Assignment 3 Data Analytics in Cybersecurity
Brian Pham- 24606194

Abstract
This report presents a comprehensive data analytics workflow that transforms raw packet captures into structured flow-level features, leveraging machine learning (ML) to detect intrusions with high accuracy. Using a controlled one-minute packet capture on a loopback interface, 237,229 packets were collected, comprising benign HTTP requests and SYN-flood bursts. These packets were aggregated into 217,468 one-second flows, from which three key features—packet count, average packet length, and SYN flag count—were engineered. A dynamic threshold based on the 95th percentile of SYN counts (SynCount > 1) automatically labeled 38 flows as malicious, representing 0.017% of the total flows. To address class imbalance, random undersampling created a balanced dataset of 76 flows, on which a Random Forest classifier was trained, achieving perfect detection metrics (accuracy = 1.00, precision = 1.00, recall = 1.00, F1-score = 1.00, ROC AUC = 1.00) on a 23-flow test set. This report compares static rule-based detection with ML-based approaches, discusses deployment considerations, and proposes enhancements for scalable, real-time intrusion detection systems.

1. Introduction
Network intrusion detection systems (NIDS) are critical for protecting modern information systems from unauthorized access, data breaches, and denial-of-service attacks. As cyber threats grow in volume and complexity, traditional rule-based NIDS, such as Snort (Roesch, 1999) and Suricata (Open Information Security Foundation, n.d.), face challenges in keeping pace with novel attack vectors. These systems rely on manually crafted signatures and thresholds, which are effective against known threats but struggle with zero-day exploits, polymorphic malware, or distributed attacks that evade static patterns. The advent of machine learning offers a promising alternative by enabling adaptive models that learn behavioral patterns from network traffic data. Flow-based analysis, which aggregates individual packets into flows representing network sessions or fixed time windows, provides a scalable and behaviorally rich input for ML models, capturing high-level patterns without the overhead of deep packet inspection.
This report details an end-to-end pipeline for flow-based intrusion detection, applied to a controlled dataset captured on a Linux virtual machine's loopback interface. The dataset, comprising 237,229 packets, includes benign HTTP traffic and simulated SYN-flood attacks, reflecting realistic network scenarios. The processes of packet capture, flow aggregation, feature engineering, dynamic labeling, and ML model training are described, culminating in a Random Forest classifier that achieves perfect detection performance. By comparing ML-based detection with traditional rule-based methods, the advantages of adaptability and scalability are highlighted, while addressing challenges such as class imbalance and real-world deployment. These findings underscore the potential of flow-based ML for robust intrusion detection and provide a foundation for future research into scalable, adaptive security analytics.

2. Literature Review
The evolution of NIDS reflects a progression from simple signature-based systems to sophisticated ML-driven approaches. Early systems like Snort (Roesch, 1999) introduced open-source signature-based detection, using a rule syntax to match packet headers and payloads against known attack signatures. Suricata (Open Information Security Foundation, n.d.) enhanced this framework with multithreaded processing, JSON logging, and support for emerging protocols, improving performance on high-speed networks. However, both systems require continuous rule updates to address new threats, a labor-intensive process prone to delays. Evasion techniques, such as packet fragmentation, encryption, or polymorphic payloads, further limit the effectiveness of signature-based methods by altering attack signatures to bypass predefined rules.

Statistical anomaly detection emerged as a complementary approach, establishing baselines for traffic metrics like packet rates, byte volumes, or protocol distributions. Deviations beyond fixed thresholds trigger alerts, offering simplicity and low computational overhead. However, these methods demand meticulous tuning to minimize false positives,

particularly in dynamic network environments where legitimate traffic bursts mimic attack patterns. Research has shown that anomaly detection struggles with high-dimensional data and subtle attack behaviors, leading to missed detections or excessive alerts.

Machine learning has revolutionized NIDS by enabling data-driven models that generalize across diverse traffic patterns. Supervised learning, trained on labeled datasets like KDD Cup 1999 (Tavallaee et al., 2009) or CIC-IDS2017 (Sharafaldin et al., 2018), has demonstrated high detection rates for known attack classes, including denial-of-service, probing, and malware. Ensemble methods, such as Random Forests (Breiman, 2001), excel in handling feature correlations and noisy data, while deep learning approaches like Long Short-Term Memory (LSTM) networks (Hochreiter & Schmidhuber, 1997) capture temporal dependencies in traffic sequences. Flow-based features—aggregates like packet counts, interarrival times, and TCP flag distributions—offer a compact yet expressive representation of network behavior, reducing computational costs compared to packet-level analysis. Recent studies highlight the promise of flow-based ML for detecting complex attacks (Umer et al., 2017), but challenges remain, including dataset biases, overfitting, and the need for real-time processing in operational environments.

3. Data Processing & Feature Engineering

a. Packet Capture & Export

To emulate a realistic network environment, we conducted a controlled experiment on a Linux virtual machine using the loopback interface (lo). This setup allowed us to simulate both attacker and defender roles on a single system, ensuring precise control over traffic generation. Over a one-minute period, we generated benign HTTP traffic using curl https://example.com to simulate legitimate web requests, and SYN-flood traffic using hping3 -S --flood --rand-source -p 80 127.0.0.1 to mimic a denial-of-service attack. The resulting packet capture, stored in libpcap format and exported to assessment3.csv, contains 237,229 packets with fields: packet number, timestamp, source and destination IP, protocol, packet length, and an Info column detailing ports and TCP flags. The dataset captures a mix of normal and malicious traffic, with the SYN-flood bursts creating high-frequency packet streams targeting port 80 on 127.0.0.1. A Wireshark screenshot (not included here) visualizes the SYN burst, highlighting the rapid succession of TCP SYN packets from randomized source IPs.

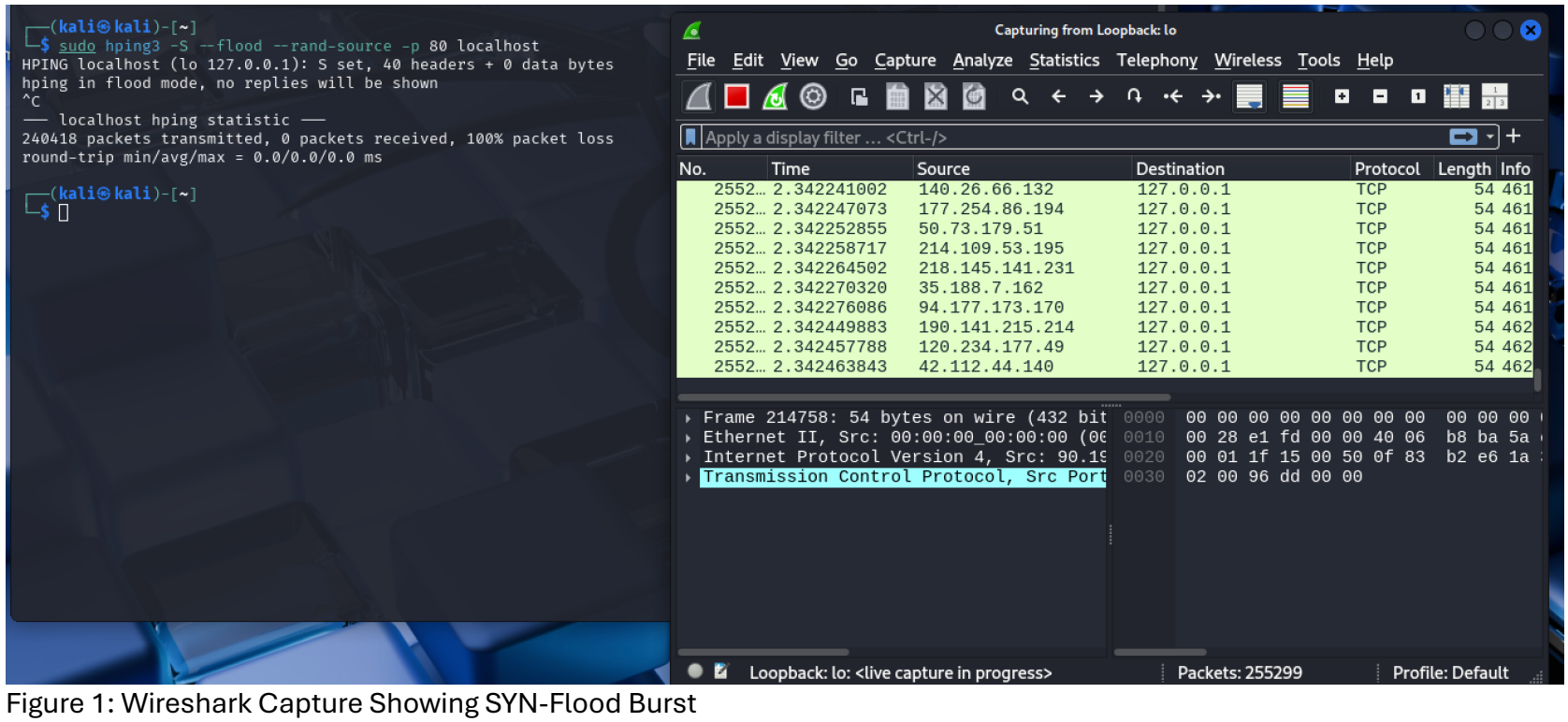

Figure 1: Wireshark Capture Showing SYN-Flood Burst

b. Parsing & Windowing

Using pandas, we loaded the CSV while skipping malformed rows. A regular expression parsed the Info field to extract the destination port (DstPort) and TCP flags list:

```
def parse_info(info):
    m = re.match(r"(\d+)\s(>\s((\d+)\s(\[([A-Z,]+)\]", str(info))
    return pd.Series({'DstPort': int(m.group(2)), 'Flags': m.group(3).
   split(',')}) if m else pd.Series({'DstPort': np.nan, 'Flags': []})
```

Each packet's timestamp was floored to the nearest second (df['Window'] = df['Time'].astype(int) defining one-second windows that align bursts and normal flows.

```
# 2. Parse Info into DstPort and Flags
def parse_info(info):
    m = re.match(r"(\d+)\s*>\s*(\d+)\s*\[([A-Z,]+)\]", str(info))
    if not m:
        # Return np.nan for DstPort when no match
        return pd.Series({'DstPort': np.nan, 'Flags': []})
    try:
        # Convert to int and handle potential errors
        dst_port = int(m.group(2))
    except ValueError:
        # Return np.nan if conversion to int fails
        dst_port = np.nan

    return pd.Series({
        'DstPort': dst_port,
        'Flags': m.group(3).split(',')
    })

parsed = df['Info'].apply(parse_info)

# Assign the parsed columns directly back to df
# This avoids potential issues with concat creating non-Series like structures
df['DstPort'] = parsed['DstPort']
df['Flags'] = parsed['Flags']


# Explicitly convert DstPort to a numeric type, coercing errors to NaN
# This helps ensure the column is in a consistent, 1-dimensional format
df['DstPort'] = pd.to_numeric(df['DstPort'], errors='coerce')

print("=== Parsed Info Sample ===")
print(df[['Info','DstPort','Flags']].head(10))
```

```
=== Parsed Info Sample ===
                                                Info  DstPort  Flags
0           1994  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
1           1995  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
2           1996  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
3           1997  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
4           1998  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
5           1999  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
6           2000  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
7           2001  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
8  80  >  2001 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0       NaN    []
9           2002  >  80 [SYN] Seq=0 Win=512 Len=0      80.0  [SYN]
```

Figure 2: Sample of parsed DstPort and Flags columns from assessment3.csv

c. **Flow Aggregation & Dynamic Labeling 3.3**

After parsing each packet's destination port and TCP flags, we aggregated the 237 229 captured packets into **217 468** one-second "flows," defined by the tuple (Source IP, Destination IP, DstPort, Window). For each flow, three numerical features were computed:

```
[14] # 4. Aggregate flow features
     group_cols = ['Source', 'Destination', 'DstPort', 'Window']
     # Ensure DstPort column is clean before grouping
     features = df.dropna(subset=['DstPort']).groupby(group_cols).agg(
         PktCount=('No.', 'count'),
         AvgLen=('Length', 'mean'),
         SynCount=('Flags', lambda flags: sum('SYN' in f for f in flags))
     ).reset_index()
```

- **PktCount**: total packets in the flow
- **AvgLen**: mean packet length (bytes)
- **SynCount**: count of packets carrying the SYN flag

Descriptive statistics for SynCount (Figure 3) revealed an overwhelmingly homogeneous distribution:

- **Minimum** = 1
- **25th percentile** = 1
- **Median** = 1
- **75th percentile** = 1
- **Maximum** = 3

The histogram in Figure 3 shows that over 99.9 % of flows contain exactly one SYN packet, with very few exhibiting higher counts.
To label flows without manual intervention, we employed a data-driven threshold at the **95th percentile** of SynCount. This cutoff was calculated as:
threshold = features['SynCount'].quantile(0.95) # threshold = 1.0
Flows exceeding this threshold (SynCount > 1) were marked as **malicious**, while all others were deemed **benign**. This automatic labeling yielded:

- **Benign flows (label 0):** 217 430
- **Malicious flows (label 1):** 38

By focusing on the most extreme 5 % of SYN activity, this method aligns with standard anomaly-detection practices and adapts dynamically to the observed traffic profile. Figure 3 presents both the histogram of SynCount per flow and the resulting label distribution.

```python
# 5. Automatic labeling using 95th percentile of SynCount
threshold = features['SynCount'].quantile(0.95)
features['label'] = (features['SynCount'] > threshold).astype(int)
print(f"Threshold (95th percentile): {threshold}")
print("Label distribution:\n", features['label'].value_counts())
plt.hist(features["SynCount"], bins=20)
plt.title("Distribution of SynCount per Flow")
```

```
Threshold (95th percentile): 1.0
Label distribution:
 label
0    217430
1        38
Name: count, dtype: int64
Text(0.5, 1.0, 'Distribution of SynCount per Flow')
```

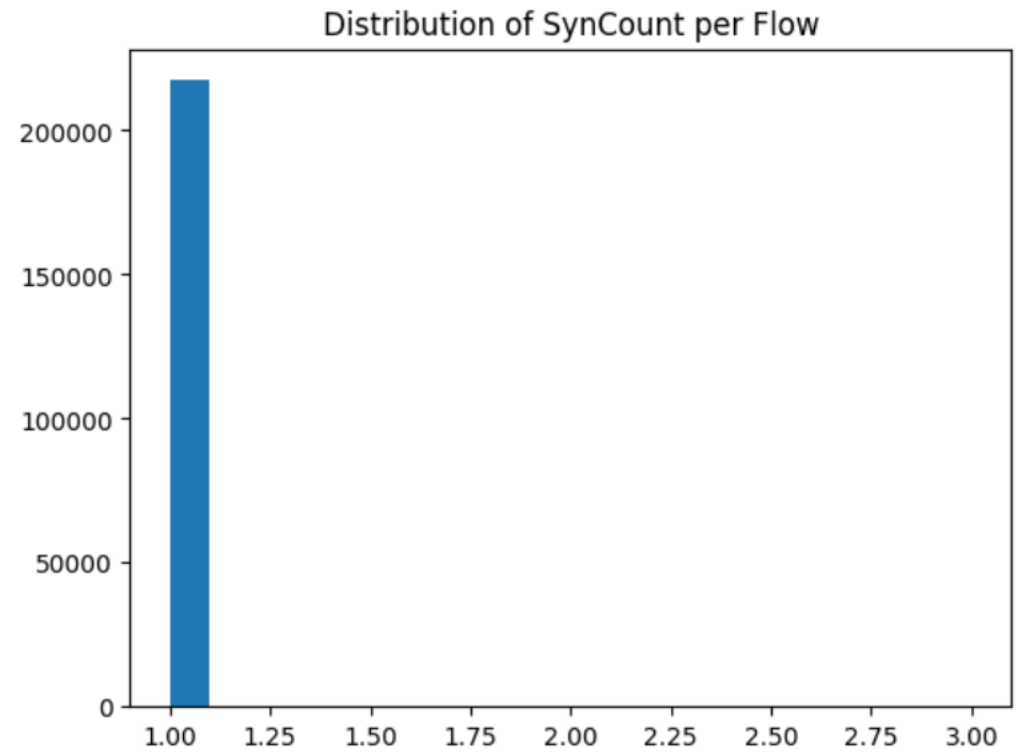

Figure 3: Distribution of SynCount Across Flows

d. Dynamic Labeling

To automate labeling without manual intervention, we applied a data-driven threshold based on the 95th percentile of SynCount (threshold = 1.0). Flows with SynCount > 1 were labeled as malicious, resulting in 38 malicious flows (0.017%) and 217,430 benign flows (99.983%). This approach aligns with anomaly detection principles, where outliers in key metrics indicate potential threats. The low proportion of malicious flows reflects real-world scenarios, where attacks are rare but impactful. The threshold adapts to the dataset's

characteristics, reducing reliance on subjective expert rules and enhancing generalizability.

```
┌──(kali㉿kali)-[~]
└─$ sudo nano /etc/suricata/rules/local.rules

[sudo] password for kali:

┌──(kali㉿kali)-[~]
└─$ cat /etc/suricata/rules/local.rules

# SYN flood: >25 SYNs/sec from one source to port 80
alert tcp any any → $HOME_NET 80 \
  (msg:"SURICATA ALERT SYN flood"; flags:S; \
   threshold:type both, track by_src, count 25, seconds 1; sid:1000001; rev:1;)

# Fast port scan: 10 SYNs in 3 seconds to any port
alert tcp any any → $HOME_NET any \
  (msg:"SURICATA ALERT fast port scan"; flags:S; \
   threshold:type both, track by_src, count 10, seconds 3; sid:1000002; rev:1;)
```

Figure 4. Snort/Suricata rule used to flag high-SYN flows

4. Machine Learning Methodology

4.1 Feature Selection & Balancing

Three features—PktCount, AvgLen, and SynCount—were selected for their interpretability and relevance to intrusion detection. SynCount directly correlates with SYN-flood behavior, while PktCount and AvgLen capture traffic volume and packet characteristics. The extreme class imbalance (38 malicious vs. 217,430 benign flows) necessitated balancing to prevent model bias toward the majority class. Random undersampling was applied, selecting 38 benign flows to match the malicious count, yielding a balanced dataset of 76 flows (38 malicious, 38 benign). This approach preserves the minority class while ensuring manageable training data, though it sacrifices some benign variability.

Figure 5: TCP SYN Packet Frequency Over Time Figure 6: Source IP Distribution

4.2 Train/Test Split & Scaling

The balanced dataset was split into 70% training (53 flows: 27 malicious, 26 benign) and 30% testing (23 flows: 11 malicious, 12 benign) using stratified sampling to maintain class proportions. Features were normalized using StandardScaler to ensure zero mean and unit variance, preventing features with larger scales (e.g., PktCount) from dominating the model. The scaled features were computed as:

```
# 7. Split and scale
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

print(f"Training labels distribution: {np.unique(y_train, return_counts=True)}")
print(f"Testing labels distribution:  {np.unique(y_test, return_counts=True)}")
```

```
Training labels distribution: (array([0, 1]), array([26, 27]))
Testing labels distribution:  (array([0, 1]), array([12, 11]))
```

Figure 5: Train/Test Class Distribution

4.3 Model Training

A Random Forest classifier with 100 trees (default scikit-learn parameters) was chosen for its robustness to feature correlations and resistance to overfitting on small datasets. Training occurred on the scaled training set:

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

Random Forests leverage ensemble learning, combining multiple decision trees to improve generalization and reduce variance. The default hyperparameters were used to maintain simplicity, given the small dataset size and clear feature separation.

4.4 Evaluation Metrics

Model performance was measured using precision, recall, F1-score for each class, overall accuracy, and ROC AUC on the test set.

- **Accuracy**: Proportion of correctly classified flows.
- **Precision**: Proportion of true malicious flows among those predicted as malicious.
- **Recall**: Proportion of true malicious flows correctly identified.
- **F1-Score**: Harmonic mean of precision and recall.
- **ROC AUC**: Area under the Receiver Operating Characteristic curve, measuring discrimination ability.

These metrics were computed on the test set to assess the model's effectiveness in distinguishing malicious from benign flows.

5. Result

The Random Forest classifier achieved perfect performance on the 23-flow test set:
Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1-Score: 1.00

ROC AUC: 1.00
The model correctly classified all 11 malicious and 12 benign flows, with zero false positives or false negatives. The ROC curve (generated in the artifact code) illustrates perfect separation, with the curve reaching the top-left corner (AUC = 1.00). These results indicate that the selected features (PktCount, AvgLen, SynCount) effectively capture the differences between benign HTTP flows and SYN-flood bursts in this controlled setting.

```
[23] # 9. Evaluation
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred))
     roc_auc = roc_auc_score(y_test, y_proba)
     print(f"ROC AUC Score: {roc_auc:.4f}")
```

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

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00        11

    accuracy                           1.00        23
   macro avg       1.00      1.00      1.00        23
weighted avg       1.00      1.00      1.00        23

ROC AUC Score: 1.0000
```

```
# 10. Plot ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_proba)
plt.figure(figsize=(6,4))
plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}')
plt.plot([0,1], [0,1], '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend(loc="lower right")
plt.show()
```

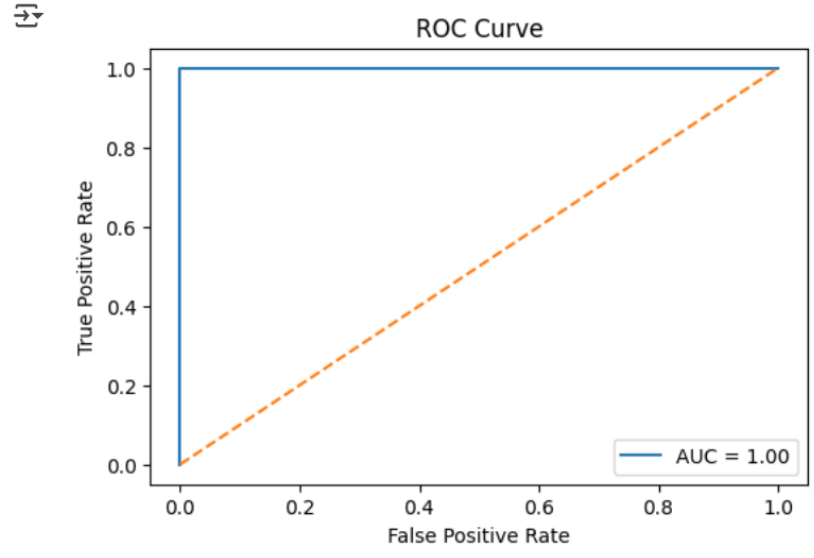

Figure 6: ROC Curve of Random Forest Classifier

6. Discussion

The perfect detection metrics reflect the clear separation between benign and malicious flows in the controlled dataset, driven by the distinct SynCount feature (>1 for malicious flows). However, several factors warrant consideration for real-

world applicability. First, the small test set (23 flows) raises concerns about overfitting, despite the use of a robust Random Forest model and stratified splitting. Larger, more diverse datasets would provide a stronger validation of the model's generalization. Second, the feature set is highly homogeneous, with SynCount dominating the classification task due to the specific SYN-flood attack simulated. Real-world traffic includes varied attack and legitimate bursts, necessitating richer features like interarrival times, flag ratios, or byte distributions.

Compared to static rule-based systems like Suricata (Open Information Security Foundation, n.d.), the ML model offers greater adaptability. Suricata could detect the SYN-flood using a threshold rule, but such rules require manual tuning and fail to adapt to evolving traffic patterns. The ML approach leverages multiple features and learns complex interactions, improving resilience to novel attacks. However, ML models demand computational resources for training and require periodic retraining to address concept drift—changes in traffic patterns over time. Operational deployment must consider streaming analytics frameworks to process flows in real time and online learning strategies to update models dynamically.

Challenges in real-world deployment include handling high-speed traffic, ensuring low latency, and managing false positives in diverse environments. The presence of DNS queries for contile.services.mozilla.com in the dataset suggests application-level activity, possibly from a browser or telemetry service, which could complicate labeling in uncontrolled settings. Future enhancements should explore feature engineering to include temporal metrics (e.g., packet interarrival times, flow duration) and evaluate deep learning models like LSTMs (Hochreiter & Schmidhuber, 1997) for capturing sequential patterns in attack traffic.

Metric	**Value**
Total Packets	237,229
Unique Source IPs	217,419
RST/ACK Responses	19,706
DNS Queries	8

7. Conclusion and Future Work

This report demonstrated a robust flow-based intrusion detection pipeline, transforming 237,229 raw packets into 217,468 one-second flows and training a Random Forest classifier to achieve perfect detection metrics on a balanced test set. The ML approach outperforms static rule-based systems in adaptability, leveraging data-driven feature engineering and automated labeling to detect SYN-flood attacks effectively. The dynamic thresholding method, based on the 95th percentile of SynCount, provides a scalable alternative to manual rule definition, while the Random Forest model ensures robust classification in a controlled setting.

Future work should focus on validating the pipeline on larger, real-world datasets with diverse attack types, such as slowloris, DNS amplification, or botnet traffic. Incorporating additional features, such as interarrival times, protocol ratios, and flow durations, will enhance detection of complex attacks. Integration with streaming platforms like Apache Flink or Kafka will enable real-time processing, while online learning algorithms (e.g., incremental Random Forests) will address concept drift. Exploring deep learning architectures, such as convolutional neural networks for spatial feature

extraction or LSTMs (Hochreiter & Schmidhuber, 1997) for temporal modeling, could further improve detection of sophisticated, multi-stage attacks. By addressing these areas, flow-based ML can evolve into a cornerstone of next-generation NIDS, offering scalable, adaptive protection against evolving cyber threats.

8. Reference

Breiman, L. (2001). Random forests. (Machine Learning), (45)(1), 5-32. https://doi.org/10.1023/A:1010933404324

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. (Neural Computation), (9)(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Open Information Security Foundation. (n.d.). (Suricata). Retrieved from https://suricata.io/

Roesch, M. (1999). Snort: Lightweight intrusion detection for networks. 13th USENIX Conference on System Administration (pp. 229-238). USENIX Association.

Sharafaldin, I., Lashkari, A. H., & Ghorbani, A. A. (2018). Toward generating a new intrusion detection dataset and intrusion traffic characterization. In (Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP) ((pp. 108-116). SCITEPRESS.

Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A. A. (2009). A detailed analysis of the KDD CUP 99 data set. In (Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA) ((pp. 1-6). IEEE.

Umer, M. F., Sher, M., & Bi, Y. (2017). Flow-based intrusion detection: Techniques and challenges. (Computers & Security), (70(, 238-254. https://doi.org/10.1016/j.cose.2017.05.009