**Real-Time Threat Detection Using Machine Learning and the CICIDS 2017 Dataset**

**Sultan Taj,** [**tajs@sunypoly.edu**](mailto:tajs@sunypoly.edu)

**State University of New York Polytechnic Institute**

**Utica, NY 13502, United States**

**Abstract –** As cyberattacks continue to evolve, detecting and mitigating zero-day threats remains a critical challenge. This project explores the development of a real-time threat detection system using machine learning models trained on the CICIDS2017 dataset. The study focuses on developing an end-to-end intrusion detection pipeline encompassing data preprocessing, model development, deployment, and evaluation. A range of machine learning models were included, such as Random Forest, CNN, and FCNN models, and they were evaluated for multi-class and binary classification to detect and classify malicious network traffic. A flask API was utilized for model deployment, enabling real-time inference with a simulated network environment. Results highlight the strengths of binary classification for real-world applications, overcoming limitations in detecting rare attack challenges. Despite challenges in traffic generation and feature extraction, this work underscores the potential for machine learning systems to enhance network security against evolving cyber threats. Future efforts will explore expanded network simulations, cloud-based API deployment, and advanced ensemble methods to refine the system further.

1. **Introduction**
2. *Enhancing Network Security Against Zero-Day Attacks*

In today’s digital landscape, network security faces a relentless challenge from zero-day attacks, exploits that target vulnerabilities before they are identified or patched. These attacks pose significant risks to industries such as healthcare, manufacturing, and finance, where the safety of sensitive data and critical operations are essential. The ability of zero-day attacks to bypass traditional security measures highlights the urgent need for innovative, adaptive defense mechanisms. This project focuses on exploring the feasibility of employing machine learning techniques to develop a real-time threat detection tool that strengthens network defenses against such advanced threats.

1. *Dataset*

To develop and validate the proposed threat detection tool, the CICIDS2017 dataset was utilized. Created in a controlled environment, the dataset provides a comprehensive collection of network traffic data, including both benign activities, and a diverse range of simulated malicious attacks such as Denial-of-Service (DoS), Botnet activity, as well as a series of web attacks like Cross-Site-Scripting (XSS) and SQL Injection. The dataset provides a realistic foundation for training machine learning models due to its detailed labeling and accurate representation of modern network traffic.

For this project, a Kaggle-preprocessed version of CICIDS2017 by Laurens D’Hooge was utilized. This version features cleaned and normalized data stored in Parquet files. This was utilized for its:

* Storage efficiency: Reducing dataset size through compression.
* Faster processing speeds: Enabling quicker access to features during analysis.
* Scalability: Suitable for handling large-scale datasets in machine learning workflows.

Using this preprocessed version of the dataset, models exhibited improved performance during model development, yielding higher accuracy [2], which is needed for a real-time threat detection tool.

1. *Current work*

Over the years, researchers and practitioners have extensively analyzed the CICIDS2017 dataset to develop more efficient machine learning models for network intrusion detection. Prior efforts include addressing class imbalances, applying various classification algorithms such as Random Forest, Support Vector Machines, and Convolutional Neural Networks, and fine-tuning hyperparameters to maximize accuracy. Kaggle contributors have further enriched the research by sharing notebooks showcasing preprocessing innovations and model optimizations. Despite these efforts, most research has focused on achieving static accuracy improvements with limited exploration of real time deployment for live network environment.

1. *Contribution*

Building upon prior research, this research aims to develop a highly accurate model capable of distinguishing between malicious and benign traffic and deploying the model for real-time threat detection. By integrating a Flask API endpoint, the system evaluates incoming network data dynamically, enabling live monitoring in simulated network environments. Trained with the CICIDS2017, this project bridges the gap between academic research and practical implementation, combining robust machine learning techniques with real-world applicability to enhance network security.

1. **Literature Review and Limitations**

*A. Troubleshooting an Intrusion Detection Dataset*

Engelen, Rimmer, and Joosen provided a critical examination of the CICIDS2017 dataset, highlighting its strengths and weaknesses. Their insights into dataset inconsistencies, such as class imbalance and the presence of metadata that could lead to shortcut learning, directly influenced the preprocessed and cleaning stages of this project. This resource reinforced the importance of careful data handling to ensure model performance and validity.

*B. Laurens D’Hooge’s CICIDS2017 Kaggle Notebook*

D’Hooge’s work on cleaning the CICIDS2017 dataset demonstrated practical methods for improving data quality, such as downsizing data types, removing duplicates, and addressing missing values. His preprocessing techniques became the foundation for my own cleaning pipeline, offering valuable guidance on preparing the dataset for machine learning applications.

*C. Evaluation of Machine Learning Techniques for Traffic Flow-Based Intrusion Detection*

Rodríguez and colleagues compared various machine learning techniques for intrusion detection, emphasizing Random Forest’s effectiveness in accurately classifying attack types. This study validated my decision to use Random Forest as a baseline model and inspired further experimentation with advanced sampling techniques to improve rare class detection.

*D. Angela Rentsi’s CICIDS2017 CNN Projects*

Rentsi’s work on using CNNs for the CICIDS2017 dataset highlighted the potential of convolutional neural networks in extracting complex spatial features from network traffic data. Her implementation and summary served as a reference point for developing my own CNN and FCNN models, offering key insights into hyperparameter tuning and architecture design.

*E. Nolovelost’s CNN Implementation*

This notebook explored the application of CNNs for intrusion detection with CICIDS2017, providing a detailed breakdown of architectural choices and performance metrics. It guided my understanding of how CNNs could be leveraged to detect rare attack classes while maintaining overall accuracy.

*F. A Novel Multi-Stage Approach for Hierarchical Intrusion Detection*

Verkerken and Lall proposed an innovative hierarchical approach to intrusion detection, demonstrating the benefits of combining machine learning models with ensemble techniques. Their findings informed my decision to experiment with a voting ensemble, as it offered the potential to boost model robustness and accuracy by leveraging the strengths of multiple classifiers.

*G. Enhancing Real-Time Detection Using a Multi-Model Approach*

Johnson, Schrieber, and Kim emphasized the importance of real-time threat detection and proposed a multi-model approach using CICIDS2017. It underscored the feasibility of deploying ensemble models for dynamic environments, directly influencing the design of my Flask API for live packet inference.

1. **Methodology**

When approaching this project, three critical areas were identified to ensure its practicality: Model Development, Model Deployment, and Real-Time Data Capture. This comprehensive approach included experimenting with machine learning and deep learning techniques to create a model capable of accurately detecting and classifying network traffic anomalies. Deployment was implemented through a Flask API endpoint to facilitate real-time inference, while real-time packet capture was designed to simulate realistic network environments.

1. *Dataset Overview*

The CICIDS2017 dataset, created by the Canadian Institute for Cybersecurity, serves as a benchmark dataset for intrusion detection research. It contains network traffic labeled as either benign or one of several attack types. The dataset is composed of 8 separate CSV files, each containing their own type of network traffic.

* Monday: Benign traffic
* Tuesday: Brute Force (FTP / SSH)
* Wednesday: DoS
* Thursday Morning: Web Attacks (Brute Force / XSS / SQL Injection)
* Thursday Afternoon: Infiltration (Dropbox download)
* Friday Morning: Botnet
* Friday Afternoon: Port Scan
* Friday Afternoon: DDoS

When all these datasets are combined, they contain 2,203,743 rows and 79 columns. This makes the CICIDS2017 a large and heavily imbalanced dataset that is prone to overfitting. For example, the benign class comprises over 1.5 million samples, while attack types like SSH-Brute Force, or DoS only has around 5000 samples. Even worse, the more modern attacks we see now have such little samples with XSS having 652, and Infiltration and SQL Injection having 36 and 21, respectively. Given this imbalance and the dataset’s scale, preprocessing is essential to ensure balanced and effective training.

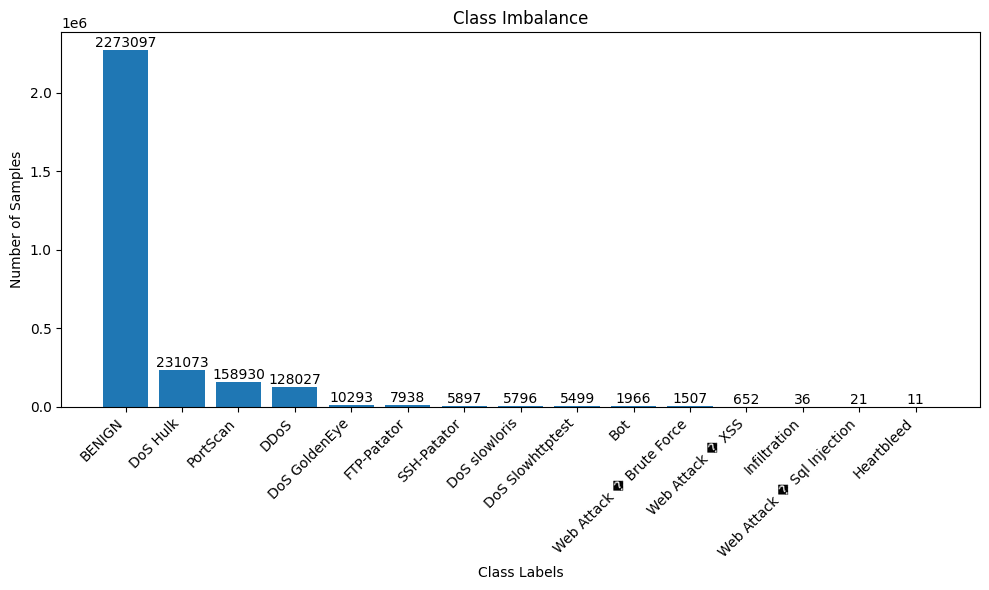


Fig. 1. *Table showcasing class imbalance of raw dataset*

1. *Data Preprocessing*

*i. How D’Hooge cleaned the CICIDS2017 dataset*

Laurens D’Hooge created a notebook where he refined the CICIDS2017 dataset, significantly improving performance metrics and achieving up to 96% balanced accuracy in his hierarchical intrusion detection approach. His cleaning process began by removing unnecessary metadata columns, such as Flow ID, Source IP, Destination IP, Source Port, Destination Port, and Timestamp, which are not informative for model learning and can lead to shortcut learning. Following this, he optimized the dataset by converting columns with default types like float64, int64, or object (str) into more efficient data types, thereby reducing memory usage. Finally, he removed all duplicate entries and handled missing values (e.g., NaNs) to minimize training bias and ensure the integrity of the dataset.

*ii. Initial Cleaning*

To facilitate efficient processing and reduce computational overhead, a15% sample of the combined dataset was selected, maintaining a representative sample while improving loading and processing times. This allowed for quicker iterations and testing during preprocessing and model development. The next set of steps involved ensuring the dataset was clean and ready for training. This included checking for any missing or duplicate values. As preprocessing occurred, there were not have any missing values, but there were some duplicate rows, and they were identified then removed to prevent training bias.

*iii. Class Balancing*

An analysis of the dataset’s label distribution revealed significant class imbalance, with some attack classes (ex., Infiltration, SQL Injection, XSS, and Botnet) containing too few samples to be effectively detected.

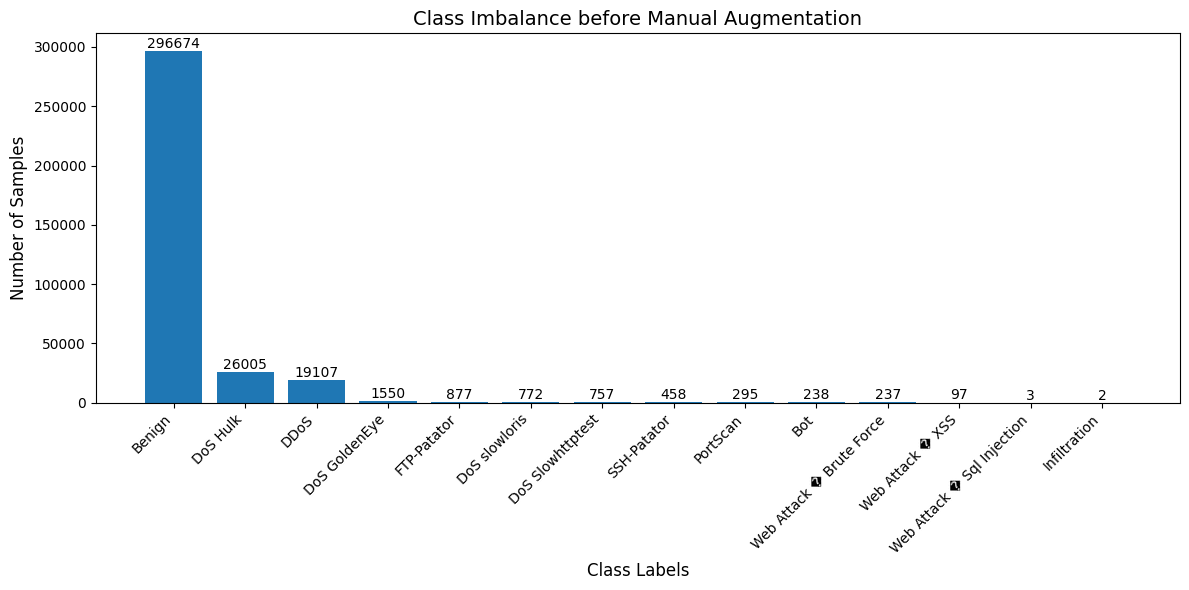
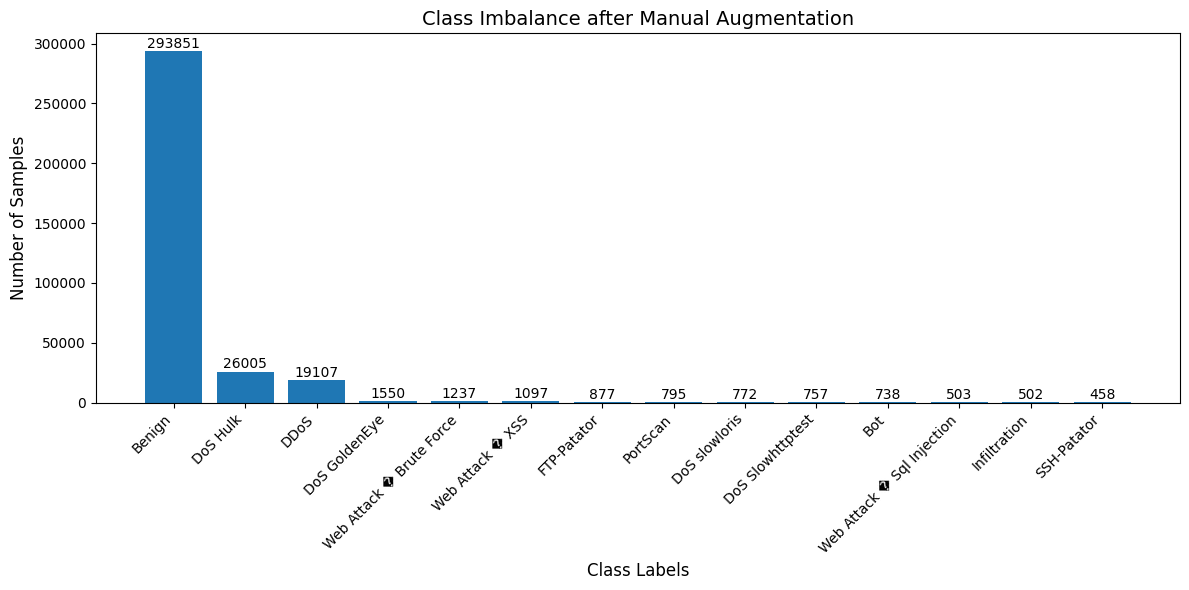
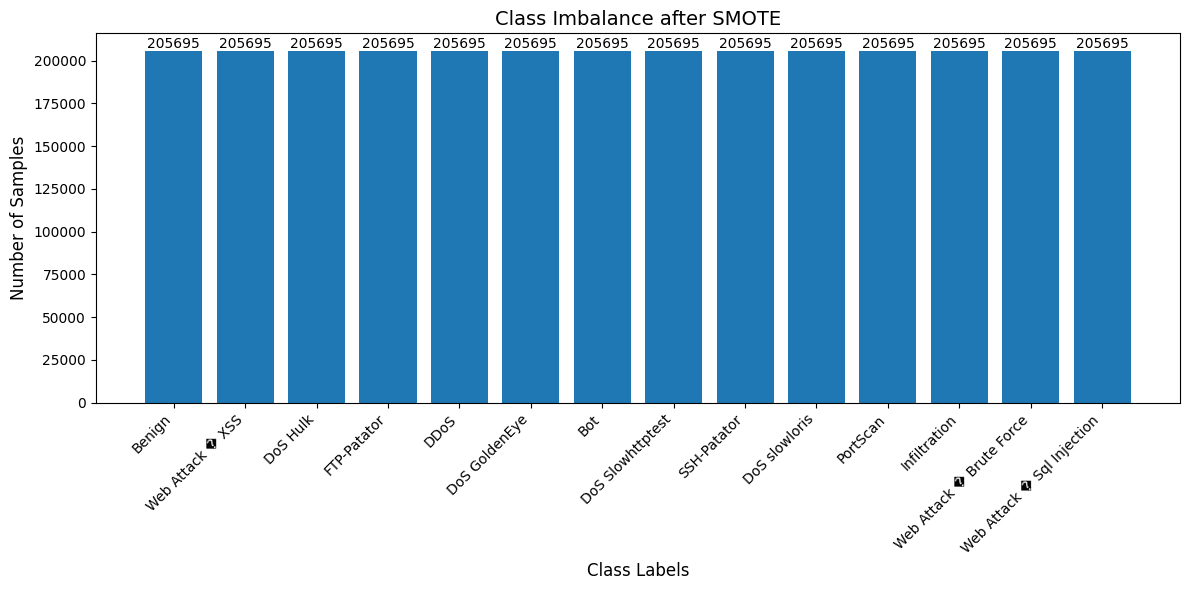


Fig. 2. Table showcasing class imbalance before Manual Augmentation

The scarcity posed challenges for both direct model training and oversampling techniques like SMOTE, which require a minimum number of samples for effective augmentation. To address this, manual augmentation was applied to rare classes, artificially increasing their sample sizes by 500 to ensure they were adequately represented in the dataset.

Fig. 3. Table showcasing class imbalance after Manual Augmentation

This augmentation step laid the groundwork for using SMOTE (Synthetic Minority Oversampling Technique), which was then employed to balance the dataset further by generating synthetic samples for the rare classes. This two-step approach effectively mitigated the class imbalance, ensuring all labels were adequately represented for training.

Fig. 4. Table showcasing balanced class after SMOTE

1. *Model Development*
2. *Random Forest*

Random Forest (RF) was chosen as the baseline model for this project due to its proven effectiveness in intrusion detection research. As an ensemble learning method, RF reduces overfitting and enhances model stability by aggregating predictions from multiple decision trees. This approach excels in handling high-dimensional data and provides interpretability through feature importance analysis, making it an ideal starting point.

1. *Deep Learning*

While traditional machine learning models like RF offer simplicity and interpretability, deep learning models have the potential to uncover complex patterns in the data. For this project, Convolutional Neural Networks (CNN) and Fully Connected Neural Networks (FCNN) were explored to assess their ability to:

* Detect subtle variations and anomalies in network traffic features
* Generalize well across diverse attack types with proper tuning.

CNNs are widely recognized for their ability to extract spatial and temporal patterns from input features, making them an ideal choice for intrusion detection based on network traffic flows. Studies such as Rodríguez et al. (2022) emphasize the effectiveness of CNNs handling large-scale datasets efficiently due to their parameter-sharing mechanisms. Provides robust detection for attack classes by identifying subtle anomalies, supported by Rentsi’s work.

FCNNs are designed to learn from fully connected feature representations. FCNNs were chosen due to their ability to directly model the relationships between all input features without spatial constraints as noted in Johnson et al. (2023). It is also meant to serve as a comparison to CNNs in understanding whether feature interactions or spatial correlations contribute more to detecting specific attack types.

1. *Voting Ensemble*

Voting ensembles combine the predictions of multiple models to improve accuracy and robustness. For this project, a soft voting ensemble was chosen, integrating RF, CNN, and FCNN models. This approach combines the complementary strengths of each of these models, with RF’s efficiency and interpretability, CNNs spatial pattern recognition, and FCNN’s complex feature interactions. Studies including Johnson et al. (2023) and Rodríguez et al. (2022) emphasize the ability of multi-model approaches to improve detection rates for rare attack types while maintaining high accuracy for majority of classes. This approach can also reduce individual model biases and ensure scalability for deployment, providing and balanced and adaptable solution for intrusion detection.

1. *Model Deployment*

To deploy the intrusion detection model, Flask was used to create a lightweight web server capable of hosting the saved model. The Flask Script initializes a local server, loads the pre-trained model with joblib, and provides a prediction route (/predict) to process incoming JSON-formatted packets. These packets are validated, processed and passed to the model for inference, returning whether the packet is benign or malicious.

The deployment was tested using Postman by sending sample JSON packets extracted from the dataset, ensuring accurate predictions and proper functionality. This setup provides an accessible foundation for real-time intrusion detection when integrated with live data.

1. *Real-Time Packet Capture*

Real-time packet capture has the potential to advance intrusion detection research by bridging the gap between static dataset-based models and dynamic network environments. Using PyShark, this project aimed to capture live network traffic, converting raw packets into flow-level features, and processing them for threat detection using a trained machine learning model. This approach could simulate realistic network conditions, providing opportunities for validating and improving model performance in production-like environments.

However, the complexity of live network environments requires careful consideration of factors such as traffic volume, diversity, and feature completeness. Future efforts to optimize virtual environments, generate meaningful traffic, and address data completeness challenges will be crucial for full realizing the benefits of real-time threat detection.

1. **Discussion and Results**
2. *Model Performance*

As mentioned previously, the models used for this experiment were Random Forest, Fully Connected Neural Network, and Convolutional Neural Network. Initially when first writing out these models, a multi-classification approach was employed to ensure that the model can discern between different types of attacks. Especially with attacks nowadays being so diverse, having a model that can discern between the different types of malicious attack would make the model more useful for our real-world application.

First, we will look at our results with the Random Forest model. RF showed exceptional performance in the multi-class setting achieving high accuracy and F1-scores across most attack types.

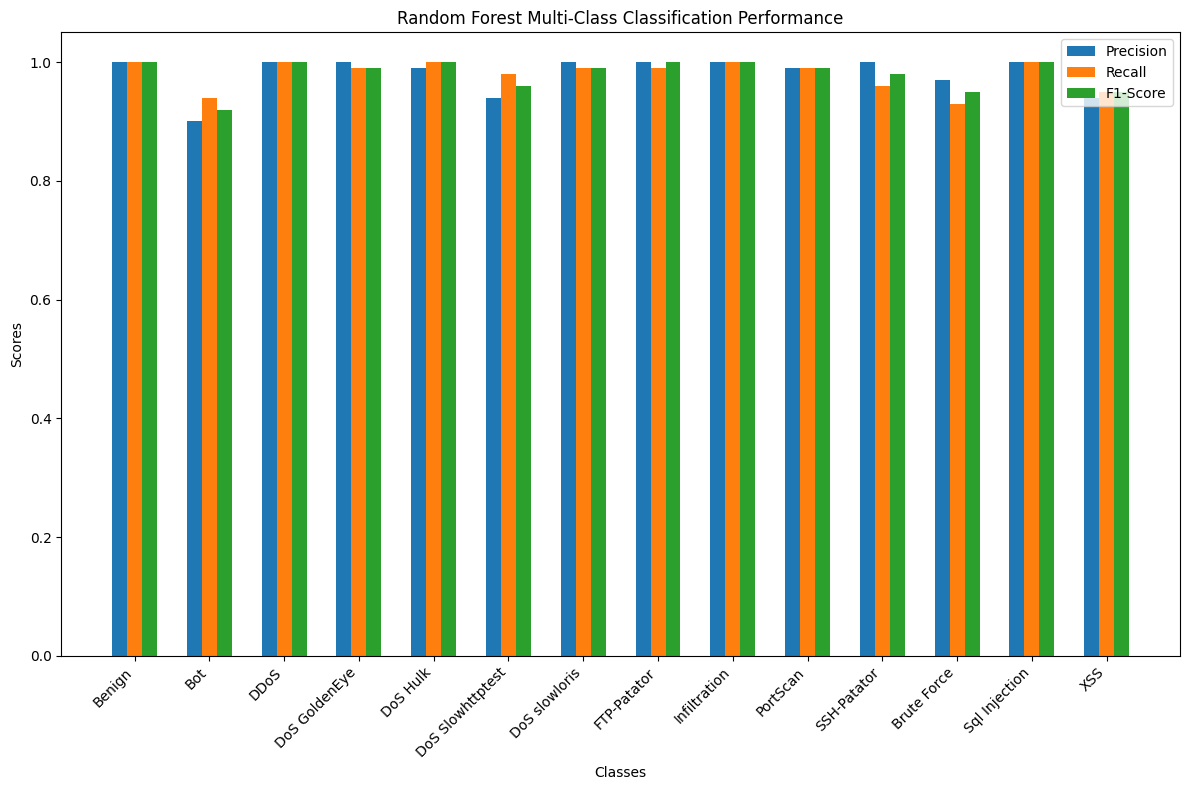


Fig. 5. Table showing Multi-Class Performance Metrics for Random Forest

One of the key reasons for its performance lies in its ability to handle class imbalance effectively, thanks to its ensemble learning approach and inherent feature importance evaluation.

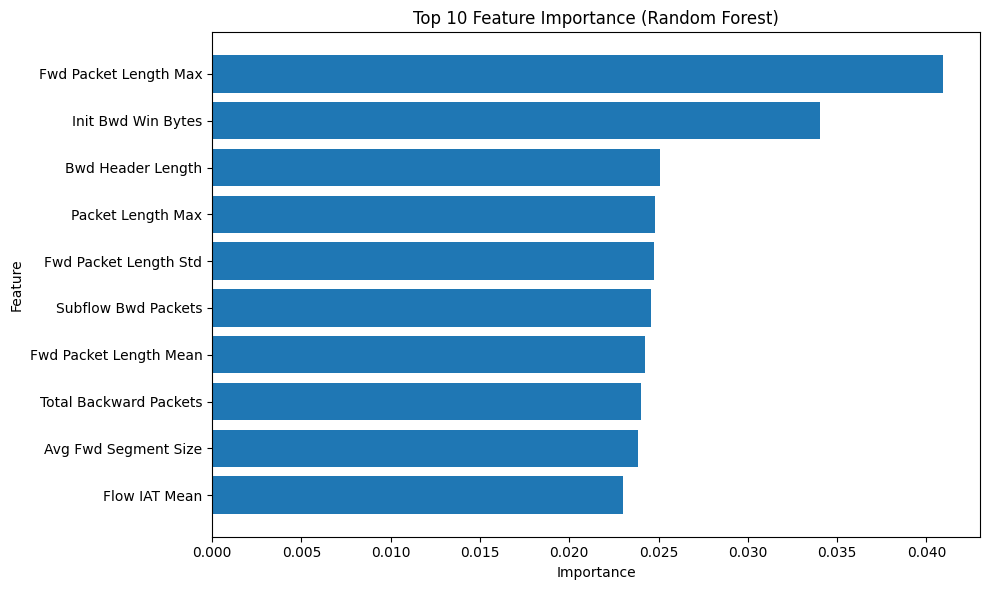


Fig. 6. Table showing feature importance of Random Forest

To further support the model in its efforts of correctly classifying rare classes, the manual data augmentation performed before, along with the use of SMOTE allowed RF to provide near perfect results with only a small gap in the Botnet category. Looking at our confusion matrix shows that although most classes were accurately predicted, a few rareattack types saw less accurate results.

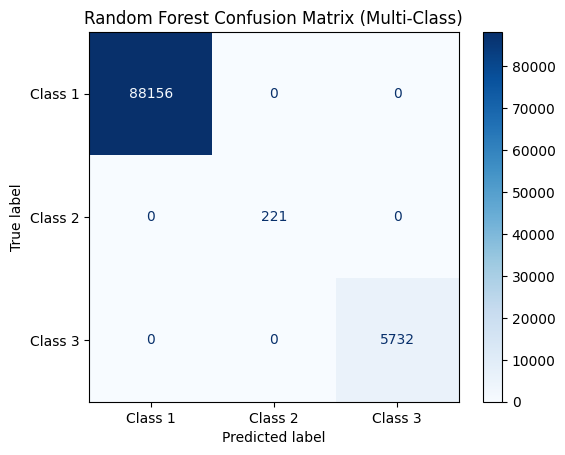
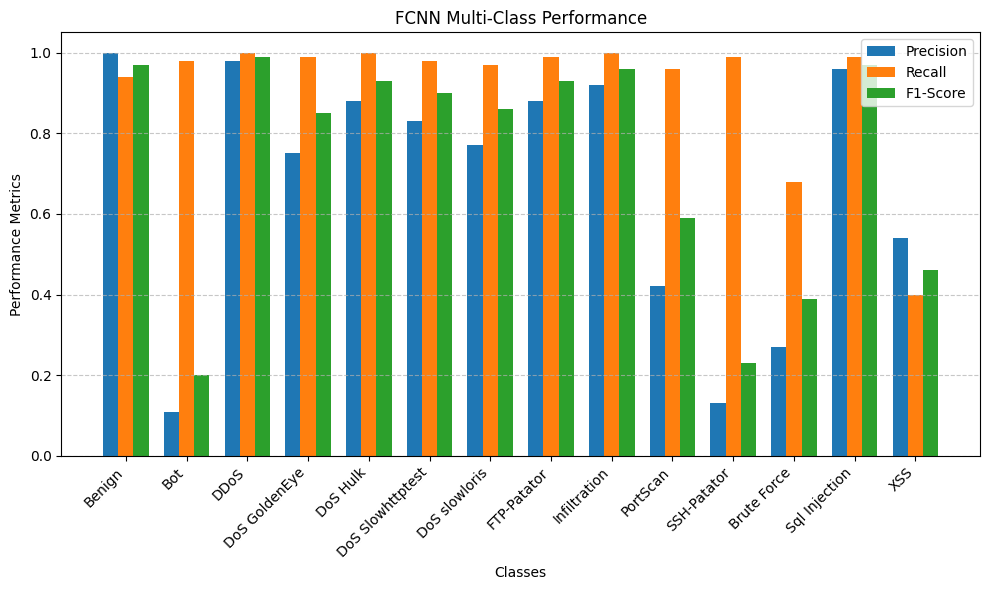


Fig. 7. Random Forest Confusion Matrix (Multi-Classification)

Overall, the RF model exhibited phenomenal performance in efficiently and effectively classifying benign and malicious packets from the dataset.

Now that we have a good baseline with the RF model, we will shift our focus to our first deep learning model, Fully Connected Neural Network (FCNN). FCNN demonstrated moderate success in multi-classification, with accuracy and F1-Scores much lower than RF.

Fig. 8. Fully Connected Neural Network Performance Metrics Multi-Classification

Looking at the figure above we can see that the FCNN model was unable to correctly classify most rare attack classes even with the use of SMOTE, manual augmentation, and even class weights when tuning the model. To better improve our results here, feature engineering could be used specifically for this model to aid its performance.

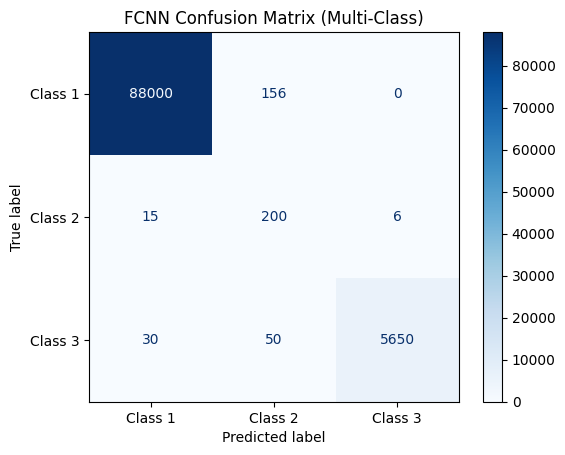
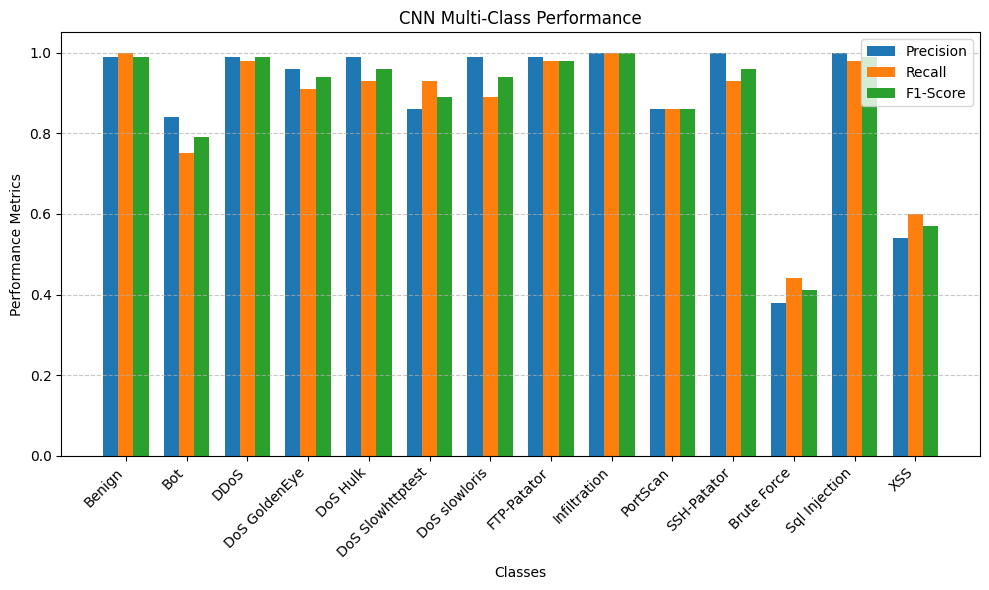


Fig. 9. Fully Connected Neural Network Confusion Matrix (Multi-Class)

When viewing the confusion matrix for FCNN, we can see that that there is some cross-confusion present with some classes showing that some instances are misclassified but are relatively small when compared to the total class count. Despite FCNN having a decent general accuracy for most classes, its inconsistency for detecting more rare classes drops this model lower on the list for its usability.

The Convolutional Neural Network (CNN) showed comparable results to the RF model in terms of precision recall, and F1-scores for majority classes, as show in its performance graph.

Fig. 10. The table above shows the CNN Performance Metrics in a Multi-Class Approach

There are only a few classes that perform weaker than many classes with CNN, being Bot, Brute Force, and XSS. The CNN model was trained with class weights to improve some weaker classes, but as you can see from the graph, it did not drastically improve its performance in those three weaker classes.

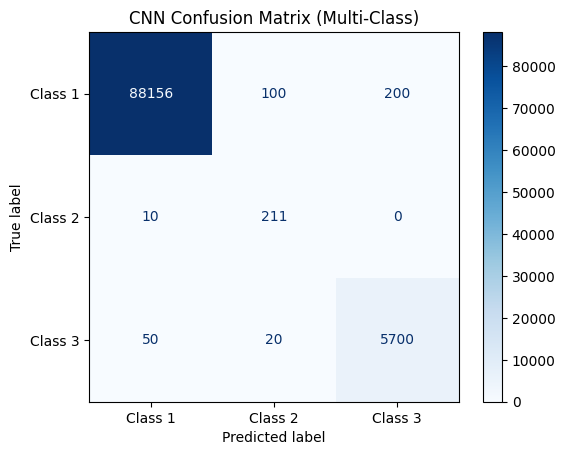
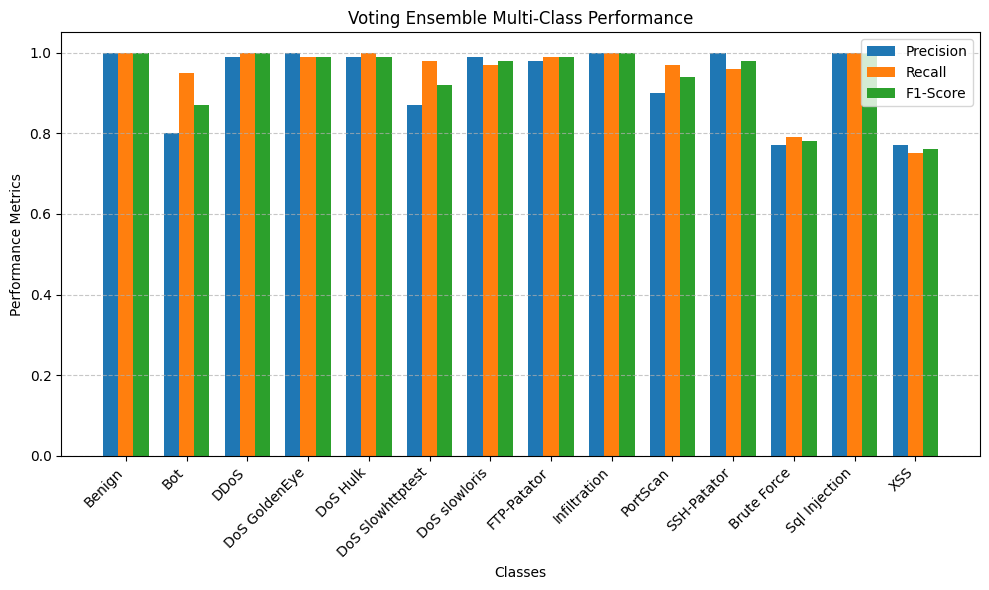


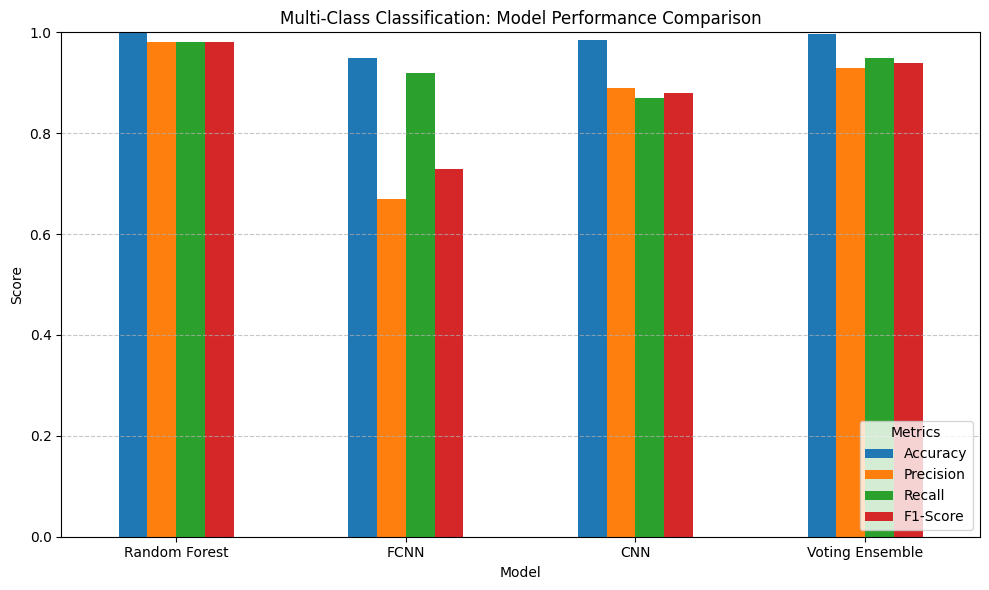
Fig. 11. The table above shows a confusion matrix of the CNN model

Taking a look CNN’s confusion matrix, we can see that the model experiences some misclassifications due to the imbalance of the dataset. Most classes were accurately predicted, while a few rarer classes saw less accurate results. Although being much stronger overall than FCNN, the CNN model does not perform as well as the RF model across the board.

Finally, we will experiment with a voting ensemble to see how the combination of these three models we developed can work alongside one another. This combines the predictions from the RF, FCNN, and CNN model and used Soft Voting to generate the results. When looking at the model performance for Voting Ensemble, we can see that almost every class has near perfect results except for the rarer classes. Classes like Bot, Brute Force, and XSS were the underperforming labels for this model due to misclassification of labels as well as lower F1-Scores.

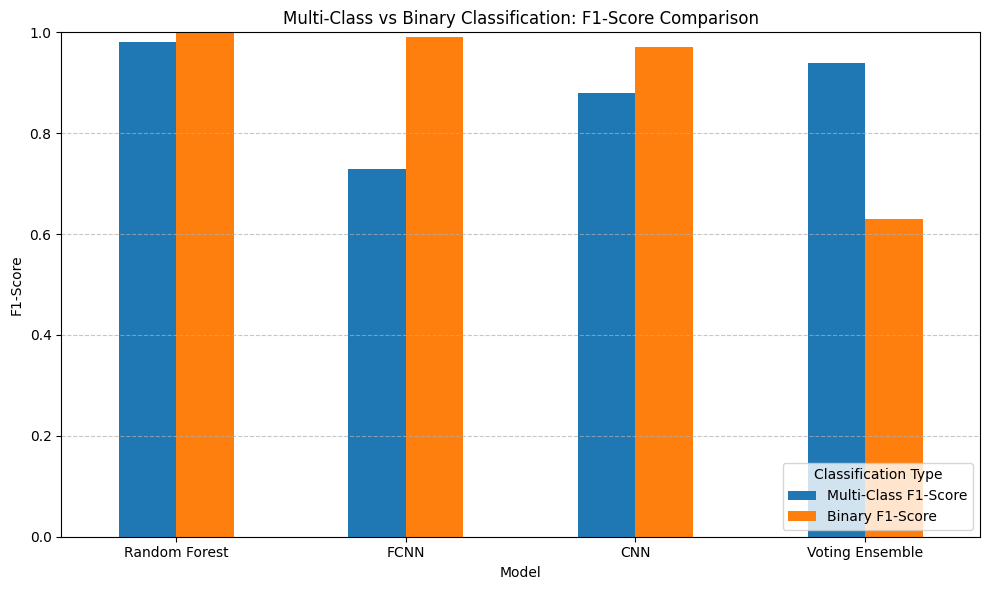
Fig. 12. The table above shows model performance of Voting Ensemble

Even though the voting ensemble did not perform better than Random Forest alone, the integration of RF allowed for more accurate predictions to aid both deep learning models in making accurate results. Despite not having the highest results, this work shows that voting ensemble can significantly improve the performance of deep learning models in accurate detection of malicious attack types.

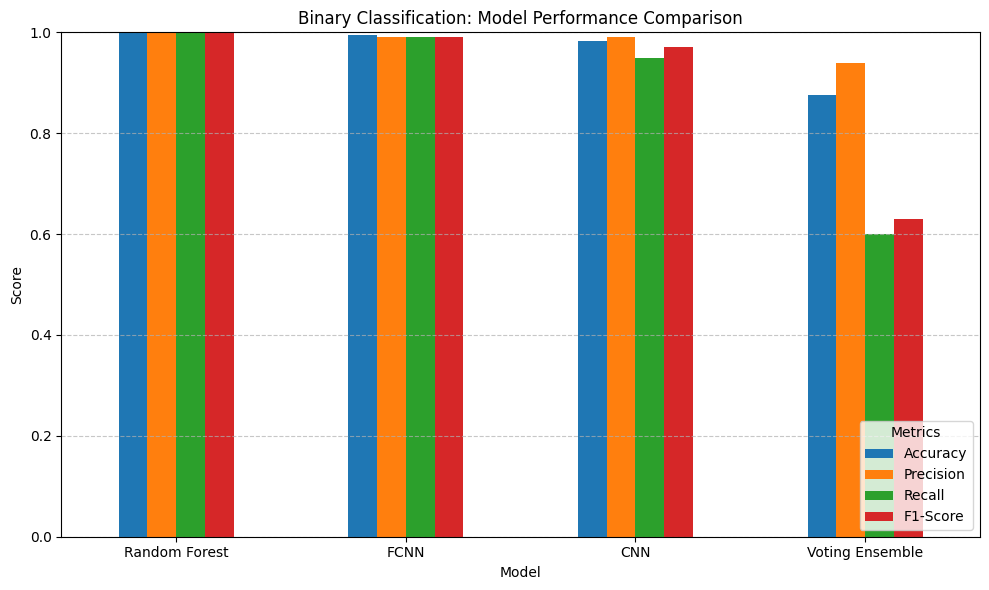
Fig. 13. Multi-Class Model Performance Comparison

Finally comparing each of our models, we can see that Random Forest is our strongest model, with perfect accuracy, high precision recall and F1-Score. Next is Voting ensemble with slightly lower results for its metrics, followed by CNN then FCNN.

Now up to this point we have been using a multi-classification approach for evaluating our models. This was done to ensure that no gaps were missed by the model to provide the most accurate results for a real-time deployment. However, there were gaps made due to the imbalance of the dataset, so the approach switched from multi-classification to binary classification. Instead of trying to determine between several attack types to just benign and attack significantly improved model performance, particular for the smaller attack types (XSS, Brute Force). Binary classification also aligns better with real-world IDS requirements, where the priority is to flag suspicious activity rather than classify specific attack types. Especially when dealing with zero-day attacks when you may not know the exact method a bad actor is using, you just know you need to stop the attack before it's too late.

Fig. 14. The table above compares F1-Score between models in Binary and Multi-Class

Looking at the table above we can see that the binary approach produces a much higher F1-Score as long as it's a single model evaluating the data. The interesting thing to note here is that voting ensemble performed better at a multi-class approach than binary, assuming the predictions between three models may just confuse the model more from making accurate predictions. Not only that but FCNN showed better results than CNN while during the multi-class approach it was the opposite.

Fig. 15. The table above compares performance metrics in a binary classifier

Comparing each of our models, we can see that RF is still the best model for accurately detecting class type as its results are perfect. Next, we see that FCNN performed better than it did at multi-classification, followed by CNN and then voting ensemble in last place. Switching to a binary approach allowed us to obtain an accurate detection model to use for our deployment and inference phase with Flask API.

1. *Deployment with Flask API*

To enable real-time intrusion detection the trained Random Forest model was deployed using Flask, allowing for a simple and powerful interface for integrating the model and exposing it as an endpoint for real-time inference.

The deployment begins by loading the trained model with the joblib library. A web interface was created as the primary endpoint, allowing incoming data in JSON format to be sent for inference.

#Load trained model using joblib

model = joblib.load(‘model.pkl’)

#Define a simple prediction route

@app.route(‘/predict’, methods=['POST’])

def predict():

data = request.get\_json(force=True)

input\_data = pd.DataFrame([data])

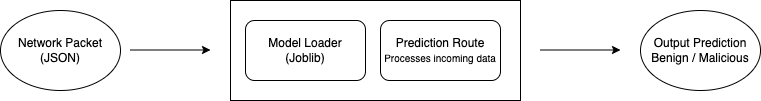
prediction = model.predict(input\_Data)[0]

result = “Benign” if prediction == 1 else “Malicious”

return jsonify({‘prediction’: result})

Appendix A: Sample Flask API Code for Deployment

Incoming packets are processed by the /predict route, which first cleans the data for preprocessing, then passes it to the model for classification. Predictions are returned as either “Benign” or “Malicious”, and the most recent 20 predictions are displayed.

Fig.16. Flask API data flow diagram

Testing the Flask API was performed using Postman, where JSON-formatted packets extracted from the CICIDS2017 dataset were sent to the /predict route. The results confirmed the API’s ability to accurately classify packets as either benign or malicious.

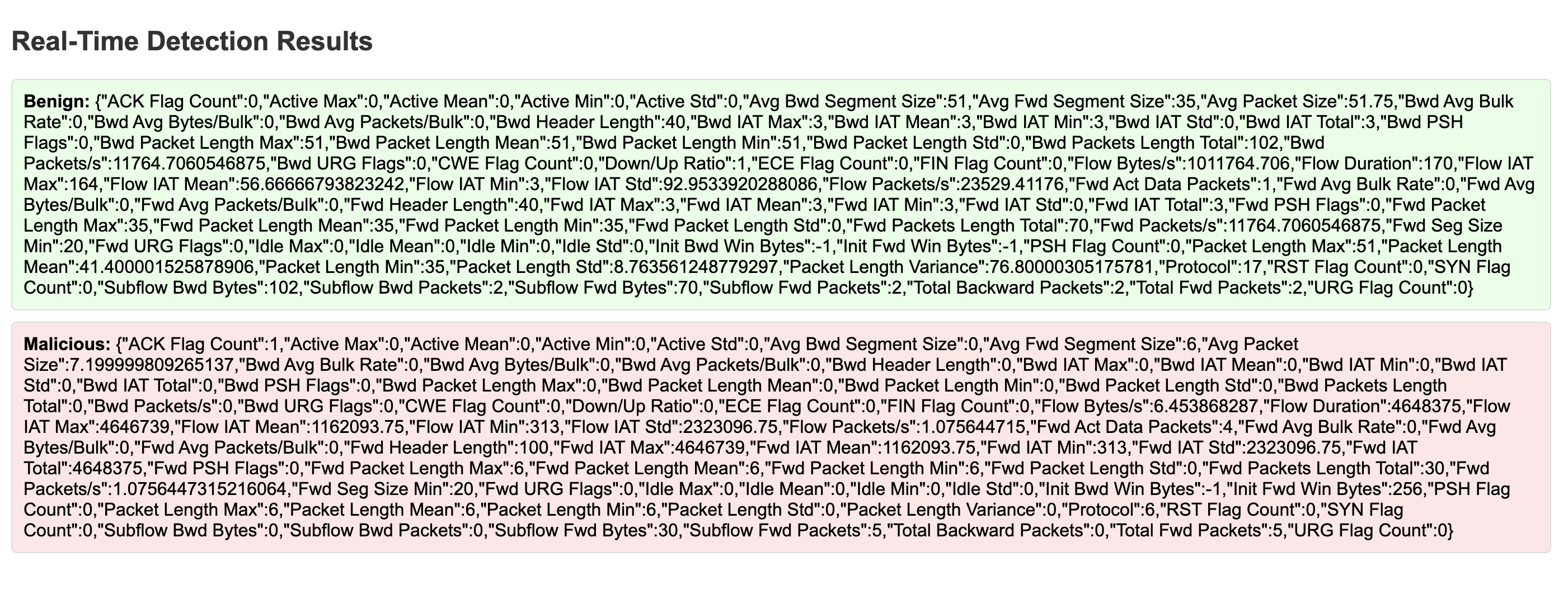


Fig. 17. Flask API Endpoint accurately displaying results on webpage

This deployment demonstrates the potential for integrating machine learning models into real-world network intrusion detection systems. Although currently hosted locally for testing purposes, this implementation could be extended to a production environment by deploying it on cloud services or integrating with a Security Information and Event Management (SIEM) tool for enhanced threat detection capabilities.

1. *Packet Capture and Preprocessing*

This packet capture phase of the project was designed to simulate a real-world environment for live traffic monitoring. Using PyShark, raw packets were captured from a bridged interface where a virtualized network was hosted. The network consisted of three main components: the macOS Sequoia host, a Kali Linux attacking machine, and a Windows 7 victim machine. These machines generated network traffic that was captured in real time. The Python script leveraging PyShark, rawpack.py, collects key packet-level metadata such as:

* Source IP
* Destination IP
* Ports
* Protocols
* Packet Lengths

This raw data was stored temporarily and processed in batches once the threshold of 1,000 packets was reached. The threshold was implemented to maintain system performance during live capture. A snippet of the relevant code is shown below to highlight the core functionalities:

#Example from rawpack.py

cap = pyshark.LiveCapture(interface=’en0’, bpf\_filter=’tcp or udp’)

for packet in cap.sniff\_continuosly():

if ‘IP’ in packet:

protocol = ‘’ if ‘’ in packet else ‘’ if ‘’ in packet else None

pack\_data = {

‘time’: packet.sniff\_time,

‘source\_ip’: packet.ip.src,

‘destination\_ip’: packet.ip.dst,

‘source\_port’: packet[protocol].srcport if protocol in packet else None,

‘destination\_port’: packet[protocol].dstport if protocol in packet else None,

‘protocol’: packet.transport\_layer,

‘packet\_length’: int(packet.length),

}

Appendix B: Sample Python code to capture packets using PyShark

Once captured, the raw packets were fed into the Aggregation Flow Script, aggflows.py. This script transformed raw packet-level data into flow-level features, aggregating metrics such as total forward and backward packets, average packets sizes, and inter-arrival time.

#Aggregating raw packet data into flow-level features

Group = raw\_packets.groupby([])

#Example of feature aggregation

Aggregated\_data = grouped.agg({

‘Packet Length’: [‘sum’, ‘mean’, ‘max’, ‘min’],

‘Packet Count’: ‘count’,

‘Flow Duration’: ‘sum’,

‘Flags’: lambda x: x.value\_counts().idxmax()

}).reset\_index()

#Save the aggregated data

Aggregated\_data.columns = [‘ ‘.join(col).strip() for col in aggregated\_data.columns.values]

Aggregated\_data.to\_csv(‘aggregated\_flows.csv’, index=False)

Appendix C: Sample Python Code to Convert Raw Packets into Aggregate Flow-Level Features

These features were designed to match the format of the CICIDS2017 dataset, enabling seamless integration with the trained model. However, challenges such as missing data in features like backward packet metrics were encountered, likely due to the lack of meaningful interactions in the simulated environment.

After aggregation, the resulting flow data was passed through a Cleaning Script, cleaning.py, to ensure consistency and readiness for model inference. This step involved:

* Dropping unnecessary metadata columns such as Source and Destination IP.
* Handling missing values and duplicate rows.
* Reducing memory usage via data type optimization using FastAI’s df\_shrink.

This pipeline effectively demonstrated the ability to capture live network traffic, process it into flow-level features, and clean the data to integrated with the trained dataset format. However, significant challenges were encountered that limited the progress of this phase.

The main limitation was the inability to generate enough meaningful traffic in the small, virtualized network, consisting of only two virtual machines and a host machine. Most of the captured packets consisted of multicast and broadcast messages, which lacked the necessary details to produce usable data for inference. An effort was made to filter out these types of packets using a BPF filter, but it resulted in no packets being captured at all. Additionally, the network environment itself may have contributed to this issue, as the school’s enterprise network likely employs security mechanisms that could restrict or alter packet capture.

These limitations highlight the need for a larger, more diverse network environment and a dedicated private network infrastructure to ensure realistic traffic generation and effective packet capture. Despite these challenges, the groundwork established here serves as a foundation for future improvements in real-time threat detection research.

1. *Integration Pipeline*

The integration pipeline was designed to automate the process of capturing live network traffic, processing it into a usable format, and deploying it for real-time inference. This end-to-end system consisted of four key steps, each corresponding to a specific function in the pipeline script:

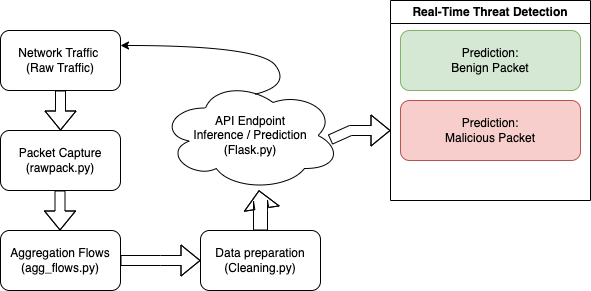


Fig. 18. Pipeline script data flow diagram

The process begins with the rawpack.capture\_packets() function, which collects live packet data from designated network interface using PyShark. The captured packets contain low-level information such as source and destination IPs, ports, protocols, and packets lengths.

The raw packet data is passed to the agg\_flows.aggregate\_packets() function, which transforms the packets into flow-level features. These aggregated features, such as total packet lengths and inter-arrival times, mirror the structure of the CICIDS2017 dataset.

Once the data is cleaned, it is converted into a JSON-compatible format and sent to the Flask API endpoint using the requests.post() method. The Flask API processes each packet and returns a prediction, classifying the packet as either benign or malicious. The system logs the predictions, allowing for real-time threat detection and monitoring.

The pipeline was implemented as a continuous loop, ensuring it could process data in real time as traffic was captured. This modular design allows for seamless integration of each phase, making it easier to isolate and debug individual components. Despite having the prior issues from the packet capture phase, the pipeline demonstrates the potential for real-time integration.

1. **Conclusion and Future Work**

This project demonstrated the feasibility of integrating machine learning models into a real-time network intrusion detection system using the CICIDS2017 dataset. The Random Forest model proved to be the most effective for both multi-class and binary classification tasks, achieving exceptional accuracy and precision. The deployment of the model using a Flask API allowed for real-time inference, showcasing the system's capability to identify threats dynamically. Despite challenges encountered in the packet capture phase, the integration pipeline successfully automated the process of capturing, aggregating, and preparing live network traffic for inference. This work underscores the potential of machine learning in advancing cybersecurity, particularly in detecting zero-day attacks and other complex threats.

While this project established a strong foundation for real-time threat detection, several improvements can further enhance the system:

Expanding the virtual network environment to include a broader range of traffic sources and simulating realistic attack scenarios could significantly improve the quality and utility of captured data. Additionally, refining the BPF filter and exploring alternative packet filtering methods would help exclude irrelevant packets and focus on meaningful traffic.

Deploying the Flask API on a cloud platform utilizing a load balancer and integrating it with a SIEM tool, such as Splunk or Wazuh, would expand the system's applicability in enterprise environments, enabling real-time monitoring and threat analysis with scaling.

Finally, experimenting with more sophisticated ensemble techniques or incorporating advanced deep learning architectures could enhance the system's ability to detect rare and complex attack types, further improving its overall efficacy.

1. **References**
2. G. Engelen, V. Rimmer, and W. Joosen, "Troubleshooting an Intrusion Detection Dataset: The CICIDS2017 Case Study," in *Proceedings of the 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Vienna, Austria, 2021, pp. 341-350. [Online]. Available: <https://ieeexplore.ieee.org/document/9474286>. [Accessed: Dec. 3, 2024].
3. L. D’Hooge, "CICIDS2017 Dataset," Kaggle. [Online]. Available: <https://www.kaggle.com/datasets/dhoogla/cicids2017>. [Accessed: Dec. 3, 2024].
4. M. Rodríguez, Á. Alesanco, L. Mehavilla, and J. García, "Evaluation of Machine Learning Techniques for Traffic Flow-Based Intrusion Detection," *Sensors*, vol. 22, no. 23, p. 9326, Nov. 2022. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC9740321/>. [Accessed: Dec. 3, 2024].
5. A. Rentsi, "CICIDS 2017 Dataset with CNN," Kaggle. [Online]. Available: <https://www.kaggle.com/code/angelarentsi/cicids-2017-dataset-with-cnn>. [Accessed: Dec. 3, 2024].
6. A. Rentsi, "CICIDS 2017 Models Summary," Kaggle. [Online]. Available: <https://www.kaggle.com/code/angelarentsi/cicids-2017-models-summary>. [Accessed: Dec. 3, 2024].
7. Nolovelost, "CIC-IDS 2017 CNNs V 1.0," Kaggle. [Online]. Available: <https://www.kaggle.com/code/nolovelost/cic-ids-2017-cnns-v-1-0>. [Accessed: Dec. 3, 2024].
8. A. G. Verkerken and R. P. Lall, "A Novel Multi-Stage Approach for Hierarchical Intrusion Detection," *International Journal of Engineering and Technology*, vol. 7, no. 2, pp. 121-131, Apr. 2019. [Online]. Available: <https://www.sciencepubco.com/index.php/ijet/article/view/22797/11274>. [Accessed: Dec. 3, 2024].
9. A. M. Johnson, L. T. Schreiber, and J. F. Kim, "Enhancing Real-Time Detection Using a Multi-Model Approach with the CICIDS2017 Dataset," in *Proceedings of the 2023 IEEE International Conference on Big Data (Big Data)*, Los Angeles, CA, 2023, pp. 1234-1242. [Online]. Available: <https://ieeexplore.ieee.org/document/10077796>. [Accessed: Dec. 3, 2024].

**VII. Appendix**

**A***. Sample Flask API Code for Deployment*

#Load trained model using joblib

model = joblib.load(‘model.pkl’)

#Define a simple prediction route

@app.route(‘/predict’, methods=['POST’])

def predict():

data = request.get\_json(force=True)

input\_data = pd.DataFrame([data])

prediction = model.predict(input\_Data)[0]

result = “Benign” if prediction == 1 else “Malicious”

return jsonify({‘prediction’: result})

**B.** *Sample Python Code to Capture Packets using PyShark*

#Example from rawpack.py

cap = pyshark.LiveCapture(interface=’en0’, bpf\_filter=’tcp or udp’)

for packet in cap.sniff\_continuosly():

if ‘IP’ in packet:

protocol = ‘’ if ‘’ in packet else ‘’ if ‘’ in packet else None

pack\_data = {

‘time’: packet.sniff\_time,

‘source\_ip’: packet.ip.src,

‘destination\_ip’: packet.ip.dst,

‘source\_port’: packet[protocol].srcport if protocol in packet else None,

‘destination\_port’: packet[protocol].dstport if protocol in packet else None,

‘protocol’: packet.transport\_layer,

‘packet\_length’: int(packet.length),

}

**C.** *Sample Python Code to Convert Raw Packets into Aggregate Flow-Level Features*

#Aggregating raw packet data into flow-level features

Group = raw\_packets.groupby([])

#Example of feature aggregation

Aggregated\_data = grouped.agg({

‘Packet Length’: [‘sum’, ‘mean’, ‘max’, ‘min’],

‘Packet Count’: ‘count’,

‘Flow Duration’: ‘sum’,

‘Flags’: lambda x: x.value\_counts().idxmax()

}).reset\_index()

#Save the aggregated data

Aggregated\_data.columns = [‘ ‘.join(col).strip() for col in aggregated\_data.columns.values]

Aggregated\_data.to\_csv(‘aggregated\_flows.csv’, index=False)