            # Extract Hashes
            hash_match = re.findall(r"\[file:hashes\..*? *= *'([^']+)'", pattern)
            ioc_list['hash'].extend(hash_match)

    # Remove duplicates
    for key in ioc_list:
        ioc_list[key] = list(set(ioc_list[key]))

    return ioc_list

# Apply fixed function
```

Data Preprocessing and Feature Engineering

Step 5: The label column is binarized (1 for botnet traffic, 0 for benign), and Medusa ransomware-related hashes are hardcoded for simulated IOC tagging.

Step 6: Unused columns like 'StartTime' and 'Dir' are dropped. New features (Byte_Ratio and PktPerSec) are created and normalized using MinMaxScaler for model compatibility.

```
# Step 1: Binary label (botnet = 1, benign = 0)
df=ctu_df
df['Label'] = df['Label'].apply(lambda x: 1 if 'Botnet' in x else 0)

# Step 2: Extracted Medusa IOC hashes
medusa_hashes = ['45D89FE2C554D1DD2AC3A8879965B35ED7E3421F', 'AC0DCE3B0F5B8D187A2E3F29EFC

# Step 3: Add IOC Matching Feature (dummy for now since hashes not in binetflow)
df['IOC_Match'] = 0  # You can set this to 1 based on any condition or external session m

# Step 4: Encode Protocol and fill missing values
df['Proto'] = df['Proto'].astype('category').cat.codes
df = df.drop(columns=['StartTime', 'Dir'])
df = df.fillna(0)

# Step 5: Add custom features
df['Byte_Ratio'] = df['SrcBytes'] / (df['TotBytes'] + 1)
df['PktPerSec'] = df['TotPkts'] / (df['Dur'] + 1)

# Step 6: Normalize numeric fields if needed
```

Train-Test Split and Model Training

Step 7: Selected features are used to train a RandomForestClassifier using a 70–30 train-test split with fixed random seed for reproducibility.

Step 8: The model is trained on the training set and used to predict both class labels and probabilities on the test set.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_au
import matplotlib.pyplot as plt
import seaborn as sns

## Drop IP address columns and any others that are still strings
X = df.drop(columns=['Label', 'SrcAddr', 'DstAddr', 'State'])
y = df['Label']

# Now re-run the train-test split and training
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ra

model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]

# Model training
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

Model Prediction and Evaluation

Step 9: The trained Random Forest model is used to make predictions on the unseen test dataset. Both the predicted class labels (y_pred) and prediction probabilities (y_prob) are calculated.
The model's performance is assessed using:

- **Confusion Matrix**: Shows the count of true vs. predicted classes.
- **Classification Report**: Provides precision, recall, and F1-score for each class.
- **ROC-AUC Score**: Measures the model's ability to distinguish between botnet and benign flows.
- **ROC Curve**: Visually depicts the trade-off between true positive rate and false positive rate.

```
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_sco
import matplotlib.pyplot as plt

# Prediction
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]

# Evaluation
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print("ROC-AUC Score:", roc_auc_score(y_test, y_prob))

# ROC Curve
RocCurveDisplay.from_estimator(model, X_test, y_test)
plt.title("ROC Curve - Random Forest")
plt.show()
```

```
[[360866      0]
 [     5  32067]]
              precision    recall  f1-score   support
```

Simulated IOC Matching Function

Step 10: The script simulates the presence of known malicious IPs and domains (Indicators of Compromise, IOCs) by inserting synthetic values from the dataset (e.g., using an IP already in the CTU dataset) and test domain names.

Step 11: A custom function add_external_ioc_matches() is defined to check whether any records in the dataset match the provided IOCs. The function adds a binary flag (External_IOC_Match) to each record marking 1 if the IP or domain matches known threat intelligence.

The result is printed at the end, showing 7,952 matches**,** confirming the simulated IOC integration step was successful.

```
# Simulate an existing IP in CTU dataset
example_ip = df['DstAddr'].iloc[0]
medusa_ips = [example_ip]  # Ensure it's a new list or append to existin

# Simulate malicious domains for testing
medusa_domains = ['malicious.example.com', 'botnet.badsite.org']

# Define the IOC matching function if not already
def add_external_ioc_matches(df, ioc_ips, ioc_domains):
    df['External_IOC_Match'] = df['DstAddr'].isin(ioc_ips).astype(int)
    if 'Domain' in df.columns:
        df['External_IOC_Match'] |= df['Domain'].isin(ioc_domains).astyp
    return df

# Apply IOC matching
df = add_external_ioc_matches(df, medusa_ips, medusa_domains)
print("✅ IOC Matches found (after simulation):", df['External_IOC_Match
```

Evaluation & ROC Curve – Model With IOC Context

Step 12: Evaluate model with IOC feature added

- This block prints the classification report, confusion matrix, and ROC-AUC score after including the External_IOC_Match feature in the training process.
-

Step 13: Display ROC Curve

- Using RocCurveDisplay, the ROC curve is plotted and titled to reflect inclusion of IOC context, allowing visual interpretation of model performance.

```
# Step 5: Evaluation
print("\n📊 Classification Report:")
print(classification_report(y_test, y_pred))

print("🔍 Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

roc_score = roc_auc_score(y_test, y_proba)
print(f"🎯 ROC-AUC Score: {roc_score:.5f}")

# Step 6: ROC Curve
RocCurveDisplay.from_estimator(model, X_test, y_test)
plt.title("ROC Curve – With IOC Context")
plt.grid()
plt.show()
```

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

           0       1.00      1.00      1.00    360866
           1       1.00      1.00      1.00     32072

    accuracy                           1.00    392938
```

XGBoost Model Evaluation

Step 14: Run evaluation on XGBoost model

- Similar to the Random Forest section, this evaluates the XGBoost model using classification metrics and confusion matrix, confirming effectiveness (perfect scores shown).

Step 15: ROC visualization

- The ROC curve is plotted for XGBoost results with the same IOC-enriched dataset, confirming high discrimination power (ROC-AUC = 1.0).

```
# Step 5: Evaluation
print("\n📊 Classification Report (XGBoost):")
print(classification_report(y_test, y_pred))

print("🔍 Confusion Loading... ')
print(confusion_matrix(y_test, y_pred))

roc_score = roc_auc_score(y_test, y_proba)
print(f"🎯 ROC-AUC Score: {roc_score:.5f}")

# Step 6: ROC Curve
RocCurveDisplay.from_estimator(xgb_model, X_test, y_test)
plt.title("ROC Curve - XGBoost with IOC Context")
plt.grid()
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning: [04:44:23] WARNING: /work
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)

📊 Classification Report (XGBoost):
              precision    recall  f1-score   support
```

Real-Time Threat Scoring Function (Code)

Step 16: IOC & Feature Setup: Extracts key fields like source/destination IP, protocol, and label from each NetFlow record, and matches against known IOC IPs.

Step 17: Model-Based Scoring: Prepares a 1-row feature-aligned DataFrame and runs predict_proba() to get the threat score; also predicts the class label.

Step 18: Alert Triggering: If the score exceeds a threshold, or there's an IOC match, or the cumulative score crosses 2.5, an alert is printed with detailed session info.

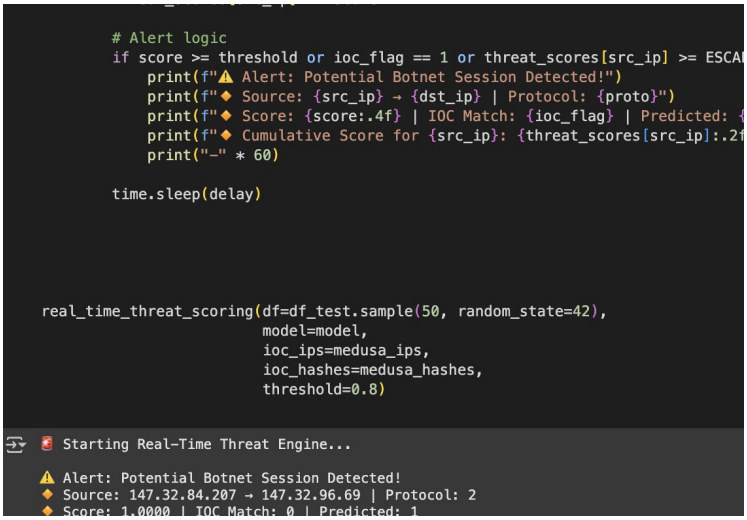
```
def real_time_threat_scoring(df, model, ioc_ips, ioc_hashes, threshold=0.8, delay=0.5):
    """
    Simulate real-time threat detection on streaming NetFlow data with threat score tracking.
    """
    from collections import defaultdict
    import numpy as np

    print("🚨 Starting Real-Time Threat Engine...\n")

    threat_scores = defaultdict(float)
    ESCALATION_THRESHOLD = 2.5

    for idx, row in df.iterrows():
        src_ip = row.get('SrcAddr', 'NA')
        dst_ip = row.get('DstAddr', 'NA')
        proto = row.get('Proto', 'NA')
        label = row.get('Label', -1)

        # Match against IOCs
        ioc_flag = int((src_ip in ioc_ips) or (dst_ip in ioc_ips))

        # Prepare sample for model prediction
        sample = row.drop(['Label', 'SrcAddr', 'DstAddr', 'Sport', 'Dport', 'State', 'Flags']
        sample = pd.DataFrame([sample])  # convert to 1-row DataFrame

        # Ensure column alignment with training features
        sample = sample.reindex(columns=model.feature_names_in_, fill_value=0)
```

Function Execution & Output

Step 19: Function Call: real_time_threat_scoring() is invoked on a 50-row sample of test data, using a trained model and provided IOCs.

Step 20: Alert Output: Real-time alerts are printed showing source/destination IPs, protocol, model prediction score, IOC match flag, and cumulative threat score.

Step 21: Live Simulation: Uses a short delay to simulate packet stream analysis in near real-time, mimicking a real-world network monitoring setup.



Zeek Log IOC Matching – Implementation Role

Step 21: Threat Intelligence Feed Integration: This code parses Zeek's conn.log file and compares IP addresses (source and destination) against a set of known Indicators of Compromise (IOCs), such as Medusa-related IPs. This bridges external threat intelligence (like Medusa IPs) with local network activity.

Step 22: Feature Enrichment for Threat Scoring: The resulting Zeek_IOC_Match flag acts as an enriched feature that can be used in your ML-based real-time scoring pipeline. It marks sessions involving suspicious IPs to improve threat model sensitivity.

Step 23: IOC-based Contextual Alerting: This module allows Zeek logs to contribute to the alerting logic (e.g., "Potential Botnet Session Detected") by identifying IOC matches in historical or live traffic logs—ensuring even non-ML threats are caught through rule-based checks.

```python
import pandas as pd

# IOC IP list (e.g., from Medusa or threat feed)
ioc_ips = {'104.81.135.169', '8.8.8.8', '64.4.54.165'}  # Example IPs; replace with medusa_ips

def parse_zeek_connlog(filepath, ioc_set):
    """
    Parses Zeek conn.log and flags IOC matches.
    """
    with open('/content/conn.log', 'r') as file:
        lines = file.readlines()

    # Skip header and comment lines (starting with '#')
    data_lines = [line for line in lines if not line.startswith("#")]

    # Get header fields from Zeek
    headers = None
    for line in lines:
        if line.startswith("#fields"):
            headers = line.strip().split("\t")[1:]
            break

    if not headers:
        raise ValueError("No '#fields' line found in Zeek log.")

    # Read data into DataFrame
    zeek_df = pd.DataFrame([line.strip().split("\t") for line in data_lines], columns=headers)
```

IOC Matching Execution:

This code runs the parse_zeek_connlog() function on the uploaded conn.log file, applying previously defined Medusa-related IPs as threat indicators. It checks for any connection where id.orig_h or id.resp_h matches those IOCs.

Step 24: Threat Session Isolation:
The filtered result ioc_hits only contains rows where the Zeek_IOC_Match flag is set to 1. These represent network sessions involving a suspicious IP, such as 104.81.135.169 or 8.8.8.8 indicating likely threat presence in DNS or HTTP traffic.

Step 25: Foundation for ML/Alerting System:
These identified sessions (sampled here using .head()) serve as the ground truth or validation layer for training, evaluating, or alerting in your pipeline enabling both real-time detection and retrospective threat hunting.

```
# ✅ Run it on your uploaded file
zeek_df = parse_zeek_connlog('/content/conn.log', ioc_ips)

# ✅ Filter and show matches
ioc_hits = zeek_df[zeek_df['Zeek_IOC_Match'] == 1]
print("✅ Sample Rows with IOC Match:")
display(ioc_hits.head())
```

✅ Sample Rows with IOC Match:

	ts	uid	id.orig_h	id.orig_p	id.resp_h	id.resp_p	proto	service	duration	orig_bytes	...	local_orig	local_resp
0	1556714128.491209	Chmpt162NcHAq0YCb	192.168.61.20	50885	104.81.135.169	80	tcp	http	0.373957	213	...	-	-
1	1556714133.054463	CP1SQx2NHC8s9Z0CBi	192.168.61.20	50883	64.4.54.165	443	tcp	-	0.058941	0	...	-	-
2	1556714128.452636	CuBvgmZmRctyexTO7	192.168.50.25	50074	8.8.8.8	53	udp	dns	0.036838	63	...	-	-
3	1556714174.956009	CBRo9C4lGZ28ds6kSj	192.168.50.25	49869	8.8.8.8	53	udp	dns	0.014214	40	...	-	-

Frequency of IOC IPs Detected in conn.log

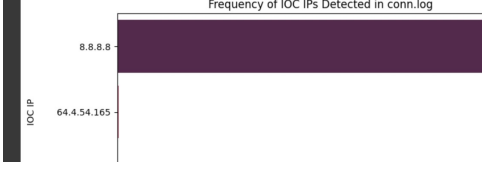
Step 26: Purpose: This plot visualizes how frequently each **IOC** appears in the Zeek

Step 27: Implementation Context: It uses the ioc_counts object (likely a value_counts() result on matched IPs) to summarize and rank the suspicious IPs based on detection volume.

Step 28: Insight Provided: Helps analysts quickly identify which IP addresses are the most frequent threats in the dataset—useful for prioritizing investigation or alerting.

```
# Bar plot of IOC IP hits
plt.figure(figsize=(8, 4))
sns.barplot(x=ioc_counts.values, y=ioc_counts.index, palette='rocket')
plt.title('Frequency of IOC IPs Detected in conn.log')
plt.xlabel('Number of Occurrences')
plt.ylabel('IOC IP')
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-34-1018570755.py:8: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.

  sns.barplot(x=ioc_counts.values, y=ioc_counts.index, palette='rocket')
```



IOC Match Distribution (Match vs. No Match)

Step 29: Purpose: This chart shows how many of the total connections had a match with the IOC list (value 1) vs. how many didn't (0).

Step 30: Implementation Context: Based on the 'Zeek_IOC_Match' column added in preprocessing (from earlier screenshots), showing binary classification results of connection logs.

Step 31: Insight Provided: Offers a **macro-level overview** of how prevalent malicious activity is in the full dataset. It also supports class balance understanding if ML models are trained later.