Cairo University
Faculty of Computers and Artificial Intelligence
Information Technology Department

HAWKEYE
Intrusion Detection System

Supervised by

Dr. Sanaa taha

TA. M. Saad

Implemented by

ID	Name
20210322	Magdy Ibrahim Sayed Ahmed
20210236	Abdullah Ali Abdeltawab
20210357	Mohamed Nabil Mahmoud
20210272	Omar Mohammed Baddour
20210578	Mahmoud Alaa Mahmoud
20210284	Fares Osama Mohammed

Graduation Project
Academic Year 2024-2025
Final Documentation

Abstract

This graduation project aims to develop an **Intrusion Detection System (IDS)** with a **web-based dashboard** from Scratch to enhance network security by monitoring, detecting, and alerting malicious activities in real-time. The system uses **signature-based detection** to identify known threats and **anomaly-based detection** to detect unusual patterns in network traffic. The IDS is designed to detect common cyber threats such as **DoS attacks**, **DDoS attacks, XSS Attacks**, and **SQL injections**.

The project includes a **user-friendly dashboard** that provides administrators with a comprehensive overview of network activity, including real-time alerts, attack summaries, and detailed logs. The dashboard allows users to filter and sort alerts, view attack trends through visual charts, and manage rules for threat detection. Additionally, the system stores historical data for analysis and reporting, enabling administrators to identify recurring threats and improve network defenses.

The IDS is implemented using **Python** for backend processing, **Scapy** for packet capture, **Flask** for the web-based dashboard, and **SQLite** for data storage. The system is evaluated on simulated network traffic to evaluate its effectiveness in detecting intrusions and generating accurate alerts. The results demonstrate that the IDS achieves a high detection rate with minimal false positives, making it a reliable tool for securing networks against cyber threats.

This project contributes to the field of **network security** by providing an accessible and efficient solution for intrusion detection, particularly for small to medium-sized networks. Future work includes integrating **machine learning** for improved anomaly detection and scaling the system for larger networks.

Acknowledgement

Our gratitude to those who helped us cannot be put into words. First, we would like to express our appreciation and thankfulness to our supervisor,

We would like to express my sincere gratitude to our supervisor, **Dr. Sanaa Taha** for her continuous support, accepting our approach, valuable guidance, patience, her utmost care and support for us throughout the duration of this project. Her guidance and mentorship have been critical to our success.

We would also like to thank **TA. Mohammed Saad** for his effort with us.

This project wouldn’t have been possible without the help and resources of our supervisors.

Table of Contents

Chapter 1 Introduction 12
1.1. Background..... 1
1.1.1. The main area of the project..... 1
1.1.2. Main techniques and applications 1
1.1.3. Motivation and Justification..... 1
1.2. Problem Statement..... 2
1.3. Objectives 2
1.4. Project Scope and Limitations..... 3
1.4.1 Project Scope..... 3
1.4.2 Limitations 3
1.5. Intrusion Detection System (IDS) overview..... 4
1.5.1. Different Approaches to the IDS 4
1.5.2 Signature-IDS VS Anomaly-IDS 6
1.5.3 What is the best? 6
1.5.4 Different Forms of the IDS..... 7
1.6. Project Methodology 9
1.7. Project Report Outline..... 11
Chapter 2 Market and Literature Survey 12
2.1. Market Landscape & Challenges…..……..……...49
2.1.1 Market Overview 13
2.1.2 Market Trends & Adoption 13
2.2. Key Competitors in the IDS Market……..50
2.2.1 Open-Source IDS Solutions..... 14
2.2.2 Commercial IDS Solutions..... 16
2.3. IDS Approaches in Research..... 16

2.3.1 Signature-Based IDS 17
2.3.2 Anomaly-Based IDS 17
2.3.3 Hybrid-Based IDS 18
2.4 Comparative Evaluation of Techniques 18
Chapter 3 Requirements & Analysis 31
3.1. Project specifications 20
3.1.1. Stakeholders 20
3.1.2. System architecture 20
3.1.3. Functional Requirements 21
3.1.4. Non-Functional Requirements 23
3.1.5. Use-case Diagram. 25
3.1.5.1. Admin 25
3.1.5.2. IDS System 27
3.1.6. System Component Diagram 30
3.1.7. Class Diagram 31
3.1.8. Sequence Diagram 32
3.1.9. Activity Diagram 41
Chapter 4 Design, Implementation & Testing 45
4.1. System Design 46
4.1.1. Signature-Based Dashboard 46
4.1.2. Anomaly-Based Dashboard 47
4.1.3. Upload-CSV File UI 48
4.1.4. Upload-PCAP File UI 49
4.1.5. Rules-Dashboard 50
4.1.6. Add-Rules 50
4.2. Implementation 51
4.2.1. Traffic PCAP Files 51

4.2.2. Database 52
4.2.3. Snort / Suricata Rules 55
4.2.4. Deep packet Inspection (DPI) 60
4.2.5. Signature-Based IDS 68
4.2.5.1. Matching Packets with Rules 68
4.2.5.2. Signature Rule Matching & Concurrent Packet Processing 73
4.2.6. Anomaly-Based IDS 75
4.2.6.1. Machine Learning Preparations 75
4.2.6.2. Model Development 101
4.2.6.3. Anomaly Detection (Unknown Attacks) 111
4.2.6.4. Hybrid Detection Method (Stacking + Isolation Forest) 112
4.2.6.5. Flow-Based Feature Extraction 114
4.2.6.6. Anomaly-Based Intrusion Detection System 119
4.2.7. if the rule matched with Packet or Detecting Anomalous 123
4.2.8. AlertAnalyzer Class 128
4.2.9. Full Stack Integration & Web Dashboard (Flask Application) 130
4.3. Testing & Evaluation 145
4.3.1. Model Testing 145
4.3.2. DPI (Deep Packet Inspection) Testing 147
4.3.3. Extracting Flow Features 148
4.3.4. Performance Evaluation 148
4.3.5. Real-Time Detection Testing 149
4.3.6. PCAP File Scanning 150
4.3.7. CSV File Prediction 151
4.3.8. Display the Rules to Admin 152
4.3.9. Adding Rules throw the Dashboard 152
4.3.10. Limitations 153

4.4. Technologies and Tools 153
4.4.1. Front-End & UI 153
4.4.2. Database 153
4.4.3. DPI 153
4.4.4. Programming Language 153
4.4.5. Tools 154
Chapter 5 Results and Discussion 155
5.1 Findings 156
5.2 Goals Achieved 156
5.3 Further Work 157
5.4 Ethical, Legal, and Social Issues 158
Chapter 6 Conclusions and Future Work 159
6.1 Conclusions 160
References 163

Chapter 1
Introduction

1.1. Background

1.1.1. The main area of the project

- The primary focus of this project is the development of an Intrusion Detection System (IDS) designed to monitor, analyze, and detect malicious activities within a network.
- Cybersecurity threats have become increasingly sophisticated, making it essential to implement an IDS that can identify potential attacks in real time and alert administrators to take appropriate actions.
- **Intrusion detection** is one of the most important and wide-ranging topics in the field of network and organizational security and protection of individuals. There are many methods used in this field, as well as many intrusion detection systems available. Despite the prominent role it plays, it still suffers from some drawbacks, so there is a need to continue researching intrusion detection systems to develop an optimal structure that achieves a high level of protection.

1.1.2. Main techniques and applications

- Web Dashboard for frontend (Flask).
- Scapy for DPI (Python).
- SQLite for the Database.
- Isolation forest
- Random forest
- Logistic regression
- KNN
- Decision Tree

1.1.3. Motivation and Justification

- **Intrusion detection** (IDS) is an essential component of modern security infrastructure for networks and organizations. IDS are designed to detect and respond to security threats, such as hacking attempts, malware infections, and unauthorized access.

- The motivation for using **IDS** is to protect critical assets and data from several types of attacks, including insider threats, external attacks, and advanced persistent threats. **IDS** can help organizations identify security breaches early on, allowing them to respond appropriately to mitigate damage and prevent further attacks. Furthermore, **IDS** can provide organizations with valuable insights into their security posture, by analyzing network traffic and identifying patterns and anomalies that may indicate potential security threats.

- This information can then be used to improve security policies and procedures, and to implement additional security measures as needed.

- The justification for **IDS** lies in the fact that threats to network security are constantly evolving, and traditional security measures such as firewalls and antivirus software may not be.

- enough to protect against them. **IDS** can provide an additional defense, allowing organizations to identify and respond to threats in real time, and to continuously monitor and improve their security posture. Overall, **IDS** are a critical tool in the fight against cyber threats, and their use is essential for organizations that want to protect their assets and data from malicious actors.

1.2. Problem Statement

- With the rapid expansion of digital networks and online services, cyber threats have become more **sophisticated, frequent, and damaging**. Organizations and individuals face a constant risk of cyberattacks such as **malware infections, unauthorized access, denial-of-service (DoS) attacks, and data breaches**. Traditional security measures like **firewalls and antivirus software** are not always sufficient to detect and prevent these advanced threats, especially those that exploit vulnerabilities in network communication.

1.3.Objectives

- **Detection of Security Threats**: The primary objective of an **IDS** is to detect security threats on the network, such as unauthorized access, malware, and other attacks.
- **Real-time Alerting: IDS** should provide real-time alerts to security personnel when a threat is detected, allowing for immediate action to be taken to mitigate the threat.
- **Incident Response: IDS** should provide information to support incident response activities, including forensic analysis of security incidents.
- **Reporting and Analytics: IDS** should provide comprehensive reporting and analytics capabilities to help security personnel identify trends and patterns in network traffic, and to help identify areas where additional security measures may be needed.
- **Detection of Security Threats:** The primary objective of an **IDS** is to detect security threats on the network, such as unauthorized access, malware, and other attacks.

1.4. Project Scope and Limitations

Project Scope

- This project focuses on designing and implementing an **Intrusion Detection System (IDS)** to enhance network security by identifying malicious activities in real time. The system is built using a Hybrid-based detection approach, leveraging **PCRE (Perl Compatible Regular Expressions)** for pattern matching and using machine learning algorithms, a structured database for storing rules, alerts and logs, a Flask-based web interface for monitoring and management.

Key Features and Functionalities:

- Network Traffic Analysis.
- Signature-Based Intrusion Detection.
- Logging and Alerting System.
- Web-Based Monitoring Interface.
- Performance and Optimization.

Limitations

- While this IDS project provides an effective solution for detecting cyber threats, it has several limitations that should be considered:
  - Limited Network Coverage.
  - Single-Node Deployment.
  - Computation Resources/Cost
  - False/Positive alarm
  - Overfitting
  - Database Performance Considerations:
    - Using an SQLite database may limit scalability when handling extremely large rule sets or high-speed network traffic.

1.5. Intrusion Detection System (IDS) overview

- In today's digital landscape, cybersecurity threats have become increasingly sophisticated, posing significant risks to organizations and individuals alike. Intrusion Detection Systems (IDS) serve as a critical component of modern cybersecurity defenses, designed to identify, alert, and sometimes respond to unauthorized access or malicious activities within a network or host system.
- IDS operate by continuously monitoring network traffic and system activities, analyzing patterns to detect potential security breaches. By providing real-time alerts and actionable insights, IDS enable security teams to take initiative-taking measures to mitigate threats before they escalate into major incidents.

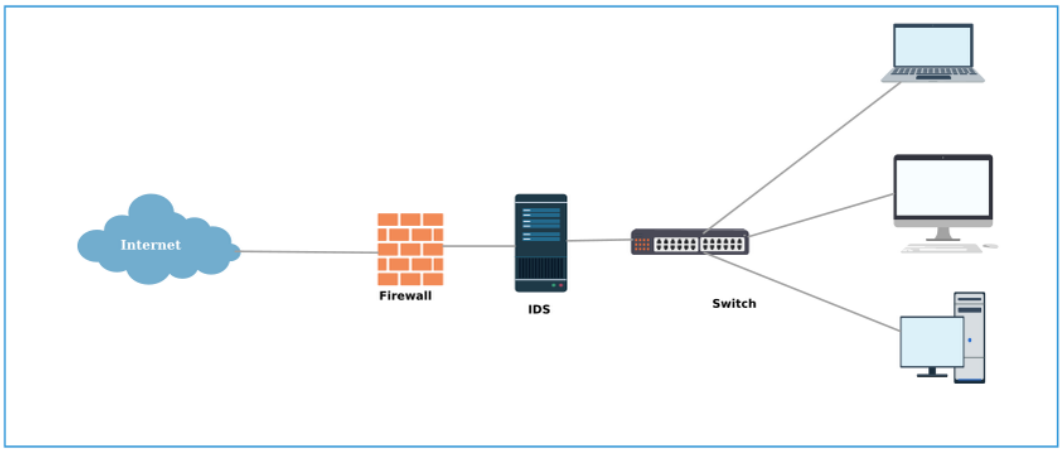

Figure 1.1

1.5.1. Different Approaches to the IDS

- IDS employ various methodologies to detect and respond to threats, with each approach tailored to specific security requirements. The primary detection methodologies include:

1. **Signature-Based IDS:** It operates by comparing incoming network traffic against a database of known attack signatures. These signatures consist of predefined patterns of malicious activity that have been documented in past cyberattacks. When the **IDS** identifies a match between observed traffic and a stored signature, it generates an alert, enabling security teams to respond accordingly.

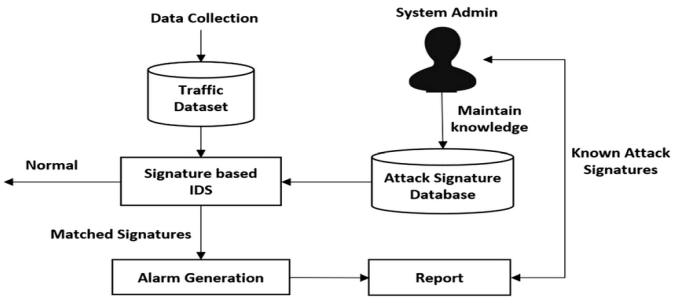

Figure 1.2

2. **Anomaly-Based IDS:** It takes a different approach by establishing a baseline of normal system behavior and identifying deviations that may indicate potential attacks. These systems use artificial intelligence (**AI**), statistical models, and machine learning algorithms to distinguish between legitimate and suspicious activities.

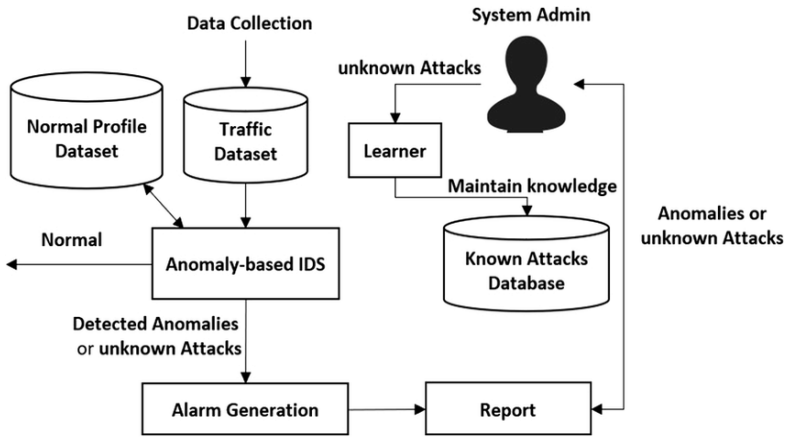

Figure 1.3

3. **Hybrid-Based IDS:** It combines both signature-based and anomaly-based detection methods to provide a more comprehensive security solution. This approach leverages the strengths of each methodology, offering precision in detecting known threats while remaining adaptable to new and emerging attack techniques.

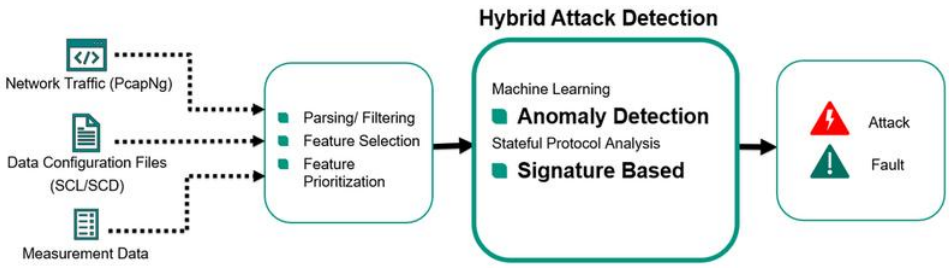

Figure 1.4

1.5.2 Signature-IDS VS Anomaly-IDS

Key	Signature IDS	Anomaly IDS
Advantages	• High accuracy in detecting known threats. • Minimal false positives when signature databases are well-maintained. • Fast and efficient detection of common attack patterns.	• Effective against zero-day attacks and novel threats. • Adaptable to evolving attack tactics. • Can detect previously unknown vulnerabilities being exploited.
Limitations	• Ineffective against new, unknown threats **(zero-day attacks).** • Requires frequent updates to the signature database to remain effective. • It can be bypassed if attackers modify their tactics to avoid detection.	• Higher false positive rates due to variations in legitimate user behavior. • Requires fine-tuning to reduce unnecessary alerts. • More resource-intensive than signature-based **IDS** due to continuous learning and analysis.

Table 1.1

1.5.3 What is the best?

- Signature-based IDS is fast and precise for known attacks.
- Anomaly-based IDS can detect zero-day threats but needs more tuning.
- Combining both methods provides stronger protection against cyber threats.

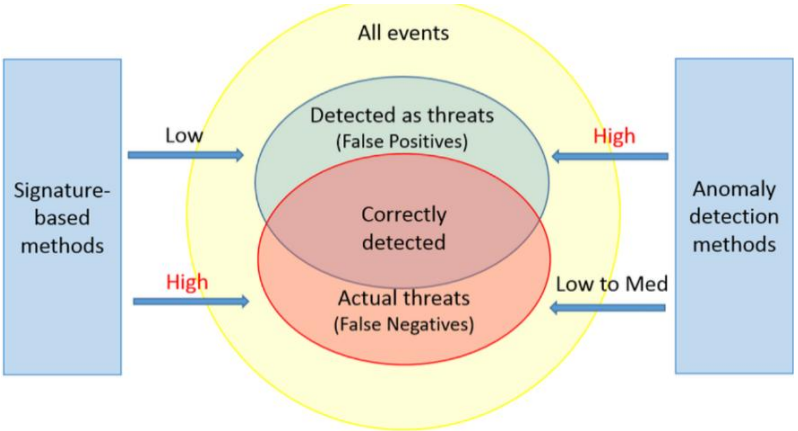

Figure 1.5

1.5.4 Different Forms of the IDS

- Intrusion Detection Systems can be implemented in different environments, each serving specific security objectives. The primary forms of **IDS** include:

1. **Network-Based IDS:** It monitors network traffic for signs of malicious activity by analyzing data packets in transit. These systems are strategically deployed at key network locations, such as firewalls, gateways, and routers, to detect unauthorized access attempts and potential cyber threats before they reach internal systems.

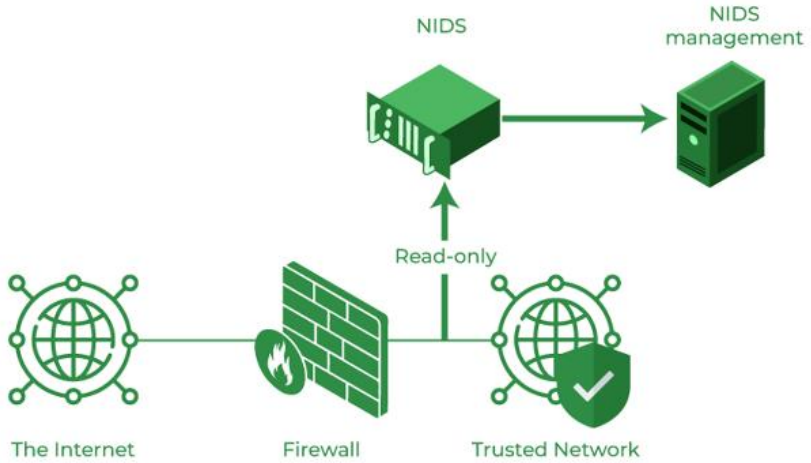

Figure 1.6

2. **Host-Based IDS:** It is installed on individual systems **(e.g., servers, workstations)** to monitor internal activities, including file integrity, system logs, and process execution. HIDS focuses on detecting unauthorized changes, insider threats, and malware infections.

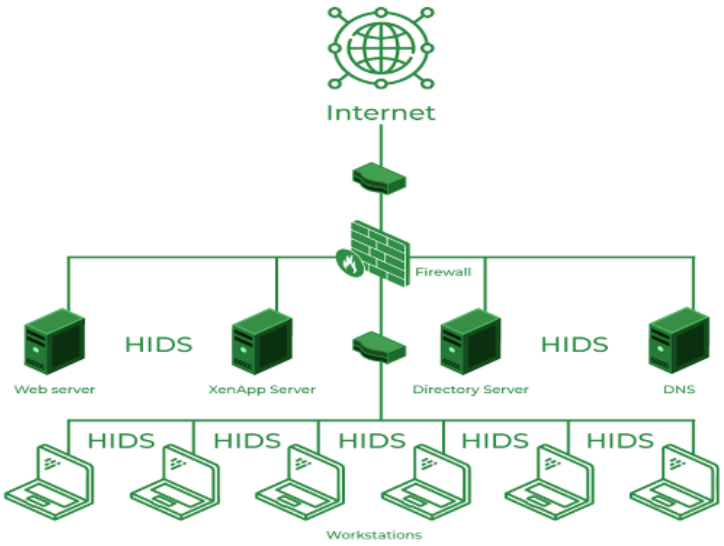

Figure 1.7

3. **Hybrid-Based IDS:** It integrates **NIDS and HIDS** capabilities to provide a holistic security framework. By combining network-wide visibility with host-specific monitoring, hybrid IDS enhances an organization's ability to detect and respond to a wide range of threats.

Key	Network IDS	Host IDS
Advantages	• Provides broad visibility across an entire network. • Detects attacks targeting multiple systems. • It can operate passively without impacting system performance.	• Provides detailed insights into host-specific activities. • Effective at detecting insider threats and advanced persistent threats (APTs). • It can operate in environments where network visibility is limited.
Limitations	• May struggle with encrypted network traffic. • High-speed networks can limit deep packet inspection capabilities. • Requires integration with other security tools for effective incident response.	• Requires installation and maintenance on each host. • Consuming system resources, potentially impacting performance. • Limited scope, as it only monitors activities on the specific host where installed.

Table 1.2

1.6. Project Methodology

- The waterfall method, also known as the waterfall model, is a sequential software development process that progresses through distinct phases, such as requirements, design, implementation, testing, deployment, and maintenance.

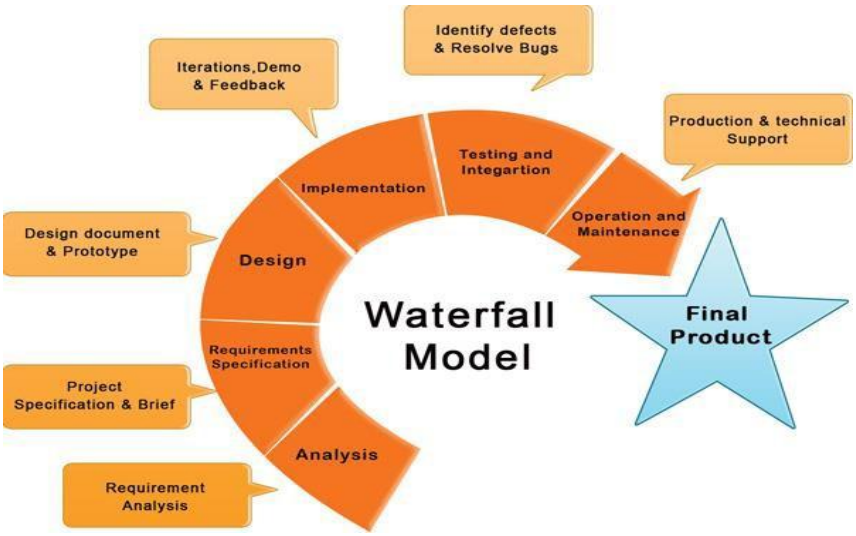

Figure 1.8

Waterfall

Waterfall model divides software development process into separate phases. The outcome of each phase is the input for the next phase. Development moves from analysis, through design, implementation, testing, installation and ends up at deployment and maintenance. Each phase of development proceeds in strict order.

Requirements

The first phase involves understanding what the user needs and what needs to design and what is its function, purpose, etc. Here, the specifications of the input and output or the final product are studied.

Design

The bridge that takes us from requirements to solutions (implementation), its objectives is to define and organize the components of the final solution as it serve as the blueprint (structure) for the next stage (implementation stage). In this stage, the design of applications, databases, and user interface are identified. Implementation The design that is identified in the previous stage is implemented through transforming each use case into a working functionality.

Testing

Once the system has been developed, it will be tested to ensure that it is working as expected without any defects. A test plan will be followed. Different test cases will be executed according to three types: unit testing, integration testing, and usability testing.

Deployment

Once the functional and non-functional testing is done, the product is deployed in the customer environment or released into the market.

Why Waterfall?

Clear Structure

The waterfall model provides a clear and structured approach to development, with each phase building upon the previous one.

Predictability

Since the requirements are gathered and documented early in the process, and changes are minimized once development begins, the waterfall model offers predictability in terms of project timelines, budgets, and deliverables.

Documentation

The waterfall method emphasizes thorough documentation at each stage of the project. This documentation can serve as a valuable reference point for understanding the project's progress, requirements, design decisions, and testing procedures.

Risk Management

By breaking down the project into sequential phases, the waterfall model allows for early identification and mitigation of risks. Issues can be addressed in the initial stages, reducing the likelihood of major problems later in the development process.

1.7. Project Report Outline

Chapter 2. Literature review

- This chapter introduces **Hawk Eye**, our Hybrid IDS that combines signature and anomaly detection. We compare it with three existing IDS solutions, explain what we used, what we added, and why we chose this approach for better security and performance.

Chapter 3. Requirements & Analysis

- This chapter identifies **what the system is supposed to do** from both the user's and developer's perspectives. It helps define the **problem clearly** and the **requirements** needed to solve it.

Chapter 4. Design, Implementation & Testing

- This chapter explains **how the system was designed, implemented, and tested**. It provides a clear view of the development process.

Chapter 5. Results & Discussions

- This chapter presents the main results obtained from the implementation and testing of the Hybrid Signature–Anomaly Intrusion Detection System (IDS). It provides a detailed discussion of the system's performance, including key findings, the extent to which the project goals were achieved, and areas for improvement. Also, this chapter outlines potential future work based on current limitations and insights gained during development. , It also addresses the ethical, legal, and social considerations related to building and deploying an IDS in real world environments

Chapter 6. Conclusion.

- In this chapter, it summarizes the key findings, achievements, and contributions of the project, reflecting on how the objectives were met. It also discusses the proposed potential areas for future improvement and expansion. By outlining future directions, this chapter aims to inspire further research and innovation in the field

Chapter 2 Market and Literature Survey

2.1 Market Landscape and Challenges

2.1.1 Market Overview

- The **IDS** market has grown substantially due to the proliferation of cyber threats and increasing regulatory demands.
- **Signature-based IDS** continue to be a foundational component in network security, offering real-time detection against known attack patterns (**Bhuyan, Bhattacharyya and Kalita, 2014).**

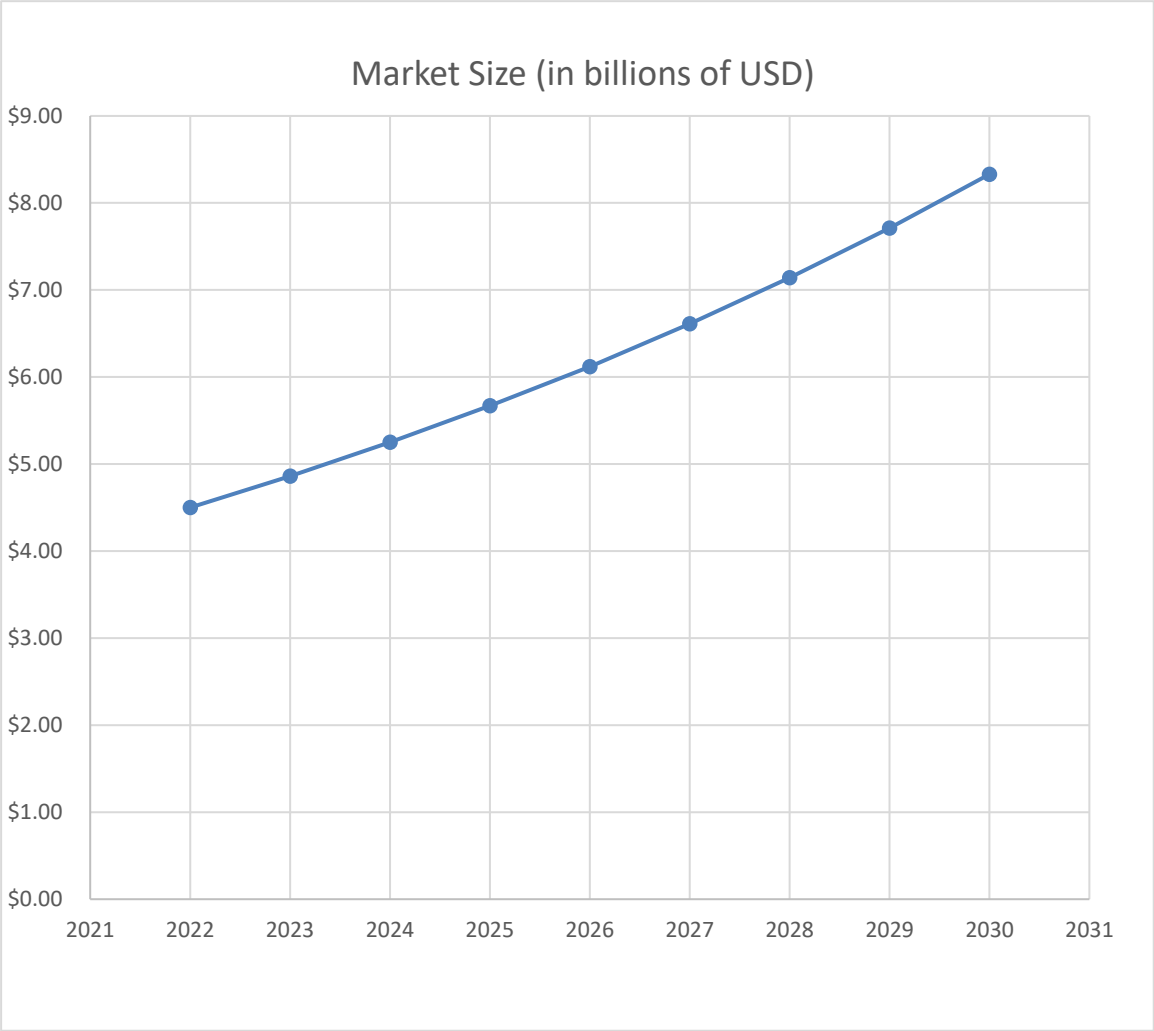

Figure2.1

2.1.2 Market Trends & Adoption

- **Increasing Demand for Network Security**: Enterprises globally are adopting IDS technologies to safeguard data and comply with cybersecurity regulations.
- **Enduring Relevance of Signature-Based IDS**: Tools-like Snort, Suricata, and Zeek remain dominant due to their effectiveness in identifying well-known threats

- **Shift Toward Cloud and Hybrid Environments**: IDS systems are being adapted to monitor traffic across cloud-native and hybrid infrastructures.
- **Leading Solutions**:

  1. **Snort (Cisco):** Community-supported, widely used open-source tool.
  2. **Suricata:** Known for multi-threading and deep packet inspection.
  3. **Zeek:** Favored for forensic analysis and scripting flexibility.
  4. **Trend Micro Tipping Point:** Offers commercial-grade **IDS** with automated responses.

- Industries such as finance, healthcare, and government agencies heavily rely on **IDS** to comply with security regulations.
- **Widespread Use of Signature-Based IDS:**
  - Despite emerging AI-driven anomaly detection, signature-based **IDS** (e.g., **Snort**, **Suricata**, **Zeek**) remains widely used due to its high accuracy in detecting known threats.
  - Businesses integrate **IDS** with firewalls and **SIEM** (Security Information and Event Management) solutions for a layered security approach.

2.2 Key Competitors in the IDS Market

2.2.1 Open-Source IDS Solutions

1. Snort:

- **Snort** is the most widely used open-source **IDS**, originally developed by **Martin Roesch** and now maintained by **Cisco**.

Figure 2.2

Key	Description
Strengths	Large community-driven rule set with frequent updates. Can function as both **IDS** and **IPS** (Intrusion Prevention System). Integration with Security Information and Event Management (**SIEM**) tools.
Weaknesses	Single-threaded processing limits performance on high-traffic networks. Manual tuning is required to reduce false positives.
Usage	Small to mid-sized enterprises, researchers, and security enthusiasts.

Table 2.1

2. Suricata:

- **Suricata** is a multi-threaded **IDS/IPS** with deep packet inspection (**DPI**) capabilities.

Figure2.3

Key	Description
Strengths	Multi-threading supports better performance in high-traffic environments. Built-in protocol detection (e.g., **HTTP**, **TLS**, **DNS**, **SMB**) enhances visibility. Supports **Snort** rules, making migration easy.
Weaknesses	Requires more system resources than **Snort**. Complex rule management and tuning needed for optimal performance.
Usage	Large enterprises and cloud environments require scalable **IDS** solutions.

Table 2.2

3. Zeek:

- Unlike traditional **IDS**, Zeek is a network security monitoring (**NSM**) tool focused on forensic analysis.

Figure2.4

Key	Description
Strengths	Provides extensive logging and analysis for threat hunting. Supports custom scripting to define security policies. Effective in detecting advanced persistent threats (**APTs**).
Weaknesses	Not designed for real-time alerting-like Snort or Suricata. Requires expertise in **Zeek** scripting language.
Usage	Security researchers, forensic analysts, and **SOC** teams are looking for deep network visibility.

Table 2.2

2.2.2 Commercial IDS Solutions

1. **Trend Micro Tipping Point:**
   - A high-performance hardware-based **IDS/IPS** solution.

Figure 2.5

Key	Description
Strengths	Zero-day threat detection with real-time updates. Inline deployment as an IPS for automated threat blocking. Scales well for large enterprise networks.
Weaknesses	Expensive licensing model compared to open-source solutions. Requires specialized hardware for full functionality.
Usage	Large corporations and critical infrastructure requiring automated threat prevention.

Table 2.3

2. **Palo Alto:**
   - Palo Alto integrates IDS capabilities within its NGFW appliances.

Figure 2.6

Key	Description
Strengths	Uses machine learning (ML) and AI for enhanced detection. Supports SSL decryption to analyze encrypted traffic. Cloud-based threat intelligence for real-time updates.
Weaknesses	High cost of deployment. Requires vendor lock-in with Palo Alto infrastructure.
Usage	Enterprises seeking an all-in-one security solution **(firewall + IDS + threat intelligence).**

Table 2.4

3. IBM QRadar IDS:

- **QRadar** is a Security Information and Event Management (**SIEM**) platform with built-in **IDS** capabilities.

Figure 2.7

Key	Description
Strengths	Advanced correlation engine reduces false positives. Automated response integration with incident management tools. Supports multi-cloud environments.
Weaknesses	Complex deployment and maintenance. High licensing costs for full **SIEM** capabilities
Usage	Enterprises looking for centralized security event management.

Table 2.5

2.3 IDS Approaches in Research

2.3.1 Signature-Based IDS

- **Signature-based IDS** rely on pattern-matching algorithms and are effective for known threats. They require frequent signature updates and are not suitable for novel or zero-day attacks.

2.3.2 Anomaly-Based IDS

- **Anomaly detection** identifies deviations from normal behavior using machine learning. Lee, Kim and Kim (2020) applied ensemble learning, while Kim, Yoon and Park (2019) used deep learning methods like LSTM and CNN for detecting IoT-based attacks. These systems are effective in identifying unknown threats but suffer from false positives.
- In this project, anomaly detection is explored through:
  - **Isolation Forest**: Efficient for unsupervised anomaly detection.
  - **Random Forest:** Provides interpretable and accurate classification.
  - **KNN.**
  - **Logistic Regression.**
  - **CatBoost.**
  - **Decision Tree.**

2.3.3 Hybrid-Based IDS

- **Hybrid** approaches integrate signature-based and anomaly-based methods. Ahmed, Mahmood and Hu (2022) reviewed hybrid IDS that balance detection accuracy with processing complexity. While effective, they are not the focus of this project.

2.4 Comparative Evaluation of Techniques

Feature	Signature-Based IDS	Traditional Anomaly-Based IDS	This Project
ML Techniques	None	Decision Tree, Random Forest, KNN	Isolation Forest, Random Forest, Decision Tree,
Detects Zero-Day	❌	❌	✅
False Positives	Low	High	Moderate
Complexity	Low	Medium	Medium
Adaptability	Low	High	Very High

Table 2.6

- This project builds on the strengths of anomaly-based systems while attempting to mitigate false positives through model tuning, feature engineering, and ensemble evaluation.

Chapter 3
Requirements & Analysis

3.1. Project specifications

3.1.1. Stakeholders

- In this Stakeholders mean individuals or groups that have an interest in any decisions or activities of the project. In the system there are four stakeholders:

1. **End Users (Network Administrators).**

2. **Cybersecurity Experts and Consultants.**

3. **System Integrators and IT Support Teams.**

4. **Security Incident Response Teams.**

3.1.2. System architecture

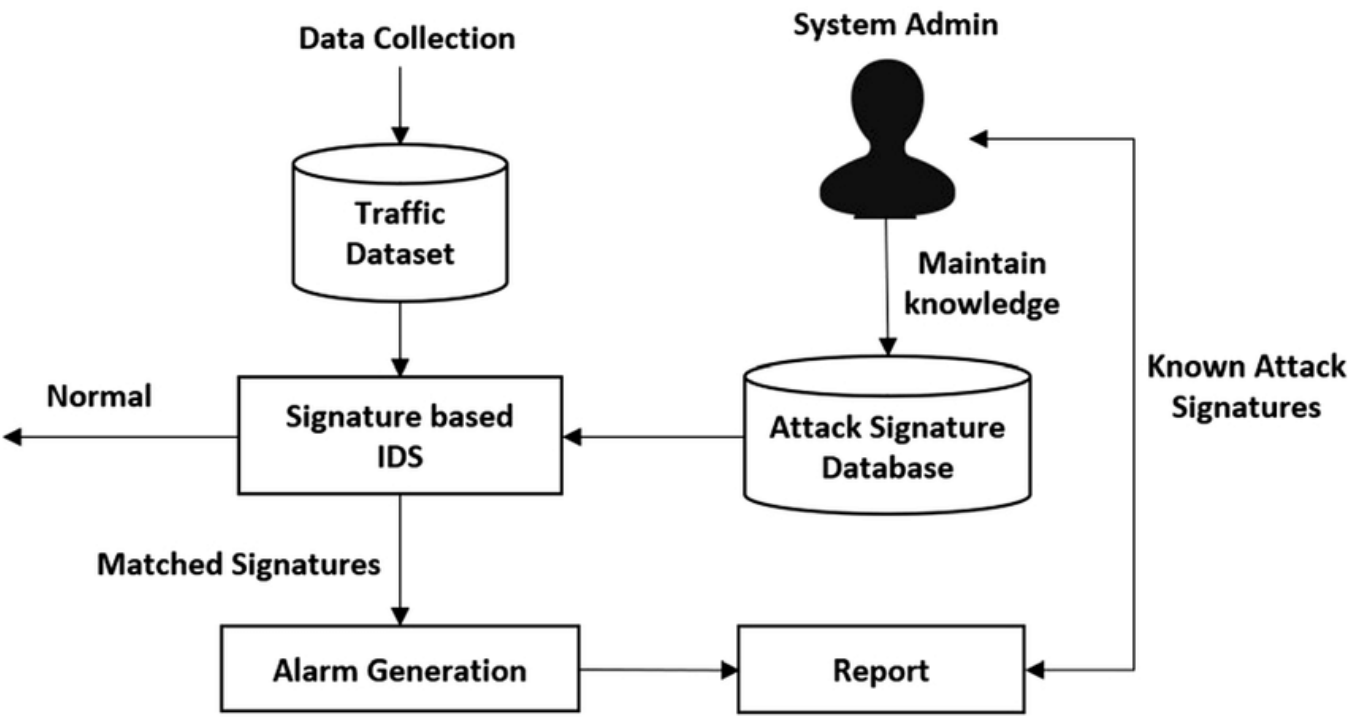

Figure 3.1

3.1.3. Functional Requirements

Requirement ID	Description
FR01	The system should **capture and analyze PCAP files** (pre-recorded traffic) using both signature and anomaly-based methods.
FR02	The system shall support multiple network protocols (e.g., **TCP, UDP, ICMP).**
FR03	The system shall **parse predefined signature rules** from .txt files or the database, and detect matches using content, flags, **dsize**, **pcre**, **itype**, and threshold options.
FR04	The system shall allow rules to be defined using **Perl-Compatible Regular Expressions (PCRE)** for advanced matching.
FR05	The administrator shall be able to **upload, view, and manage custom rules** from the dashboard.
FR06	The system shall apply **signature detection** logic and raise alerts when a rule is matched.
FR07	The system shall apply anomaly detection on traffic using a trained hybrid model **(Stacking Model + Isolation Forest)**, and raise alerts if anomalies are detected.
FR08	Alerts should include details such as: ○ Timestamp. ○ Source IP address.

	○ Destination IP address. ○ Attack type **(e.g., DDoS, port scanning, SQL injection)**. ○ Detection Method **(Signature or Anomaly)**.
FR09	Alerts shall be searchable and filterable by timestamp, source IP, destination IP, and attack type.
FR10	The system shall store all alerts and logs in a structured SQLite database.
FR11	Logs shall be searchable and filterable by timestamp, source IP, destination IP, and attack type.
FR12	The system shall include a **web-based dashboard** built with Flask to allow real-time interaction and monitoring.
FR13	○ The dashboard shall allow the admin to: - View real-time signature alerts - View anomaly detection results - View logs - Upload PCAP files - Upload CSV files (anomaly detection only)
FR14	The dashboard shall show **statistical summaries**: - Top source IPs - Top destination IPs - Count of alerts by type and severity

FR15	The system shall generate reports summarizing network activity and detected threats.
FR16	Reports shall be exportable in multiple formats (e.g., PDF, CSV).
FR17	The system should send email or SMS notifications to administrators for high-severity alerts.

Table 3.1

3.1.4. Non-Functional Requirements

Requirement name	Description
Usability	○ The web dashboard shall be **intuitive, modern, and clear**. ○ Tooltips and field hints will guide the admin during rule upload or CSV/PCAP selection.
Performance & Response Time	○ The system shall process uploaded traffic and display results **within 1–3 seconds**. ○ Signatures and anomaly engines run in **parallel threads** to reduce latency.

Scalability	○ The system shall support **extending rule sets**, larger PCAP files, and growing database logs. ○ It should support **multi-threaded detection** for faster processing.
Security	○ Sensitive data (alerts, logs, rule content) must be **stored securely** in the local database. ○ The admin interface shall require **authentication** (to be implemented). ○ Support for **role-based access control** is planned.
Maintainability	○ The codebase shall be **modular**, separating components like rules, packets, logs, alerts, and models. ○ Each component (e.g., Signature IDS, Anomaly IDS) can be **updated independently**.
Reliability	○ The system must handle **unexpected inputs** (e.g., malformed PCAP or corrupted CSV) gracefully. ○ In case of failure, the system shall display user-friendly errors and continue serving the dashboard.

Table 3.2

3.1.5. Use-case Diagram.

3.1.5.1. Admin

- This Use Case Diagram illustrates the interaction between the admin **(the only human actor)** and **the Intrusion Detection System (IDS)** User Interface. The system allows the admin to upload data, manage logs, and view alerts and statistics generated by both **signature-based** and **anomaly-based** detection engines.

Use Case	Description
Upload PCAP File	The admin uploads a. **pcap** or. **pcapng** file containing raw network traffic data. This data is used for real-time detection by both signature and anomaly detection modules.
Upload Rule File	The admin uploads a rule set **(e.g., in Snort syntax)**. These rules are parsed, extracted, and stored in the database. The signature-based IDS engine later uses them for matching packets.
View Signature Alerts	The admin can view all alerts generated by the signature-based IDS. These alerts are based on pattern matches between the traffic and uploaded rules.
Clear Logs & Alerts	The admin can clear all logs and alerts from the database to reset the detection cycle or after maintenance.
Upload CSV File	The admin uploads a CSV file that contains pre-extracted features from traffic. This is used by the anomaly-based IDS to perform machine learning classification.
View Anomaly Alerts	The admin can view alerts generated by the anomaly-based IDS, which uses a trained ML model to detect unknown or rare behavior in network data.

View Alert Statistics	The admin can analyze statistical summaries such as the most frequent source/destination IPs or the distribution of attack types using an Alert Analyzer module. This helps prioritize responses.

Table 3.3

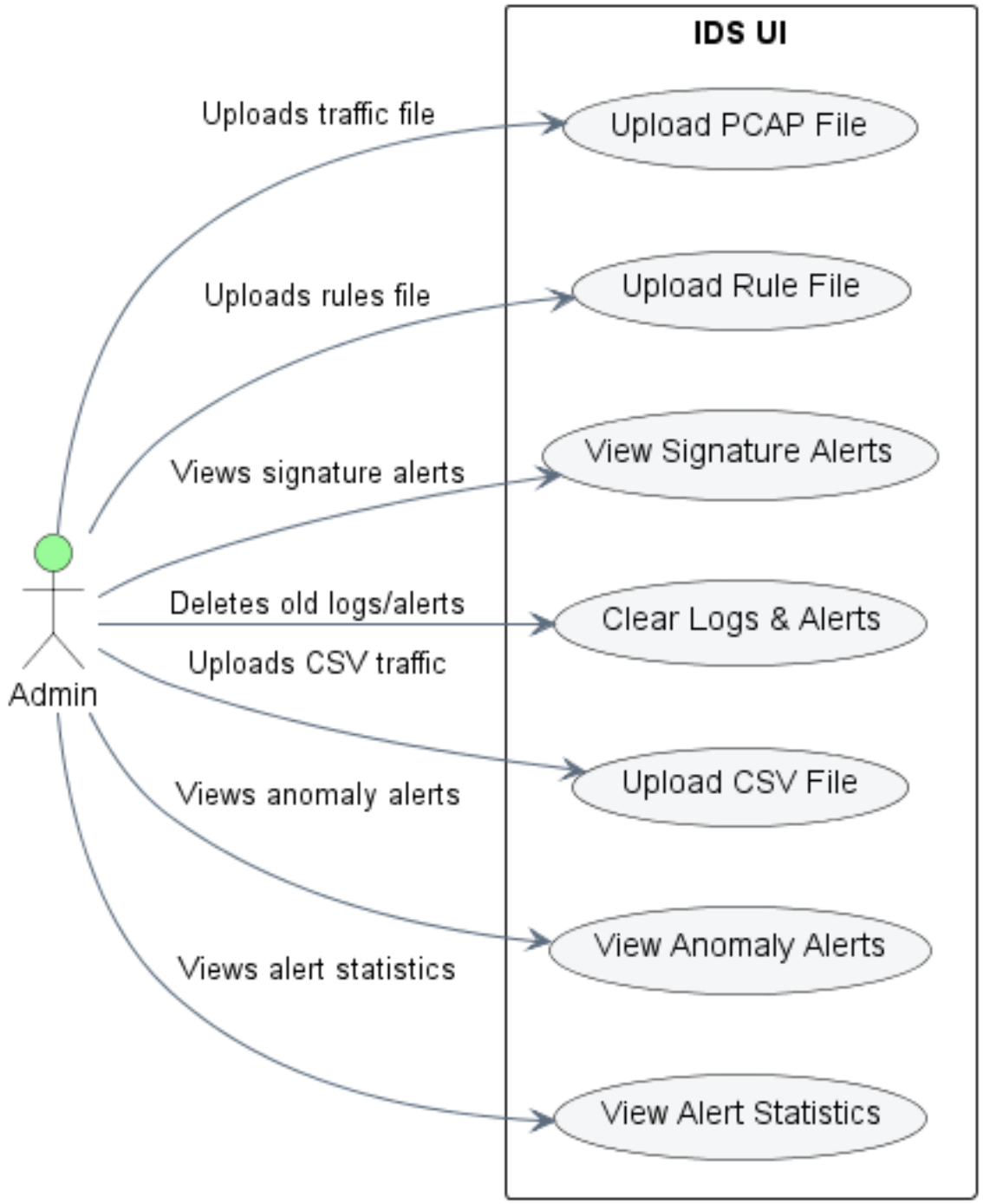

Figure 3.2

3.1.5.2. IDS System

- This Use Case Diagram focuses on internal automated operations handled by the System actor, which includes:

- Parsing and managing rule sets,
- Running detection engines (signature & anomaly),
- Logging & alerting,
- Statistical analysis on collected data.

Use Cases by category:

- **Shared Operations:**

Use Case	Description
Analyze Alerts	The system reviews alerts to perform further analysis such as frequency, IPs, and attack patterns. This feeds into the UI for visualization.
Analyze Top Src_Ips	Identifies and ranks the most frequent source IPs from which alerts have been generated. Useful in tracking potential attackers.
Analyze Top Dst_Ips	Like the source IPs, the system lists top targeted destination IPs for deeper inspection.

Table 3.4

- **Signature-Based IDS:**

Use Case	Description
Parse & Store Rules	When the Admin uploads a rule file, the system extracts the header and options and stores them in the database for real-time matching.

Start Signature Detection	Initiated after receiving packets (e.g., from a PCAP), the system starts comparing them with the stored signature rules.
Generate Logs	If a rule matches a packet, a log is written on the logs table indicating detection event details.
Generate Alerts	Simultaneously, an alert is issued indicating potential intrusion, stored in the alerts table with metadata.

Table 3.5

- **Anomaly-Based IDS:**

Use Case	Description
Start Anomaly Detection	After feature extraction from traffic or CSV upload, the anomaly engine (ML model) starts evaluating traffic behavior.
Generate Logs	For each classified anomaly, a log entry is generated for audit trail and review.
Generate Alerts	If a behavior is considered malicious or unusual by the model, an alert is triggered and recorded in the alert database.

Table 3.6

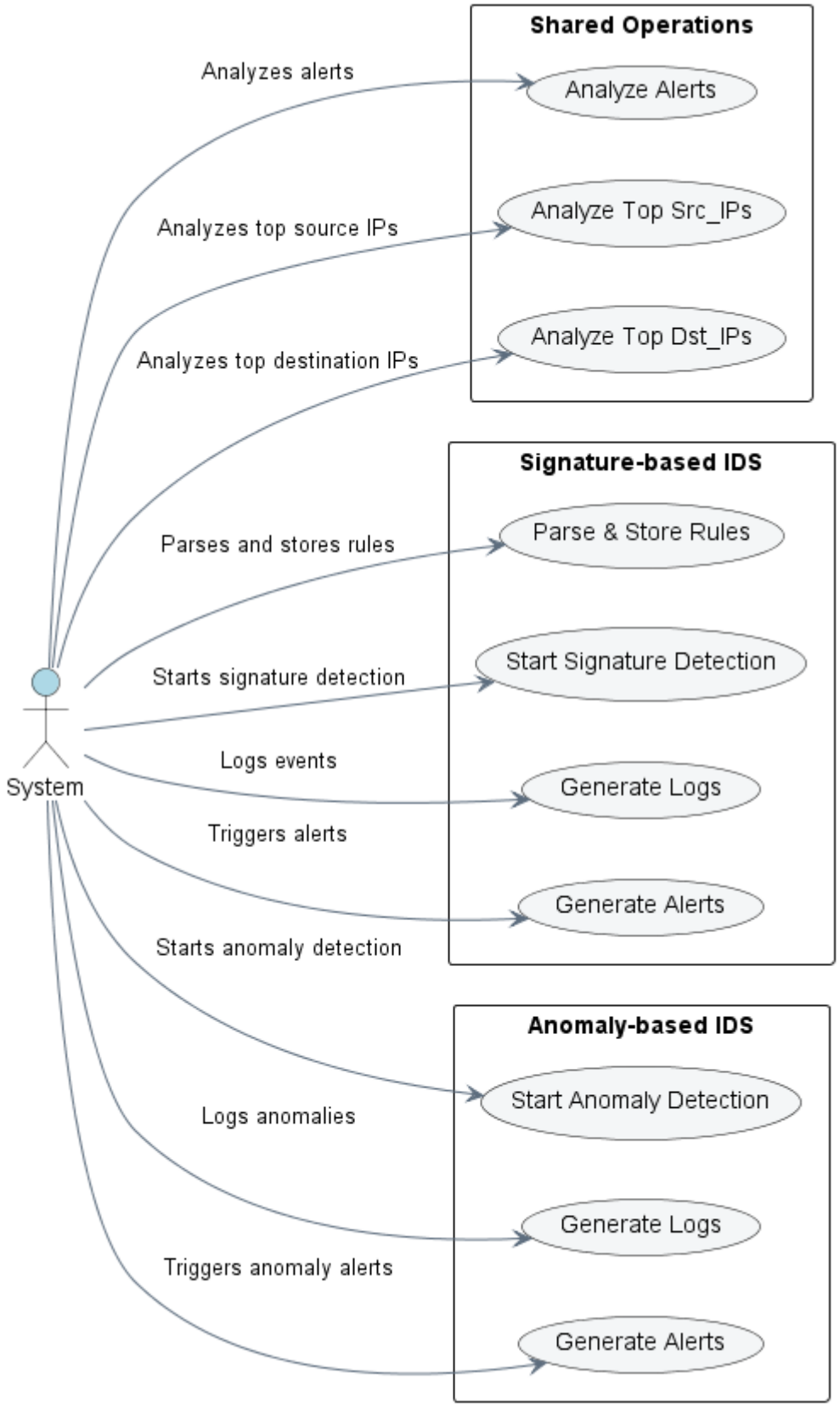
Shared Operations
Analyze Alerts
Analyze Top Src_IPs
Analyze Top Dst_IPs
Signature-based IDS
Parse & Store Rules
Start Signature Detection
Generate Logs
Generate Alerts
Anomaly-based IDS
Start Anomaly Detection
Generate Logs
Generate Alerts
System
Analyzes alerts
Analyzes top source IPs
Analyzes top destination IPs
Parses and stores rules
Starts signature detection
Logs events
Triggers alerts
Starts anomaly detection
Logs anomalies
Triggers anomaly alerts

Figure 3.3

3.1.6. System Component Diagram

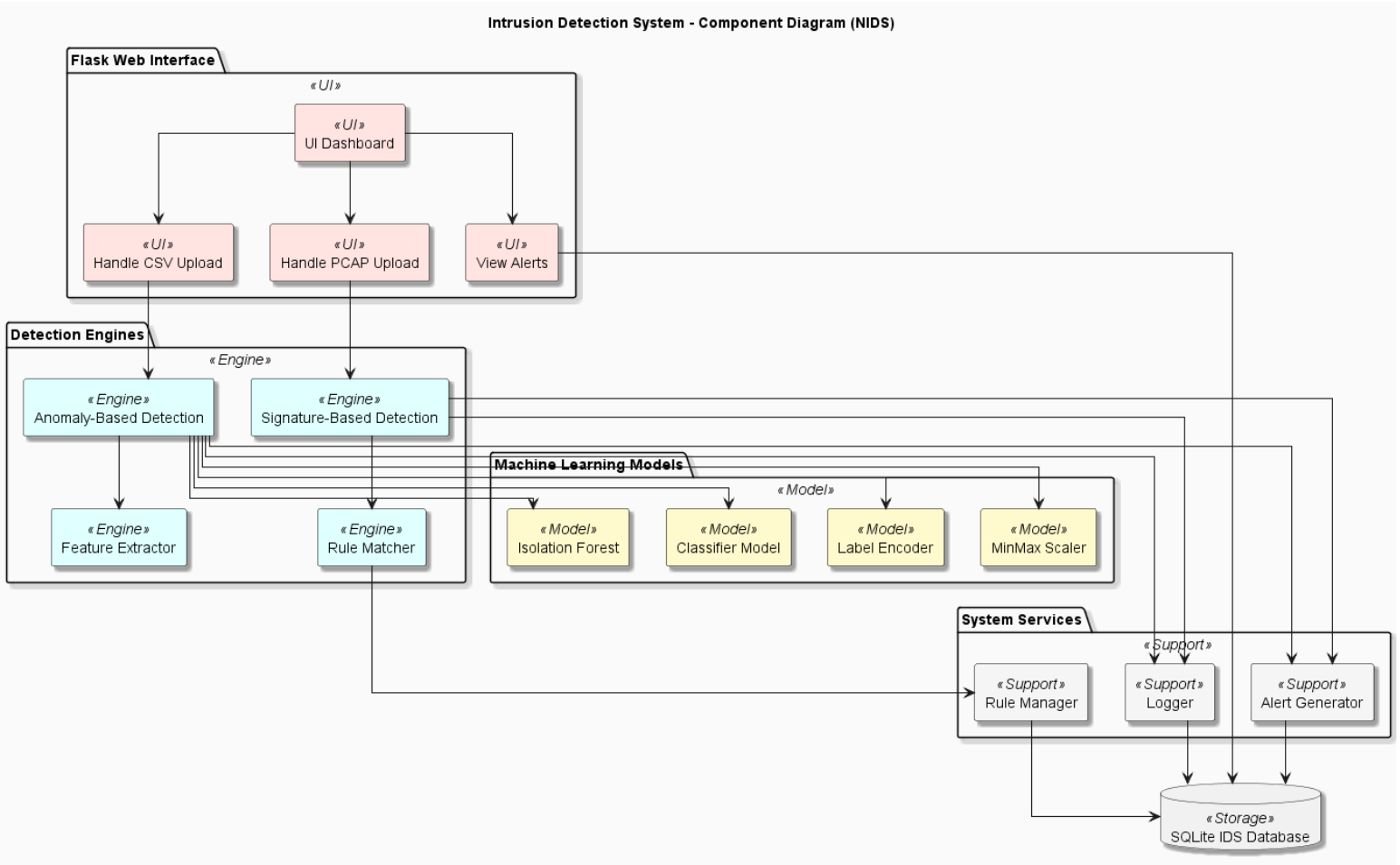

Figure 3.4

3.1.7. Class Diagram

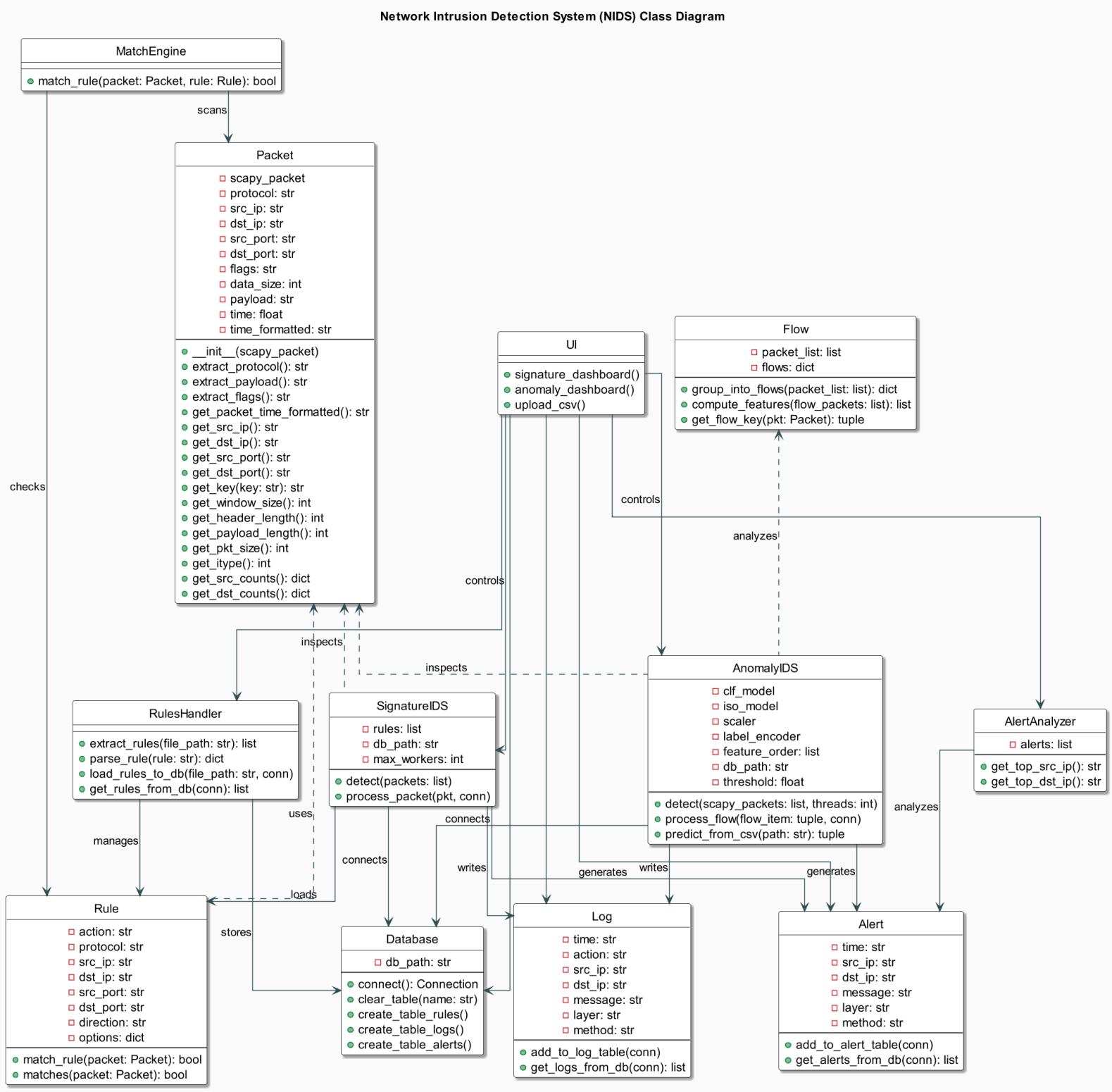

Figure 3.5

3.1.8. Sequence Diagram

3.1.8.1. System Initialization

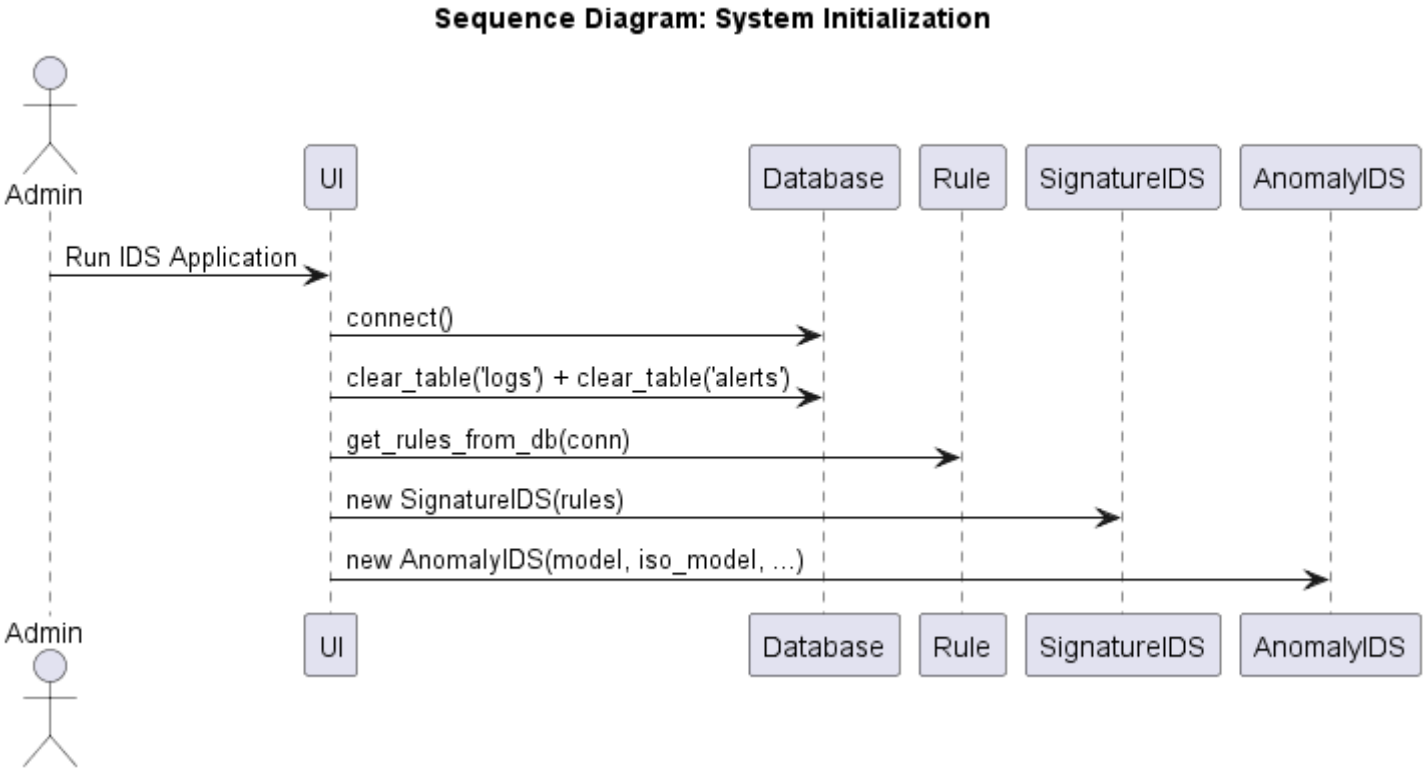

Figure 3.6

3.1.8.2. Add Rules to Database

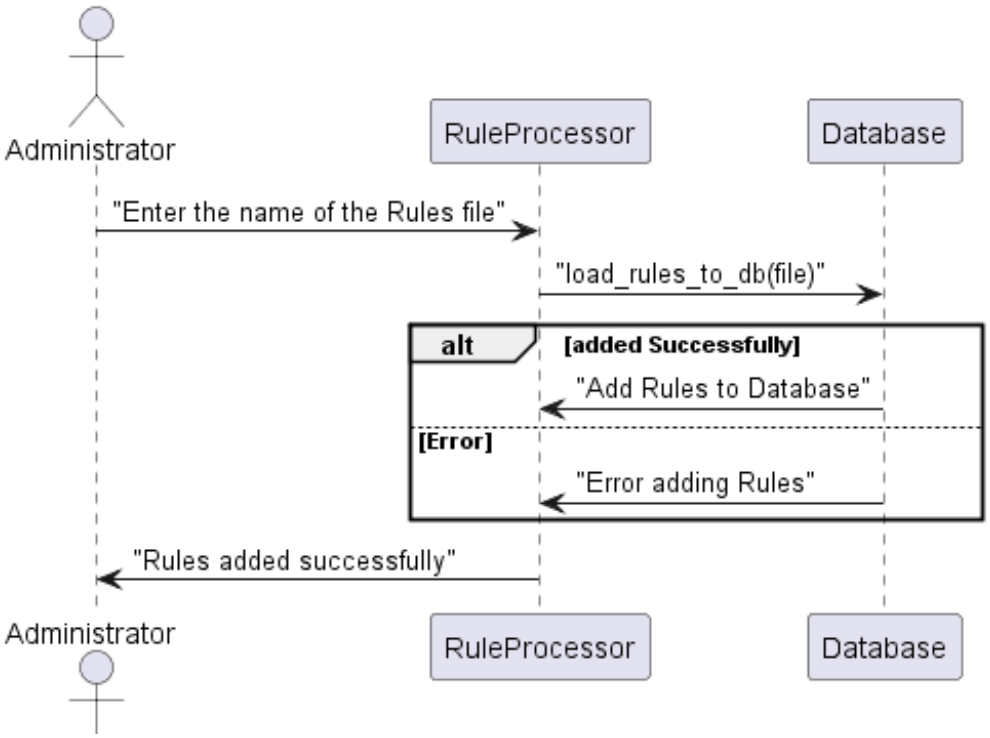

Figure 3.7

3.1.8.3. Detecting Attack with Signature-Based Method

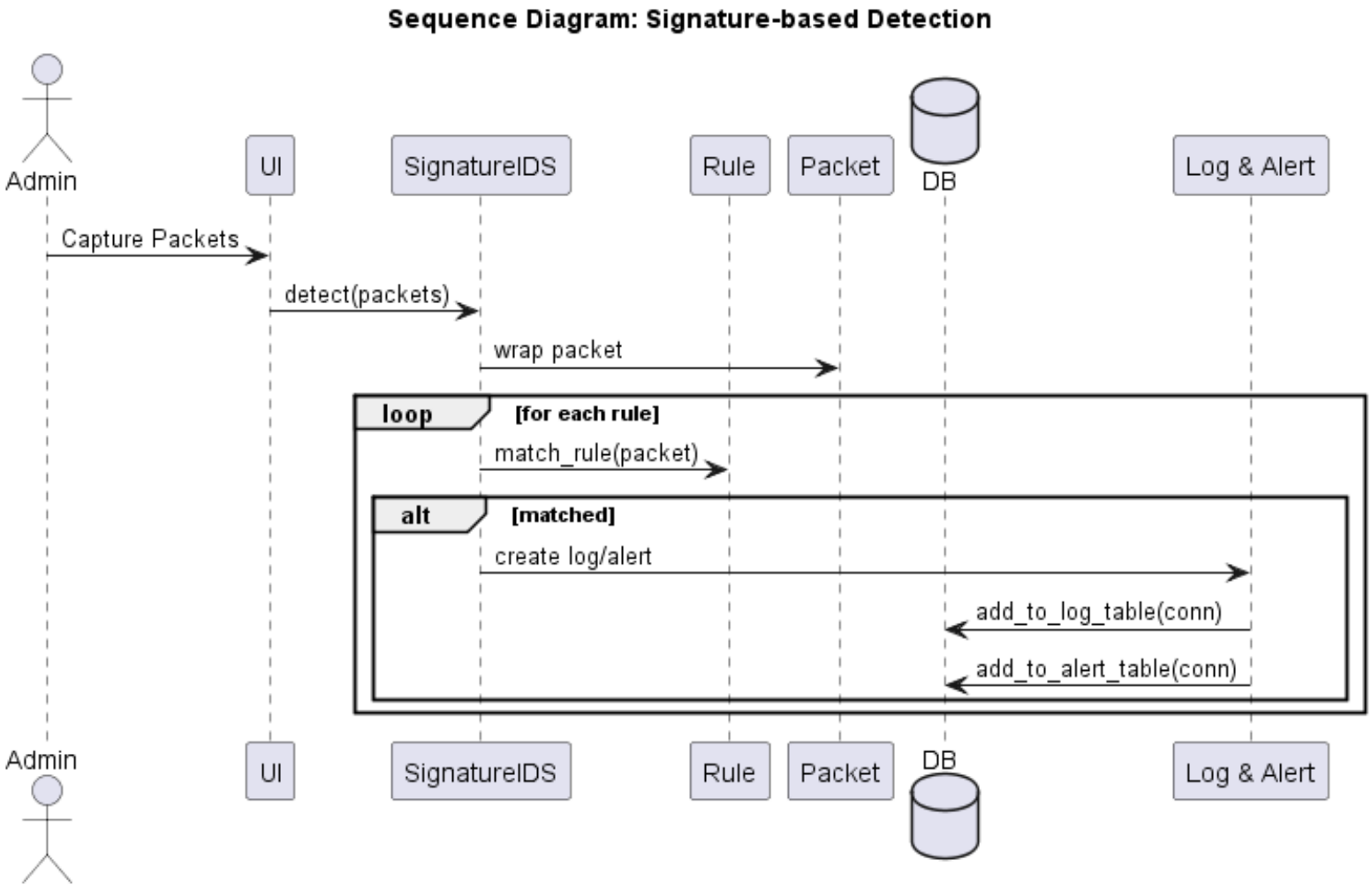

Figure 3.8

3.1.8.4. Detecting Attack with Anomaly-Based Method

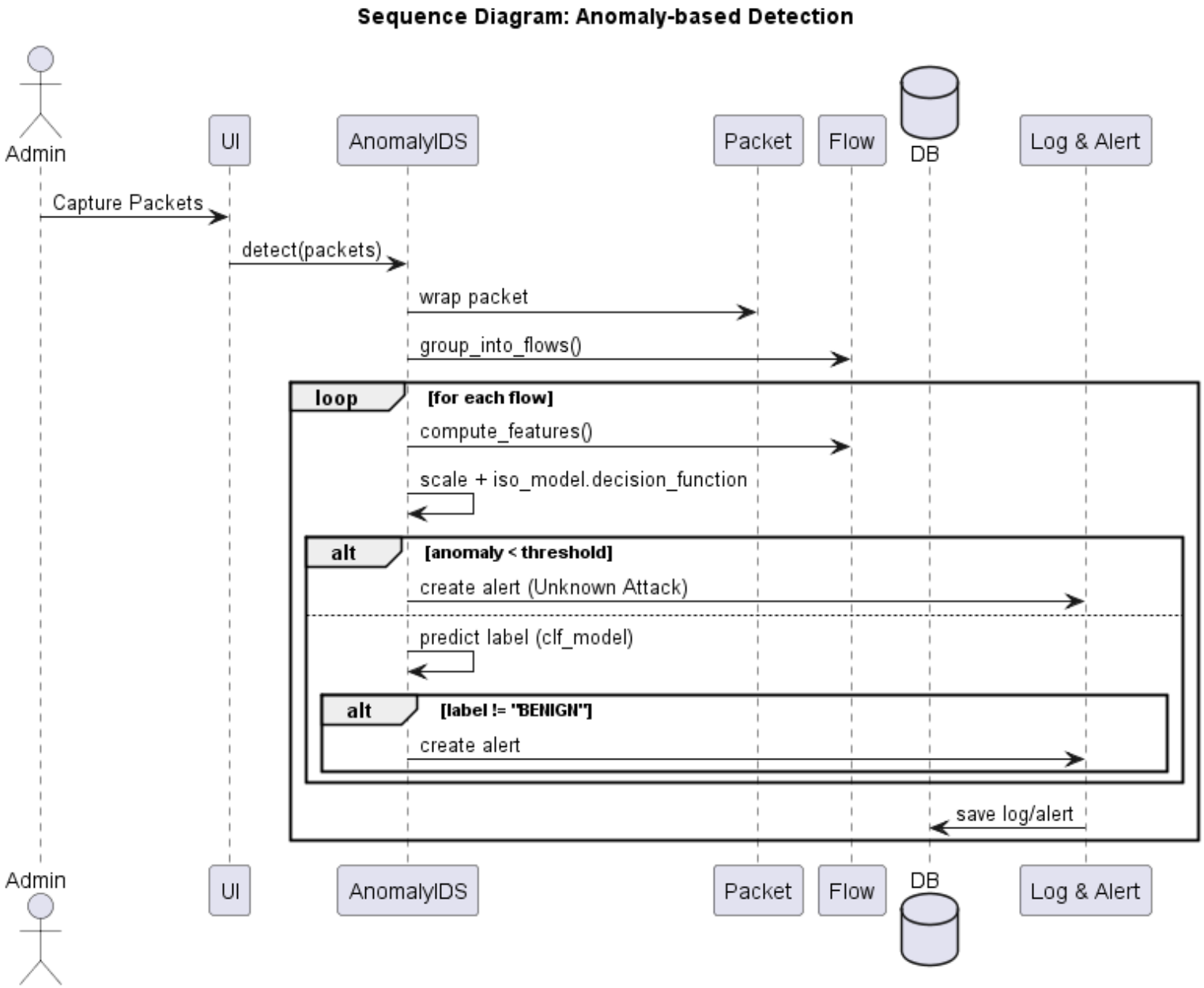

Figure 3.9

3.1.8.5. CSV Upload Detection

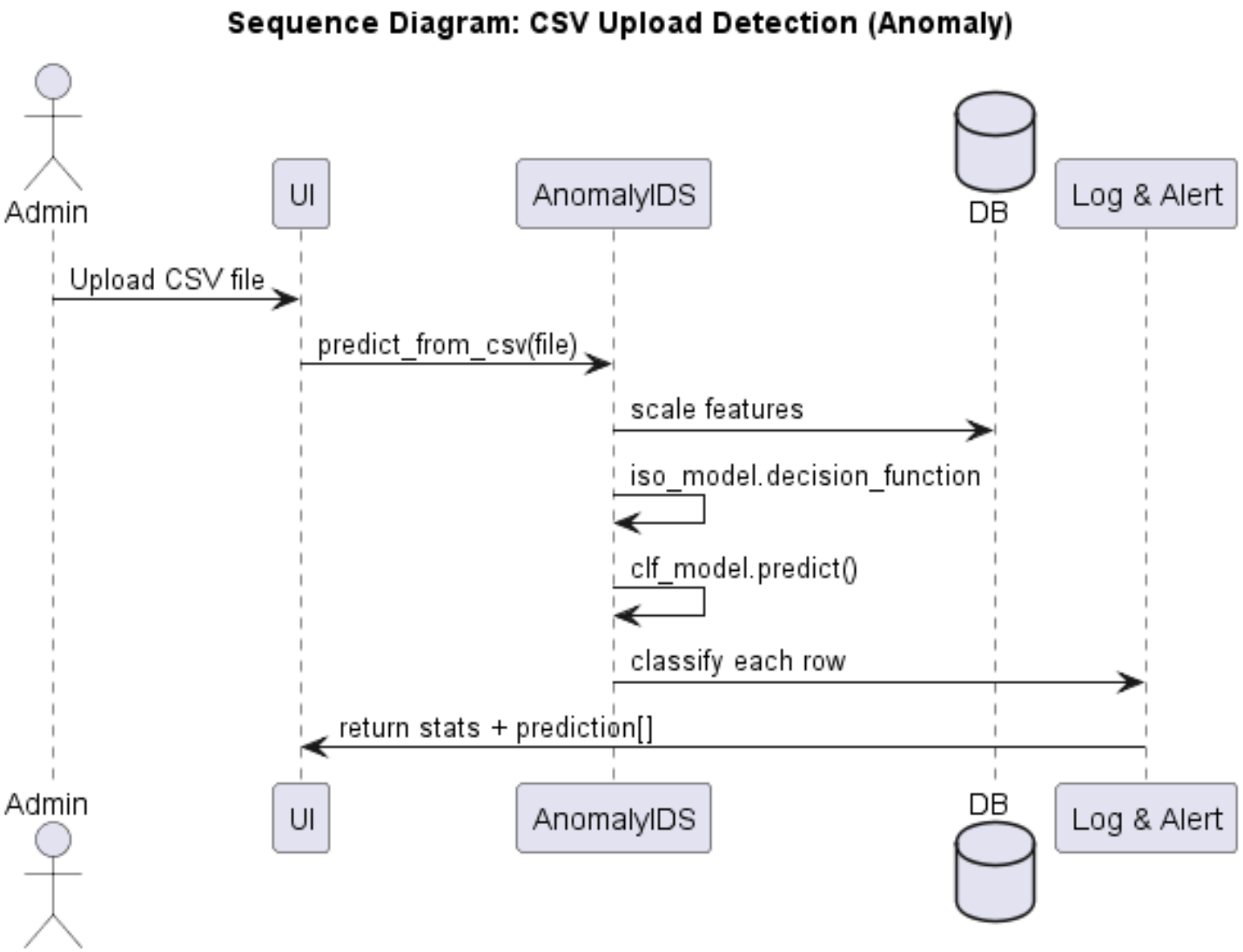

Figure 3.10

3.1.8.6. PCAP Upload Detection

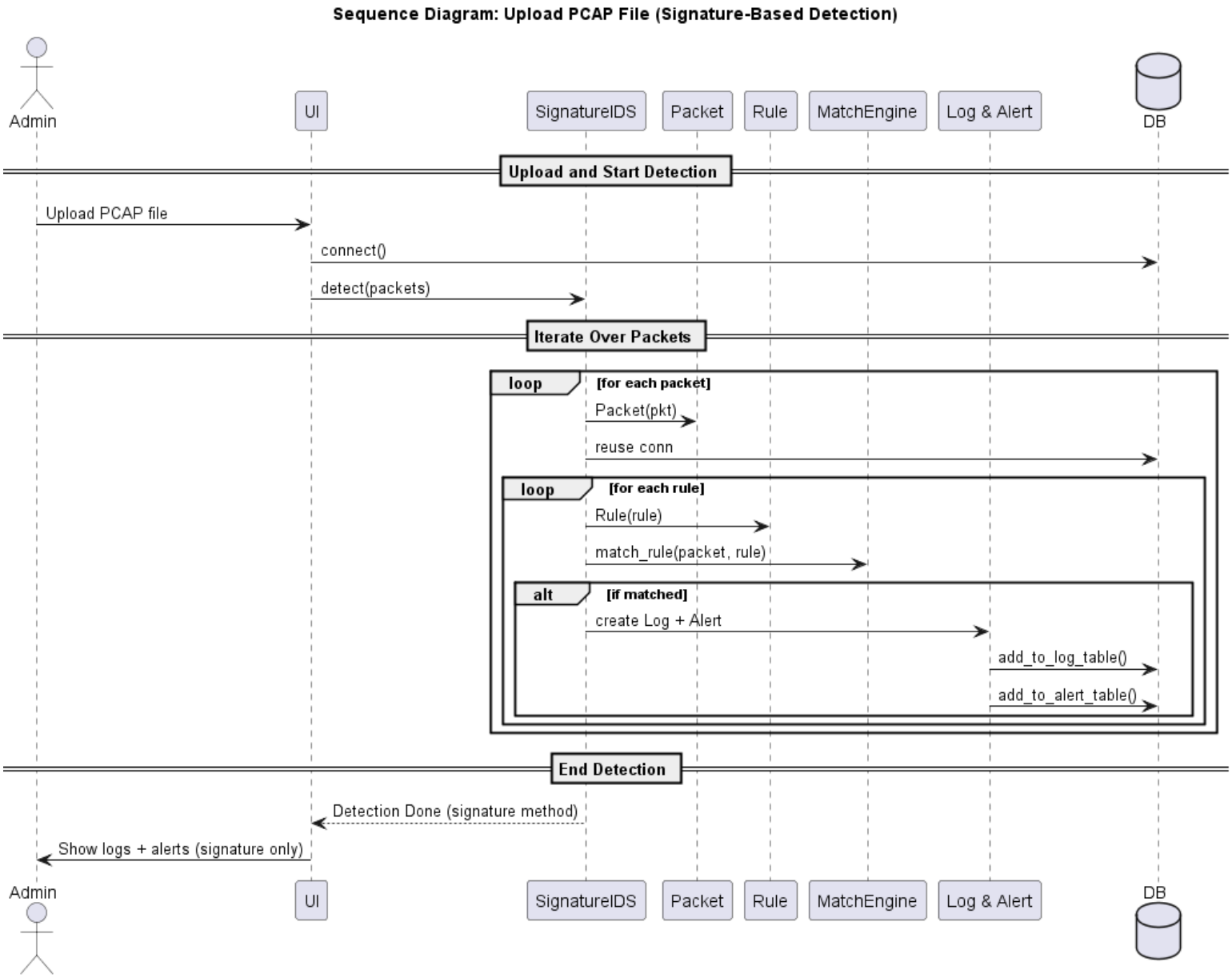

Figure 3.11

3.1.8.7. Web UI View

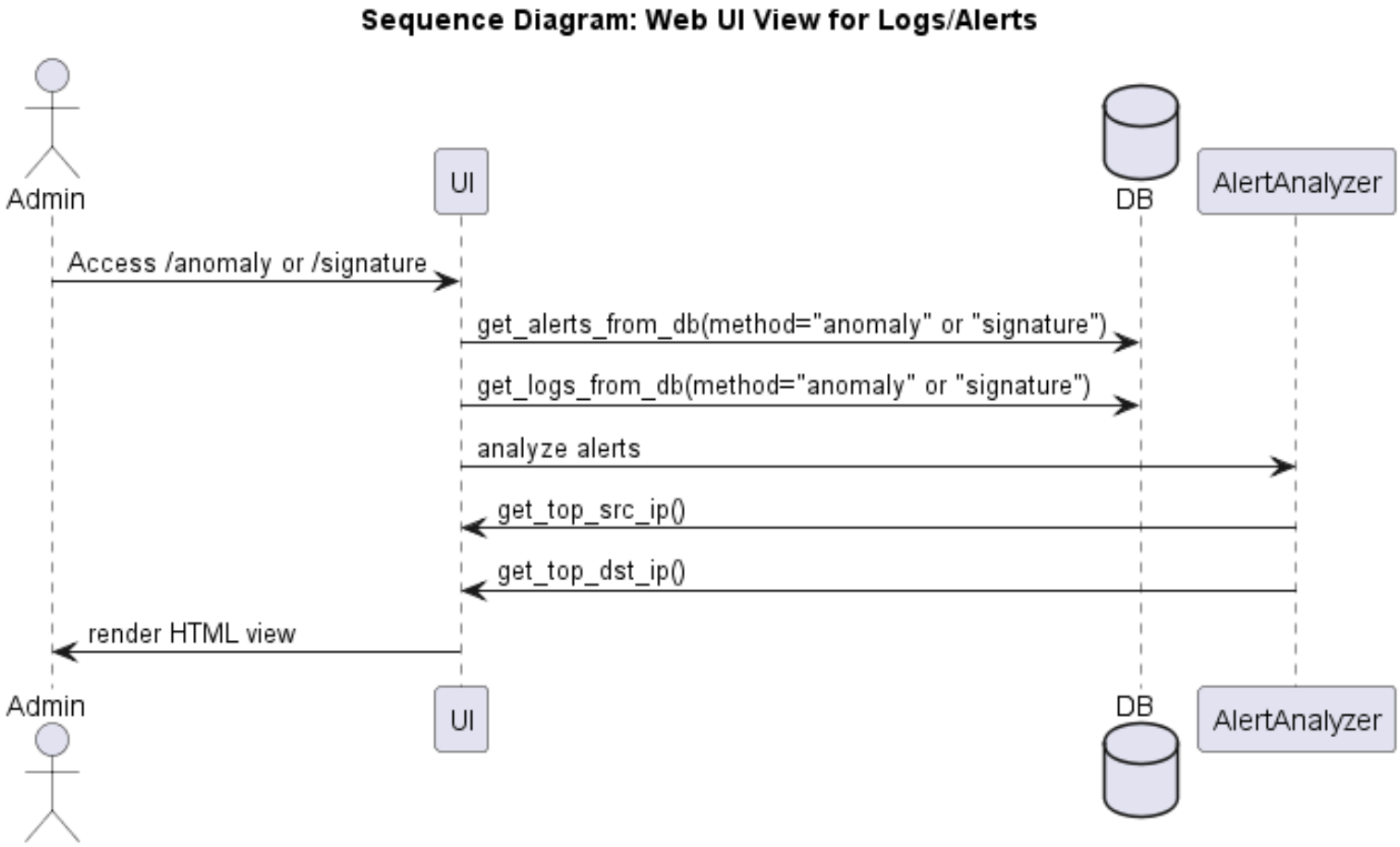

Figure 3.12

3.1.8.8. Display Rules for Admin

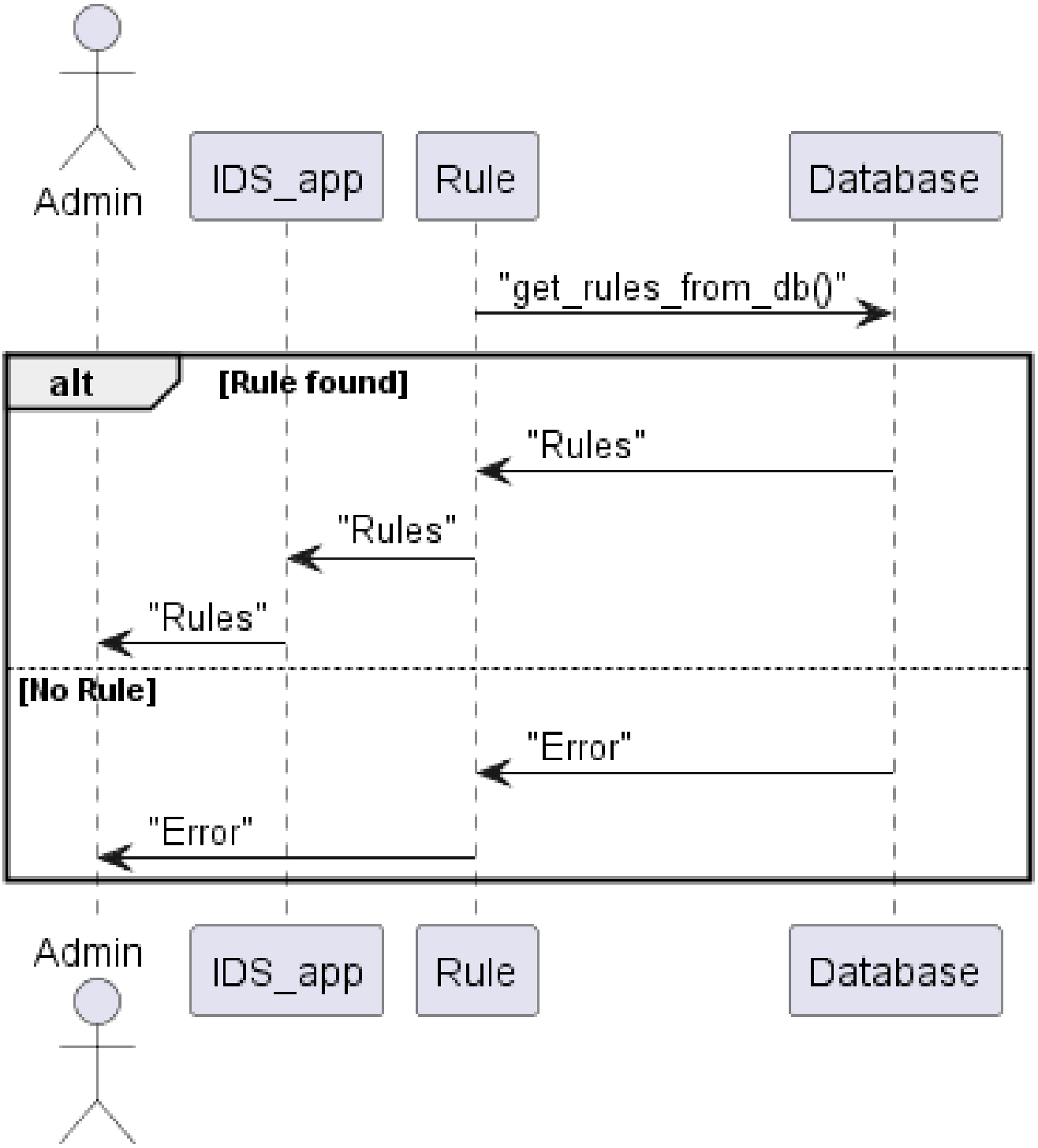

Figure 3.13

3.1.8.9. Display Alerts for Admin

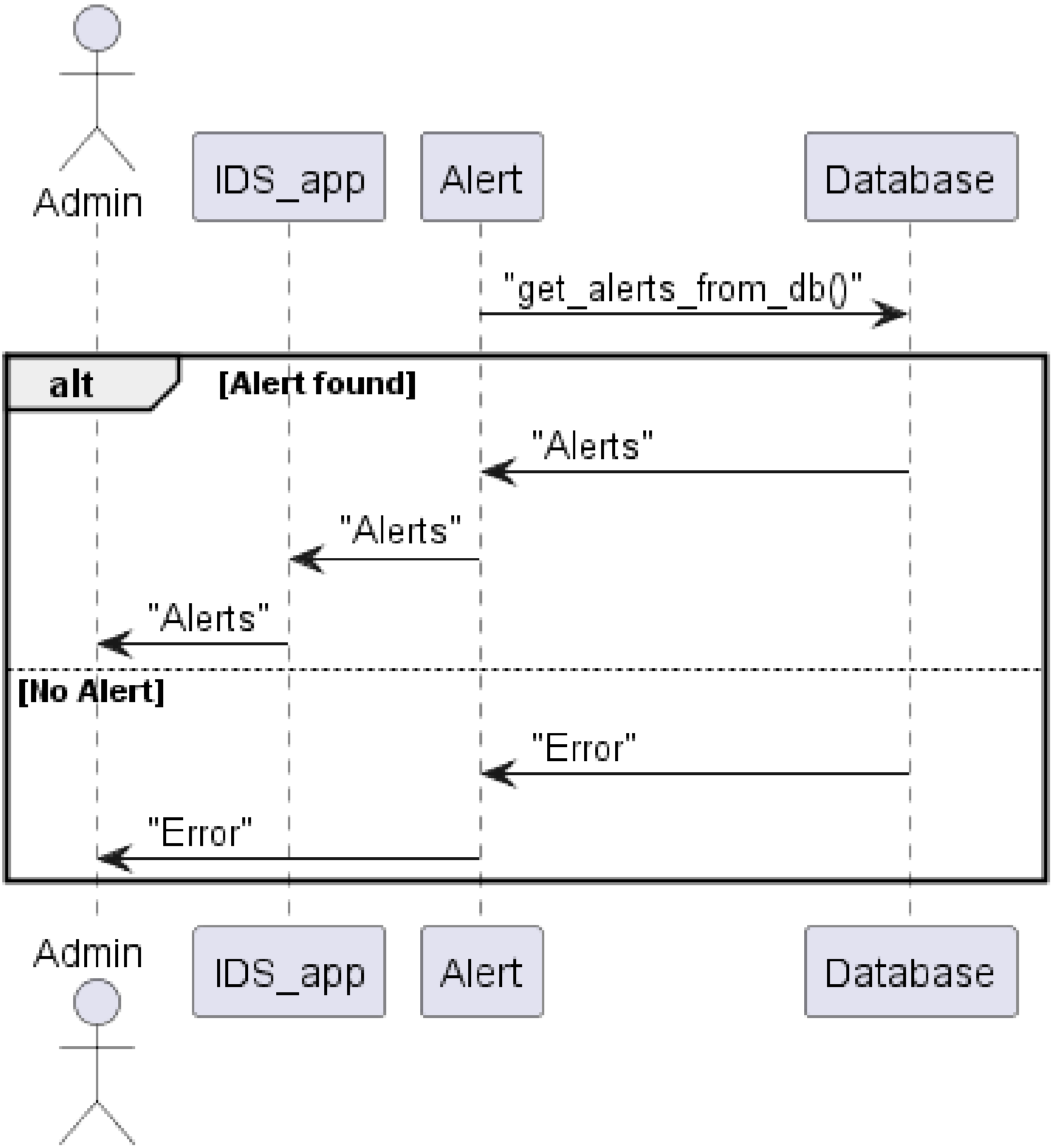

Figure 3.14

3.1.8.10.Display Logs for Admin

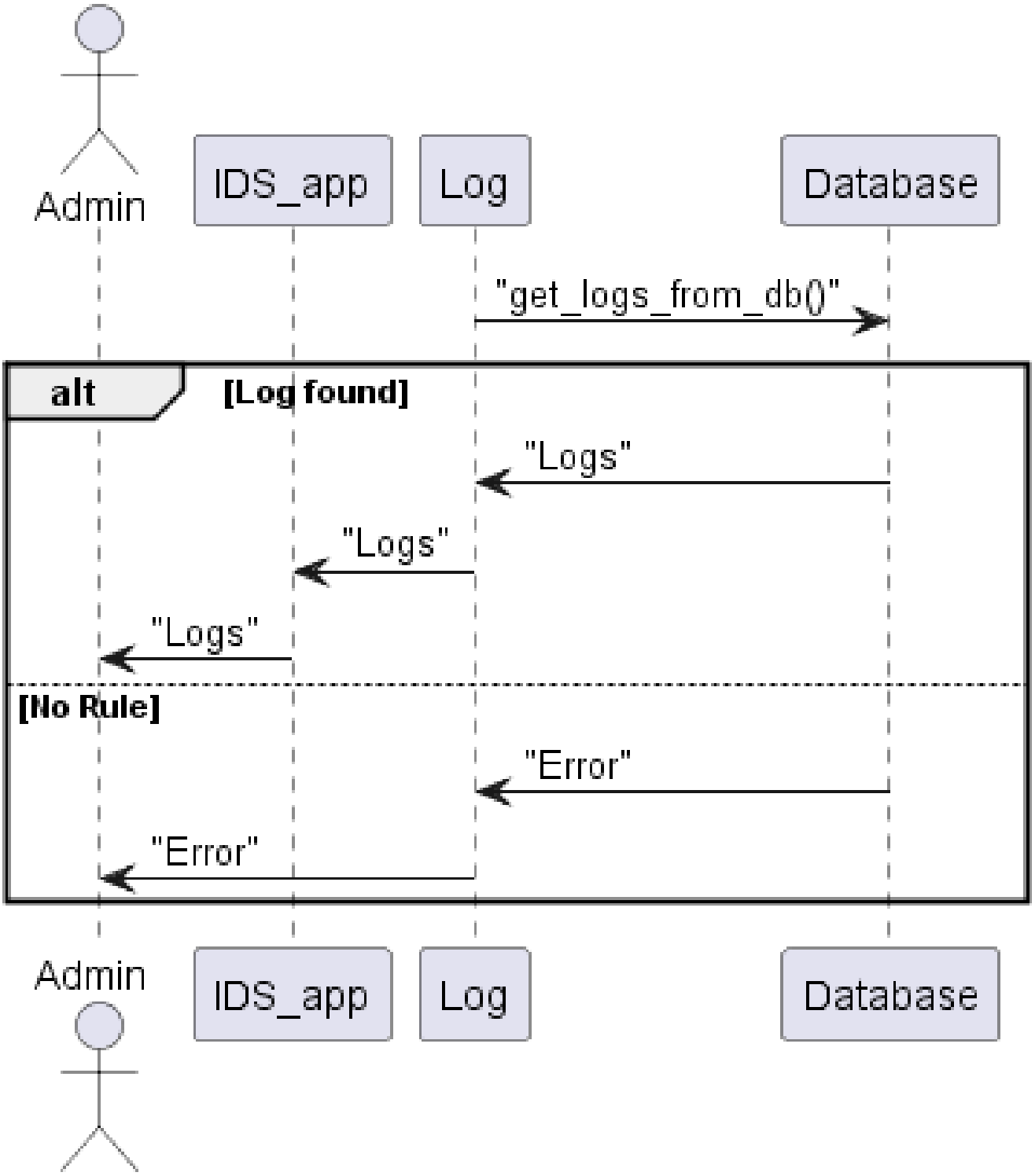

Figure 3.15

3.1.9. Activity Diagram

3.1.9.1. Display Logs for Admin

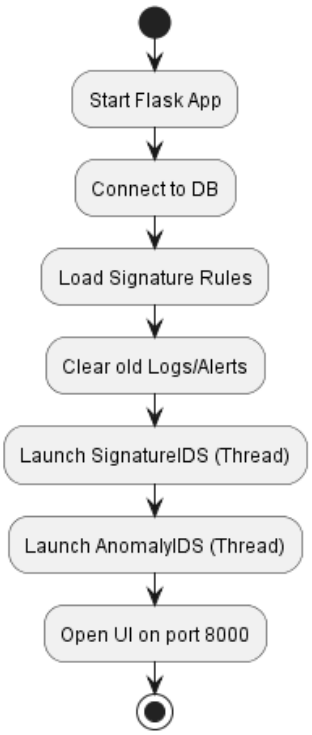

Figure 3.16

3.1.9.2. Adding Rules from File

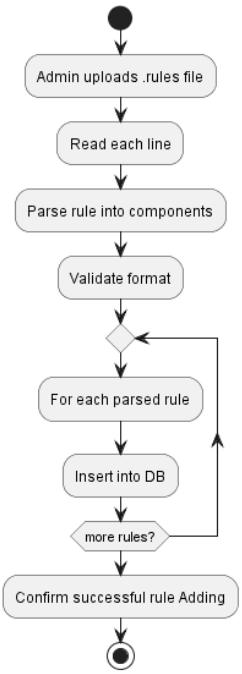

Figure 3.17

3.1.9.3. Alert Analyzer Dashboard

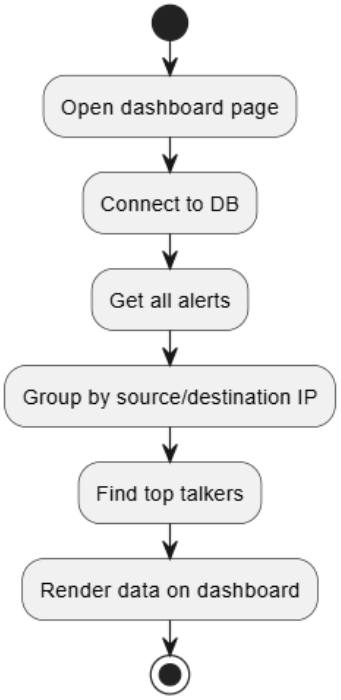

Figure 3.18

3.1.9.4. Anomaly Detection from CSV Upload

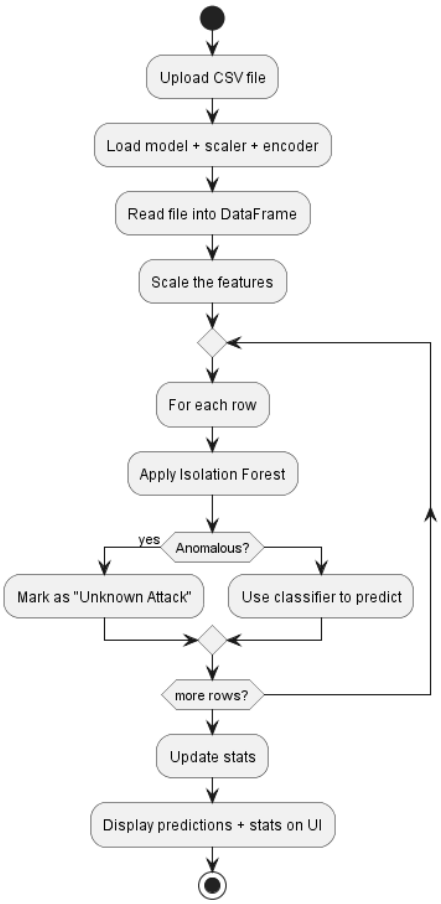

Figure 3.19

3.1.9.5. Real-Time Signature + Anomaly Detection

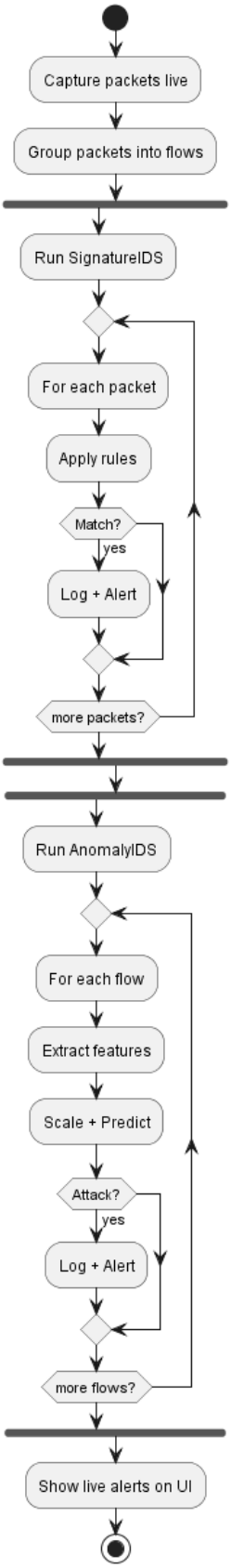

Figure 3.20

3.1.9.6. Signature-Based Detection from PCAP File

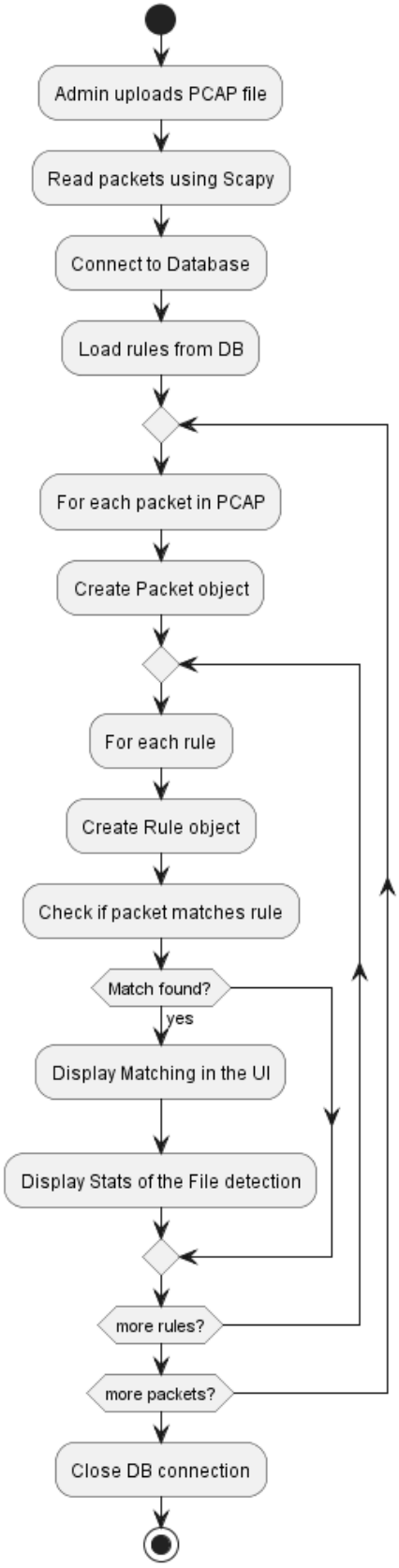

Figure 3.21

Chapter 4
Design, Implementation & Testing

4.1. System Design

4.1.1. Signature-Based Dashboard

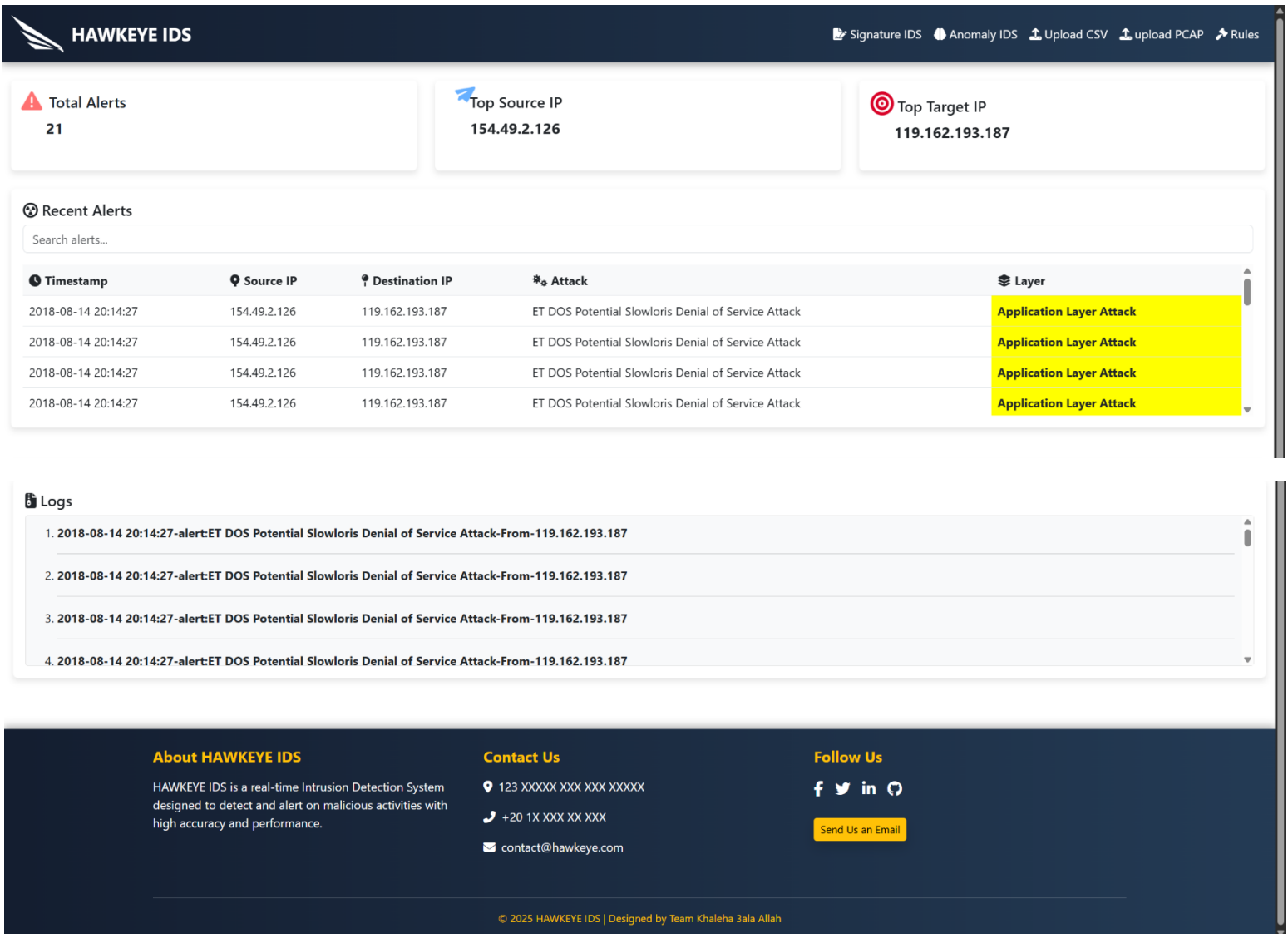

Figure 4.1

4.1.2. Anomaly-Based Dashboard

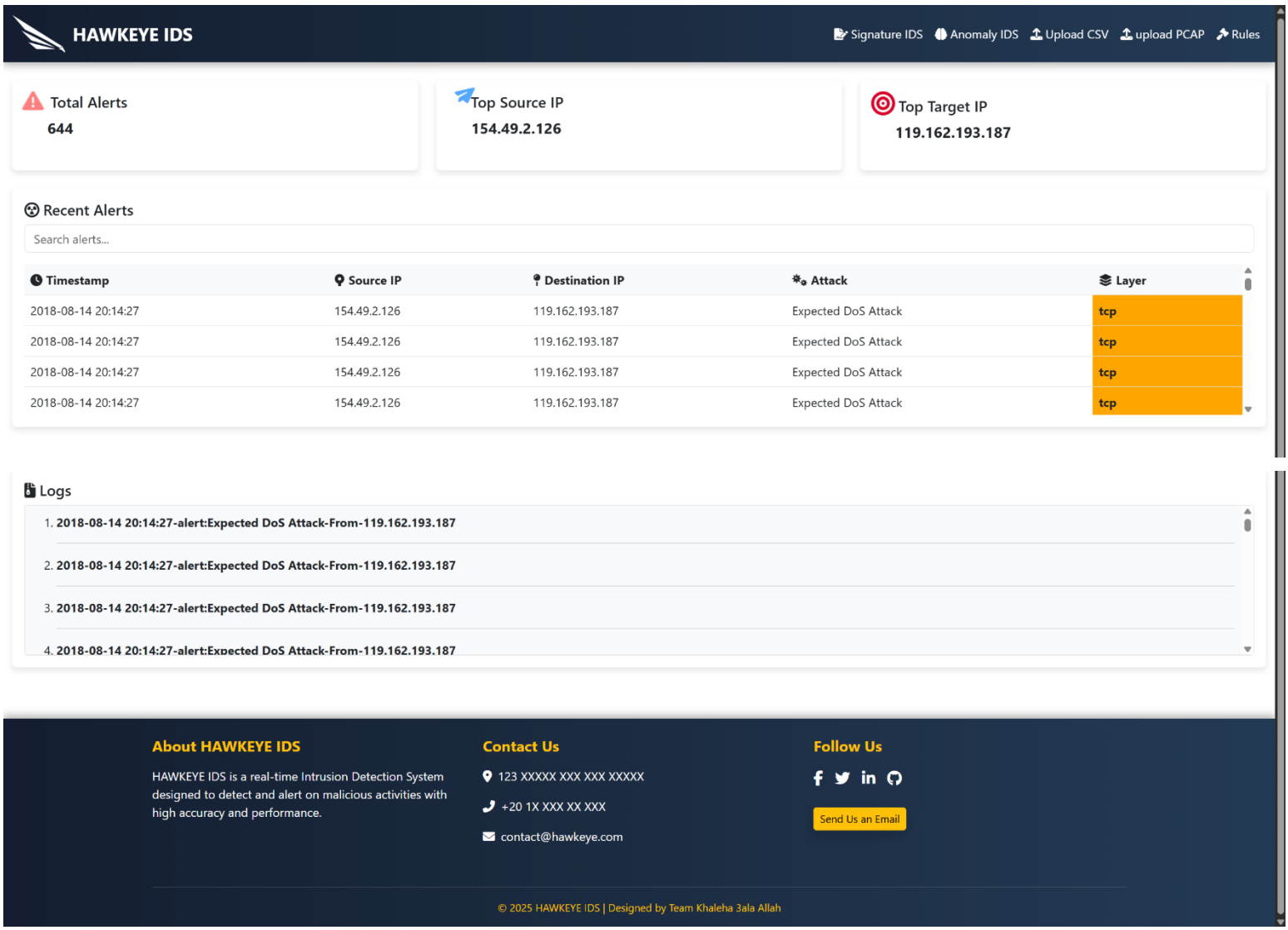

Figure 4.2

4.1.3. Upload-CSV File UI

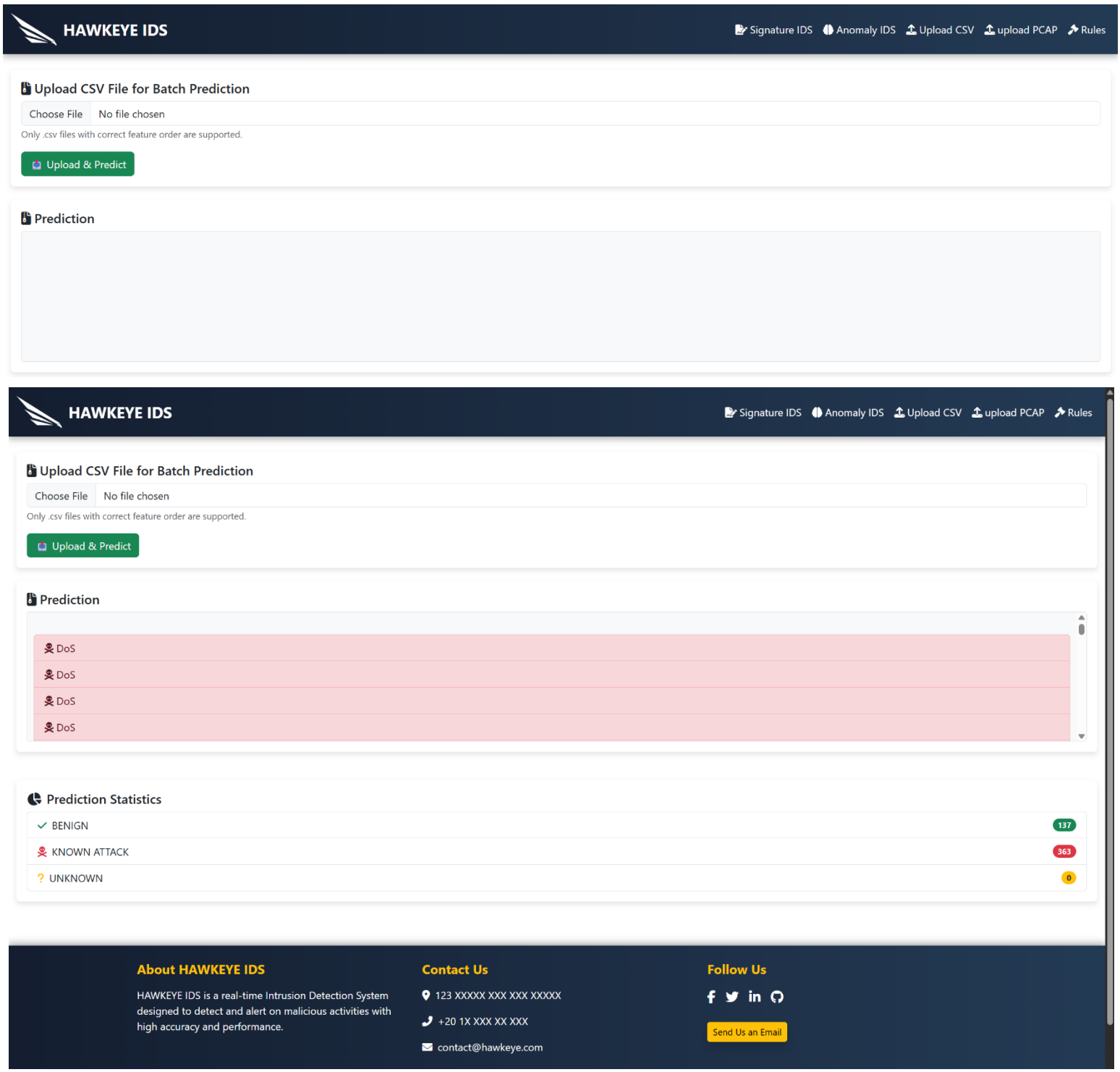

Figure 4.3

4.1.4. Upload-PCAP File UI

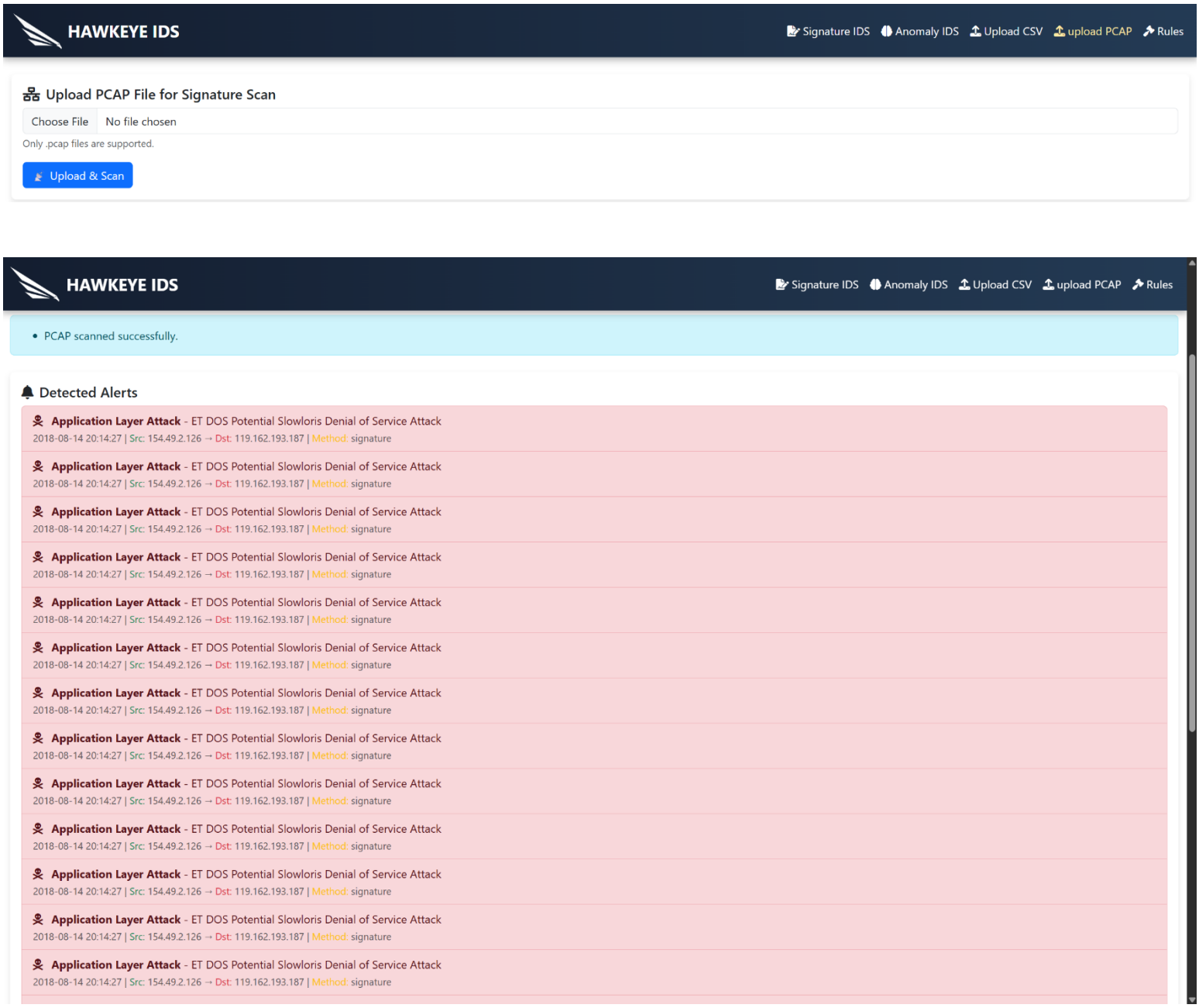

Figure 4.4

4.1.5. Rules-Dashboard

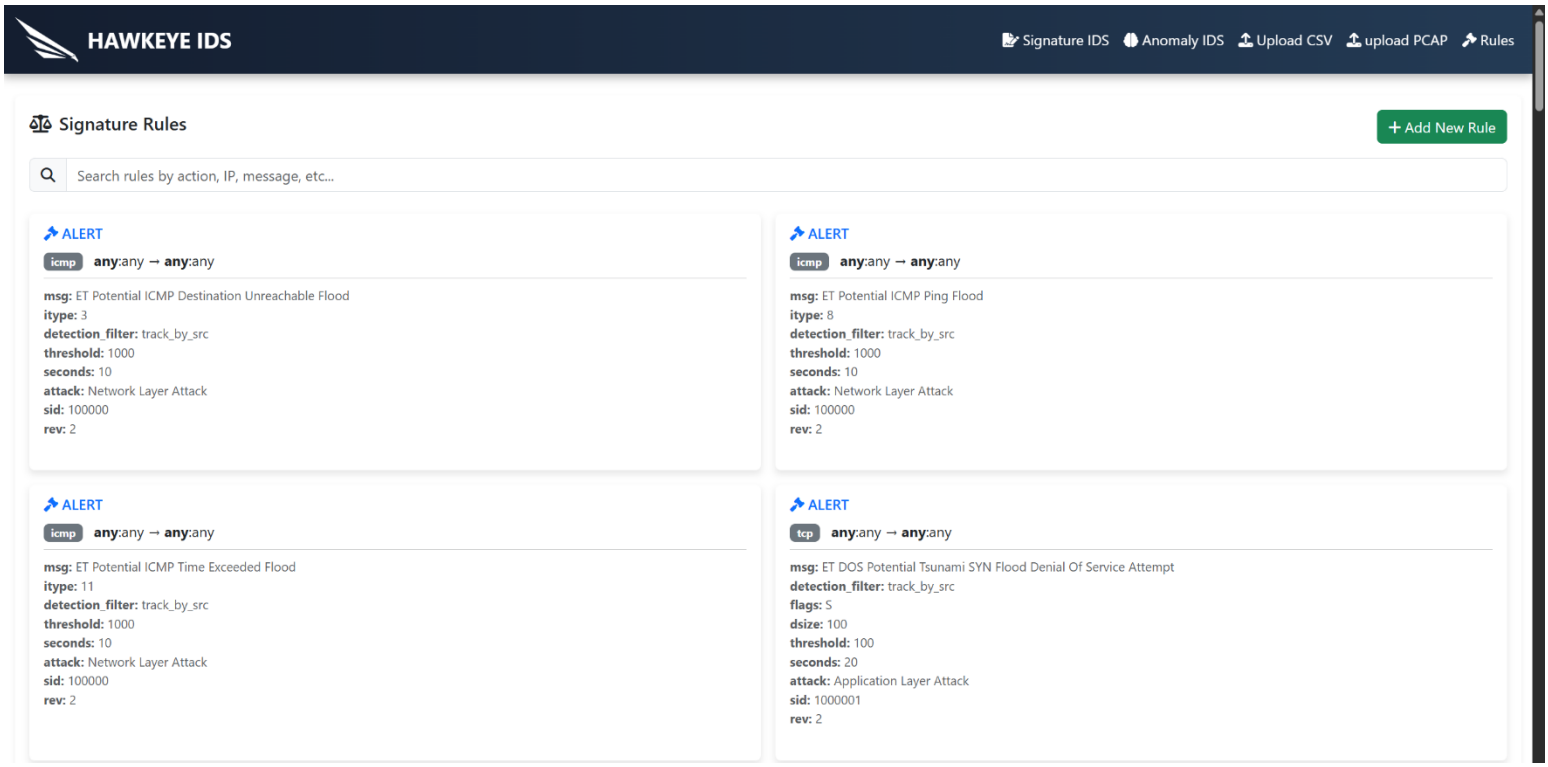

Figure 4.5

4.1.6. Add-Rules

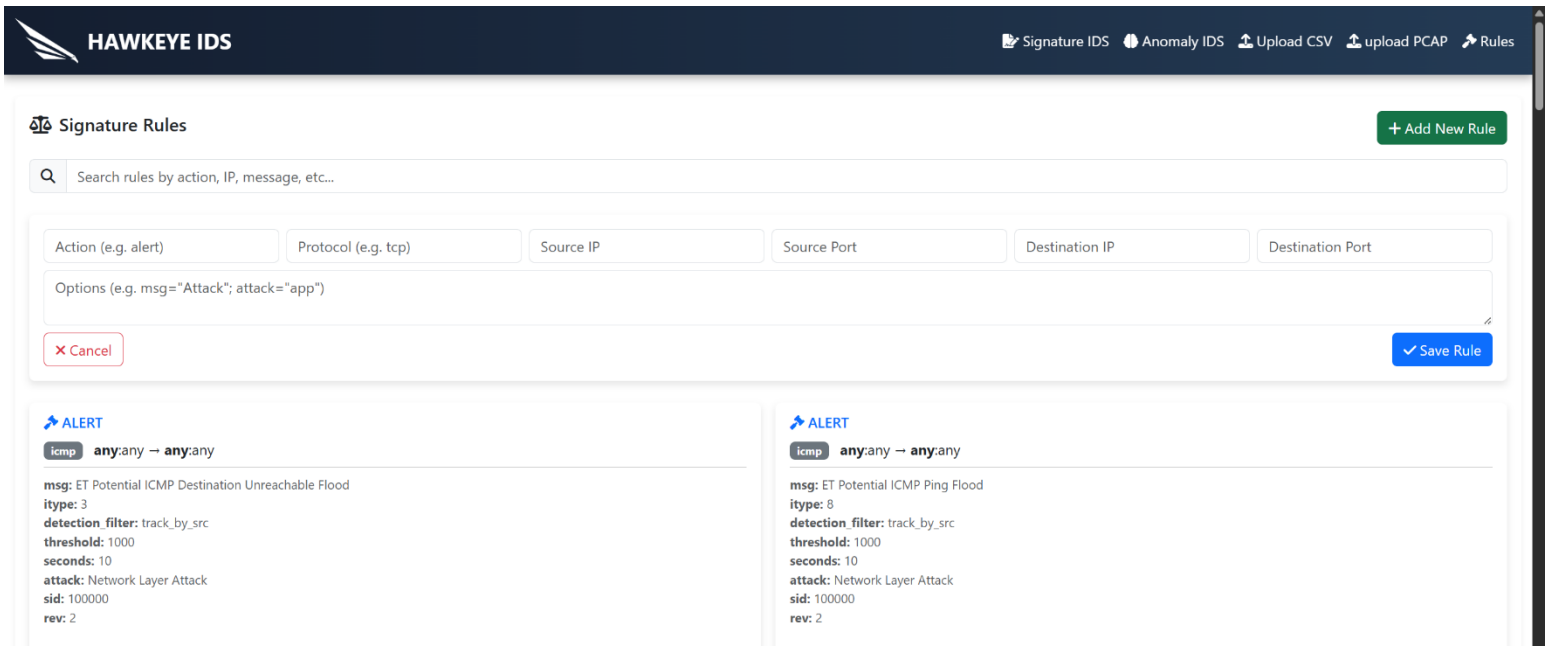

Figure 4.6

4.2. Implementation

4.2.1. Traffic PCAP Files

- **PCAP** (Packet Capture) files are a common format for storing network traffic data. These files contain the raw data packets captured from a network interface, including headers and payloads. **PCAP** files are widely used in network analysis, troubleshooting, and security investigations. They are particularly useful for testing and evaluating Intrusion Detection Systems (**IDS**) because they provide real-world or simulated network traffic that can be used to assess the IDS's ability to detect malicious activity.

- **PCAP** files used to evaluate **IDS** typically contain network traffic that includes both normal (**benign**) and **malicious** activities. These files are designed to evaluate the effectiveness, accuracy, and performance of an **IDS** in detecting and responding to several types of threats.

- It is used to validate the detection capabilities of an **IDS**, to test the **IDS's** ability to differentiate, between normal and malicious traffic, and to evaluate the **IDS's** performance under different traffic conditions.

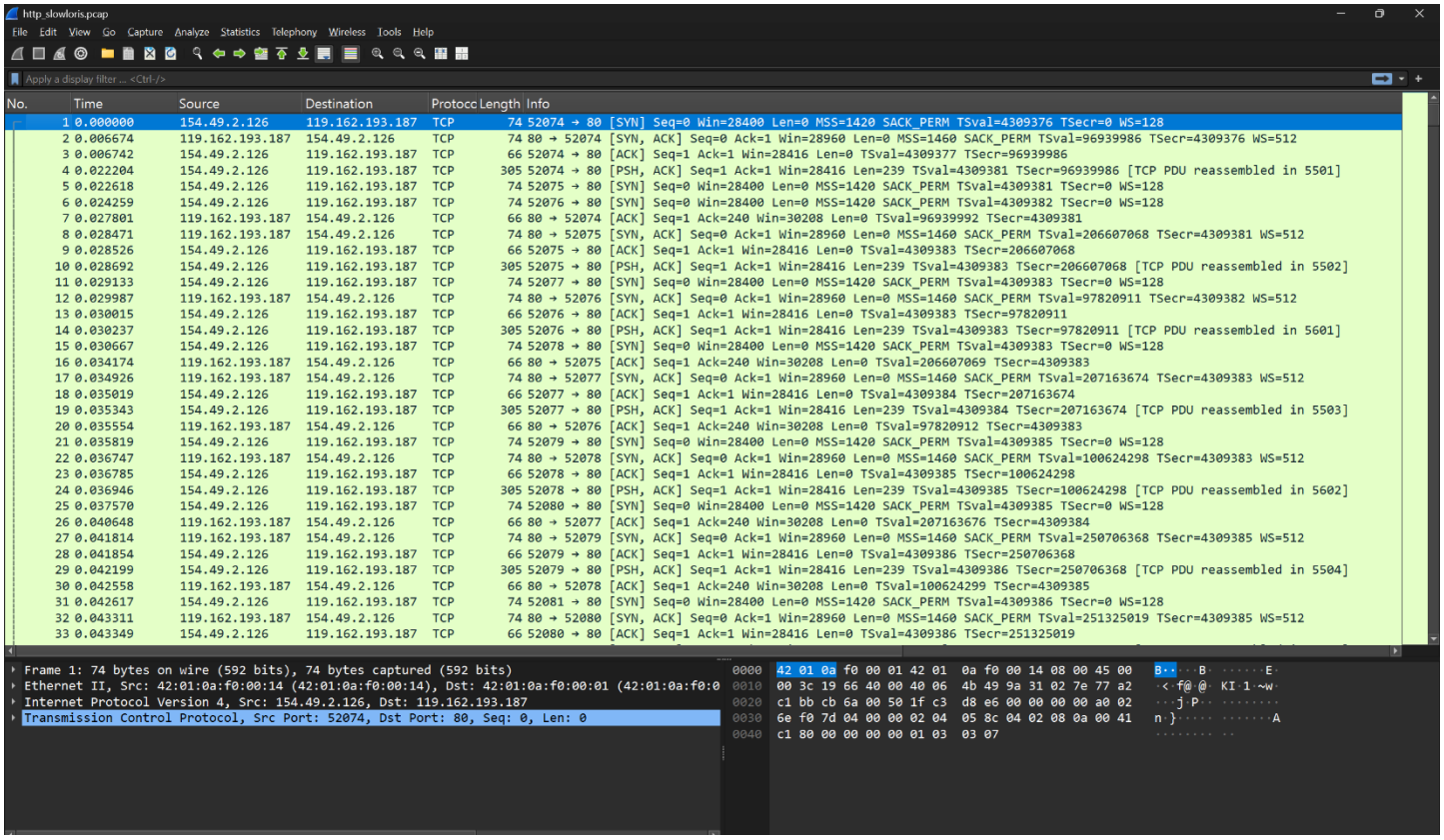

Figure 4.7

4.2.2. Database

- **The Intrusion Detection System (IDS)** relies on an **SQLite-based** relational database to support both **Signature-based** and **Anomaly-based** detection components. **SQLite** was selected due to its lightweight architecture and seamless integration with Python, making it suitable for real-time processing without requiring a separate database server.

- **Database Structure:**
  - The database consists of the following key tables:
    - **Rules**
      - Stores signature-based detection rules, each containing fields such as action, protocol, source and destination addresses, ports, and options.
    - **Alerts**
      - Records all generated alerts, regardless of whether they were triggered by a signature match or an anomaly detection model. Each alert entry includes metadata such as timestamp, source/destination IPs, attack type, and detection method (e.g., **'signature'** or **'anomaly'**).
    - **Logs**
      - Captures system events, packet evaluations, and audit trails. This table ensures visibility into the system's decision-making process and provides context for each detection.

- **Functionality:**
  - During runtime, the system:
    - Loads predefined signature rules from the Rules table.
    - Processes incoming network packets using both detection engines.
    - If a packet matches a rule or is classified as anomalous, a corresponding entry is created in the Alerts table.
    - All events, including normal traffic evaluations and generated alerts, are stored in the Logs table for auditing.
  - A centralized **Database** class handles:
    - Connecting to **IDS.db** located under the **DB/** directory.
    - Creating tables if they do not already exist.
    - Resetting or cleaning data at system startup.
    - Executing all insert, update, and retrieval operations for **logs**, **alerts**, and **rules**.

- **Integration:**
  - The database plays a pivotal role in unifying the outputs of the two detection methodologies and acts as the central repository for:
    - Storing detection logic (rules).
    - Logging analysis history.
    - Facilitating alert presentation in the web interface.

- Now we show the code used to perform all of this:

```
class Database:
    def __init__(self, db_name):
        self.db_name = db_name
        self.conn = None


    def connect(self):

        try:
            self.conn = sqlite3.connect(self.db_name, check_same_thread=False, timeout=10)
            self.conn.row_factory = sqlite3.Row
            self.conn.execute('PRAGMA journal_mode=WAL;')
            return self.conn

        except sqlite3.Error as e:
            print(f"[Error] Failed to connect to the database: {e}")
            return None

    def create_table_rules(self):

        create_table_rules = """
        CREATE TABLE IF NOT EXISTS rules
        (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            action TEXT,
            protocol TEXT,
            src_ip TEXT,
            src_port TEXT,
            direction TEXT,
            dst_ip TEXT,
            dst_port TEXT,
            options TEXT
        );
        """

        try:
            cursor = self.conn.cursor()
            cursor.execute(create_table_rules)
            self.conn.commit()
            print("[Success] 'rules' table created or already exists.")

        except sqlite3.Error as e:
            print(f"[Database Error] {e}")
```

Figure 4.8

```python
def create_table_logs(self):

        create_table_logs = """
        CREATE TABLE IF NOT EXISTS logs
        (
            id INTEGER PRIMARY KEY,
            timestamp DATETIME,
            event_type TEXT,
            src_ip TEXT,
            dst_ip TEXT,
            message TEXT,
            attack TEXT,
            method TEXT
        );
        """

        try:
            cursor = self.conn.cursor()
            cursor.execute(create_table_logs)
            self.conn.commit()
            print("[Success] 'logs' table created or already exists.")

        except sqlite3.Error as e:
            print(f"[Database Error] {e}")

    def create_table_alerts(self):

        create_table_alerts = """
        CREATE TABLE IF NOT EXISTS alerts
        (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            timestamp DATETIME,
            src_ip TEXT,
            dst_ip TEXT,
            message TEXT,
            attack TEXT,
            method TEXT
        );
        """

        try:
            cursor = self.conn.cursor()
            cursor.execute(create_table_alerts)
            self.conn.commit()
            print("[Success] 'alerts' table created or already exists.")

        except sqlite3.Error as e:
            print(f"[Database Error] {e}")

    def clear_table(self, table_name):

        try:
            if not self.conn:
                print("[Error] No database connection.")
                return False

            cursor = self.conn.cursor()
            cursor.execute(f"DELETE FROM {table_name};")
            self.conn.commit()
            print(f"[Success] All records from '{table_name}' have been deleted.")
            return True

        except sqlite3.Error as e:
            print(f"[Database Error] {e}")
            return False
```

Figure 4.9

- Now this is the result after running the previous code:

Name	Type	Schema
Tables (4)		
alerts		CREATE TABLE alerts (id INTEGER PRIMARY KEY AUTOINCREMENT, timestamp DATETIME, src_ip TEXT, dst_ip TEXT, message TEXT, attack TEXT)
logs		CREATE TABLE logs (id INTEGER PRIMARY KEY, timestamp DATETIME, event_type TEXT, src_ip TEXT, dst_ip TEXT, message TEXT, attack TEXT)
rules		CREATE TABLE rules (id INTEGER PRIMARY KEY AUTOINCREMENT, action TEXT, protocol TEXT, src_ip TEXT, src_port TEXT, direction TEXT, dst_ip TEXT, dst_port TEXT, options TEXT)

Figure 4.10

4.2.3. Snort / Suricata Rules

- **Snort** and **Suricata** are open-source **Intrusion Detection and Prevention Systems (IDS/IPS)**, meaning they are freely available and can be used across multiple platforms including Linux, Windows, and others.
- These systems monitor network traffic in real time and analyze it against a predefined set of rules with the objective of identifying suspicious or potentially malicious activities.
- Rules in Snort or Suricata serve as detailed instructions that define how the system should respond to specific patterns or behaviors within the network traffic. They function as structured guidelines or filters to detect anomalies, threats, or known attack signatures.
- Each rule typically specifies the **protocol**, **source** and **destination IP** addresses and **ports**, the **direction** of the traffic, and a set of **options** such as content matching, message tags, or classification types.
- So, we use the same Syntax as **Snort** Rules.

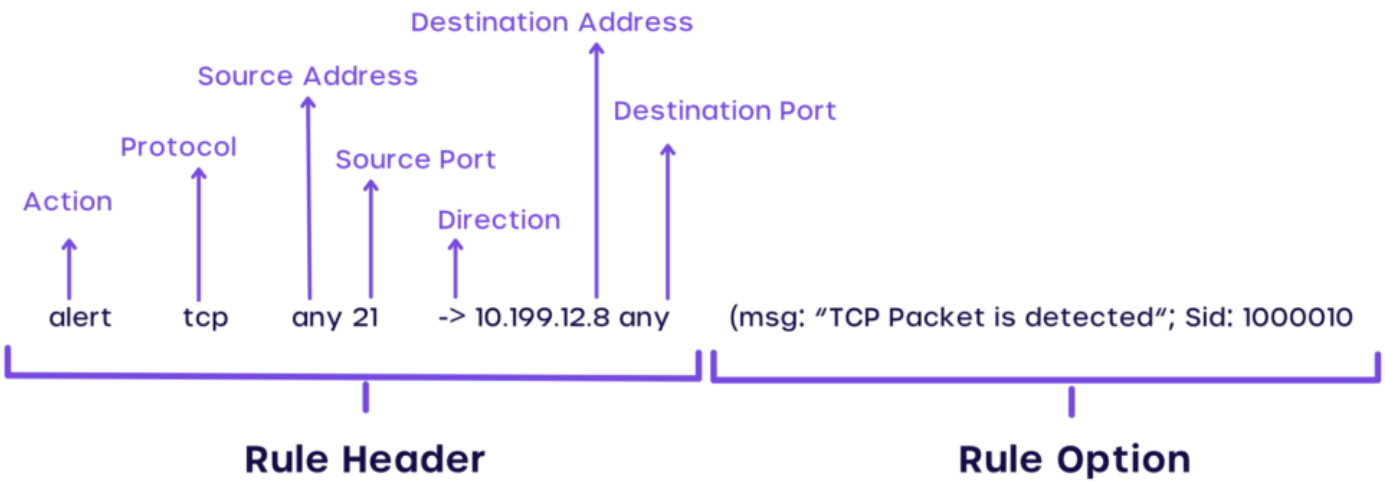

Figure 4.11

- A **Snort** rule is composed of two main components:
  - the **Rule Header** and the **Rule Options**
  - The header defines the basic structure **(such as protocol and IP addresses)**.
  - while the options provide detailed matching criteria and metadata.

- To ensure clarity and consistency, all rules were written and processed using a standardized format that is easy to read and maintain.

- The implementation followed a structured pipeline:
  - **Initial Rule Collection**: Each team member wrote Snort rules in a plain .txt file. Initially, the rules varied in formatting and structure.
  - **Rule Extraction**: A script was developed to read and extract individual rules from these text files.
  - **Normalization Process**: Another script was implemented to unify the format of all rules, standardizing their structure to match a consistent template.
  - **Database Insertion**: Once the rules were standardized, they were inserted into the database in a structured and searchable format.
  - This process ensured that all Snort rules are handled uniformly and can be programmatically retrieved, displayed, or updated through the application interface**.**

- Now this shows how it is implemented by code.

```python
def extract_rules(file_path):
        rules = []
        current_rule = ""

        try:
            with open(file_path, "r") as f:
                for line in f:
                    line = line.strip()
                    if line.startswith("#") or not line:
                        continue
                    if "(" in line and ")" not in line:
                        current_rule = line
                    elif current_rule:
                        current_rule += " " + line
                        if ")" in line:
                            rules.append(current_rule)
                            current_rule = ""
                    else:
                        rules.append(line)
        except FileNotFoundError:
            print(f"Error: File not found at {file_path}")
        except Exception as e:
            print(f"Error while reading rules: {e}")

        return rules
```

Figure 4.12

```python
def parse_rule(rule):
    parts = rule.split('(')
    header = parts[0].strip()
    options = parts[1].strip(')').split(';')

    header_parts = header.split()

    action = header_parts[0]
    proto = header_parts[1]
    src_ip = header_parts[2]
    src_port = header_parts[3]
    direction = header_parts[4]
    dst_ip = header_parts[5]
    dst_port = header_parts[6]

    options_dict = {}

    for option in options:

        if ':' in option:
            key, value = option.split(':', 1)
            key = key.strip()
            value = value.strip().strip('"')

            if key == 'dsize' or key == 'threshold' or key == 'itype':
                options_dict[key] = int(value)

            else:
                options_dict[key] = value

    return {
        "action": action,
        "protocol": proto,
        "src_ip": src_ip,
        "src_port": src_port,
        "dst_ip": dst_ip,
        "dst_port": dst_port,
        "direction": direction,
        "options": options_dict
    }
```

Figure 4.13

```python
def load_rules_to_db(file_path, conn):
    cursor = conn.cursor()
    rules = extract_rules(file_path)
    for rule in rules:
        parsed_rule = parse_rule(rule)
        cursor.execute(
            """
            INSERT INTO rules
            (
                action,
                protocol,
                src_ip,
                src_port,
                direction,
                dst_ip,
                dst_port,
                options
            )
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
            """,
            (
                parsed_rule['action'],
                parsed_rule['protocol'],
                parsed_rule['src_ip'],
                parsed_rule['src_port'],
                parsed_rule['direction'],
                parsed_rule['dst_ip'],
                parsed_rule['dst_port'],
                json.dumps(parsed_rule['options'])
            )
        )
    conn.commit()
    print(f"Inserted {len(rules)} rules into the database.")
```

Figure 4.14

```python
def get_rules_from_db(conn):
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM rules")
    rows = cursor.fetchall()
    rules = []
    for row in rows:
        rules.append({
            "id": row[0],
            "action": row[1],
            "protocol": row[2],
            "src_ip": row[3],
            "src_port": row[4],
            "direction": row[5],
            "dst_ip": row[6],
            "dst_port": row[7],
            "options": json.loads(row[8])
        })
    return rules
```

Figure 4.15

4.2.4. Deep packet Inspection (DPI)

- **Deep Packet Inspection (DPI)** is an advanced technique used for analyzing and managing network traffic. Unlike traditional packet inspection methods that only examine packet headers, DPI inspects the entire packet, including the payload. This provides a more comprehensive understanding of the traffic, enabling improved security, monitoring, and network performance optimization.
- **Key Components of DPI:**
  - **Packet Capture:**
  - DPI tools begin by capturing network packets as they pass through a network interface.
  - **Header Inspection:**
    - The headers of packets (e.g., IP addresses, port numbers, and protocol types) are analyzed to determine fundamental information about the flow.
  - **Payload Inspection:**
    - The data portion (payload) of each packet is inspected to identify:
      - Application-layer protocols such as HTTP, FTP, and DNS.
      - Specific keywords or patterns, including malware signatures or sensitive information.
      - File types such as documents, images, or videos.
  - **Analysis and Action:**
    - Based on the inspection, **DPI** can:
      - Detect malicious activity (e.g., **malware**, **intrusions**).
      - Enforce network policies (e.g., **blocking certain applications**).
      - Optimize traffic (e.g., **prioritizing critical applications**).
- **DPI Implementation Using Scapy in Python:**
  - To implement the DPI mechanism, we utilized the **Scapy** library in Python — a powerful packet manipulation tool.
  - **Installation:**
    - **Scapy** can be installed using standard Python package managers.

```
pip install scapy
```

Figure 4.16

- By importing Scapy Library and importing from it
  - **TCP:** Enables inspection of the TCP layer and extraction of relevant fields.
  - **UDP:** Allows analysis of UDP layer attributes.
  - And the same for the remaining layers.
  - **Raw:** Grants access to the packet payload if present.
  - Additional protocol layers supported by **Scapy** can also be utilized as needed.

```
from scapy.all import TCP,UDP,ICMP,Ether,IP,ARP,Raw
```

Figure 4.17

- **Custom DPI Class:**
  - To extract all necessary packet details, we implemented a custom Packet class, which reads packets from a PCAP traffic file and:
    - Parses headers and payloads,
    - Extracts key features,
    - Prepares the packet for comparison against existing **signature-based rules & Anomaly-Based Detection**.

```
class Packet:
    src_ip_count = {}
    dst_ip_count = {}


    def __init__(self, scapy_packet):

        self.scapy_packet = scapy_packet
        self.protocol = self.extract_protocol()
        self.src_ip = self.get_src_ip()
        self.dst_ip = self.get_dst_ip()
        self.src_port = self.get_src_port()
        self.dst_port = self.get_dst_port()
        self.flags = self.extract_flags()
        self.data_size = self.get_pkt_size()
        self.payload = self.extract_payload()
        self.time = self.get_packet_time()
        self.time_formatted = self.get_packet_time_formatted()
        self.src_ip_count = self.update_src_ip_count()
        self.dst_ip_count = self.update_dst_ip_count()
```

Figure 4.18

- **Purpose and Responsibilities:**
  - To parse, inspect, and extract relevant metadata and payload content from each network packet.
  - To convert raw Scapy packets into structured feature sets that can be analyzed and compared with signature rules.
  - To maintain statistics on packet source and destination IP occurrences.
  - To facilitate the classification and detection of suspicious behavior based on header fields, payload, and protocol-specific flags.

- **Key Features and Methods:**
  - **Initialization and Feature Extraction:**
    - Upon instantiation, the class automatically extracts and stores the following features:
      - Protocol Type **(TCP, UDP, ICMP, ARP, etc.).**
      - Source and Destination IP Addresses.
      - Source and Destination Ports.
      - Packet Flags **(e.g., TCP SYN, ACK, FIN).**
      - Packet Size.
      - Raw Payload Content.
      - Timestamp (Raw and Formatted).

  - **Traffic Statistics Tracking:**
    - Maintains global counters for source and destination IPs across all parsed packets using **src_ip_count** and **dst_ip_count**.
    - These statistics are useful in identifying potential scanning or flooding behavior.

  - **Payload and Header Analysis:**
    - Supports extraction of the **TCP/UDP** payload using the Raw layer.
    - Inspects **TCP/IP** flags to detect behavior patterns **(e.g., SYN flood)**.
    - Provides detailed header metrics such as header length and TCP window size.

  - **Utility Methods:**
    - **get_src_counts() / get_dst_counts()**: Returns cumulative source/destination IP frequencies.
    - get_key(key): Retrieves a specific feature by name.

- **__str__()**: String representation of packet attributes in dictionary format.

- **Layer-Specific Handling:**
  - Handles both Layer 3 **(IP, ARP) and Layer 4 (TCP, UDP, ICMP)** protocols.
  - Supports payload decoding, with fallback to hexadecimal if decoding fails.

- **Error Handling and Flexibility:**
  - Handles corrupted or non-decodable payloads gracefully.
  - Uses dynamic detection to ensure compatibility with various packet structures.

- For each Packet comes from the **PCAP**:
  - Get the Protocol of the **Packet**:

```python
def extract_protocol(self):

        pkt = self.scapy_packet

        if pkt.haslayer(TCP):
            return "tcp"

        elif pkt.haslayer(UDP):
            return "udp"

        elif pkt.haslayer(ARP):
            return "arp"

        elif pkt.haslayer(ICMP):
            return "icmp"

        elif pkt.haslayer(IP):
            return "ip"

        else:
            return f"N/A"
```

Figure 4.19

- Get the Source IP Address of the **Packet**:

```
def get_src_ip(self):

        pkt = self.scapy_packet

        if pkt.haslayer(IP):#network
            return pkt[IP].src#srcip

        elif pkt.haslayer(ARP):#data
            return pkt[ARP].psrc#srcip

        else:
            return "N/A"
```

Figure 4.20

- Get the Destination IP Address of the **Packet**:

```
def get_dst_ip(self):

        pkt = self.scapy_packet

        if pkt.haslayer(IP):
            return pkt[IP].dst

        elif pkt.haslayer(ARP):
            return pkt[ARP].pdst

        else:
            return "N/A"
```

Figure 4.21

- Get the Source Port Number of the **Packet**:

```
def get_src_port(self):

        pkt = self.scapy_packet

        if pkt.haslayer(TCP):
            return str(pkt[TCP].sport)

        elif pkt.haslayer(UDP):
            return str(pkt[UDP].sport)

        else:
            return "N/A"
```

Figure 4.22

- Get the Destination Port Number of the **Packet**:

```
def get_dst_port(self):

        pkt = self.scapy_packet

        if pkt.haslayer(TCP):
            return str(pkt[TCP].dport)

        elif pkt.haslayer(UDP):
            return str(pkt[UDP].dport)

        else:
            return "N/A"
```

Figure 4.23

- Get the **Flags** of the **Packet**:

```python
def extract_flags(self):
        pkt = self.scapy_packet

        if pkt.haslayer(TCP):
            tcp_flags = pkt[TCP].flags

            flags = []

            if tcp_flags & 0x02:  # SYN
                flags.append("S")

            if tcp_flags & 0x10:  # ACK
                flags.append("A")

            if tcp_flags & 0x01:  # FIN
                flags.append("F")

            if tcp_flags & 0x08:  # PSH
                flags.append("P")

            if tcp_flags & 0x04:  # RST
                flags.append("R")

            if tcp_flags & 0x20:  # URG
                flags.append("U")

            if tcp_flags & 0x40:  # ECE
                flags.append("E")

            if tcp_flags & 0x80:  # CWR
                flags.append("C")

            return ''.join(flags)

        elif pkt.haslayer(IP):
            ip_flags = pkt[IP].flags
            flags = []

            if ip_flags & 0x02:  # DF (Don't Fragment)
                flags.append("DF")

            if ip_flags & 0x01:  # MF (More Fragments)
                flags.append("MF")

            return ', '.join(flags) if flags else "No Flags"

        else:
            return "N/A"
```

Figure 4.24

- Get the **Size** of the **Packet**:

```
def get_pkt_size(self):
        pkt = self.scapy_packet
        return len(pkt)
```

Figure 4.25

- Get the **Time** of the **Packet**:

```
def get_packet_time_formatted(self):
        pkt = self.scapy_packet
        packet_time = float(pkt.time)

        formatted_time = datetime.fromtimestamp(packet_time).strftime("%Y-%m-%d %H:%M:%S")

        return formatted_time

    def get_packet_time(self):
        return float(self.scapy_packet.time)
```

Figure 4.26

- Get the **Payload** of the **Packet**:

```
def get_payload_length(self):
        # Assuming you want the length of the TCP/UDP payload, not including headers
        if self.scapy_packet.haslayer('TCP'):
            tcp_layer = self.scapy_packet['TCP']
            return len(tcp_layer.payload)
        elif self.scapy_packet.haslayer('UDP'):
            udp_layer = self.scapy_packet['UDP']
            return len(udp_layer.payload)
        else:
            # fallback: maybe no TCP/UDP layer, return 0 or total length minus headers
            return 0

    def extract_payload(self):
        pkt = self.scapy_packet

        if pkt.haslayer(Raw):

            try:
                payload = pkt[Raw].load.decode(errors="ignore")

                return payload

            except Exception as e:
                return pkt[Raw].load.hex()

        else:
            return "
```

Figure 4.27

- To store for each IP Address the number of packets it sends:
- To store for each IP Address the number of packets it sends:

```
@staticmethod
    def get_src_counts():
        return Packet.src_ip_count

    @staticmethod
    def get_dst_counts():
        return Packet.dst_ip_count
```

Figure 4.28

4.2.5. Signature-Based IDS

4.2.5.1. Matching Packets with Rules

- In our Signature-Based IDS, we rely on a set of predefined Snort-like rules to detect known patterns of malicious behavior in the network traffic. These rules are written following the standard Snort syntax, which consists of two main components:
  - **Rule Header**: Defines the general structure of the rule, including the action, protocol, source and destination IP addresses, and ports.
  - **Rule Options**: Provides detailed matching criteria such as message descriptions, content matching, pattern matching (PCRE), thresholds, and more.

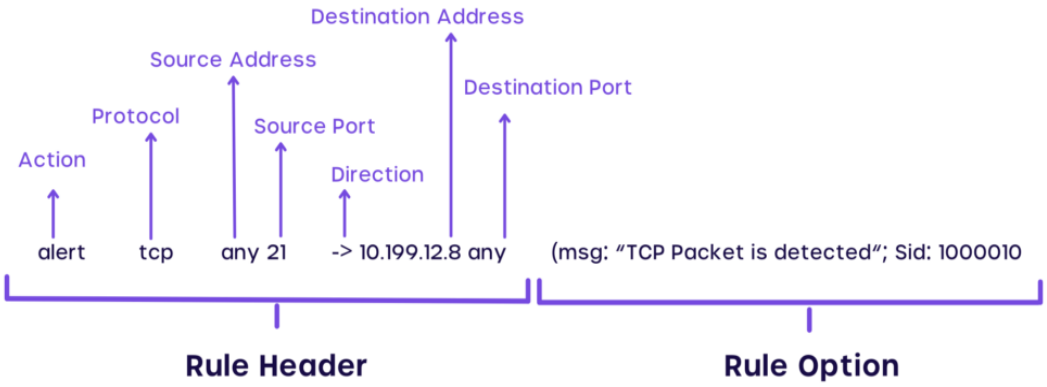

Figure 4.29

- Once these rules are written and stored in the database (as described in Section 4.2.3), the detection process is carried out as follows:

I. Rule Extraction:

- All rules are retrieved from the database using a dedicated database handler.
- For each rule retrieved, an object of the Rule class is created.
- This object encapsulates all components of the rule, including the header and the associated options.

```
class Rule:
    def __init__(self, rule_dict):
        self.action = rule_dict["action"]
        self.protocol = rule_dict["protocol"]
        self.src_ip = rule_dict["src_ip"]
        self.dst_ip = rule_dict["dst_ip"]
        self.src_port = rule_dict["src_port"]
        self.dst_port = rule_dict["dst_port"]
        self.options = rule_dict.get("options", {})
```

Figure 4.30

II. Packet Matching Process:

- Each incoming network packet is parsed into an object of the Packet class, which contains all relevant information extracted from the packet using Deep Packet Inspection (DPI). The matching process proceeds in two phases:

a. Header Matching

- The following header fields from the packet and the rule are compared:
  - Protocol.
  - Source IP and Port.
  - Destination IP and Port.
  - Direction (e.g., → or ↔).
- A match is considered successful only if all corresponding fields between the packet and rule header align correctly.

```python
def matches(self, packet):##header
        if self.protocol != "any" and packet.get_key("protocol") != self.protocol:
            return False
        if self.src_ip != "any" and packet.get_key("src_ip") != self.src_ip:
            return False
        if self.dst_ip != "any" and packet.get_key("dst_ip") != self.dst_ip:
            return False
        if self.src_port != "any" and packet.get_key("src_port") != self.src_port:
            return False
        if self.dst_port != "any" and packet.get_key("dst_port") != self.dst_port:
            return False
        return True
```

Figure 4.31

b. Option Matching

- After the header is matched, the system proceeds to compare the rule's options with the contents of the packet.
- Key fields checked include:
  - **content**: Searches for specific byte patterns or strings in the packet payload.
  - **pcre**: Performs regular expression matching against the payload.
  - **itype**: Used for ICMP-specific matching.
  - **threshold**: Checks if the number of matching packets exceeds a predefined count within a time interval.
  - **detection_filter**: Applies rate-based conditions on packet detection.
- The system ensures that each of these fields, if present in the rule, must also be satisfied by the packet for the rule to be considered a match.

```python
def match_rule(self,packet):#Option
        if not self.matches(packet):
            return False
        if "content" in self.options:
            content = self.options["content"]
            payload = packet.payload
            if content.startswith("!"):
                if content[1:] in payload:
                    return False
            else:
                if content not in payload:
                    return False
```

Figure 4.32

```
if "pcre" in self.options:
            pcre = self.options["pcre"]
            pcre = pcre.strip("/")
            if pcre.startswith("(?i)"):
                pcre = pcre[4:]
                pattern = re.compile(pcre, re.IGNORECASE)

            else:
                pattern = re.compile(pcre)

            payload = packet.payload

            if not pattern.search(payload):
                return False


        if "flags" in self.options:

            flags = self.options["flags"]

            packet_flags = packet.flags

            if (flags != packet_flags):
                return False


        if "dsize" in self.options:

            dsize = self.options["dsize"]

            data_size = packet.data_size

            if data_size < dsize:
                return False
```

Figure 4.33

```
if "threshold" in self.options:

            threshold = self.options["threshold"]

            if self.options["detection_filter"] == "track_by_src":

                src_ip_counts = Packet.get_src_counts()
                ips_to_delete = []

                for src_ip, count in src_ip_counts.items():

                    if src_ip == packet.src_ip:

                        if count < threshold:
                            return False

                        else :
                            ips_to_delete.append(src_ip)

                for ip in ips_to_delete:

                    del src_ip_counts[ip]


            elif self.options["detection_filter"] == "track_by_dst":

                dst_ip_counts = Packet.get_dst_counts()
                ips_to_delete = []

                for dst_ip, count in dst_ip_counts.items():

                    if dst_ip == packet.dst_ip:

                        if count < threshold:
                            return False

                        else :
                            ips_to_delete.append(dst_ip)

                for ip in ips_to_delete:

                    del dst_ip_counts[ip]


        if "itype" in self.options:

            itype = Packet.get_itype(packet)

            itype_rule = self.options["itype"]

            if itype != itype_rule:
                return False


        return True
```

Figure 4.34

4.2.5.2. Signature Rule Matching & Concurrent Packet Processing

- To implement the signature-based intrusion detection mechanism effectively, the **SignatureIDS** class was developed. This class is responsible for processing packets, applying matching logic using the predefined rules, and triggering alerts or logging events when malicious behavior is detected.
- The **SignatureIDS** class supports both real-time detection and offline prediction (**from PCAP files)** using multithreading to improve performance. It integrates with the rule engine, packet parser, and database interface to provide a complete signature detection pipeline.

```
class SignatureIDS:
    def __init__(self, rules, db_path='DB/IDS.db', max_workers=5):
        self.rules = [
            Rule(rule) for rule in rules
        ]

        self.db_path = db_path
        self.max_workers = max_workers
```

Figure 4.35

- **Functionalities**
  - **process_packet(pkt, conn):** Processes a single packet and checks if it matches any rule.

```
def process_packet(self, pkt, conn):
        try:
            packet = Packet(pkt)
            for rule in self.rules:
                if rule.match_rule(packet):
                    time = packet.get_packet_time_formatted()
                    src_ip = packet.get_src_ip()
                    dst_ip = packet.get_dst_ip()
                    action = rule.action
                    message = rule.options.get("msg", "No message specified")
                    layer = rule.options.get("attack", "No attack type")
                    method = "signature"
                    log = Log(time,action,src_ip,dst_ip,message,layer,method)
                    alert = Alert(time,src_ip,dst_ip,message,layer,method)
                    log.add_to_log_table(conn)
                    alert.add_to_alert_table(conn)
        except Exception as e:
            print(f"[ERROR] Failed to process packet: {type(e).__name__} - {e}")
```

Figure 4.36

- **detect(packets):** Processes a list of Scapy packets using multithreading.

```
def detect(self, packets):
        DB_obj = Database(self.db_path)
        conn = DB_obj.connect()

        if not conn:
            print("[ERROR] Failed to establish database connection.")
            return

        with ThreadPoolExecutor(max_workers=self.max_workers) as executor:

            futures = [executor.submit(self.process_packet, pkt, conn) for pkt in packets]

            for future in futures:
                future.result()

        conn.close()
```

Figure 4.37

- **predict_from_pcap(packets):** Performs detection on packets without writing to the database.

```
def predict_from_pcap(self, packets):
        alerts = []
        def process_single_packet(pkt):
            local_alerts = []
            try:
                packet = Packet(pkt)
                for rule in self.rules:
                    if rule.match_rule(packet):
                        alert = {
                            "time": packet.get_packet_time_formatted(),
                            "src_ip": packet.get_src_ip(),
                            "dst_ip": packet.get_dst_ip(),
                            "message": rule.options.get("msg", "No message specified"),
                            "layer": rule.options.get("attack", "No attack type"),
                            "method": "signature"
                        }
                        local_alerts.append(alert)
            except Exception as e:
                print(f"[ERROR] Predict packet failed: {type(e).__name__} - {e}")
            return local_alerts
        with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
            results = executor.map(process_single_packet, packets)
            for res in results:
                alerts.extend(res)
        return alerts
```

Figure 4.38

4.2.6. Anomaly-Based IDS

4.2.6.1. Machine Learning Preparations

1. **Data Collection**

a. **CIC-IDS2017**

- The **CIC-IDS2017** dataset is one of the most comprehensive and realistic datasets available for intrusion detection system (**IDS**) research. It was developed by the **Canadian Institute for Cybersecurity** (**CIC**) and includes both benign traffic and contemporary attack scenarios, making it suitable for training and evaluating **anomaly-based IDS models.**
- **Diverse Attack Types:** Includes DoS, DDoS, Brute-force, Botnet, Port Scanning, Heartbleed, Web Attacks, and Infiltration.
- **Realistic Traffic:** Simulated in a controlled environment using real user behavior.
- **Labelled Data:** All flows are clearly labeled as either benign or as a specific attack class.
- **Rich Features:** More than 80 flow-based features per record, such as packet length, inter-arrival time, and header flags.
- **Multiple Days of Traffic:** Data is organized per day, where each day corresponds to a different attack scenario.
- High **feature dimensionality** (~80+), increasing complexity in preprocessing.
- **Class imbalance**, with some attack types underrepresented.
- Data volume can be large, requiring substantial computation.

b. **KDD'99(KDD Cup '99)**

- Groundbreaking dataset derived from DARPA 1998–99 data, originally used in the 1999 KDD Cup. It includes ~4.9 million records with 41 features and five attack classes plus normal.
- Extensive historical use – widely benchmarked in IDS research.
- Very large; suitable for testing scalability.
- Contains ~75% redundant records, leading to biased model training.
- Severe class imbalance; small classes (U2R) poorly represented.
- Noisy and outdated; it does not adequately reflect modern network behavior or attacks.
- Non-stationarity: training and test distributions differ, undermining generalization.

c. **NSL-KDD**

- An improved version of the KDD Cup 1999 dataset, with duplicate records removed and more balanced classes. Contains 41 features covering basic, content, and traffic-related attributes. Classified into Normal, DoS, Probe, R2L, and U2R categories.
- Represents modern network protocols and attack types not available in NSL-KDD.
- Balanced statistical distributions between training and test sets.

- Rich multi-class labeling, enabling advanced classification tasks.
- Complex: many features (~42–49) require careful engineering.
- Minority class issues: some rare attack types (e.g. Worms, Shellcode) are underrepresented, causing detection difficulty.
- Requires substantial preprocessing (encoding, normalization).

Dataset	Features	Classes	Records (Train/Test)	Main Challenges
KDD'99	41	Normal + 4 attack types	~494k / 311k	Duplication, imbalance, outdated attack types
NSL-KDD	41	Same as KDD'99	~126k / 22k	Aging data; residual imbalance and noise
UNSW-NB15	42–49	Normal + 9 modern attacks	175k / 82k	Feature complexity; minority class sparsity
CIC-IDS2017	~80+	Real-world attacks	Varies (days-long capture)	High dimensionality; preprocessing overhead; imbalance

Table 4.1

Feature	CIC-IDS 2017	NSL-KDD	UNSW-NB15	KDD'99
Realistic Traffic	✅	❌	❌	❌
Modern Attack Types	✅	❌	❌	❌
Labelled Data	✅	✅	✅	✅

Suitable for Deep Learning	✅	❌	✅	❌
Imbalanced Classes	⚠️	⚠️	✅ Balanced	❌ Highly imbalanced
Feature Richness	✅ 80+ flow features	❌ Limited	✅ Medium (~49)	❌ Very limited
Year Released	2017	2009	2015	1999

Table 4.2

- In this project, we selected the **CIC-IDS2017** dataset as the primary source for building and evaluating our Intrusion Detection System (IDS). This decision was based on several key factors:

  1. **Realistic and Modern Traffic**:

     - The CIC-IDS2017 dataset was generated in a controlled environment to closely mimic real-world network behavior. It includes up-to-date attack types such as **Brute Force, DDoS, Port Scanning, Botnets**, and more, making it highly relevant to today's security challenges.

  2. **Comprehensive Feature Set**:

     - It contains over **80 flow-based features** extracted using **CICFlowMeter**, which offers a detailed view of both benign and malicious traffic. This rich set of attributes supports both traditional machine learning and deep learning approaches.

  3. **Labeled and Structured Format**:

     - Each traffic flow in the dataset is **clearly labeled**, allowing for supervised learning models to be trained and evaluated effectively. The labels cover multiple attack categories, enabling binary and multi-class classification.

4. **Payload-Level Inspection**:

   - Unlike older datasets, **CIC-IDS2017** includes **payload data**, allowing for Deep Packet Inspection (DPI) and enabling signature-based detection techniques alongside statistical analysis.

5. **Balanced Representation of Days and Attacks**:

   - The dataset is organized by day, with specific attack scenarios scheduled, which allows us to **split data cleanly** into training and testing sets—improving the robustness of evaluation.

6. **Academic and Community Acceptance**:

   - **CIC-IDS2017** is widely cited and used in academic research, making it a **benchmark dataset** for comparing the performance of intrusion detection techniques.

```python
for dirname ,_, filenames in os.walk('../Datasets/'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
../Datasets/Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv
../Datasets/Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv
../Datasets/Friday-WorkingHours-Morning.pcap_ISCX.csv
../Datasets/Monday-WorkingHours.pcap_ISCX.csv
../Datasets/Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv
../Datasets/Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv
../Datasets/Tuesday-WorkingHours.pcap_ISCX.csv
../Datasets/Wednesday-workingHours.pcap_ISCX.csv
```

Figure 4.39

```python
df = pd.DataFrame()

for dirname ,_, filenames in os.walk('../Datasets/'):
    for filename in filenames:
        if filename.endswith('.csv'):
            df_temp = pd.read_csv(os.path.join(dirname, filename))
            print(f"Loaded {filename} with shape {df.shape}")
            df = pd.concat([df, df_temp])
            del df_temp
```

```
Loaded Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv with shape (0, 0)
Loaded Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv with shape (225745, 79)
Loaded Friday-WorkingHours-Morning.pcap_ISCX.csv with shape (512212, 79)
Loaded Monday-WorkingHours.pcap_ISCX.csv with shape (703245, 79)
Loaded Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv with shape (1233163, 79)
Loaded Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv with shape (1521765, 79)
Loaded Tuesday-WorkingHours.pcap_ISCX.csv with shape (1692131, 79)
Loaded Wednesday-workingHours.pcap_ISCX.csv with shape (2138040, 79)
```

Figure 4.40

2. Exploratory Data Analysis (EDA)

- After collecting and organizing the CIC-IDS2017 dataset, we conducted a thorough Exploratory Data Analysis (EDA) to gain deeper insights into the structure, quality, and characteristics of the data. This step was critical to understanding the distribution of attacks, detecting anomalies in feature values, identifying data quality issues (such as missing or infinite values), and guiding the preprocessing phase. The following outlines the major observations and processes during EDA:

  - Shape of the Dataset

```python
nRow,ncol = df.shape
print(f"Total rows: {nRow}, Total columns: {ncol}")
```

```
Total rows: 2830743, Total columns: 79
```

Figure 4.41

  - Preview the First Few Rows

```python
df.head()
```

Python

	Destination Port	Flow Duration	Total Fwd Packets	Total Backward Packets	Total Length of Fwd Packets	Total Length of Bwd Packets	Fwd Packet Length Max	Fwd Packet Length Min	Fwd Packet Length Mean	Fwd Packet Length Std	...	min_seg_size_forward	Active Mean	Active Std	A
0	54865	3	2	0	12	0	6	6	6.0	0.0	...	20	0.0	0.0	
1	55054	109	1	1	6	6	6	6	6.0	0.0	...	20	0.0	0.0	
2	55055	52	1	1	6	6	6	6	6.0	0.0	...	20	0.0	0.0	
3	46236	34	1	1	6	6	6	6	6.0	0.0	...	20	0.0	0.0	
4	54863	3	2	0	12	0	6	6	6.0	0.0	...	20	0.0	0.0	

Figure 4.42

  - Dataset Information Summary

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2830743 entries, 0 to 692702
Data columns (total 79 columns):
 #   Column                       Dtype
---  ------                       -----
 0    Destination Port            int64
 1    Flow Duration               int64
 2    Total Fwd Packets           int64
 3    Total Backward Packets      int64
 4   Total Length of Fwd Packets  int64
 5    Total Length of Bwd Packets int64
 6    Fwd Packet Length Max       int64
 7    Fwd Packet Length Min       int64
 8    Fwd Packet Length Mean      float64
 9    Fwd Packet Length Std       float64
 10  Bwd Packet Length Max        int64
 11   Bwd Packet Length Min       int64
 12   Bwd Packet Length Mean      float64
 13   Bwd Packet Length Std       float64
 14  Flow Bytes/s                 float64
 15   Flow Packets/s              float64
 16   Flow IAT Mean               float64
 17   Flow IAT Std                float64
 18   Flow IAT Max                int64
 19   Flow IAT Min                int64
...
 77   Idle Min                    int64
```

Figure 4.43

- Statistical Summary

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Destination Port	2830743.0	8.071483e+03	1.828363e+04	0.0	53.0	80.0	443.0	65535.0
Flow Duration	2830743.0	1.478566e+07	3.365374e+07	-13.0	155.0	31316.0	3204828.5	119999998.0
Total Fwd Packets	2830743.0	9.361160e+00	7.496728e+02	1.0	2.0	2.0	5.0	219759.0
Total Backward Packets	2830743.0	1.039377e+01	9.973883e+02	0.0	1.0	2.0	4.0	291922.0
Total Length of Fwd Packets	2830743.0	5.493024e+02	9.993589e+03	0.0	12.0	62.0	187.0	12900000.0
...	...	...	...	...	...	...	...	...
Active Min	2830743.0	5.829582e+04	5.770923e+05	0.0	0.0	0.0	0.0	110000000.0
Idle Mean	2830743.0	8.316037e+06	2.363008e+07	0.0	0.0	0.0	0.0	120000000.0
Idle Std	2830743.0	5.038439e+05	4.602984e+06	0.0	0.0	0.0	0.0	76900000.0
Idle Max	2830743.0	8.695752e+06	2.436689e+07	0.0	0.0	0.0	0.0	120000000.0
Idle Min	2830743.0	7.920031e+06	2.336342e+07	0.0	0.0	0.0	0.0	120000000.0

78 rows × 8 columns

Figure 4.44

- Column Names

```
df.columns
```

```
Index([' Destination Port', ' Flow Duration', ' Total Fwd Packets',
       ' Total Backward Packets', 'Total Length of Fwd Packets',
       ' Total Length of Bwd Packets', ' Fwd Packet Length Max',
       ' Fwd Packet Length Min', ' Fwd Packet Length Mean',
       ' Fwd Packet Length Std', 'Bwd Packet Length Max',
       ' Bwd Packet Length Min', ' Bwd Packet Length Mean',
       ' Bwd Packet Length Std', 'Flow Bytes/s', ' Flow Packets/s',
       ' Flow IAT Mean', ' Flow IAT Std', ' Flow IAT Max', ' Flow IAT Min',
       'Fwd IAT Total', ' Fwd IAT Mean', ' Fwd IAT Std', ' Fwd IAT Max',
       ' Fwd IAT Min', 'Bwd IAT Total', ' Bwd IAT Mean', ' Bwd IAT Std',
       ' Bwd IAT Max', ' Bwd IAT Min', 'Fwd PSH Flags', ' Bwd PSH Flags',
       ' Fwd URG Flags', ' Bwd URG Flags', ' Fwd Header Length',
       ' Bwd Header Length', 'Fwd Packets/s', ' Bwd Packets/s',
       ' Min Packet Length', ' Max Packet Length', ' Packet Length Mean',
       ' Packet Length Std', ' Packet Length Variance', 'FIN Flag Count',
       ' SYN Flag Count', ' RST Flag Count', ' PSH Flag Count',
       ' ACK Flag Count', ' URG Flag Count', ' CWE Flag Count',
       ' ECE Flag Count', ' Down/Up Ratio', ' Average Packet Size',
       ' Avg Fwd Segment Size', ' Avg Bwd Segment Size',
       ' Fwd Header Length.1', 'Fwd Avg Bytes/Bulk', ' Fwd Avg Packets/Bulk',
       ' Fwd Avg Bulk Rate', ' Bwd Avg Bytes/Bulk', ' Bwd Avg Packets/Bulk',
       'Bwd Avg Bulk Rate', 'Subflow Fwd Packets', ' Subflow Fwd Bytes',
       ' Subflow Bwd Packets', ' Subflow Bwd Bytes', 'Init_Win_bytes_forward',
       ' Init_Win_bytes_backward', ' act_data_pkt_fwd',
       ' min_seg_size_forward', 'Active Mean', ' Active Std', ' Active Max',
       ' Active Min', 'Idle Mean', ' Idle Std', ' Idle Max', ' Idle Min',
       ' Label'],
      dtype='object')
```

Figure 4.45

- Label Distribution

```
# Dataset conatains 15 labels.
print(df[' Label'].unique())
```

```
['BENIGN' 'DDoS' 'PortScan' 'Bot' 'Infiltration'
 'Web Attack ⊕ Brute Force' 'Web Attack ⊕ XSS'
 'Web Attack ⊕ Sql Injection' 'FTP-Patator' 'SSH-Patator' 'DoS slowloris'
 'DoS Slowhttptest' 'DoS Hulk' 'DoS GoldenEye' 'Heartbleed']
```

```
len(df[' Label'].unique())
```

```
15
```

Figure 4.46

3. Data Cleaning & Preprocessing

- To ensure the dataset was ready for machine learning model training and evaluation, we performed a thorough cleaning and preprocessing process. The following steps summarize the procedures taken:

- Remove Duplicate Entries

```
df = df.drop_duplicates()
df.reset_index(drop=True, inplace=True)
```

```
nRow,ncol = df.shape
print(f"Total rows: {nRow}, Total columns: {ncol}")
```

```
Total rows: 2522362, Total columns: 79
```

Figure 4.47

- Fix Column Names

```
col_names = [col.replace(' ', '') for col in df.columns]
df.columns = col_names
df.head()
```

	DestinationPort	FlowDuration	TotalFwdPackets	TotalBackwardPackets	TotalLengthofFwdPackets	TotalLengthofBwdPackets
0	54865	3	2	0	12	0
1	55054	109	1	1	6	6
2	55055	52	1	1	6	6
3	46236	34	1	1	6	6
4	54863	3	2	0	12	0

Figure 4.48

- Clean and Standardize Label Values

```
df["Label"].unique()
```

```
array(['BENIGN', 'DDoS', 'PortScan', 'Bot', 'Infiltration',
       'Web Attack 	Brute Force', 'Web Attack 	XSS',
       'Web Attack 	Sql Injection', 'FTP-Patator', 'SSH-Patator',
       'DoS slowloris', 'DoS Slowhttptest', 'DoS Hulk', 'DoS GoldenEye',
       'Heartbleed'], dtype=object)
```

```
label_names = df['Label'].unique()
```

- Clean Label Values (Remove Special Characters)

```
label_names = [re.sub("[^a-zA-Z ]+", "", l) for l in label_names]
label_names = [re.sub("[\s\s]", '_', l) for l in label_names]
label_names = [lab.replace("__", "_") for lab in label_names]
```

```
label_names, len(label_names)
```

```
(['BENIGN',
  'DDoS',
  'PortScan',
  'Bot',
  'Infiltration',
  'Web_Attack_Brute_Force',
  'Web_Attack_XSS',
  'Web_Attack_Sql_Injection',
  'FTPPatator',
  'SSHPatator',
  'DoS_slowloris',
  'DoS_Slowhttptest',
  'DoS_Hulk',
  'DoS_GoldenEye',
  'Heartbleed'],
 15)
```

Figure 4.49

- Label Distribution Check

```
df["Label"].value_counts()
```

```
Label
BENIGN                      2096484
DoS_Hulk                     172849
DDoS                         128016
PortScan                      90819
DoS_GoldenEye                 10286
FTPPatator                     5933
DoS_slowloris                  5385
DoS_Slowhttptest               5228
SSHPatator                     3219
Bot                            1953
Web_Attack_Brute_Force         1470
Web_Attack_XSS                  652
Infiltration                     36
Web_Attack_Sql_Injection         21
Heartbleed                       11
Name: count, dtype: int64
```

Figure 4.50

- NULL Value Detection and Removal

```
df.isnull().values.any()
```

```
np.True_
```

- Identify Columns With Null Values

```
# Checking which column/s contain NULL values.
[col for col in df if df[col].isnull().values.any()]
```

```
['FlowBytes/s']
```

```
# Checking how many NULL values it this column contains.
df['FlowBytes/s'].isnull().sum()
```

```
np.int64(353)
```

- Remove Rows With NULLs

```
# Removing rows that contain NULL values and checking if number of removed rows is equal to the number of null values.
before = df.shape
df.dropna(inplace=True)
after = df.shape
before[0] - after[0]
```

```
353
```

Figure 4.51

```
df.isnull().any().any()
```

```
np.False_
```

- Ensure All Columns Are Numeric

```
print(df.shape)
```

```
(2522009, 79)
```

Figure 4.52

- Ensure All Features are Numeric

```
labl = df['Label']
df = df.loc[:, df.columns != 'Label'].astype('float64')
```

- Check for Non-Finite Values (Inf or NaN)

```
# Checking if all values are finite.

np.all(np.isfinite(df))
```

```
np.False_
```

```
# Checking what column/s contain non-finite values.

nonfinite = [col for col in df if not np.all(np.isfinite(df[col]))]

nonfinite
```

```
['FlowBytes/s', 'FlowPackets/s']
```

```
# Checking how many non-finite values each column contains.

finite = np.isfinite(df['FlowBytes/s']).sum()

df.shape[0] - finite
```

```
np.int64(1211)
```

```
# We can see that now we have Nan values again.

np.any(np.isnan(df))
```

```
np.False_
```

Figure 4.53

```python
df.replace([np.inf, -np.inf], np.nan, inplace=True)
df.drop_duplicates(inplace=True)
df.dropna(inplace=True)
```

Python

```python
df.shape
```

Python

```
(2520798, 79)
```

```python
print(df.columns.tolist())
```

Python

```
['DestinationPort', 'FlowDuration', 'TotalFwdPackets', 'TotalBackwardPackets', 'TotalLengthofFwdPackets', 'TotalLengthofBwdPackets', 'FwdPacketLengthMax',
```

Figure 4.54

- Export Cleaned Dataset

- Save Final Preprocessed Dataset

```python
df.to_csv("../Datasets/CIC_IDS2017_full_week.csv", index=False)
```

Figure 4.55

4. EDA: Feature Distribution & Class Balance

-To gain deeper insights into the structure of the CIC-IDS2017 dataset, we conducted **visual Exploratory Data Analysis (EDA)** by plotting the distribution of each feature and examining the distribution of class labels (normal vs. attack types).

- Feature-Wise Distribution Analysis
  - We plotted the distributions of all features with limited unique values (less than 50) to understand the spread of data. Both **histograms** and **bar plots** were used depending on the nature of the variable (numerical or categorical).

```python
def plot_per_column_distribution(df,n_graph_shown,n_graph_per_row):
    nunique_values = df.nunique()
    columns_to_plot = df.columns[(nunique_values > 1) & (nunique_values < 50)]
    df_to_plot = df[columns_to_plot]

    n_rows,n_cols = df_to_plot.shape
    column_names = list(df_to_plot.columns)

    n_graph_row = (n_cols + n_graph_per_row - 1) // n_graph_per_row
    plt.figure(num=None,figsize=(6*n_graph_per_row,8*n_graph_row),dpi=80,facecolor='w',edgecolor='k')

    for i, col_name in enumerate(columns_to_plot[:n_graph_shown]):
        plt.subplot(n_graph_row,n_graph_per_row,i+1)
        column_data = df_to_plot[col_name]

        if not np.issubdtype(type(column_data.iloc[0]),np.number):
            value_counts = column_data.value_counts()
            value_counts.plot.bar()
        else:
            column_data.plot.hist()

        plt.ylabel('Count')
        plt.xticks(rotation=90)
        plt.title(f"{col_name}(column {i})")

    plt.tight_layout(pad = 1.0, w_pad = 1.0, h_pad = 1.0)
    plt.show()
```

```python
plt.rcParams['figure.figsize'] = (30, 30)

plot_per_column_distribution(df, 79, 5)
```

Figure 4.56

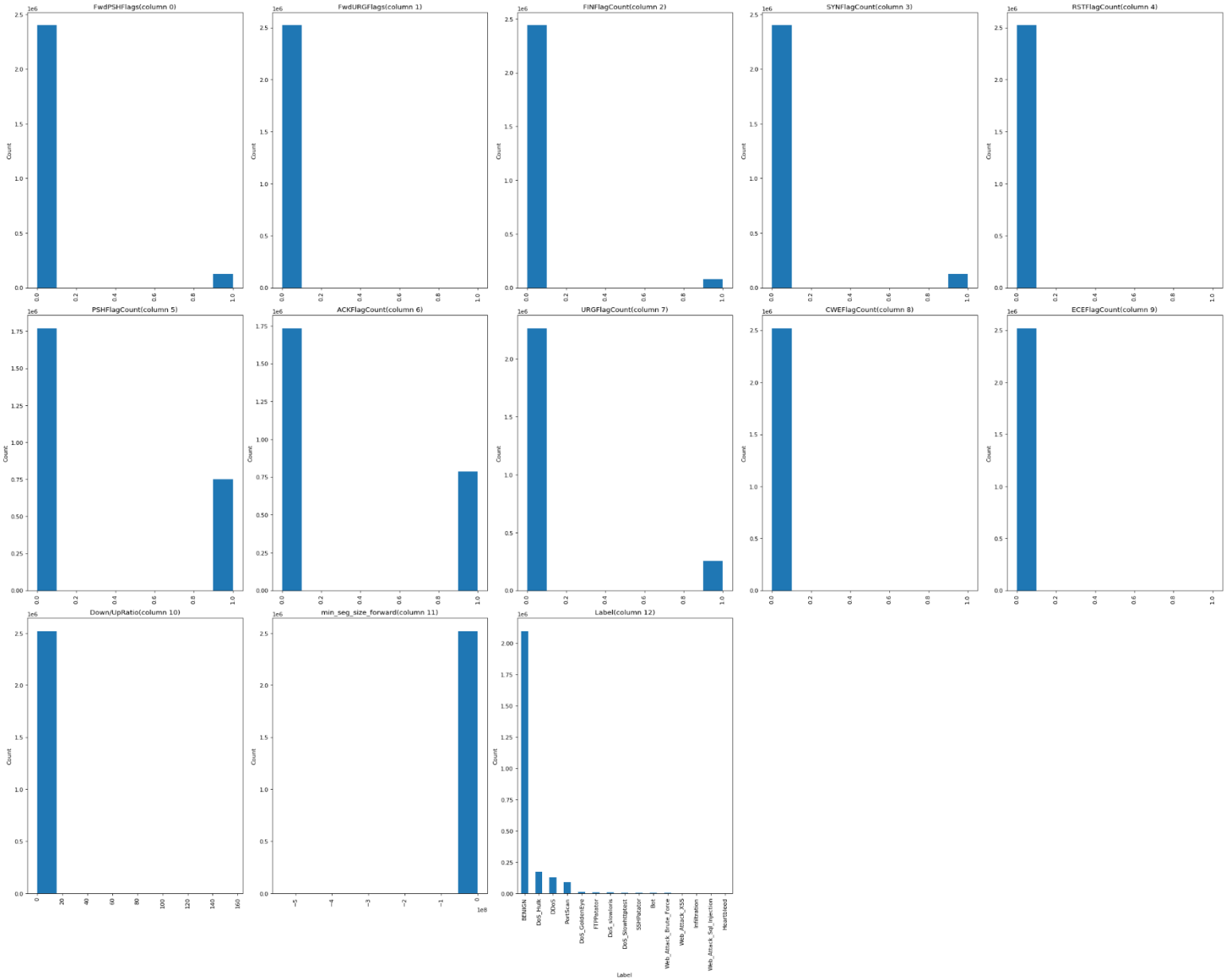

Figure 4.57

- Label Distribution Analysis
  - To assess the **class imbalance**, we visualized the distribution of the Label column using both **bar graphs** and **pie charts**.

```
def bar_graph(feature):
    df[feature].value_counts().plot(kind='bar')


plt.rcParams['figure.figsize'] = [15, 15]
bar_graph("Label")
```

Figure 4.58

- The dataset is **highly imbalanced**, with many samples belonging to the **BENIGN** class.
- Certain attack types such as **DoS Hulk, PortScan**, and **DDoS** had significantly higher frequencies compared to others like **Heartbleed** or **Web Attack – Brute Force**.
- This imbalance can affect model performance, so appropriate **resampling** or **weighting techniques** may be required during training

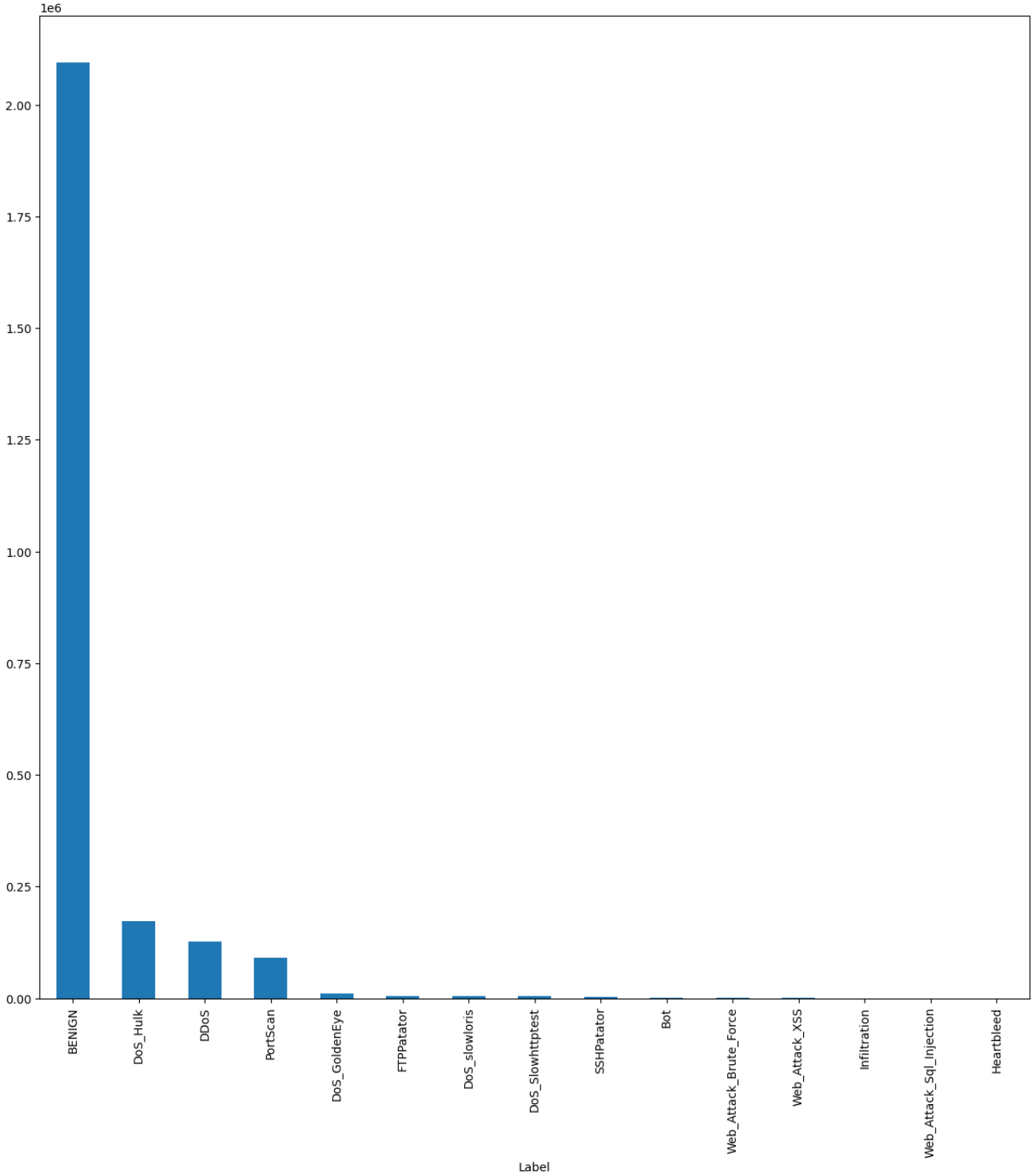

Figure 4.59

- Feature Distribution for Key Metrics
  - To analyze specific numerical features, such as **FlowDuration**, we used a distribution plot:

```
p = sns.distplot(a = df["FlowDuration"])
plt.show()
```

Figure 4.60

  - This helped detect extreme values and skewness in network flow durations which could be critical indicators of attack behaviors.

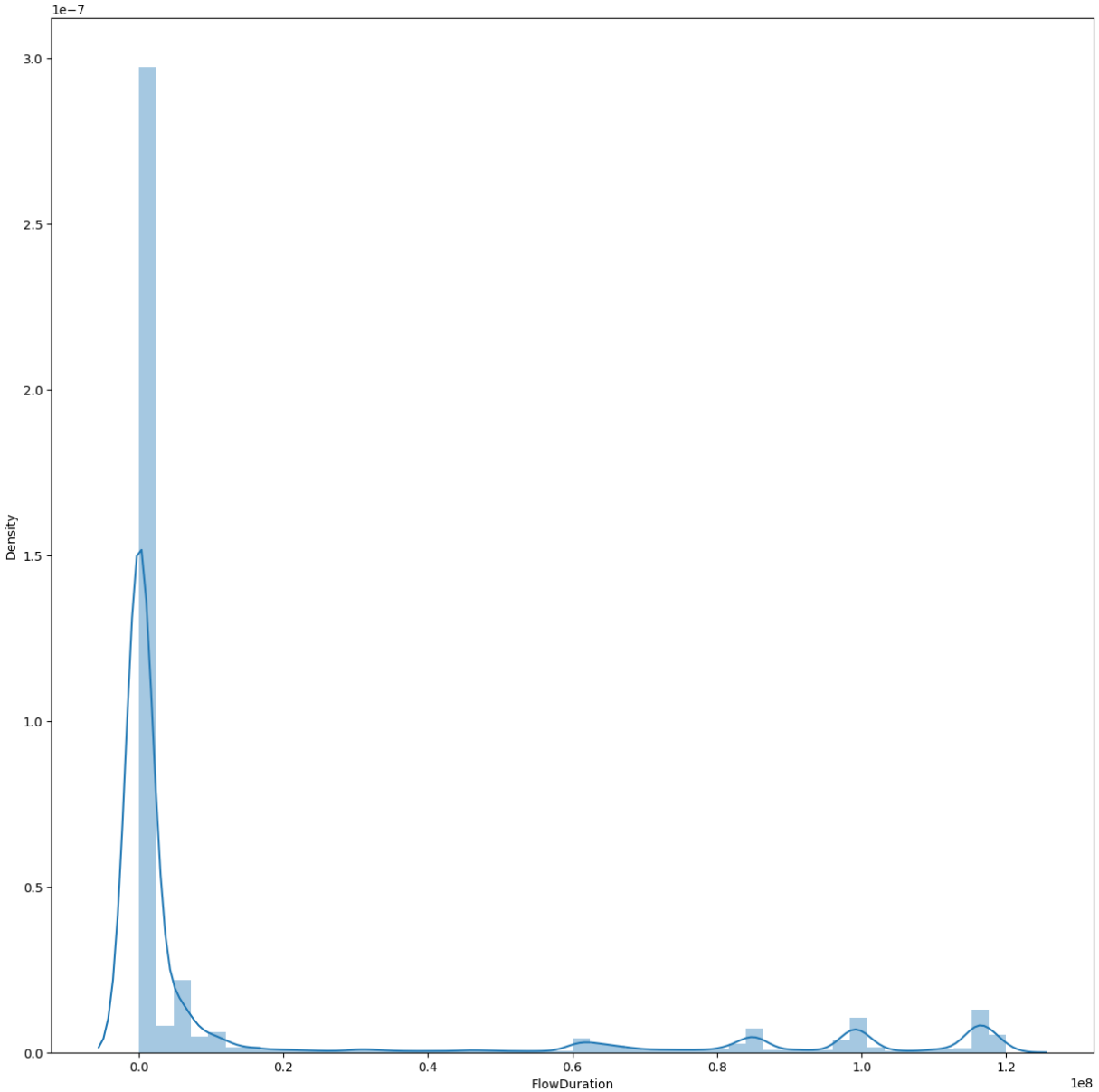

Figure 4.61

- Label Distribution Pie Chart
  - To understand the **class composition** of the CIC-IDS2017 dataset, we generated a **pie chart** showing the percentage distribution of each traffic type (normal and attack labels).

```
plt.figure(figsize=(20,20))
plt.pie(df['Label'].value_counts(), labels = df['Label'].unique(),autopct='%0.2f%%')
plt.title('Distribution of Labels')
plt.legend()
plt.show()
```

Figure 4.62

  - The dataset is **dominated by a few classes**, especially **BENIGN**, **DoS Hulk**, and **PortScan**.
  - Some attack types are **underrepresented** (e.g., **Heartbleed**, **Infiltration**, **Web Attack variants**).

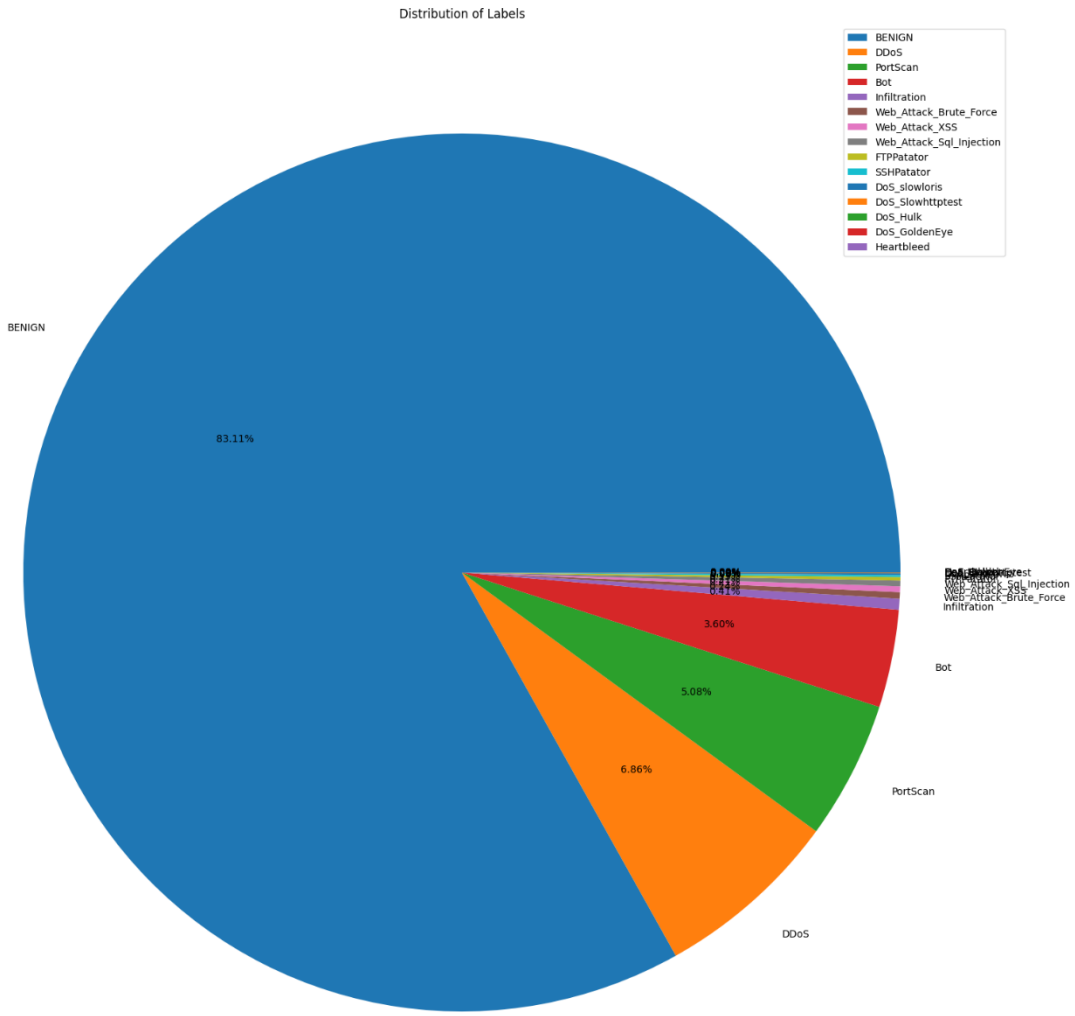

Figure 4.63

- This confirms the **imbalance issue** in the dataset, which may lead to **biased model predictions** if not addressed.

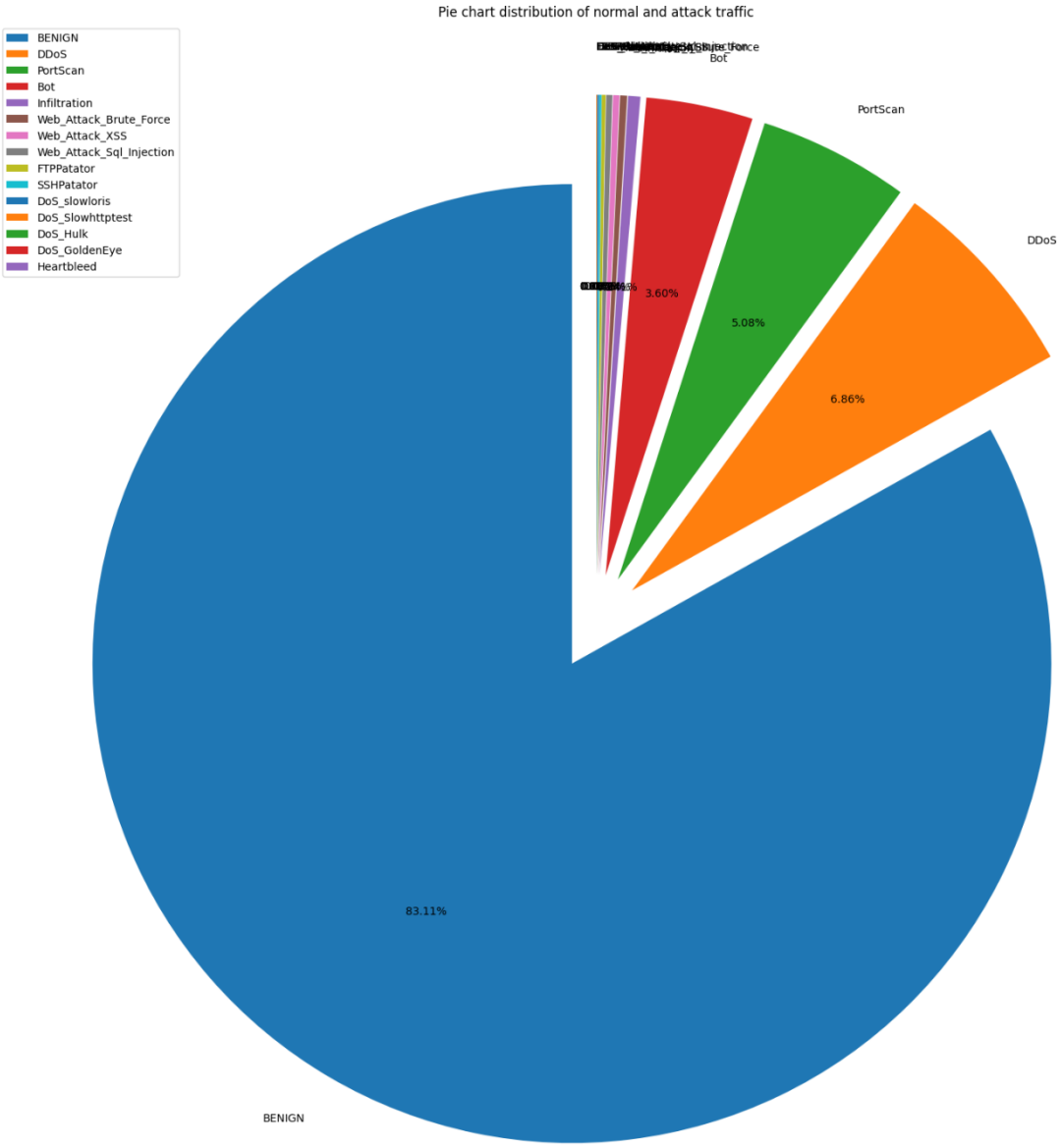

Figure 4.64

5. Balancing the Dataset

-After performing **Exploratory Data Analysis (EDA),** it became clear that the CIC-IDS2017 dataset suffers from **class imbalance**, where certain attack types (like **BENIGN, DoS Hulk, or PortScan)** dominate the dataset, while others (like Heartbleed or Infiltration) are severely underrepresented.

-To address the class imbalance issue in the CIC-IDS2017 dataset, a systematic approach was adopted to ensure fair representation of all attack types while preventing model bias toward dominant classes. The dataset originally contains highly imbalanced label distributions, with some attacks such as **BENIGN**, **DoS_Hulk**, **DDoS**, and **PortScan**

occurring in extremely large numbers compared to rare and critical attacks like **Heartbleed** and **Infiltration**.

-Why Balancing is Important

- Machine learning models trained on imbalanced data tend to be **biased toward majority classes**.
- Minority classes (often the most dangerous attacks) may be **misclassified** or completely ignored.
- Balancing improves model **generalization** and **fairness** across all classes.

- Dataset Before Balancing

```
df['Label'].value_counts()
```

```
Label
BENIGN                      2095057
DoS_Hulk                     172846
DDoS                         128014
PortScan                      90694
DoS_GoldenEye                 10286
FTPPatator                     5931
DoS_slowloris                  5385
DoS_Slowhttptest               5228
SSHPatator                     3219
Bot                            1948
Web_Attack_Brute_Force         1470
Web_Attack_XSS                  652
Infiltration                     36
Web_Attack_Sql_Injection         21
Heartbleed                       11
Name: count, dtype: int64
```

Figure 4.65

- Label Filtering and Limiting

- **Retained in full:** We kept all instances of the rare and underrepresented classes such as: **DoS_GoldenEye, FTPPatator, DoS_slowloris, DoS_Slowhttptest, SSHPatator, Bot, Web_Attack_Brute_Force, Web_Attack_XSS, Web_Attack_Sql_Injection, Infiltration, Heartbleed**
- **Downsampled (limited):** The most dominant labels were truncated to the top 50,000 samples each to prevent overrepresentation. These include: **BENIGN, DoS_Hulk, DDoS, PortScan.**
- A new balanced DataFrame (result_df) was then created by combining the full rare classes with the limited samples of dominant classes and shuffling them randomly.

```
classes_to_keep = ['DoS_GoldenEye','FTPPatator','DoS_slowloris','DoS_Slowhttptest',
                   'SSHPatator','Bot','Web_Attack_Brute_Force','Web_Attack_XSS','Web_Attack_Sql_Injection',
                   'Infiltration','Heartbleed'
                  ]

classes_to_limit = ['BENIGN','DoS_Hulk','DDoS','PortScan']

df_keep = df[df['Label'].isin(classes_to_keep)]

df_limit = df[df['Label'].isin(classes_to_limit)].groupby('Label').head(50000)

result_df = pd.concat([df_keep,df_limit])

result_df = result_df.sample(frac=1,random_state = 42).reset_index(drop=True)
```

Figure 4.66

```
result_df['Label'].value_counts()

Label
PortScan                    50000
DDoS                        50000
BENIGN                      50000
DoS_Hulk                    50000
DoS_GoldenEye               10286
FTPPatator                   5931
DoS_slowloris                5385
DoS_Slowhttptest             5228
SSHPatator                   3219
Bot                          1948
Web_Attack_Brute_Force       1470
Web_Attack_XSS                652
Infiltration                   36
Web_Attack_Sql_Injection       21
Heartbleed                     11
Name: count, dtype: int64
```

Figure 4.67

- Oversampling Rare Classes
  - Even after filtering, some attack types such as Heartbleed and Infiltration remained severely underrepresented. To mitigate this, we applied manual **oversampling** by replicating each of these rare classes 30 times. This increased their presence in the training data and allowed the model to learn better patterns from them.

```
df_rare = df_top_features[df_top_features['Label'].isin(['Infiltration', 'Heartbleed'])]
df_rare_oversampled = pd.concat([df_rare]*30).reset_index(drop=True)

df_top_features = pd.concat([df_top_features, df_rare_oversampled])
result_df = df_top_features.sample(frac=1, random_state=42).reset_index(drop=True)
```

Figure 4.68

- Attack Type Mapping and Normalization

  - To simplify classification and group semantically similar attacks, we introduced a mapping (**attack_map)** that consolidated specific attack types under general categories. For instance:

    - **DoS_Hulk, DoS_GoldenEye, DoS_slowloris → DoS**
    - **Web_Attack_XSS, Web_Attack_Brute_Force → Web_Attack**
    - **FTPPatator, SSHPatator → Brute_Force**

  - This mapping was applied using the. map **()** function to overwrite the Label column with normalized attack families.

```python
# Creating a dictionary that maps each label to its attack type
attack_map = {
    'BENIGN': 'BENIGN',
    'DDoS': 'DDoS',
    'DoS_Hulk': 'DoS',
    'DoS_GoldenEye': 'DoS',
    'DoS_slowloris': 'DoS',
    'DoS_Slowhttptest': 'DoS',
    'PortScan': 'Port_Scan',
    'FTPPatator': 'Brute_Force',
    'SSHPatator': 'Brute_Force',
    'Bot': 'Bot',
    'Web_Attack_Brute_Force': 'Web_Attack',
    'Web_Attack_XSS': 'Web_Attack',
    'Web_Attack_Sql_Injection': 'Web_Attack',
    'Infiltration': 'Infiltration',
    'Heartbleed': 'Heartbleed'
}

# Creating a new column 'Attack Type' in the DataFrame based on the attack_map dictionary
df_top_features['Label'] = df_top_features['Label'].map(attack_map)
```

Figure 4.69

- Visualizing the Balanced Label Distribution

  - A pie chart was plotted to visualize the class distribution post-balancing. This helped ensure that no class excessively dominates the dataset and that rare attacks are adequately represented for meaningful model training and evaluation.

- Final Dataset Preparation

  - The resulting dataset (**result_df**) was shuffled and stored with a clean distribution of attack types, suitable for training robust and fair classification models.

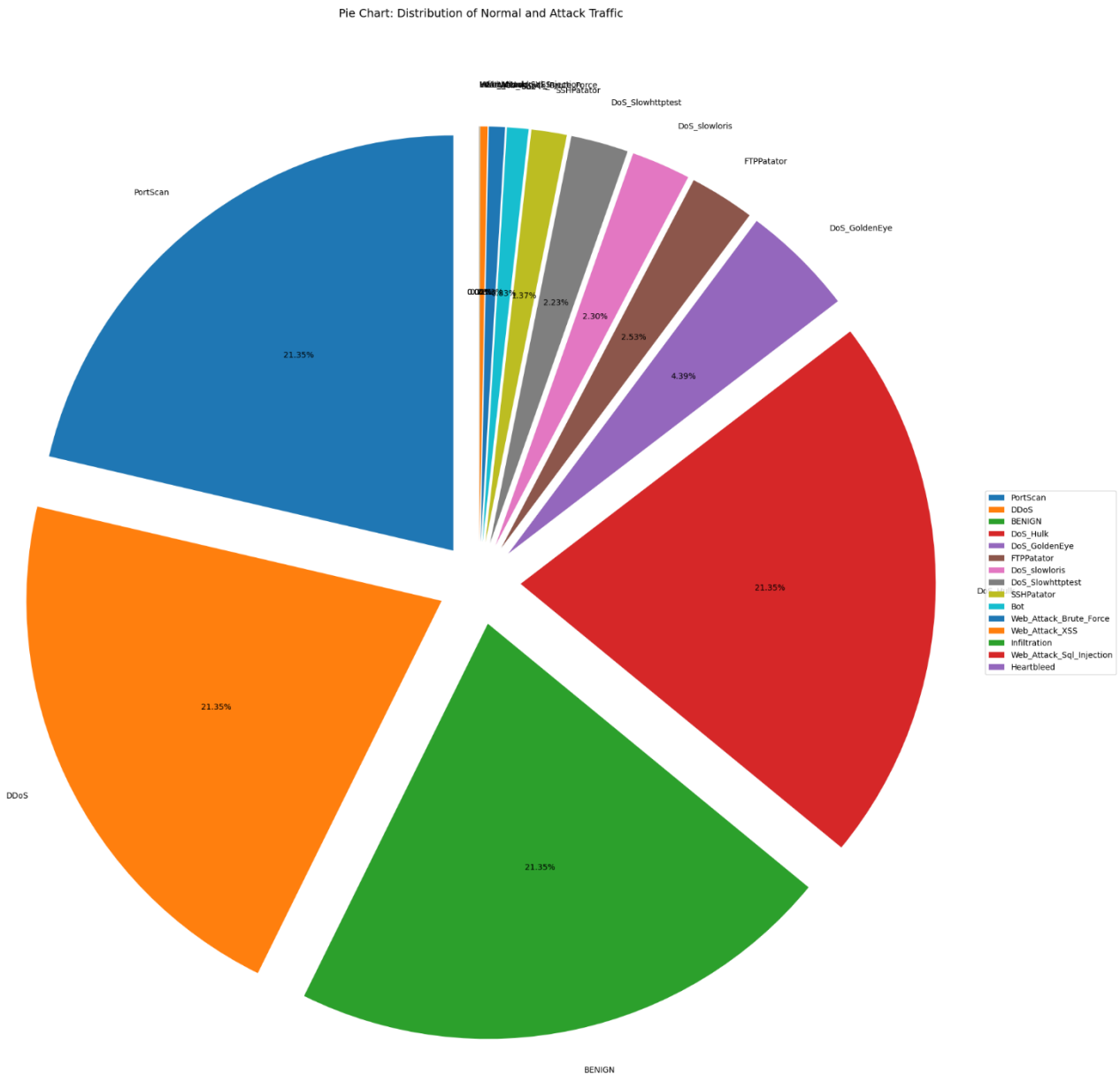

Figure 4.70

6. Label Encoding

- To make the target variable suitable for machine learning models, especially those that only accept numerical inputs, the categorical labels (attack types) were converted into numerical forms using **Label Encoding**.

- The **LabelEncoder** from **sklearn.preprocessing** was applied to the '**Label**' column.
- A dictionary was generated to store the mapping between the original attack names and their corresponding numerical codes.
- The encoder was saved using **joblib** for reuse during model inference to ensure consistent label mapping.
- **{'BENIGN': np.int64(0), 'Bot': np.int64(1), 'Brute_Force': np.int64(2), 'DDoS': np.int64(3), 'DoS': np.int64(4), 'Heartbleed': np.int64(5), 'Infiltration': np.int64(6), 'Port_Scan': np.int64(7), 'Web_Attack': np.int64(8)}.**

```
df_encoded = df_top_features.copy()

le = LabelEncoder()

df_encoded['Label'] = le.fit_transform(df_top_features['Label'])

label_mapping = dict(zip(le.classes_, le.transform(le.classes_)))

print(label_mapping)

print("\nEncoded DataFrame:")
df_encoded.head()
```

Figure 4.71

7. Correlation Analysis

- To gain deeper insights into the relationship between the features and the target variable (Label), a **correlation heatmap** was generated. This step aims to identify the most relevant features based on their statistical correlation with the class's output.

- A **Pearson correlation matrix** was calculated using the encoded dataset (**df_encoded**), where all features and the target label were numerical.

- A **correlation threshold** of 0.1 was applied to filter out features that have little to no linear relationship with the target variable.

- Only the features whose absolute correlation value with the label exceeded 0.1 were included in the heatmap.

- This analysis helps identify **highly correlated features** that are more likely to influence the classification outcome.

- Features that show **strong positive or negative correlation** with the label are valuable for training efficient and accurate models.

- This also aids in **reducing redundancy** and **selecting informative features**, which improves both performance and interpretability.

- The filtered correlation matrix was visualized using a **Seaborn heatmap** with a diverging color palette, making it easier to distinguish between positive and negative correlations.

- Each cell in the matrix shows the correlation value (ranging from -1 to +1), where:

- +1 indicates a strong positive correlation.

- -1 indicates a strong negative correlation.

- 0 indicates no correlation.

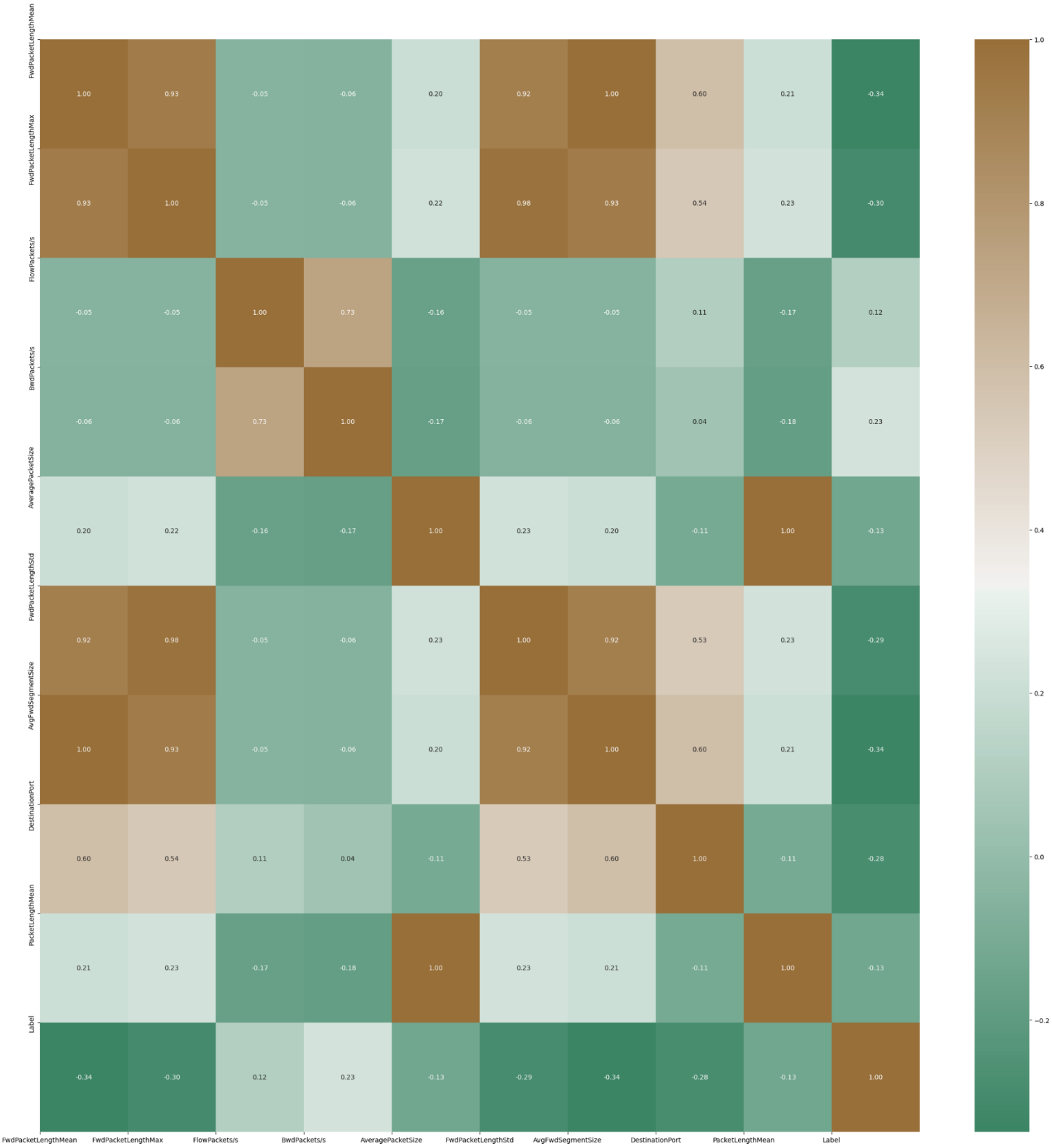
FwdPacketLengthMean
FwdPacketLengthMax
FlowPackets/s
BwdPackets/s
AveragePacketSize
FwdPacketLengthStd
AvgFwdSegmentSize
DestinationPort
PacketLengthMean
Label
1.00 0.93 -0.05 -0.06 0.20 0.92 1.00 0.60 0.21 -0.34
0.93 1.00 -0.05 -0.06 0.22 0.98 0.93 0.54 0.23 -0.30
-0.05 -0.05 1.00 0.73 -0.16 -0.05 -0.05 0.11 -0.17 0.12
-0.06 -0.06 0.73 1.00 -0.17 -0.06 -0.06 0.04 -0.18 0.23
0.20 0.22 -0.16 -0.17 1.00 0.23 0.20 -0.11 1.00 -0.13
0.92 0.98 -0.05 -0.06 0.23 1.00 0.92 0.53 0.23 -0.29
1.00 0.93 -0.05 -0.06 0.20 0.92 1.00 0.60 0.21 -0.34
0.60 0.54 0.11 0.04 -0.11 0.53 0.60 1.00 -0.11 -0.28
0.21 0.23 -0.17 -0.18 1.00 0.23 0.21 -0.11 1.00 -0.13
-0.34 -0.30 0.12 0.23 -0.13 -0.29 -0.34 -0.28 -0.13 1.00
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

Figure 4.72

8. Feature Selection using Random Forest

- After performing data preprocessing and balancing, it became essential to reduce the dimensionality of the dataset by selecting only the most relevant features for model training. Feature selection helps improve model performance, training speed, and generalization. For this purpose, the Random Forest Classifier was used to evaluate feature importance.

- Steps Followed:

- **Splitting the Dataset:**
  - The dataset was divided into input features X and target labels y.
  - A train-test split was applied using an 80/20 ratio to ensure model evaluation on unseen data.

```
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
✓ 0.1s
```

Figure 4.73

- **Training the Random Forest Model:**
  - A **Random Forest Classifier** was trained using 100 estimators (trees) with a fixed random seed.
  - The classifier's built-in. **feature importances_** attribute was used to compute the importance of each feature.

```
re_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
✓ 0.0s

re_classifier.fit(X_train, y_train)
✓ 53.4s

RandomForestClassifier
RandomForestClassifier(random_state=42)
```

Figure 4.74

- **Extracting and Ranking Feature Importances:**
  - Feature importance values were extracted and stored in a DataFrame.
  - The features were sorted in descending order based on their importance scores.

- **Selecting the Top 20 Features:**
  - The top 20 most important features were selected for further analysis and model training.

- A reduced dataset **df_top_features** was created using these selected features and the label column.

```
feature_importances = re_classifier.feature_importances_

feature_importances_df = pd.DataFrame({'Feature': x.columns, 'Importance': feature_importances})

feature_importances_df = feature_importances_df.sort_values(by='Importance', ascending=False)

top_features = feature_importances_df.head(20)['Feature'].tolist()

df_top_features = result_df[top_features + ['Label']]

df_top_features.head()
```

✓ 0.0s

Figure 4.75

- **Visualization:**

  - Bar charts and horizontal plots were used to visualize the importance of values clearly.

```
plt.figure(figsize=(10, 8))
sns.barplot(x='Importance', y='Feature', data=feature_importances_df.head(20), palette='viridis')
plt.title('Top 20 Important Features (Random Forest)', fontsize=14)
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

Figure 4.76

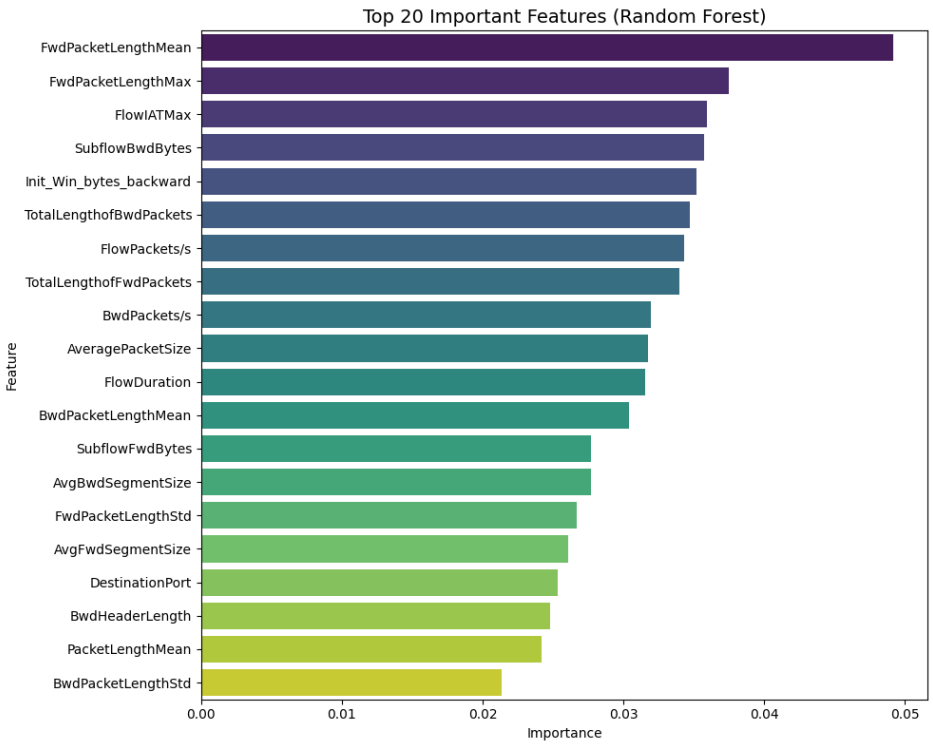

Figure 4.77

- This step was essential to ensure that the models only use the most relevant input attributes, thus reducing overfitting and improving detection accuracy. These selected features were later used in training both the anomaly-based and signature-based IDS components.

9. **Splitting the Data into Features and Labels**

- After completing the preprocessing, cleaning, balancing, and feature selection stages, we proceeded to split the dataset into features (**X**) and labels (**y**) in preparation for training and evaluation of the machine learning models.

- Steps Followed:

- **Separating Features and Labels:**
  - The target column (**Label**) was separated from the dataset.
  - **X** contains all the features (input variables).
  - **y** contains the corresponding labels (output classes).
- **Train-Test Split:**
  - The dataset was split into training and testing sets using an **80/20** ratio.
  - The **stratify=y** parameter ensures that the class distribution in both training and test sets remains balanced and consistent with the original dataset.
  - The **random_state=42** was set to ensure reproducibility of results.

```
X = dataset.drop('Label',axis=1)
y = dataset['Label']


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

Figure 4.78

10. **Feature Scaling**

- After splitting the dataset into training and testing sets, feature scaling was applied to normalize the feature values and improve the performance of the machine learning models.

- Why Feature Scaling?

- Many machine learning algorithms are sensitive to the range and scale of input features, especially those based on distance (e.g., **KNN, SVM, Neural Networks**). Features with larger magnitudes may dominate others, leading to biased model behavior. To address this, we used Min-Max Scaling, which transforms all feature values into a range between **0** and **1**.

- Steps Followed:

- **Min-Max Scaling:**
  - The **MinMaxScaler** from **sklearn.preprocessing** was used.
  - The scaler was fitted only on the training set to avoid data leakage.
  - The same scaler was then used to transform both training and testing sets.
- **Converted to DataFrame:**
  - The scaled arrays were converted back to DataFrame format to retain feature names and ease debugging or inspection.
- **Saving the Scaler:**
  - The fitted scaler object was saved using **joblib** to ensure consistent scaling during real-time inference or deployment.

```
# Fit only on training data
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Fit and transform on train
X_test_scaled = scaler.transform(X_test)        # Transform only on test


# Optional: convert back to DataFrame (use original column names if needed)
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_train.columns)


joblib.dump(scaler, 'Scaler/scaler_minmax.save')

['Scaler/scaler_minmax.save']
```

Figure 4.79

11. Exporting Data:

- The scaled and original features and labels were exported to .csv files for backup and testing purposes.

```
X_train_scaled.to_csv('test/features_scaled.csv', index=False)


X_train.to_csv('test/features.csv', index=False)


y_train.to_csv('test/labels.csv', index=False)
```

Figure 4.80

12. Validation:

- Checked for **NaN** or Inf values in the dataset to ensure data integrity before feeding into models.

```
print("Infinity values in X_train:",np.any(np.isinf(X_train)))
print("NaN values in X_train:",np.any(np.isnan(X_train)))

Infinity values in X_train: False
NaN values in X_train: False
```

Figure 4.81

13. Memory Management:

- Unused large variables were set to None to free memory space during the training pipeline.

```
X = None
y = None
X_train = None
dataset = None
df = None
df_encoded = None
df_keep = None
df_limit = None
df_temp = None
df_top_features = None
result_df = None
feature_importances_df = None
```

Figure 4.82

4.2.6.2. Model Development

1. Model Creating

- **Stacking Classifier**

  - To improve the robustness and performance of our Intrusion Detection System (IDS), we implemented an ensemble learning technique called Stacking, which combines multiple machine learning classifiers and uses a meta-model to produce the final prediction**.**
  - We use it to detect the Known (pre-Trained) labels only.
  - **Stacking Model Architecture:**
    - The stacking classifier consists of the following base learners:
      - **Decision Tree** – Criterion: Entropy, Max Depth: 4:
        - is a supervised learning model that splits the data based on feature values to form a tree-like structure of decisions. Each node splits the dataset using a condition that best separates the classes.

$$\text{Information Gain} = \text{Entropy(parent)} - \sum \frac{n_i}{n} \cdot \text{Entropy(child}_i)$$

- In the model, the **criterion is "entropy"**, which uses **Information Gain** to choose the best split:
- **Max Depth = 4** restricts the tree from growing too deep, which:
  - Prevents overfitting.
  - Forces the model to learn general patterns.
  - Makes it faster and interpretable.
- **Why it's useful in IDS:**
  - Good for capturing **simple decision rules**.
  - Handles both numerical and categorical features.
  - Fast to train and test on.

- **Logistic Regression:**
  - Logistic Regression is a **linear classifier** that models the **probability** of a class using the **sigmoid function**:

$$P(y = 1|x) = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

  - In It outputs a value between 0 and 1, which is **thresholded** to make class predictions.
  - **Max Depth = 4** restricts the tree from growing too deep, which:
    - Prevents overfitting.
    - Forces the model to learn general patterns.
    - Makes it faster and interpretable.
  - **Why it's useful in IDS:**
    - Simple and fast.
    - Good for linearly separable attack types.
    - Works well as a baseline and provides probability outputs, which is important in stacking.
- **K-Nearest Neighbors (KNN)** – Neighbors: 3
  - KNN is an **instance-based** (or lazy) learning algorithm.
  - For a new sample, it:
    - Finds the **k=3 closest points** in the training set.

        - Predicts the label by **majority vote**.
    - Distance metric: usually **Euclidean**.
    - **Why it's useful in IDS:**
        - Effective when **similar attacks cluster** together in feature space.
        - Captures **local patterns** missed by other classifiers.
    - Non-parametric: doesn't make assumptions about data distribution.
  - **CatBoost Classifier** (Boosting Algorithm)
    - **CatBoost** is a **gradient boosting** algorithm developed by Yandex. It builds an ensemble of **decision trees**, each correcting the errors of the previous ones.
    - For a new sample, it:
        - iterations=100: Number of trees.
        - learning_rate=0.1: Controls how much each tree contributes.
        - depth=6: Controls tree complexity.
    - **What makes CatBoost special:**
        - Efficient with **categorical features** (even if not present here).
        - Handles **imbalanced data** better.
        - Uses **ordered boosting** to reduce overfitting.
    - **Why it's useful in IDS:**
        - Strong overall accuracy.
        - Excellent at modeling **non-linear relationships** between features.
        - Boosting is often the **best standalone classifier** for structured/tabular data like network features.
- The **meta-model** (final estimator) used is a **Random Forest Classifier** with 100 estimators
    - Each of the 4 base models makes **individual predictions** on the input features.
    - These predictions (or class probabilities) are then **used as new features**.
    - The **meta-model (Random Forest)** learns from these to produce the final prediction.
    - **Why Random Forest is used as meta-model:**
        - Robust to overfitting.
        - Aggregates decisions from multiple trees.
        - Handles non-linear patterns in the **meta-feature space** (outputs of base learners).

- **Why Use Stacking in IDS:**
  - Each base learner contributes diverse perspectives:
  - Linear (LogReg), Local (KNN), Tree-based logic (DT), Ensemble power (CatBoost).
  - The final estimator learns how to trust each base model depending on the situation.
  - This leads to higher accuracy, robustness, and generalization, which are critical in detecting both known and unknown attacks.

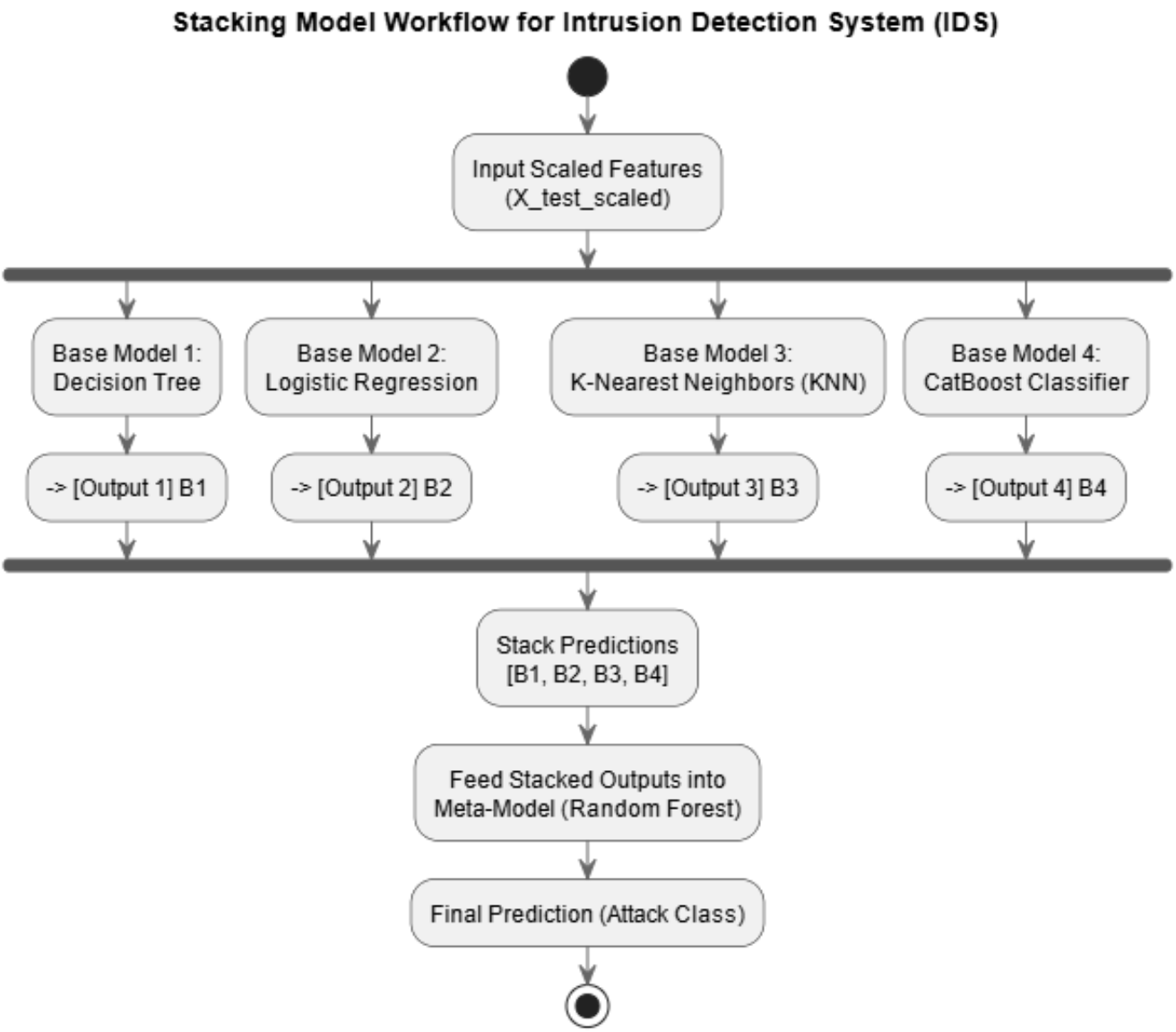

Figure 4.83

2. Model Training

- The model was trained on the **scaled training data (X_train_scaled, y_train)** using 5-fold cross-validation (cv=5) to ensure a balanced and generalizable model.Linear (LogReg), Local (KNN), Tree-based logic (DT), Ensemble power (CatBoost).

```
stacked_model.fit(X_train_scaled, y_train)
```

```
0:      learn: 1.5201101        total: 253ms    remaining: 25s
1:      learn: 1.2316514        total: 339ms    remaining: 16.6s
2:      learn: 1.0430819        total: 421ms    remaining: 13.6s
3:      learn: 0.8954925        total: 507ms    remaining: 12.2s
4:      learn: 0.7753531        total: 595ms    remaining: 11.3s
5:      learn: 0.6838872        total: 678ms    remaining: 10.6s
6:      learn: 0.6073777        total: 767ms    remaining: 10.2s
7:      learn: 0.5456176        total: 853ms    remaining: 9.81s
8:      learn: 0.4899374        total: 935ms    remaining: 9.45s
9:      learn: 0.4446307        total: 1.02s    remaining: 9.17s
10:     learn: 0.4037306        total: 1.1s     remaining: 8.89s
11:     learn: 0.3689204        total: 1.19s    remaining: 8.69s
12:     learn: 0.3346471        total: 1.28s    remaining: 8.56s
13:     learn: 0.3073022        total: 1.36s    remaining: 8.37s
14:     learn: 0.2812147        total: 1.45s    remaining: 8.21s
15:     learn: 0.2569282        total: 1.53s    remaining: 8.02s
16:     learn: 0.2357404        total: 1.61s    remaining: 7.86s
17:     learn: 0.2180863        total: 1.69s    remaining: 7.72s
18:     learn: 0.2014116        total: 1.77s    remaining: 7.57s
19:     learn: 0.1863322        total: 1.87s    remaining: 7.47s
20:     learn: 0.1730179        total: 1.94s    remaining: 7.31s
21:     learn: 0.1607258        total: 2.03s    remaining: 7.19s
22:     learn: 0.1499681        total: 2.1s     remaining: 7.04s
23:     learn: 0.1394975        total: 2.19s    remaining: 6.94s
24:     learn: 0.1307575        total: 2.27s    remaining: 6.83s
...
96:     learn: 0.0204839        total: 7.8s     remaining: 241ms
97:     learn: 0.0202241        total: 7.96s    remaining: 162ms
98:     learn: 0.0199578        total: 8.07s    remaining: 81.5ms
99:     learn: 0.0197265        total: 8.22s    remaining: 0us
```

Figure 4.84

3. Model Prediction

```
# Get predictions
y_pred = stacked_model.predict(X_test_scaled)
y_proba = stacked_model.predict_proba(X_test_scaled)
```

Figure 4.85

4. Model Evaluation Metrics:

- The trained model was evaluated on the **test set** (X_test_scaled, y_test) using the following performance metrics:

1. **Accuracy**

   - **Definition**: The ratio of correctly predicted samples to the total number of samples.
   - **Purpose**: Gives an overall sense of how often the model is correct.
   - **Limitation**: Not reliable alone in imbalanced datasets (e.g., where "BENIGN" is much more frequent than attacks).
   - **Formula**:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

2. **Precision (Weighted)**

   - **Definition**: How many of the predicted positive samples are correct, averaged across all classes, weighted by support (number of samples).
   - **Formula (for each class)**:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Weighted Average**:

$$\text{Weighted Precision} = \sum_{i=1}^{n} \frac{n_i}{N} \times P_i$$

  - n_i: number of samples in class i
  - N: total number of samples
  - P_i: precision of class i

- **Purpose**: Indicates how *precise* the model is when labeling something as a certain attack. Critical for minimizing **false alarms**.

3. **Recall (Weighted)**

   - **Definition**: Measures how well the model captures *all actual instances* of each class.
   - **Formula (for each class)**:

$$\text{Recall} = \frac{TP}{TP + FN}$$

   - **Weighted Average**:
     Like precision but using recall.
   - **Purpose**: Essential for **detecting all attack types**, especially important in intrusion detection to avoid missing real threats.

4. **F1 Score (Weighted)**

   - **Definition**: The harmonic mean of precision and recall, giving a balance between them.
   - **Formula**:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- **Weighted Average**: Averages F1 across all classes, weighted by class size.
- **Purpose**: Balanced view of performance, especially useful for **imbalanced datasets** where some classes are underrepresented.

5. **F1 Score (Macro)**

   - **Definition**: Average of F1 scores for all classes, **without considering class imbalance**.
   - **Formula**:

$$\text{Macro F1} = \frac{1}{n}\sum_{i=1}^{n} F1_i$$

   - **Purpose**: Evaluates the model's ability to treat **all classes equally**, regardless of their frequency. Useful when rare attack types of matter as much as frequent ones.

6. **ROC AUC Score (OvR - One vs Rest)**

   - **Definition**: Measures the model's ability to distinguish between classes.
   - **How it works**:
     - For each class, treat it as "positive" and the rest as "negative" (One-vs-Rest).
     - Plot the ROC curve and calculate the **Area Under Curve (AUC)**.
     - Average the AUC values (weighted or macro).
   - **Purpose**: Gives insight into the **probabilistic separability** of classes. Higher AUC indicates better discrimination between attack types and benign traffic.

```python
# Metrics for multi-class
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
f1_macro =f1_score(y_test, y_pred, average='macro')
```

```python
# ROC AUC for multi-class (One-vs-Rest)
try:
    roc_auc = roc_auc_score(y_test, y_proba, multi_class='ovr', average='weighted')
except:
    roc_auc = "Not computable (possibly missing probability output or wrong label format)"
```

Figure 4.86

- **Results:**
  - The model achieved the following performance:

Metric	Score
Accuracy	0.9972410865874364*100 = 99.7241%
Precision (Weighted)	0.9972764667822465*100 = 99.7276%
Recall (Weighted)	0.9972410865874364*100 = 99.7241%
F1 Score (Weighted)	0.9972502096897812*100 = 99.7250%
F1 Score (Macro)	0.9929805115023242*100 = 99.2981%
ROC AUC Score	0.999956591477652*100 = 99.9957%

Table 4.3

  - These values indicate **exceptionally high performance** of the stacked IDS model across all key metrics — showing both strong general accuracy and good balance between precision and recall, even across multiple classes.
  - **Classification Report:**
    - A detailed classification report was generated showing per-class precision, recall, and F1-score.
  - **Confusion Matrix:**
    - A confusion matrix was plotted to visualize the performance across different classes. It helps identify which classes are more likely to be misclassified and provides insight into class-wise mode

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

           0       0.99      0.99      0.99     10000
           1       0.94      1.00      0.97       390
           2       1.00      1.00      1.00      1830
           3       1.00      1.00      1.00     10000
           4       1.00      1.00      1.00     14180
           5       1.00      1.00      1.00        68
           6       1.00      1.00      1.00       223
           7       1.00      1.00      1.00     10000
           8       0.98      0.99      0.98       429

    accuracy                           1.00     47120
   macro avg       0.99      1.00      0.99     47120
weighted avg       1.00      1.00      1.00     47120
```

Figure 4.87

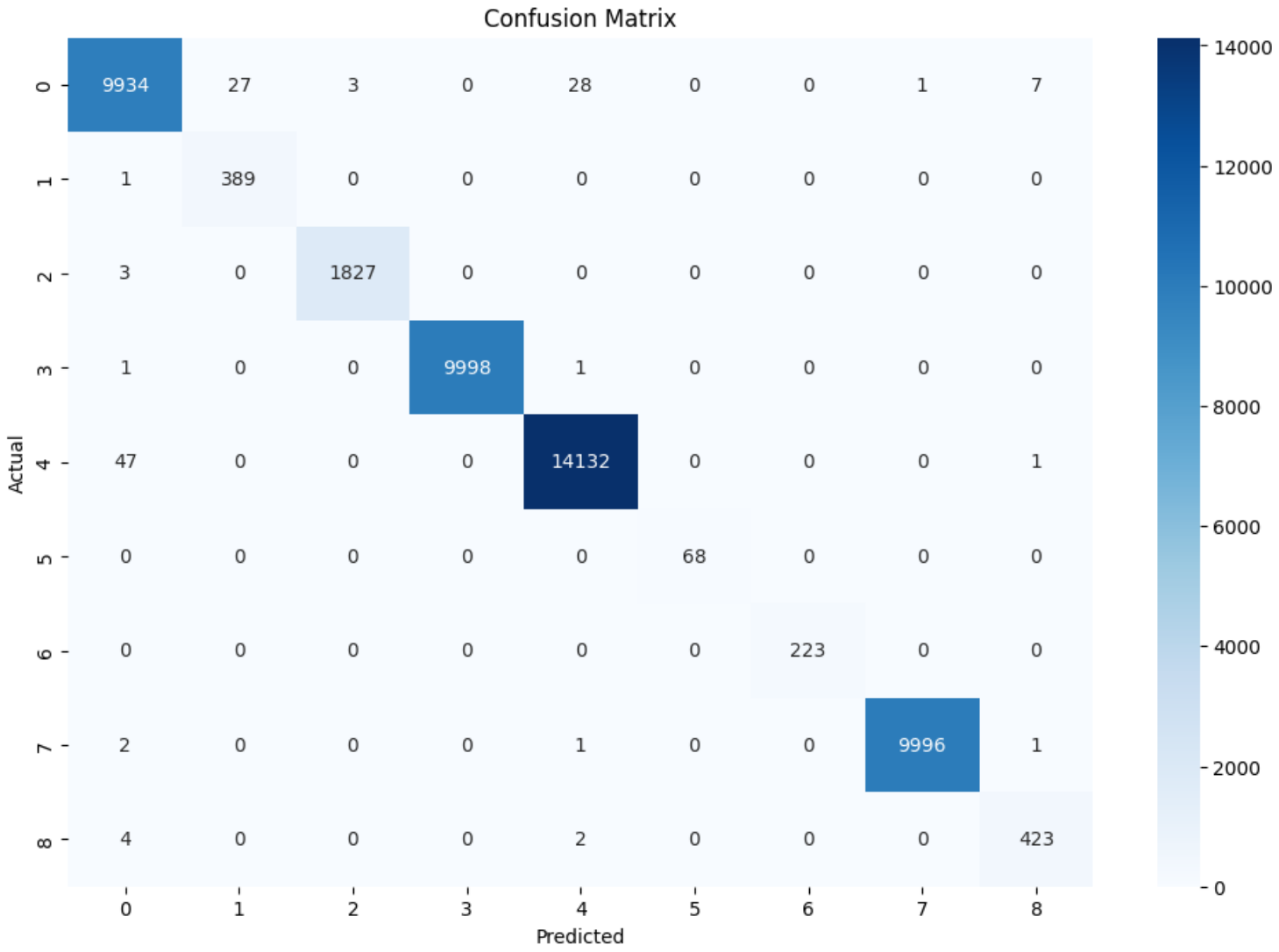

Figure 4.88

4.2.6.3. Anomaly Detection (Unknown Attacks)

- To enhance the system's capability of detecting unknown or zero-day attacks, we implemented an anomaly-based detection module using the Isolation Forest algorithm. The key idea is to model what *normal (benign)* traffic looks like and then identify any data points that significantly deviate from this norm as *potential anomalies*.
- **Steps:**
  - Extract Only Benign Traffic:
    - We transformed the class label "**BENIGN**" using the previously saved **LabelEncoder**.
    - Using the encoded label, we filtered the **X_train_scaled** DataFrame to extract only benign records **(X_benign)**.
    - This dataset is assumed to represent normal network behavior.

```
X_train_df = pd.DataFrame(X_train_scaled, columns=features)
X_train_df.index = y_train.index

benign_class = le.transform(['BENIGN'])[0]
X_benign = X_train_df[y_train == benign_class]
```

Figure 4.89

- **Train Isolation Forest:**
  - We used **IsolationForest** from **sklearn.ensemble** to train on **X_benign**.
  - Parameters used:
    - **n_estimators**=100: Number of trees in the forest.
    - contamination=0.05: An estimate of the expected proportion of outliers (this helps in scoring).
    - random_state=42: For reproducibility.

```
X_train_df = pd.DataFrame(X_train_scaled, columns=features)
if not isinstance(y_train, pd.Series):
    y_train = pd.Series(y_train)

y_train.index = X_train_df.index
benign_class = le.transform(['BENIGN'])[0]
X_benign = X_train_df[y_train == benign_class]

print(f"BENIGN: {X_benign.shape[0]}")

iso_forest = IsolationForest(n_estimators=100, contamination=0.05, random_state=42)
iso_forest.fit(X_benign)
```

Figure 4.90

- **Save the Model:**
  - The trained model is saved using **joblib** to be used later in live or offline prediction mode.
- This anomaly detector can be used to flag suspicious traffic not seen during training or not part of the labeled classes (e.g., new types of attacks).
- It works well in combination with the signature-based and ML-based classifiers to form a hybrid IDS system.

```
joblib.dump(iso_forest, "Models/isolation_forest.joblib")
```

Figure 4.91

4.2.6.4. Hybrid Detection Method (Stacking + Isolation Forest)

- To build a more robust and intelligent IDS, we designed a hybrid detection system that leverages the strengths of both supervised learning and unsupervised anomaly detection. Specifically, we combined a Stacking Classifier with an Isolation Forest to detect both known and unknown (zero-day) attacks.
- Components of the Hybrid System:
  - Stacking Classifier (Supervised Learning):
    - Trained on labeled attack and benign data.
    - It consists of multiple base classifiers (e.g., Decision Tree, Logistic Regression, KNN, CatBoost).
    - A meta-model (Random Forest) aggregates predictions from the base classifiers.
    - Handles known attack types with high accuracy.
    - Predicts both class label and confidence/probability.
  - Isolation Forest (Unsupervised Learning):
    - Trained only in BENIGN traffic.
    - Learning the patterns of normal behavior.
    - Flags anything that deviates as anomalous.
    - Effective in detecting new/unknown/zero-day attacks.

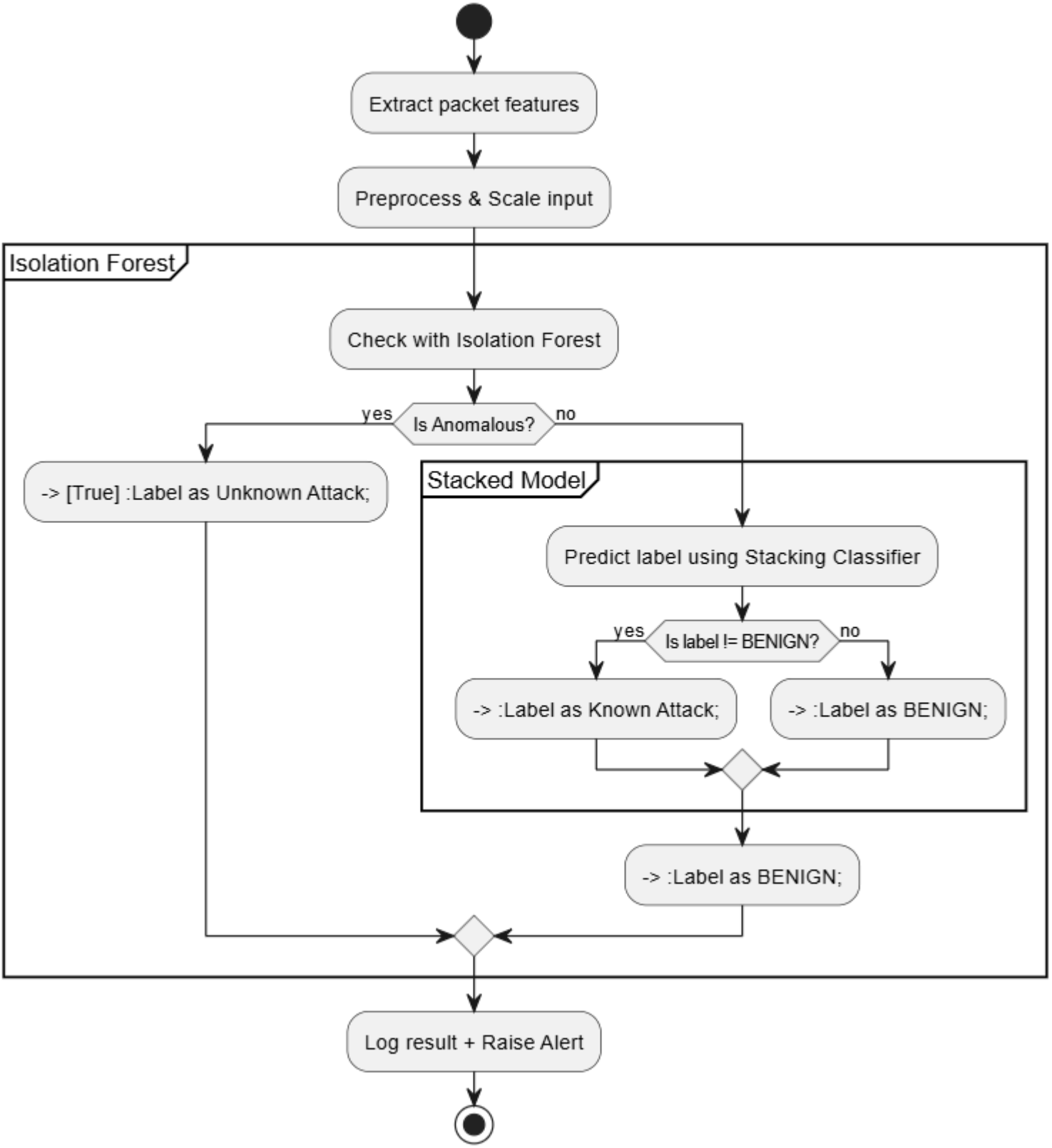
Extract packet features
Preprocess & Scale input
Isolation Forest
Check with Isolation Forest
yes
Is Anomalous?
no
-> [True] :Label as Unknown Attack;
Stacked Model
Predict label using Stacking Classifier
yes
Is label != BENIGN?
no
-> :Label as Known Attack;
-> :Label as BENIGN;
-> :Label as BENIGN;
Log result + Raise Alert

Figure 4.92

4.2.6.5. Flow-Based Feature Extraction

- In our proposed **Intrusion Detection System (IDS)**, we developed a custom Flow class designed to extract meaningful flow-level features from raw packet data.
- This process is crucial for transforming low-level network traffic into structured input suitable for machine learning models.

1. **Flow Construction:**
   - Network packets are grouped into ***flows*** using a 5-tuple key:
     - **Source IP**
     - **Destination IP**
     - **Source Port**
     - **Destination Port**
     - **Protocol**
   - The grouping is **bidirectional** meaning forward, and reverse traffic are combined to represent a complete communication session. This grouping allows for a more comprehensive analysis of each flow's behavior.

```
class Flow:
    def __init__(self, packet_list):
        self.packet_list = packet_list
        self.flows = self.group_into_flows(packet_list)
```

Figure 4.93

```
@staticmethod
    def safe_int(value):
        try:
            return int(value)
        except (ValueError, TypeError):
            return 0

    @staticmethod
    def group_into_flows(packet_list):
        flows = defaultdict(list)
        for pkt in packet_list:
            key = (pkt.src_ip, pkt.dst_ip, pkt.src_port, pkt.dst_port, pkt.protocol)
            reverse_key = (pkt.dst_ip, pkt.src_ip, pkt.dst_port, pkt.src_port, pkt.protocol)

            if reverse_key in flows:
                flows[reverse_key].append(pkt)
            else:
                flows[key].append(pkt)
        return flows
```

Figure 4.94

2. **Feature Extraction:**

- Once the flows are formed, we extract 20 handcrafted features for each flow. These features were selected based on their relevance to distinguishing between benign and malicious behavior. The extracted features include:
  - **Statistical metrics:** e.g., mean and standard deviation of packet lengths.
  - **Traffic direction:** separate calculations for forward and backward streams.
  - **Temporal properties:** such as inter-arrival times and flow duration.
  - **Protocol-related attributes:** like header lengths, segment sizes, and initial window sizes.
- The full list of features includes:
  - **FwdPacketLengthMean, FwdPacketLengthMax, FlowIATMax**
  - **SubflowBwdBytes, Init_Win_bytes_backward, TotalLengthofBwdPackets**
  - **FlowPackets/s, TotalLengthofFwdPackets, BwdPackets/s**
  - **AveragePacketSize, FlowDuration, BwdPacketLengthMean**
  - **SubflowFwdBytes, AvgBwdSegmentSize, FwdPacketLengthStd**
  - **AvgFwdSegmentSize, DestinationPort, BwdHeaderLength**
  - **PacketLengthMean, BwdPacketLengthStd**

NO	Feature Name	Description	Calculation Method
1	FwdPacketLengthMean	Average length of forward direction packets	np.mean(fwd_sizes)
2	FwdPacketLengthMax	Maximum length among all forward packets	np.max(fwd_sizes)
3	FlowIATMax	Maximum inter-arrival time between packets in the flow	max (fwd_times + bwd_times)
4	SubflowBwdBytes	Total bytes transferred in the backward subflow	sum(bwd_sizes)

5	Init_Win_bytes_backward	Initial TCP window size from the first backward packet	Extracted from first backward packet: pkt.get_window_size()
6	TotalLengthofBwdPackets	Total byte length of all backward packets	sum(bwd_sizes)
7	FlowPackets/s	Number of packets per second in the flow	(len(fwd_sizes) + len(bwd_sizes)) / flow_duration
8	TotalLengthofFwdPackets	Total byte length of all forward packets	sum(fwd_sizes)
9	BwdPackets/s	Number of backward packets per second	len(bwd_sizes) / flow_duration
10	AveragePacketSize	Average size of all packets in the flow	np.mean(fwd_sizes + bwd_sizes)
11	FlowDuration	Total duration of the flow	flow_end - flow_start
12	BwdPacketLengthMean	Average length of backward packets	np.mean(bwd_sizes)
13	SubflowFwdBytes	Total bytes transferred in the forward subflow	sum(fwd_sizes)
14	AvgBwdSegmentSize	Average segment size of backward packets	np.mean(bwd_seg_size)

15	**FwdPacketLengthStd**	**Standard deviation of forward packet lengths**	**np.std(fwd_sizes)**
16	**AvgFwdSegmentSize**	**Average segment size of forward packets**	**np.mean(fwd_seg_size)**
17	**DestinationPort**	**Destination port number from the first packet**	**pkt.dst_port**
18	**BwdHeaderLength**	**Total length of headers in backward packets**	**sum(header_length for each backward packet)**
19	**PacketLengthMean**	**Mean of all packet lengths in both directions**	**np.mean(fwd_sizes + bwd_sizes)**
20	**BwdPacketLengthStd**	**Standard deviation of backward packet lengths**	**np.std(bwd_sizes)**

Table 4.4

```python
@staticmethod
    def compute_features(flow_packets):
        fwd_sizes, bwd_sizes = [], []
        fwd_seg_size, bwd_seg_size = [], []
        fwd_times, bwd_times = [], []

        fwd_header_len = bwd_header_len = 0
        subflow_fwd_bytes = subflow_bwd_bytes = 0
        init_win_bytes_bwd = 0

        flow_start = flow_packets[0].time
        flow_end = flow_start

        src_ip = flow_packets[0].src_ip
        dst_ip = flow_packets[0].dst_ip

        prev_fwd_time = prev_bwd_time = None

        for pkt in flow_packets:
            direction = 'fwd' if pkt.src_ip == src_ip else 'bwd'
            pkt_time = pkt.time
            pkt_size = pkt.data_size
            flow_end = max(flow_end, pkt_time)

            header_len = pkt.get_header_length()
            seg_size = pkt.get_payload_length()

            if direction == 'fwd':
                fwd_sizes.append(pkt_size)
                subflow_fwd_bytes += pkt_size
                fwd_seg_size.append(seg_size)
                fwd_header_len += header_len

                if prev_fwd_time is not None:
                    fwd_times.append(pkt_time - prev_fwd_time)
                prev_fwd_time = pkt_time
            else:
                bwd_sizes.append(pkt_size)
                subflow_bwd_bytes += pkt_size
                bwd_seg_size.append(seg_size)
                bwd_header_len += header_len

                if init_win_bytes_bwd == 0:
                    init_win_bytes_bwd = pkt.get_window_size()

                if prev_bwd_time is not None:
                    bwd_times.append(pkt_time - prev_bwd_time)
                prev_bwd_time = pkt_time

        flow_duration = flow_end - flow_start if flow_end > flow_start else 1e-6
        total_pkt_count = len(fwd_sizes) + len(bwd_sizes)
        total_bytes = subflow_fwd_bytes + subflow_bwd_bytes

        dst_port = Flow.safe_int(flow_packets[0].dst_port) if flow_packets else 0

        features = [
            np.mean(fwd_sizes) if fwd_sizes else 0,                     # FwdPacketLengthMean
            np.max(fwd_sizes) if fwd_sizes else 0,                      # FwdPacketLengthMax
            max(fwd_times + bwd_times) if (fwd_times + bwd_times) else 0,# FlowIATMax
            subflow_bwd_bytes,                                          # SubflowBwdBytes
            init_win_bytes_bwd,                                         # Init_Win_bytes_backward
            sum(bwd_sizes),                                             # TotalLengthofBwdPackets
            total_pkt_count / flow_duration if flow_duration else 0,    # FlowPackets/s
            sum(fwd_sizes),                                             # TotalLengthofFwdPackets
            len(bwd_sizes) / flow_duration if flow_duration else 0,     # BwdPackets/s
            np.mean(fwd_sizes + bwd_sizes) if total_pkt_count else 0,   # AveragePacketSize
            flow_duration,                                              # FlowDuration
            np.mean(bwd_sizes) if bwd_sizes else 0,                     # BwdPacketLengthMean
            subflow_fwd_bytes,                                          # SubflowFwdBytes
            np.mean(bwd_seg_size) if bwd_seg_size else 0,               # AvgBwdSegmentSize
            np.std(fwd_sizes) if fwd_sizes else 0,                      # FwdPacketLengthStd
            np.mean(fwd_seg_size) if fwd_seg_size else 0,               # AvgFwdSegmentSize
            dst_port,                                                   # DestinationPort
            bwd_header_len,                                             # BwdHeaderLength
            np.mean(fwd_sizes + bwd_sizes) if total_pkt_count else 0,   # PacketLengthMean
            np.std(bwd_sizes) if bwd_sizes else 0                       # BwdPacketLengthStd
        ]

        return features
```

Figure 4.95

3. **Purpose and Integration:**

- These features serve as the input for both:
  - The **stacked classification model** (used to detect known attacks).
  - The **Isolation Forest is an anomaly detector** (used to detect unknown threats based on deviations from benign traffic patterns).
- The design ensures compatibility with the models trained earlier using the **CIC-IDS2017** dataset, thereby allowing seamless inference in real-time environments.

4. **Real-Time Readiness:**

- The Flow class is optimized for use in real-time scenarios, enabling it to process packets and extract features efficiently without relying on third-party flow processors.

4.2.6.6. Anomaly-Based Intrusion Detection System

- To implement an intelligent anomaly-based intrusion detection mechanism, the **AnomalyIDS** class was developed. This class plays a critical role in identifying both **known attack types** using supervised learning and **unknown/zero-day attacks** using unsupervised anomaly detection techniques.
- It combines both supervised learning and unsupervised anomaly detection techniques to achieve high accuracy in identifying known attacks and flexibility in detecting unknown threats.
- The **AnomalyIDS** class operates by extracting features from packet flows, normalizing them using a pre-fitted scaler, and then passing them through a hybrid detection pipeline. It integrates seamlessly with the database (for logging and alerting), the custom Packet and Flow classes (for feature extraction), and supports parallel flow processing using multithreading for real-time performance.
- The AnomalyIDS class integrates two machine learning models:
  - The **stacked classification model** (used to detect known attacks).
  - The **Isolation Forest is an anomaly detector** (used to detect unknown threats based on deviations from benign traffic patterns).

- By combining both models, the class enhances the detection capabilities of the IDS, providing:
  - High precision for known threats, and
  - Flexibility and adaptability for detecting novel attacks.

```
class AnomalyIDS:
    def __init__(self, clf_model,iso_model, scaler, label_encoder, feature_order, db_path="DB/IDS.db",threshold=-0.000001):
        self.clf_model = clf_model
        self.iso_model = iso_model
        self.scaler = scaler
        self.label_encoder = label_encoder
        self.feature_order = feature_order
        self.threshold = threshold
        self.db_path = db_path
```

Figure 4.96

- The **AnomalyIDS** Class perform the following:
  - **Flow Processing:** Groups raw packets into flows and extracts statistical features per flow.
  - **Feature Scaling:** Uses a saved **MinMaxScaler** to normalize features before prediction.
  - **Anomaly Scoring:** Computes an anomaly score using Isolation Forest; detects unseen behaviors.
  - **Classification:** For normal flows, uses a trained stacking model to classify known attacks.
  - **Logging & Alerting:** Records suspicious activity into a structured SQLite database.
  - **Batch Prediction:** Supports prediction from CSV files to test or validate model behavior offline.
- **Functionalities:**
  - **process_flow(flow_item, conn)**
    - Processes a single bidirectional flow of packets:
      - Extracts 20 handcrafted statistical features using the Flow class.
      - Scales the features using a **MinMaxScaler**.
      - Detects anomalies using the **IsolationForest** model.
      - If not anomalous, classifies the traffic using the **StackingClassifier**.
      - If the result is an attack (not "**BENIGN**"), it logs and raises an alert in the database.

```python
def process_flow(self, flow_item, conn=None):
        flow_key, flow_packets = flow_item
        own_conn = False
        try:
            if conn is None:
                DB = Database(self.db_path)
                conn = DB.connect()
                own_conn = True
            flow_handler = Flow(flow_packets)
            stats = flow_handler.compute_features(flow_packets)
            pkt_obj = flow_packets[0]
            raw_features = stats
            if len(raw_features) != len(self.feature_order):
                raise ValueError("Mismatch between extracted features and model's feature order")
            df = pd.DataFrame([raw_features], columns=self.feature_order)
            df_scaled = pd.DataFrame(self.scaler.transform(df), columns=self.feature_order)
            score = self.iso_model.decision_function(df_scaled)[0]
            if score < self.threshold:
                predicted_class = "Unknown Attack"
            else:
                pred = self.clf_model.predict(df_scaled)[0]
                predicted_class = self.label_encoder.inverse_transform([pred])[0]
            if predicted_class != "BENIGN":
                timestamp = datetime.fromtimestamp(pkt_obj.time).strftime("%Y-%m-%d %H:%M:%S")
                src_ip, dst_ip, sport, dport, proto = flow_key
                message = f"Expected {predicted_class} Attack"
                method = "anomaly"
                action = "alert"
                log = Log(timestamp,action,src_ip,dst_ip,message,proto,method)
                alert = Alert(timestamp,src_ip,dst_ip,message,proto,method)
                log.add_to_log_table(conn)
                alert.add_to_alert_table(conn)
        except Exception as e:
            print(f"[ERROR] Failed to process flow {flow_key}: {e}")
        finally:
            if own_conn and conn:
                conn.close()
```

Figure 4.97

- **detect(scapy_packets,threads=10)**
  - Accepts a list of **Scapy** packets and performs the following:
    - Converts packets into Packet objects.
    - Group them into flows using a 5-tuple connection key.
    - Detects and classifies traffic in parallel using thread pools.
    - Updates the total number of flows and detection status.

```python
def detect(self, scapy_packets, threads=10):
        try:
            packets = [Packet(pkt) for pkt in scapy_packets]
            flows = Flow(packets).flows

            with ThreadPoolExecutor(max_workers=threads) as executor:
                futures = [executor.submit(self.process_flow, flow) for flow in flows.items()]
                for future in futures:
                    future.result()

            print(f"[INFO] Anomaly detection complete. Total flows processed: {len(flows)}")

        except Exception as e:
            print(f"[CRITICAL] Anomaly detection failed: {e}")
```

Figure 4.98

- **predict_from_CSV(csv_path)**
  - Allows offline prediction and evaluation using a structured **CSV file**:
    - Reads the input features and scales them.
    - Applies the hybrid anomaly model.
    - Returns predictions and summary statistics (e.g., counts of benign, attack, and unknown).

```python
def predict_from_csv(self, csv_path):
        try:
            df = pd.read_csv(csv_path)
            df = df[self.feature_order]
            df_scaled = self.scaler.transform(df)
            df_scaled_named = pd.DataFrame(df_scaled, columns=self.feature_order)
            predictions = []
            stats = {"BENIGN": 0,"ATTACK": 0,"UNKNOWN": 0}
            for i in range(len(df_scaled_named)):
                row_df = df_scaled_named.iloc[[i]]
                score = self.iso_model.decision_function(row_df)[0]
                if score < self.threshold:
                    predicted_class = "Unknown Attack"
                    stats["UNKNOWN"] += 1
                else:
                    pred = self.clf_model.predict(row_df)[0] #1
                    predicted_class = self.label_encoder.inverse_transform([pred])[0]
                    if predicted_class == "BENIGN":
                        stats["BENIGN"] += 1
                    else:
                        stats["ATTACK"] += 1
                predictions.append(predicted_class)
            return predictions, stats
        except Exception as e:
            print(f"[ERROR] Failed to predict from CSV: {e}")
            return [], {}
```

Figure 4.99

- **Output & Alert Handling:**
  - If an attack is detected (either known or unknown), a new Log and Alert record is created.
  - Each alert includes metadata such as timestamp, source IP, destination IP, protocol, and attack type.
  - The alert is stored in a local SQLite database for later visualization or response.

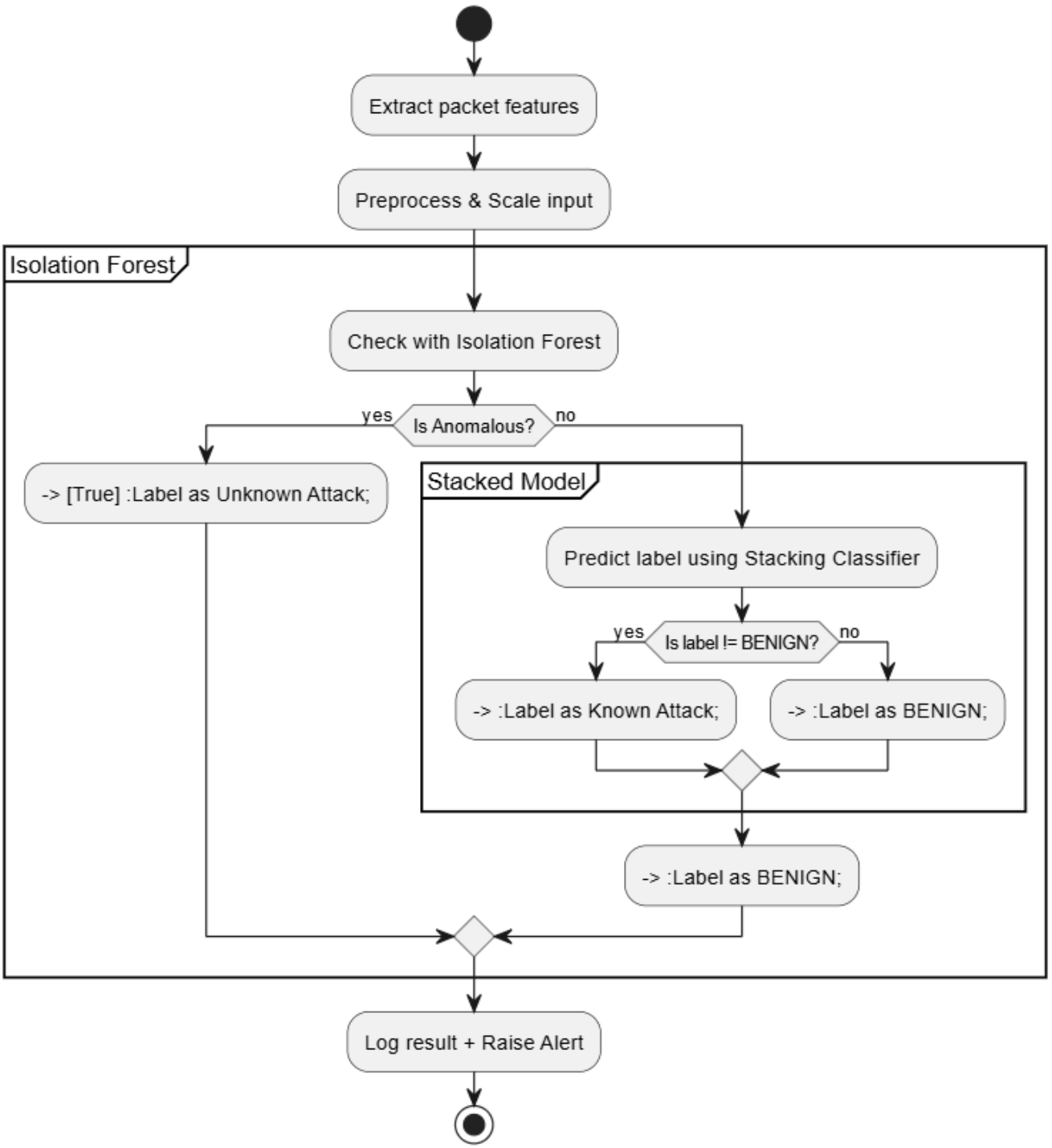

Figure 4.100

4.2.7. if the rule matched with Packet or Detecting Anomalous

- **it would do two things**:

1. **Alerting:**

   - To ensure proper tracking and response to detected threats, the Alert class was implemented as a key component of the intrusion detection system. This class is responsible for storing alert information into a persistent database and retrieving it when needed. It enables both anomaly-based and signature-based detection modules to log alerts in a unified format.

```
class Alert:
    def __init__(self, time, src_ip, dst_ip, message, attack,method):
        self.time = time
        self.src_ip = src_ip
        self.dst_ip = dst_ip
        self.message = message
        self.attack = attack
        self.method = method
```

Figure 4.101

   - Inserts the current alert instance into the alerts table within the SQLite database. It ensures that all key fields are captured and committed to storage for later analysis or UI display.

```
def add_to_alert_table(self, conn):
        cursor = conn.cursor()
        cursor.execute(
            """
            INSERT INTO alerts(timestamp,src_ip,dst_ip,message,attack,method)
            VALUES (?, ?, ?, ?, ?, ?)
            """,
            (
                self.time,
                self.src_ip,
                self.dst_ip,
                self.message,
                self.attack,
                self.method
            )
        )
        conn.commit()
```

Figure 4.102

- Retrieves all existing alerts from the database, reconstructs them into Alert objects, and returns them in a list. This is particularly useful for reporting, visualization in dashboards, or audit purposes.

```python
@classmethod
    def get_alerts_from_db(cls, conn):
        cursor = conn.cursor()
        query = "SELECT * FROM alerts"

        cursor.execute(query)
        rows = cursor.fetchall()

        alerts = []

        for row in rows:
            alert = cls(
                time = row[1],
                src_ip = row[2],
                dst_ip = row[3],
                message = row[4],
                attack = row[5],
                method = row[6]
            )

            alerts.append(alert)

        return alerts
```

Figure 4.103

- **Integration in the IDS Pipeline:**
  - Used by both the **AnomalyIDS** and **SignatureIDS** classes after successful attack detection.
  - Allows consistent and modular handling of alerts across different detection techniques.
  - Supports future extension such as email notifications, **SMS**, or **SIEM** integration.

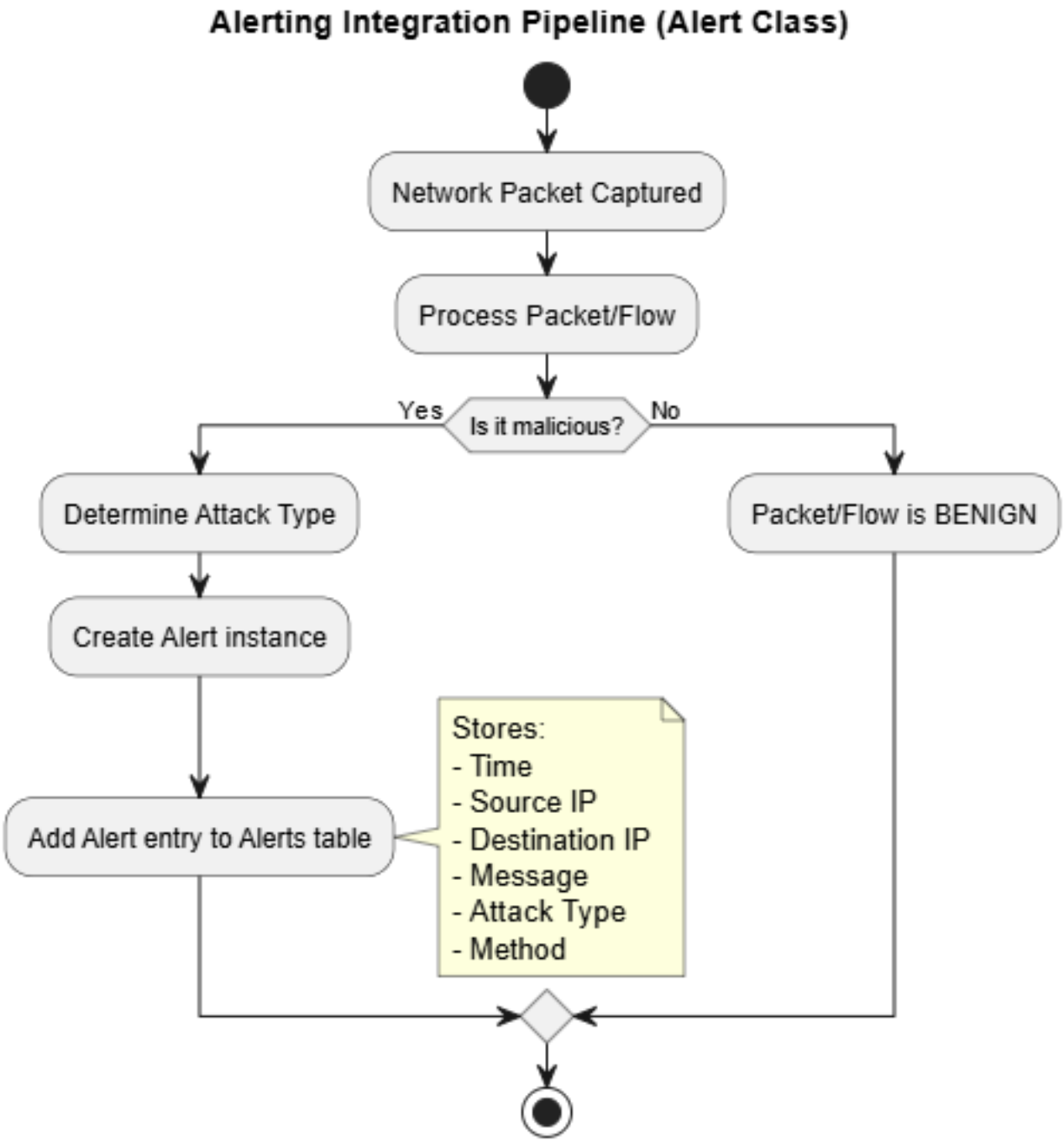
Alerting Integration Pipeline (Alert Class)
Network Packet Captured
Process Packet/Flow
Is it malicious?
Yes
No
Determine Attack Type
Packet/Flow is BENIGN
Create Alert instance
Add Alert entry to Alerts table
Stores:
- Time
- Source IP
- Destination IP
- Message
- Attack Type
- Method

Figure 4.104

2. **Logging:**
   - The **Log** class is responsible for recording and retrieving logging events related to the **Intrusion Detection System (IDS)**.
   - These events capture important details about network activities, including suspicious or malicious traffic, and store them in a persistent database table (logs).

```
class Log:
    def __init__(self, time, action, src_ip, dst_ip, message, attack,method
        self.time = time
        self.action = action
        self.src_ip = src_ip
        self.dst_ip = dst_ip
        self.message = message
        self.attack = attack
        self.method = method
```

Figure 4.105

   - Inserts the log entry into the logs table of the SQLite database using the provided connection.

```
def add_to_log_table(self, conn):
        cursor = conn.cursor()

        cursor.execute("""
            INSERT INTO logs (timestamp, event_type, src_ip, dst_ip, message, attack,method)
            VALUES (?, ?, ?, ?, ?, ?, ?)
        """, (self.time, self.action, self.src_ip, self.dst_ip, self.message, self.attack,self.method)
        )

        conn.commit()
```

Figure 4.106

   - Retrieves all stored logs from the logs table and returns them as a list of Log objects for further analysis or display.

```
@classmethod
    def get_logs_from_db(cls, conn):
        cursor = conn.cursor()
        query = "SELECT * FROM logs"

        cursor.execute(query)
        rows = cursor.fetchall()

        logs = []

        for row in rows:
            log = cls(
                time = row[1],
                action = row[2],
                src_ip = row[3],
                dst_ip = row[4],
                message = row[5],
                attack = row[6],
                method = row[7]
            )

            logs.append(log)

        return logs
```

Figure 4.107

- **Integration:**
  - Works closely with both **AnomalyIDS** and **SignatureIDS** systems.
  - An event called whenever (especially an alert or detection) needs to be persistently recorded.
  - The stored data supports system auditing, dashboard visualization, and forensic analysis.

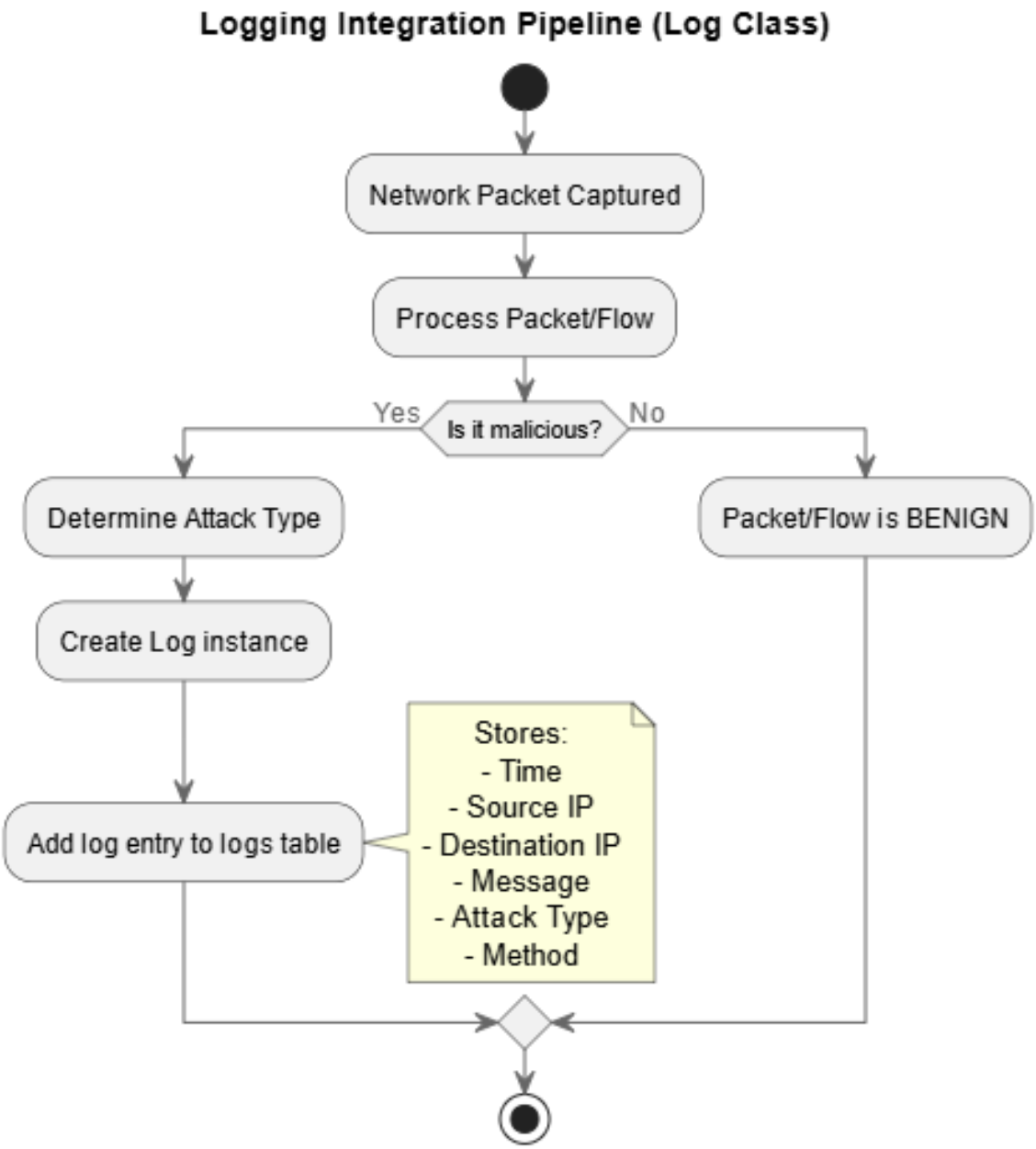

Figure 4.108

4.2.8. AlertAnalyzer Class

- The **AlertAnalyzer** class is a simple analytical utility designed to process and extract insights from a list of alert objects (instances of the Alert class). It provides statistical summaries such as identifying the most frequent source and destination IPs that triggered alerts. This helps administrators understand the most common offenders and targets in the network, assisting in further investigation and mitigation.
- This class is typically used in the **admin dashboard or backend analytics module** to highlight common sources of attacks and potential targets, allowing network admins to identify patterns or possible compromised devices.

```python
class AlertAnalyzer:
    def __init__(self, alerts):
        self.alerts = alerts

    def get_top_src_ip(self):
        src_ip_count = {}

        for alert in self.alerts:
            src_ip_count[alert.src_ip] = src_ip_count.get(alert.src_ip, 0) + 1

        top_ip = max(src_ip_count, key=src_ip_count.get, default="N/A")
        return top_ip

    def get_top_dst_ip(self):
        dst_ip_count = {}

        for alert in self.alerts:
            dst_ip_count[alert.dst_ip] = dst_ip_count.get(alert.dst_ip, 0) + 1

        top_ip = max(dst_ip_count, key=dst_ip_count.get, default="N/A")
        return top_ip

    def get_all_counts(self):
        return {
            "top_src_ip": self.get_top_src_ip(),
            "top_dst_ip": self.get_top_dst_ip(),
        }
```

Figure 4.109

- **Integration Pipeline:**

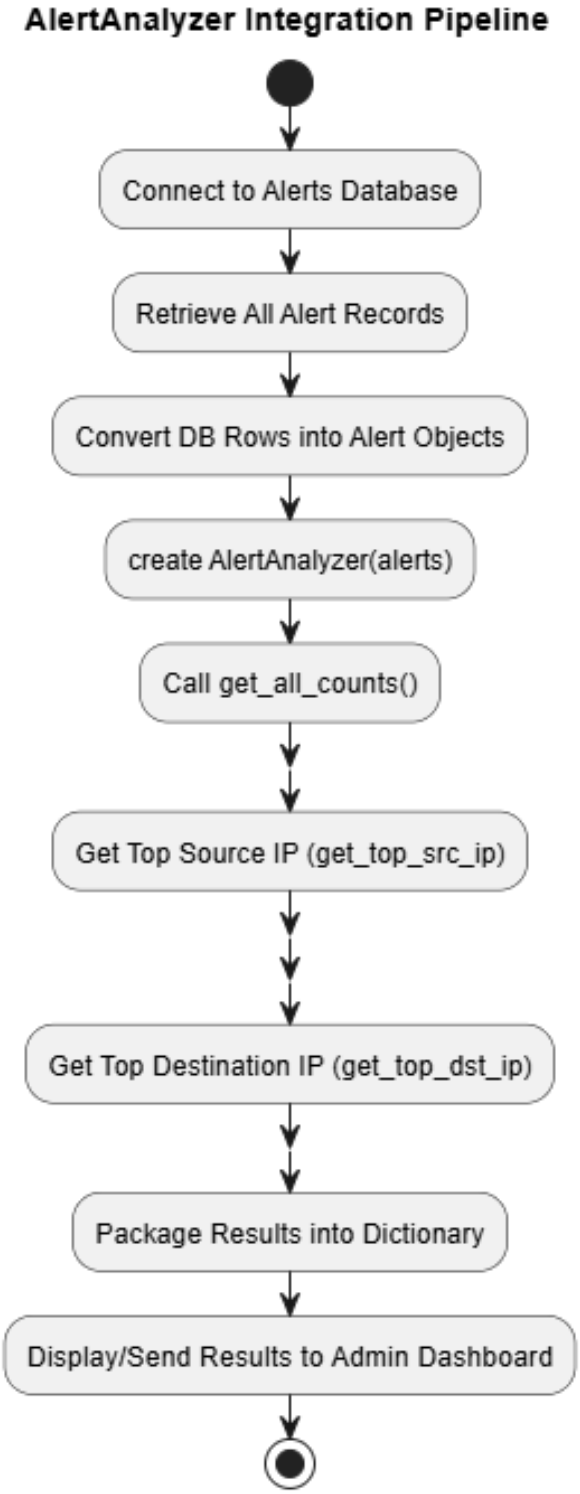

Figure 4.110

- To ensure proper tracking and response to detected threats, the Alert class was implemented as a key component of the intrusion detection system. This class is responsible for storing alert information into a persistent database and retrieving it when needed. It enables both anomaly-based and signature-based detection modules to log alerts in a unified format.

4.2.9. Full Stack Integration & Web Dashboard (Flask Application)

- All the previous it was into the core, how Matching done and what happen if matched. Now we have reached the last point, which is how we will display all this to the admin.
- We used it for visualizing and using the UI Flask framework.
- Flask is a lightweight and flexible Python web framework used for building web applications. It is classified as a micro-framework because it provides the essentials for web development without imposing too many restrictions or dependencies. Flask is known for its simplicity, ease of use, and extensibility, making it a popular choice for both small and large projects.
- The Flask-based web system serves as the central interface for monitoring, controlling, and analyzing the behavior of the hybrid Intrusion Detection System (IDS). It combines both Signature-Based and Anomaly-Based detection mechanisms with an interactive dashboard for real-time visibility.
- **Install FLASK:**

```
pip install flask
```

Figure 4.111

- **Import FLASK:**

```
from flask import Flask, render_template
```

Figure 4.112

- **Create a name for the Application:**

```
IDS_app = Flask(__name__)
```

Figure 4.113

- **Implement The Front-End Pages with HTMl, CSS & JS:**

1. Creating Partial Templates

a. Navbar

```html
<nav class="navbar navbar-expand-lg navbar-dark sticky-top">
    <div class="container-fluid">
        <a class="navbar-brand d-flex align-items-center" href="#">
            <img class="flying-logo" src="{{url_for('static',filename='IMG/w1.png')}}" alt="Logo">
            <span class="fw-bold fs-4">HAWKEYE IDS</span>
        </a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navMenu"
            aria-controls="navMenu" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse justify-content-end" id="navMenu">
            <ul class="navbar-nav mb-2 mb-lg-0">

                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('signature_dashboard') }}"><i class="fa-solid fa-file-signature"></i> Signature IDS</a>
                </li>

                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('anomaly_dashboard') }}"><i class="fa-solid fa-brain"></i> Anomaly IDS</a>
                </li>


                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('upload_csv') }}"><i class="fas fa-upload me-1"></i>Upload CSV</a>
                </li>

                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('upload_pcap')}}"><i class="fas fa-upload me-1"></i>upload PCAP</a>
                </li>

                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('rules_dashboard')}}"><i class="fa-solid fa-gavel me-1"></i>Rules</a>
                </li>

            </ul>
        </div>
    </div>
</nav>
```

Figure 4.114

b. Footer

```html
<footer class="site-footer text-white mt-5 pt-4 pb-3">
    <div class="container">
        <div class="row text-start text-md-left">
            <!-- About Section -->
            <div class="col-md-4 mb-4">
                <h5 class="fw-bold">About HAWKEYE IDS</h5>
                <p>HAWKEYE IDS is a real-time Intrusion Detection System designed to detect and alert on malicious activities with high accuracy and performance.</
            </div>

            <!-- Contact Section -->
            <div class="col-md-4 mb-4">
                <h5 class="fw-bold">Contact Us</h5>
                <p><i class="fas fa-map-marker-alt me-2"></i>123 XXXXX XXX XXX XXXXX</p>
                <p><i class="fas fa-phone-alt me-2"></i>+20 1X XXX XX XXX</p>
                <p><i class="fas fa-envelope me-2"></i><a href="mailto:contact@hawkeye.com" class="text-white text-decoration-none">contact@hawkeye.com</a></p>
            </div>

            <!-- Social Media Section -->
            <div class="col-md-4 mb-4">
                <h5 class="fw-bold">Follow Us</h5>
                <div class="d-flex gap-3">
                    <a href="#" class="text-white fs-5 social-icon"><i class="fab fa-facebook-f"></i></a>
                    <a href="#" class="text-white fs-5 social-icon"><i class="fab fa-twitter"></i></a>
                    <a href="#" class="text-white fs-5 social-icon"><i class="fab fa-linkedin-in"></i></a>
                    <a href="#" class="text-white fs-5 social-icon"><i class="fab fa-github"></i></a>
                </div>
                <form class="mt-3" action="mailto:contact@hawkeye.com" method="POST" enctype="text/plain">
                    <button type="submit" class="btn btn-warning btn-sm mt-2 shadow">Send Us an Email</button>
                </form>
            </div>
        </div>

        <hr class="bg-white">
        <div class="text-center small">
            © 2025 HAWKEYE IDS | Designed by IDS Team
        </div>
    </div>
</footer>
```

Figure 4.115

2. **Create A Base.html**
   - defines the main layout of the Flask-based Intrusion Detection System (IDS) dashboard. It is a reusable and responsive structure that loads dynamic content based on Flask routes, and ensures consistency across all pages using Bootstrap, custom style sheets, and partial templates

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{title}}</title>

    <!-- Bootstrap & Font Awesome -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">

    <!-- Custom CSS -->
    <link rel="stylesheet" href="{{url_for('static',filename='CSS/style.css')}}">
    <link rel="stylesheet" href="{{url_for('static',filename='CSS/navbar.css')}}">
    <link rel="stylesheet" href="{{url_for('static',filename='CSS/footer.css')}}">

    <link rel="shortcut icon" href="{{url_for('static',filename='IMG/w1.png')}}">
</head>

<body>

    <!-- Page Content -->
    {% include 'nav.html'%}
    <div id="page-content" class="container-fluid mt-4">
        {% block content %}
        {% endblock %}
    </div>
    {% include 'footer.html'%}

    <!-- JS Scripts -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <script src="{{url_for('static',filename='JS/script.js')}}"></script>
    <!-- <script src="{{url_for('static',filename='JS/reload.js')}}"></script> -->
</body>

</html>
```

Figure 4.116

3. **Create HTML Code for each page to display its content and use Base.html into it.**
   a. **CSV.html.**
   b. **PCAP.html.**
   c. **Index.html.**
   d. **Anomaly.html.**
   e. **Rules.html.**

```
{% extends 'base.html'%}

{% block content %}
<!-- Summary Section -->
    <div class="row row-cols-1 row-cols-md-3 g-4 mb-4">

        <div class="col">
            <div class="card h-100">
                <div class="card-body">
                    <h5 class="card-title">
                        <i class="fa-solid fa-triangle-exclamation alert-icon" style="padding-right: 5px;"></i>
                        Total Alerts
                    </h5>
                    <p class="card-text" id="total-alerts"><b>{{num_alerts}}</b></p>
                </div>
            </div>
        </div>

        <div class="col">
            <div class="card h-100" >
                <div class="card-body">
                    <h5 class="card-title"><i class="fa-sharp fa-solid fa-paper-plane send-icon" style="padding-right: 6px;"></i>Top Source IP</h
                    <p class="card-text" id="top-source-ip"><b>{{top_src_ip}}</b></p>
                </div>
            </div>
        </div>

        <div class="col">
            <div class="card h-100" >
                <div class="card-body">
                    <h5 class="card-title"><i class="fa-solid fa-bullseye target-icon" style="padding-right: 5px;"></i>Top Target IP</h5>
                    <p class="card-text" id="top-target-ip"><b>{{top_dst_ip}}</b></p>
                </div>
            </div>
        </div>
    </div>
```

Figure 4.117

```
{% extends 'base.html' %}

{% block content %}
<div id="page-content">

    <!-- PCAP Upload Section -->
    <div class="row">
        <div class="col">
            <div class="card">
                <div class="card-body">
                    <h5 class="card-title">
                        <i class="fa-solid fa-network-wired me-2"></i>Upload PCAP File for Signature Scan
                    </h5>
                    <form method="POST" enctype="multipart/form-data" id="pcap-form">
                        <div class="mb-3">
                            <input type="file" class="form-control" name="pcap_file" accept=".pcap" required onchange="updatePCAPFileName(this)">
                            <small id="pcapHelp" class="form-text text-muted mt-1">Only .pcap files are supported.</small>
                        </div>
                        <button type="submit" class="btn btn-primary">🚀 Upload & Scan</button>
                    </form>
                </div>
            </div>
        </div>
    </div>
```

Figure 4.118

```
{% extends 'base.html'%}

{% block content %}
<div id="page-content">
    <div class="row">
        <div class="col">
            <div class="card">
                <div class="card-body">
                    <h5 class="card-title"><i class="fa-solid fa-file-zipper" style="padding-right: 5px;"></i>Upload CSV File for Batch Prediction</h5>
                    <form method="POST" enctype="multipart/form-data" id="csv-form">
                        <div class="mb-3">
                            <input type="file" class="form-control" name="csv_file" accept=".csv" required onchange="updateFileName(this)">
                            <small id="fileHelp" class="form-text text-muted mt-1">Only .csv files with correct feature order are supported.</small>
                        </div>
                        <button type="submit" class="btn btn-success">🔮 Upload & Predict</button>
                    </form>
                </div>
            </div>
        </div>
```

Figure 4.119

4. **We use Jinja2, the default Flask templating engine to:**
   - To take the Alerts, Logs and Rules from the Flask and display it onto the HTML Page we use Jinja2 Syntax.
   - Jinja2 Syntax is a powerful and widely used templating engine for Python. It is commonly used on web **frameworks like** Flask **and** Django **to generate dynamic** HTML**,** XML**, or other markup formats.** Jinja2 **allows you to embed Python-like expressions and control structures**
     - **i.** Inject variables.
     - **ii.** Serve routes dynamically.
     - **iii.** Include partial templates for reusability.

```
{% with messages = get_flashed_messages() %}
    {% if messages %}
        <div class="alert alert-danger mt-4">
            <ul class="mb-0">
                {% for message in messages %}
                    <li>{{ message }}</li>
                {% endfor %}
            </ul>
        </div>
    {% endif %}
    {% endwith %}
```

Figure 4.120

5. Write the **CSS(Bootstrab)** for the **HTML** File to give styling for the Page**.**

```
1  <html lang="en">
2      <head>
3          <meta charset="UTF-8">
4          <meta name="viewport" content="width=device-width, initial-scale=1.0">
5          <title>{{title}}</title>
6          <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css">
7          <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
8          <link rel="stylesheet" href="{{url_for('static',filename='CSS/style.css')}}">
9          <link rel="shortcut icon" href="{{url_for('static',filename='IMG/intrusion-detection.png')}}" >
10     </head>
```

Figure 4.121

```
<tbody >
    {%for alert in alerts%}
    <tr>
        <td>{{alert.time}}</td>
        <td>{{alert.src_ip}}</td>
        <td>{{alert.dst_ip}}</td>
        <td>{{alert.message}}</td>
        <td class="{% if 'Application Layer' in alert.attack %}app-layer
                    {% elif 'Transport Layer' in alert.attack %}tra-layer
                    {% elif 'Network Layer' in alert.attack %}net-layer
                    {% elif 'Data Link Layer' in alert.attack %}data-layer
                    {% elif 'Physical Layer' in alert.attack %}phy-layer
                    {% endif %}">
            {{alert.attack}}
    </td>
    </tr>
    {%endfor%}
</tbody>
```

Figure 4.122

6. Write the **JS** for the **HTML** File to add flexibility to the web page**.**

```
function toggleForm() {
    const form = document.getElementById("addRuleForm")
    form.classList.toggle("d-none");
    if (!form.classList.contains("d-none")) {
        form.classList.add("fade-in");
    } else {
        form.classList.remove("fade-in");
    }
}
```

Figure 4.123

- **Implement The Controller of the System:**
  1. **Implement the Routes of each Page.**
     a. **Siganture-Based IDS.**

```python
@IDS_app.route("/")
def signature_dashboard():
    top_src_ip = 'N/A'
    top_dst_ip = 'N/A'

    alerts = []
    logs = []
    num_alerts = 0
    loaded_rules = []

    try:
        db = Database("DB/IDS.db")
        conn = db.connect()

        logs = [
            log for log in Log.get_logs_from_db(conn)
            if log.method == "signature"
        ]

        alerts = [
            alert for alert in Alert.get_alerts_from_db(conn)
            if alert.method == "signature"
        ]

        loaded_rules = Rule.get_rules_from_db(conn)

        num_alerts = len(alerts)

        analyzer = AlertAnalyzer(alerts)
        top_src_ip = analyzer.get_top_src_ip()
        top_dst_ip = analyzer.get_top_dst_ip()

        conn.close()

    except Exception as e:
        print(f"[Error] {e}")
        return "Dashboard error", 500

    return render_template(
        "index.html",
        title="Signature IDS Dashboard",
        logs=logs,
        alerts=alerts,
        rules=loaded_rules,
        num_alerts=num_alerts,
        top_src_ip=top_src_ip,
        top_dst_ip=top_dst_ip
    )
```

Figure 4.124

b. Anomaly-Based IDS.

```
@IDS_app.route("/anomaly")
def anomaly_dashboard():
    top_src_ip = 'N/A'
    top_dst_ip = 'N/A'

    alerts = []
    logs = []

    num_alerts = 0

    try:
        db = Database("DB/IDS.db")
        conn = db.connect()

        logs = [
            log for log in Log.get_logs_from_db(conn)
            if log.method == "anomaly"
        ]

        alerts = [
            alert for alert in Alert.get_alerts_from_db(conn)
            if alert.method == "anomaly"
        ]

        num_alerts = len(alerts)

        analyzer = AlertAnalyzer(alerts)
        top_src_ip = analyzer.get_top_src_ip()
        top_dst_ip = analyzer.get_top_dst_ip()

        conn.close()

    except Exception as e:
        print(f"[Error] {e}")
        return "Anomaly dashboard error", 500

    return render_template(
        "anomaly.html",
        title="Anomaly IDS Dashboard",
        logs=logs,
        alerts=alerts,
        num_alerts=num_alerts,
        top_src_ip=top_src_ip,
        top_dst_ip=top_dst_ip
    )
```

Figure 4.125

c. **Scanning PCAP Files.**

```
@IDS_app.route("/upload_pcap", methods=["GET", "POST"])
def upload_pcap():
    alerts = []

    if request.method == "POST":
        file = request.files.get("pcap_file")

        if not file or file.filename == "":
            flash("Please upload a PCAP file")
            return redirect(request.url)

        filepath = os.path.join(UPLOAD_FOLDER, file.filename)
        file.save(filepath)

        try:

            packets = rdpcap(filepath)

            sig_ids = SignatureIDS(rules)
            alerts = sig_ids.predict_from_pcap(packets)

            flash("PCAP scanned successfully.")

        except Exception as e:
            print(f"[PCAP Error] {type(e).__name__}: {e}")
            flash("Failed to process PCAP.")

    return render_template(
        "pcap.html",
        alerts=alerts,
        title="PCAP Signature Detection"
    )
```

Figure 4.126

d. Scanning CSV Files.

```
@IDS_app.route("/csv", methods=["GET", "POST"])
def upload_csv():
    predictions = []
    stats = {}

    if request.method == "POST":
        file = request.files.get("csv_file")

        if not file or file.filename == "":
            flash("Please upload a valid CSV file")
            return redirect(request.url)

        try:
            df = pd.read_csv(file)
            path = os.path.join("Uploads", "temp_uploaded.csv")
            df.to_csv(path, index=False)

            anomaly_ids = AnomalyIDS(
                model,
                iso_forest,
                scaler,
                label_encoder,
                feature_order
            )

            predictions, stats = anomaly_ids.predict_from_csv(path)

        except Exception as e:
            print(f"[CSV Error] {e}")
            flash("An error occurred while processing the file.")
            return redirect(request.url)

    return render_template(
        "csv.html",
        predictions=predictions,
        stats=stats,
        title="CSV Anomaly Detection"
    )
```

Figure 4.127

e. **Display Rules.**

```
@IDS_app.route("/rules")
def rules_dashboard():
    rules = []

    try:
        db = Database("DB/IDS.db")
        conn = db.connect()

        if not conn:
            print("[Error] DB Connection Failed.")
            return "Database connection error", 500

        rules = Rule.get_rules_from_db(conn)
        conn.close()

    except Exception as e:
        print(f"[Error] {e}")
        return "Rules dashboard error", 500

    return render_template(
        "rules.html",
        title="Rules Dashboard",
        rules=rules
    )
```

Figure 4.128

2. **Connect to the database.**

```
db = Database("DB/IDS.db")
        conn = db.connect()

        if not conn:
            print("[Error] DB Connection Failed.")
            exit(1)
```

Figure 4.129

3. **Load a sample test. PCAP File to act as a real time traffic.**

```
# Read test pcap
        packets = rdpcap("test/BENIGN.pcapng")
```

Figure 4.130

4. **Load the Signatures rules.**

```
# Load rules
        rules = Rule.get_rules_from_db(conn)
```

Figure 4.131

5. **Model Integration**

```
# ========== Load model components ========== #
model = joblib.load(
    "Models/Models/tst1_stk_classifier.joblib"
)
iso_forest = joblib.load(
    "Models/Models/isolation_forest.joblib"
)
scaler = joblib.load(
    "Models/Scaler/scaler_minmax.save"
)
label_encoder = joblib.load(
    "Models/Label Encoder/lb_encoder.pkl"
)
feature_order = joblib.load(
    "Models/Features_Order/features_order.pkl"
)
```

Figure 4.132

6. **Launch:**
   i. **Signature-Based IDS in the background thread.**

```
# Start Signature IDS
        threading.Thread(
            target=lambda:
            SignatureIDS(rules).detect(packets),
            daemon=True
        ).start()
```

Figure 4.133

ii. **Anomaly-Based IDS in a parallel thread.**

```
Start Anomaly IDS
      threading.Thread(
          target=lambda:
          AnomalyIDS(
              model,
              iso_forest,
              scaler,
              label_encoder,
              feature_order
          ).detect(packets),
          daemon=True
          ).start()
```

Figure 4.134

7. **Start the Flask Server**

```
IDS_app.run(
            debug=True,
            port=8000
        )
```

Figure 4.135

- The Pipeline of the UI:

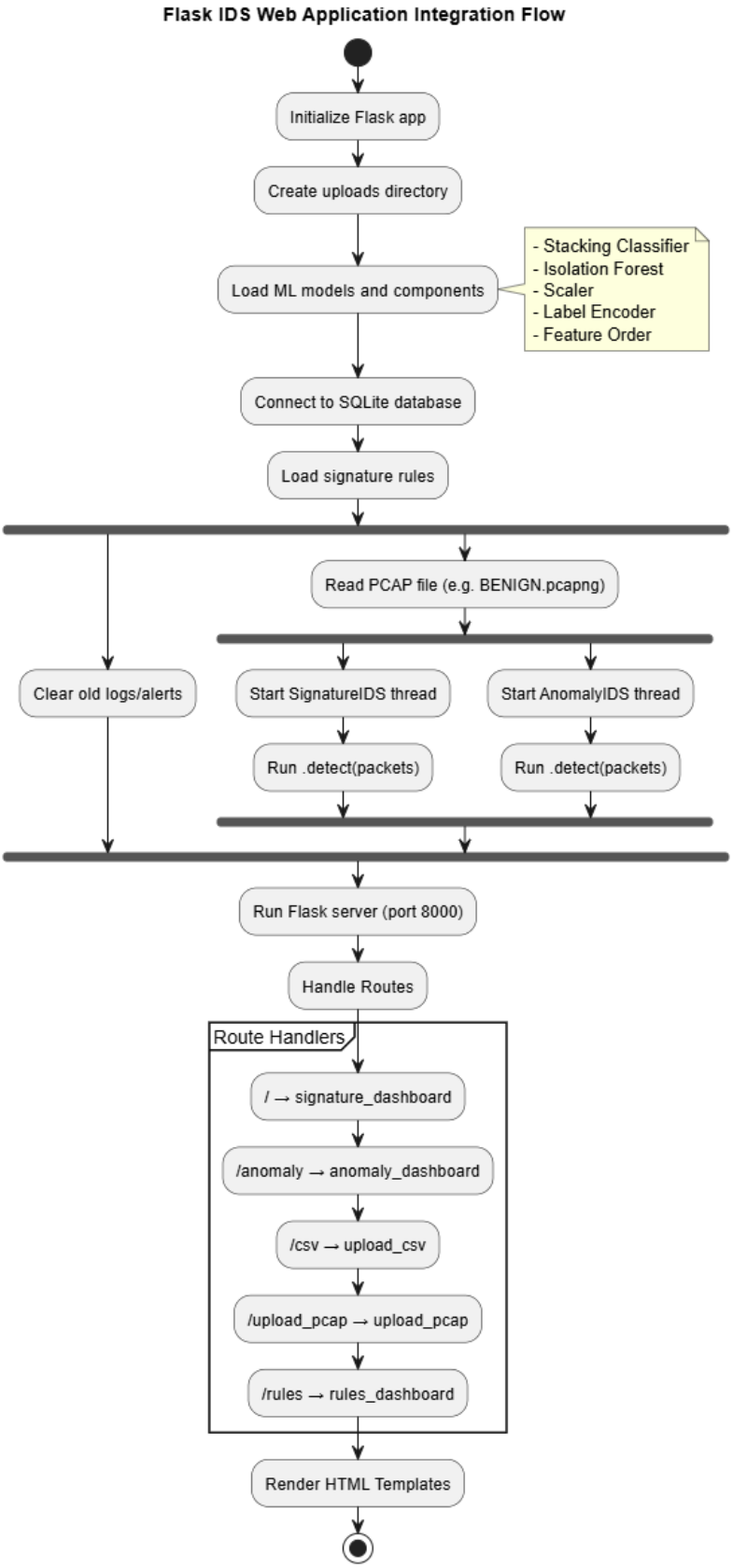

Figure 4.136

4.3. Testing & Evaluation

- This presents the testing methodology and performance evaluation of the developed **Intrusion Detection System (IDS)**, which integrates both Signature-based detection and Anomaly-based detection techniques.
- The aim is to ensure the system functions correctly, detects intrusions accurately, and maintains performance under different operational conditions.

4.3.1. Model Testing

- After training our **hybrid Anomaly Intrusion Detection System (IDS)** composed of a **Stacking Model and Isolation Forest (IF)**, we conducted comprehensive testing using multiple approaches to evaluate its accuracy, generalization, and robustness.

```
print(X_top_features.iloc[0])
print("\n\noutput:",y_top_features.iloc[0])
```

```
FwdPacketLengthMean         6.796157e-03
FwdPacketLengthMax          1.357021e-02
FlowIATMax                  6.916667e-01
SubflowBwdBytes             1.470993e-03
Init_Win_bytes_backward     3.601074e-03
TotalLengthofBwdPackets     1.470993e-03
FlowPackets/s               5.056239e-08
TotalLengthofFwdPackets     1.126963e-04
BwdPackets/s                3.608886e-08
AveragePacketSize           3.611734e-01
FlowDuration                6.927379e-01
BwdPacketLengthMean         4.421502e-01
SubflowFwdBytes             1.126963e-04
AvgBwdSegmentSize           4.421502e-01
FwdPacketLengthStd          1.585837e-02
AvgFwdSegmentSize           6.796157e-03
DestinationPort             1.220778e-03
BwdHeaderLength             1.809758e-03
PacketLengthMean            4.102229e-01
BwdPacketLengthStd          3.247607e-01
Name: 0, dtype: float64


output: Label    4
Name: 0, dtype: int64
```

```
prediction = loaded_model.predict(first_row_df)
print("Predicted class:", prediction[0])
```

```
Predicted class: 4
```

Figure 4.137

```
print(X_top_features.iloc[1])
print("\n\noutput:",y_top_features.iloc[1])
```

```
FwdPacketLengthMean         3.366518e-04
FwdPacketLengthMax          8.561644e-05
FlowIATMax                  1.666667e-07
SubflowBwdBytes             7.611864e-07
Init_Win_bytes_backward     1.525879e-05
TotalLengthofBwdPackets     7.611864e-07
FlowPackets/s               3.174603e-02
TotalLengthofFwdPackets     6.978099e-07
BwdPackets/s                2.380952e-02
AveragePacketSize           2.121341e-03
FlowDuration                1.666674e-07
BwdPacketLengthMean         1.372782e-03
SubflowFwdBytes             6.978099e-07
AvgBwdSegmentSize           1.372782e-03
FwdPacketLengthStd          0.000000e+00
AvgFwdSegmentSize           3.366518e-04
DestinationPort             2.229445e-02
BwdHeaderLength             1.809758e-04
PacketLengthMean            1.721022e-03
BwdPacketLengthStd          0.000000e+00
Name: 1, dtype: float64


output: Label    7
Name: 1, dtype: int64
```

```
first_row_df = X_top_features.head(2).tail(1)
```

```
prediction = loaded_model.predict(first_row_df)
print("Predicted class:", prediction[0])
```

```
Predicted class: 7
```

Figure 4.138

- After this we apply live testing mechanism by entering the values of the features and make the prediction.

```
loaded_model = joblib.load('Models/tst1_stk_classifier.joblib')


features = [
    'FwdPacketLengthMean',
    'FwdPacketLengthMax',
    'FlowIATMax',
    'SubflowBwdBytes',
    'Init_Win_bytes_backward',
    'TotalLengthofBwdPackets',
    'FlowPackets/s',
    'TotalLengthofFwdPackets',
    'BwdPackets/s',
    'AveragePacketSize',
    'FlowDuration',
    'BwdPacketLengthMean',
    'SubflowFwdBytes',
    'AvgBwdSegmentSize',
    'FwdPacketLengthStd',
    'AvgFwdSegmentSize',
    'DestinationPort',
    'BwdHeaderLength',
    'PacketLengthMean',
    'BwdPacketLengthStd'
]


joblib.dump(features, 'Features_Order/features_order.pkl')
# scaler = joblib.load("scaler.pkl")
scaler = joblib.load('Scaler/scaler_minmax.save')
```

Figure 4.139

```python
def make_prediction():
    user_input={}
    for column in features:
        value = input(f"Enter value for {column}: ")
        user_input[column] = float(value)if 'float' in str(df.dtypes[column]) else int(value)

    user_data = pd.DataFrame([user_input])
    # ✅ Apply scaling
    scaled_data = scaler.transform(user_data)
    prediction = loaded_model.predict(scaled_data)

    print(f"Predicted class: {prediction[0]}")


make_prediction()
```

```
Predicted class: 4
```

Figure 4.140

4.3.2. DPI (Deep Packet Inspection) Testing

- To ensure that the **Packet DPI Class** Already working as it needed.

```python
def main():
    pcap_path = "test/BENIGN.pcapng"

    print("[INFO] Reading packets...")
    scapy_packets = rdpcap(pcap_path)

    print(f"[INFO] Total packets loaded: {len(scapy_packets)}\n")

    for i, pkt in enumerate(tqdm(scapy_packets), 1):
        try:
            p = Packet(pkt)

            print(f"\nPacket #{i}")
            print(f"Time: {p.time_formatted}")
            print(f"Protocol: {p.protocol}")
            print(f"Source IP: {p.src_ip}")
            print(f"Destination IP: {p.dst_ip}")
            print(f"Source Port: {p.src_port}")
            print(f"Destination Port: {p.dst_port}")
            print(f"Flags: {p.flags}")
            print(f"Data Size: {p.data_size}")
            print(f"Payload: {p.payload[:100]}")

        except Exception as e:
            print(f"[ERROR] Failed to process packet #{i}: {e}")

    print("\n[INFO] Done processing packets.")
```

Figure 4.141

```
Packet #179
Time: 2025-05-17 15:29:44
Protocol: udp
Source IP: 142.251.37.202
Destination IP: 192.168.1.2
Source Port: 443
Destination Port: 53371
Flags: DF
Data Size: 69
Payload: KhIvxYcg$^G>3

Packet #180
Time: 2025-05-17 15:29:45
Protocol: tcp
Source IP: 192.168.1.2
Destination IP: 195.181.174.164
Source Port: 52140
Destination Port: 443
Flags: A
Data Size: 55
Payload:

Packet #181
Time: 2025-05-17 15:29:45
Protocol: tcp
Source IP: 195.181.174.164
Destination IP: 192.168.1.2
Source Port: 443
Destination Port: 52140
Flags: A
Data Size: 70
Payload:

Packet #182
Time: 2025-05-17 15:29:47
Protocol: tcp
Source IP: 192.168.1.2
Destination IP: 50.19.50.5
Source Port: 54654
Destination Port: 443
Flags: A
Data Size: 55
Payload:

Packet #183
Time: 2025-05-17 15:29:47
Protocol: tcp
Source IP: 50.19.50.5
Destination IP: 192.168.1.2
Source Port: 443
Destination Port: 54654
Flags: A
Data Size: 70
Payload:
100%|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 183/183 [00:00<00:00, 440.56it/s]
```

Figure 4.142

4.3.3. Extracting Flow Features

```
--- Flow #25 ---
Key: ('192.168.1.2', '50.19.50.5', '54654', '443', 'tcp')
Features:
  Feature 1: 55.0000
  Feature 2: 55.0000
  Feature 3: 0.0000
  Feature 4: 70.0000
  Feature 5: 466.0000
  Feature 6: 70.0000
  Feature 7: 14.2200
  Feature 8: 55.0000
  Feature 9: 7.1100
  Feature 10: 62.5000
  Feature 11: 0.1406
  Feature 12: 70.0000
  Feature 13: 55.0000
  Feature 14: 0.0000
  Feature 15: 0.0000
  Feature 16: 1.0000
  Feature 17: 443.0000
  Feature 18: 32.0000
  Feature 19: 62.5000
  Feature 20: 0.0000
```

Figure 4.143

4.3.4. Performance Evaluation

- To We evaluated the performance of the **Anomaly-Based Detection** model using standard classification metrics. The results show high detection, accuracy and robustness:
- Interpretation:
  - High Accuracy and ROC AUC indicate excellent discrimination between normal and Malicious traffic.

- The Macro F1 score suggests the model performs consistently across all classes, even if some classes have fewer samples.

Metric	Value (%)
Accuracy	99.72%
Precision (Weighted)	99.73%
Recall (Weighted)	99.72%
F1 Score (Weighted)	99.72%
F1 Score (Macro)	99.29%
ROC AUC Score	99.99%

4.3.5. Real-Time Detection Testing

- A **PCAP** file (e.g., **httpslowloris.pcac**) was loaded into the system.
- Both **SignatureIDS** and **AnomalyIDS** were executed **concurrently** using multithreading.
- The system generated alerts **in real-time** and stored them in the **SQLite** database.
- Results:
  - Alerts were generated instantly after detecting suspicious flows.
  - Flows were classified correctly based on rules or anomaly thresholds.
  - Multithreading ensured quick processing even for large packet volumes.
- For Signature-Based IDS:

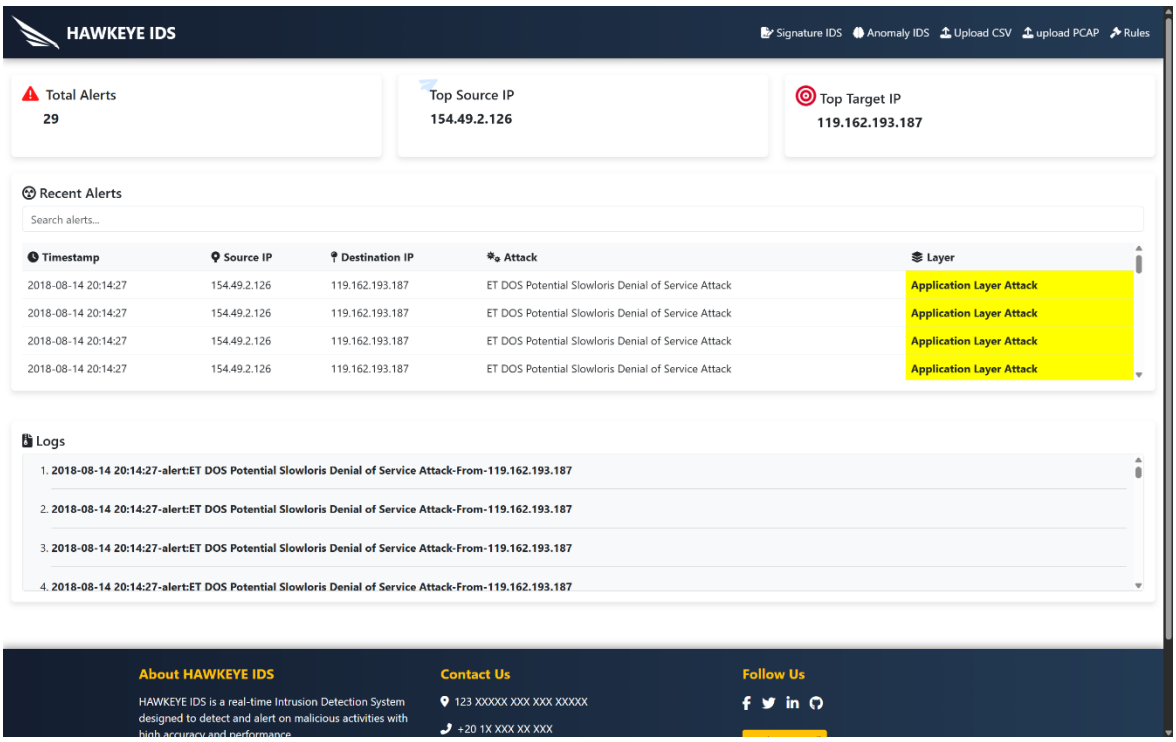

Figure 4.144

- For the Anomaly-Based IDS:

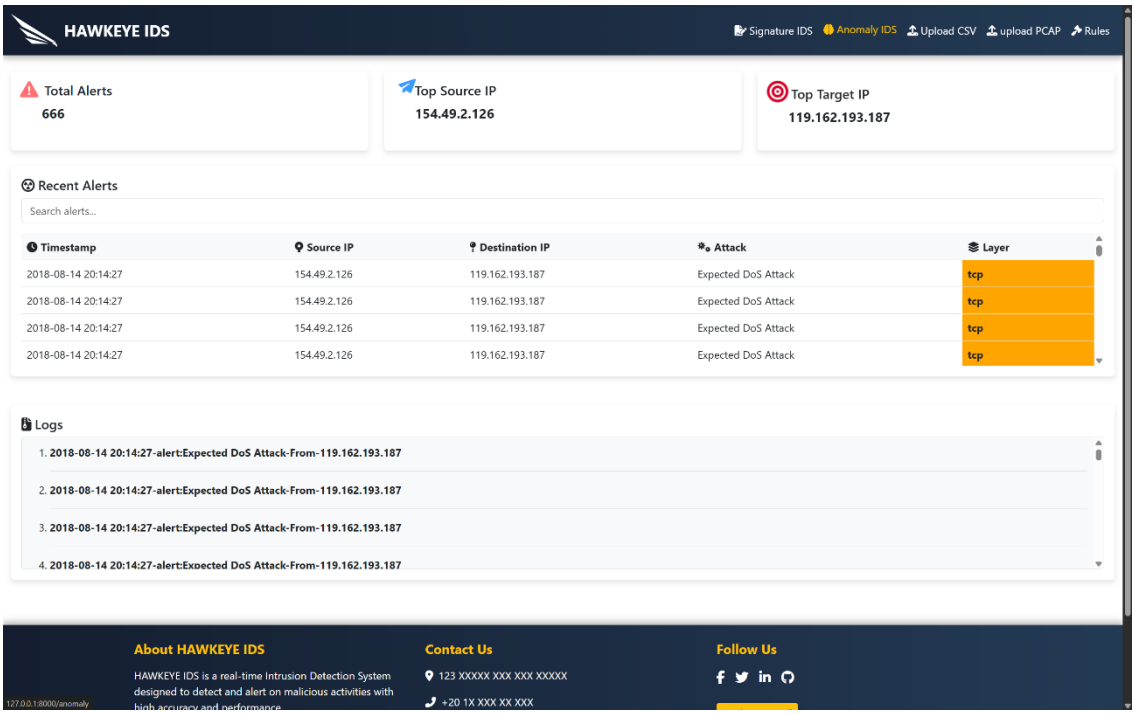

Figure 4.145

4.3.6. PCAP File Scanning

- By Selecting a file that is already infected with specific attack such as:**HTTPSlowloris**:

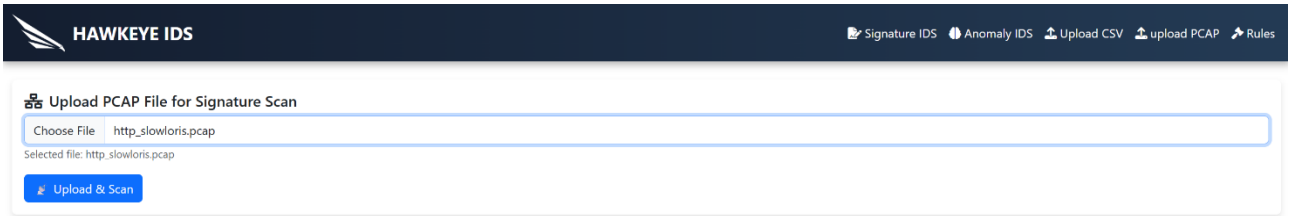

Figure 4.146

- By clicking Upload and scan it will process the file throw the Signature IDS and detect what attack in this file if it exist.
- Results:

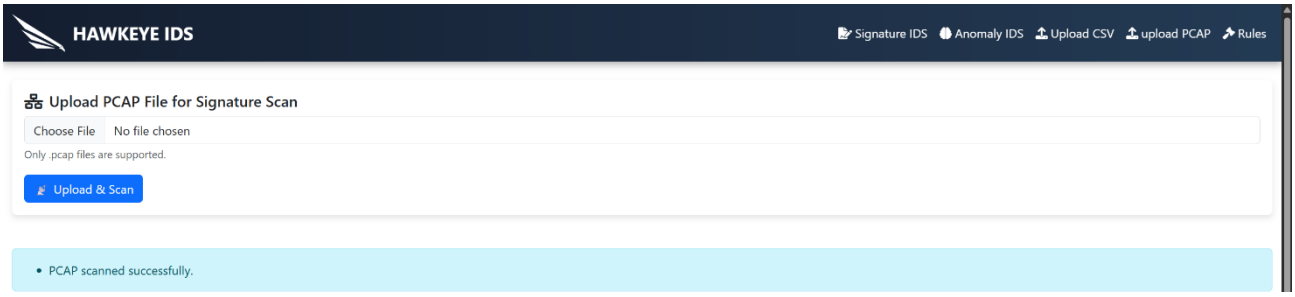

Figure 4.147

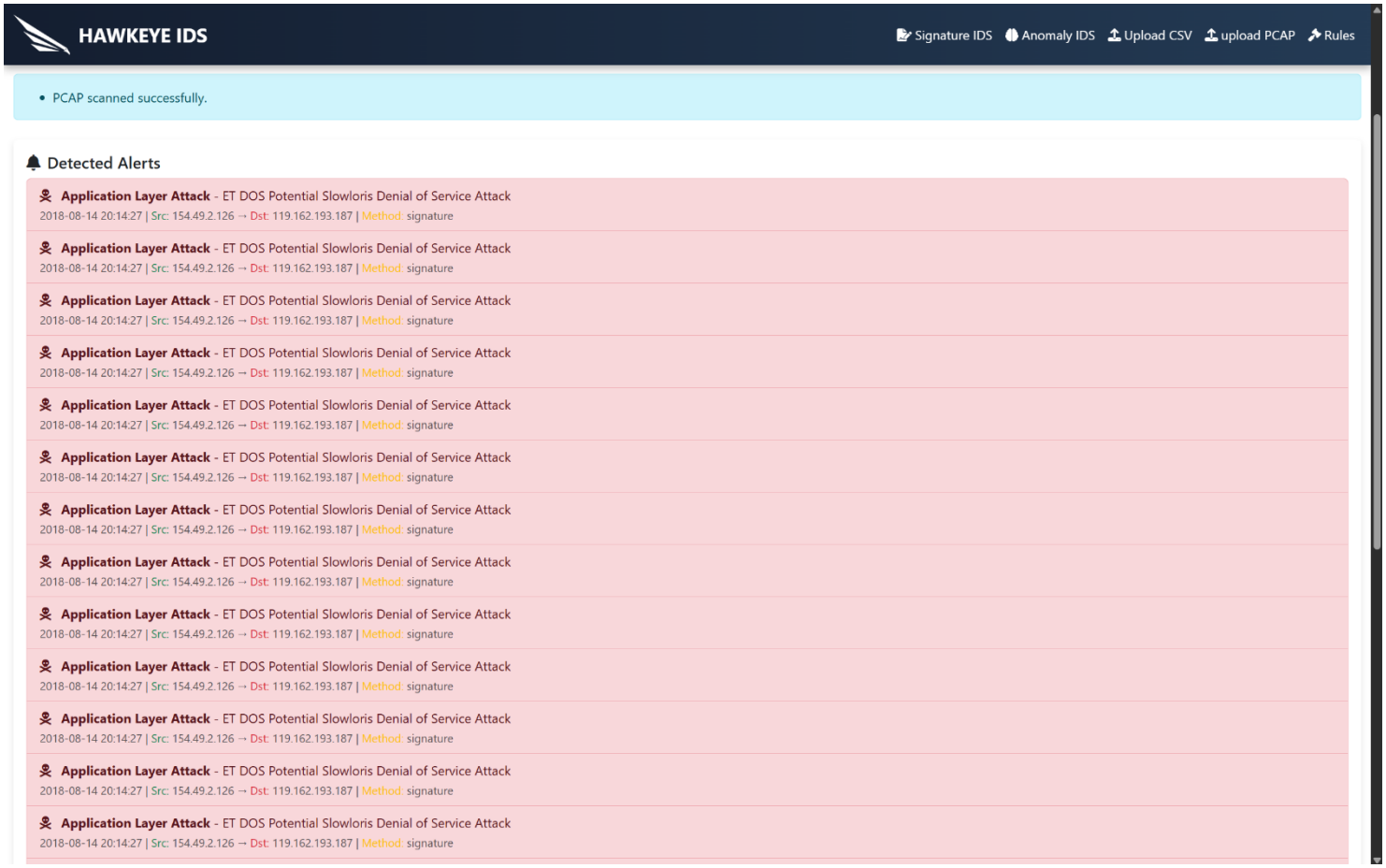

Figure 4.148

4.3.7. CSV File Prediction

- By Selecting a file that is already infected with specific attack such as: HTTPSlowloris or here in the anomaly detection now we can detect the Unknown attacks:
- By clicking Upload and scan it will process the file throw the Anomaly IDS and detect what attack in this file if it exist.

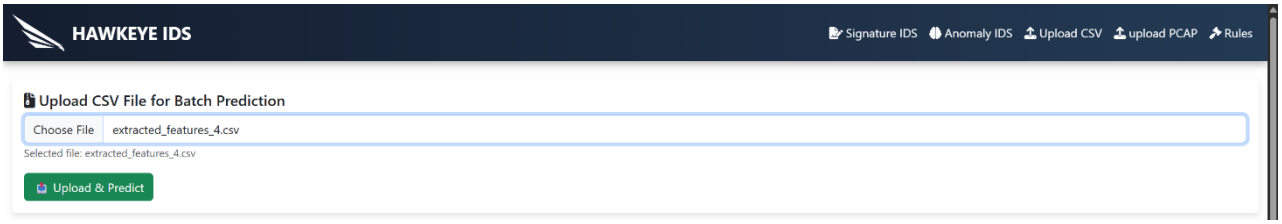

Figure 4.149

- Results:

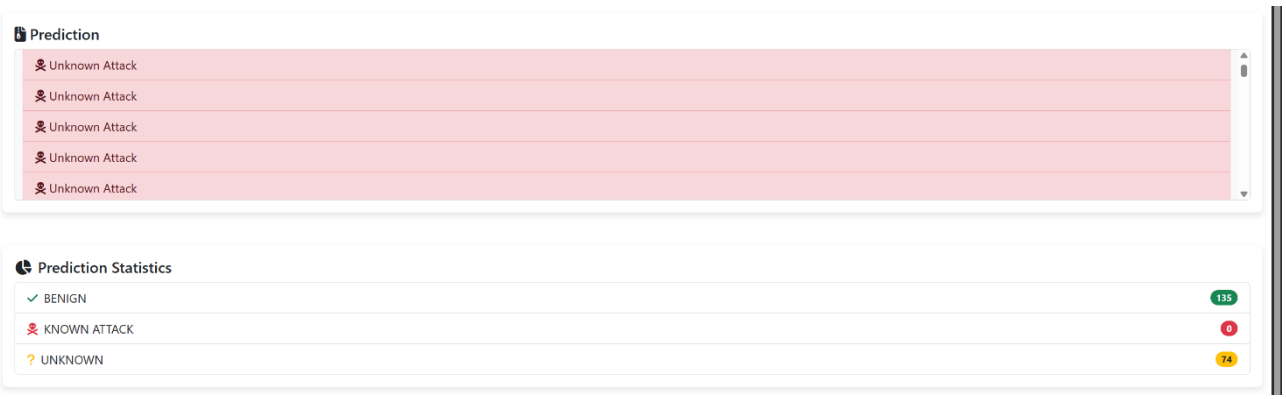

Figure 4.150

4.3.8. Display the Rules to Admin

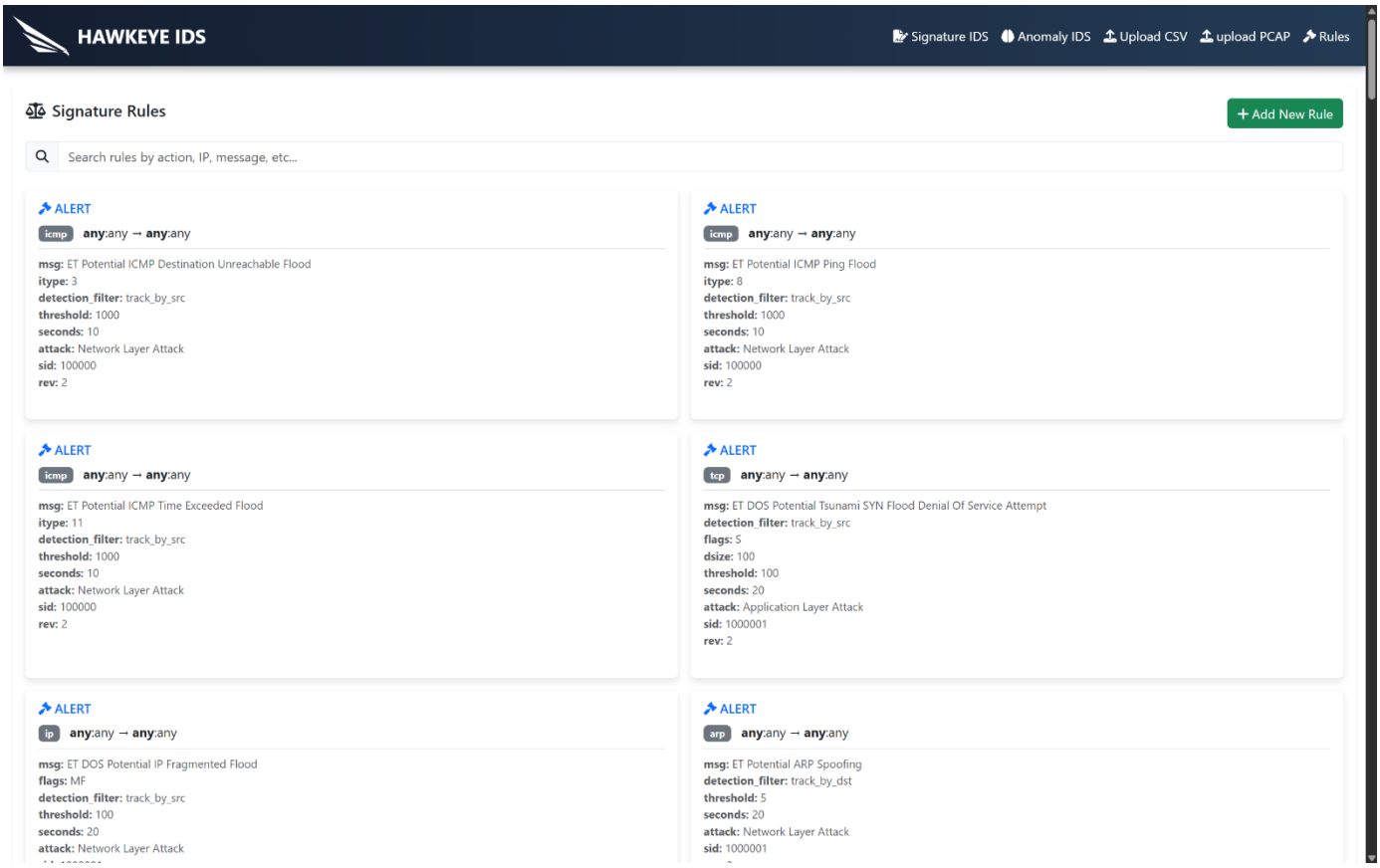

Figure 4.151

4.3.9. Adding Rules throw the Dashboard

- By Clicking **Add New Rule** it will display the Form of adding new rule:

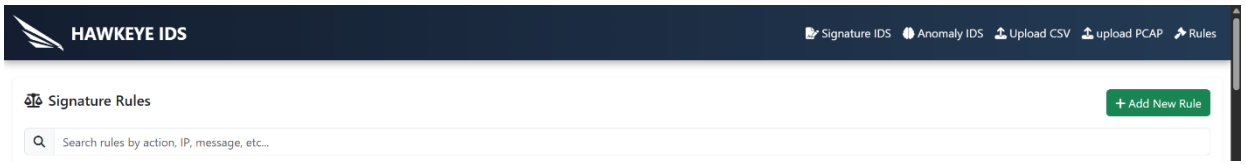

Figure 4.152

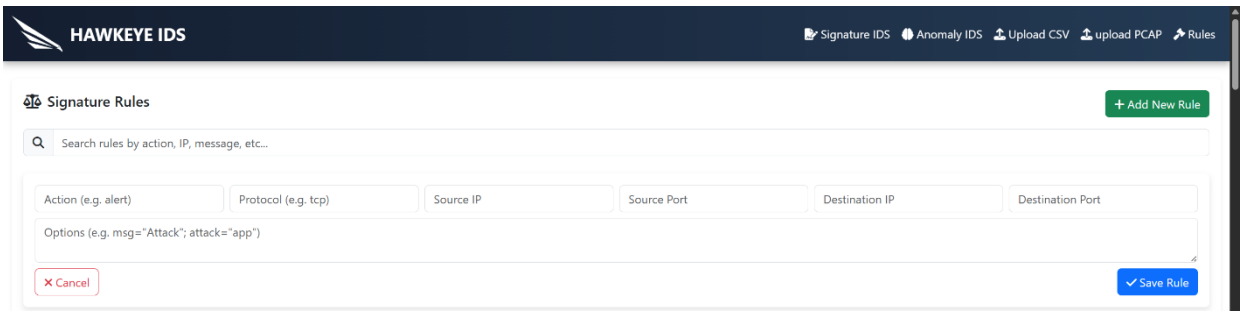

Figure 4.153

4.3.10. Limitations

- While the system performed well in most scenarios, a few limitations were noted:

  - **False Positives**: The Isolation Forest occasionally flagged rare benign traffic as malicious due to unfamiliar patterns.
  - **Signature Limitation**: New or zero-day attacks are not detected by **SignatureIDS** unless rules are updated.
  - **Threshold Sensitivity**: The anomaly detection threshold required fine-tuning to avoid misclassification.
  - **Feature Dependence**: Anomaly model heavily depends on correct feature extraction; malformed packets may reduce accuracy.

4.4. Technologies and Tools

4.4.1. Front-End & UI

- **Flask.**
- **HTML.**
- **CSS.**
- **Bootstrap.**
- **Java Script.**

4.4.2. Database

- **DB SQLite**

4.4.3. DPI

- **Scapy**

4.4.4. Programming Language

- **Python**

4.4.5. Tools

- **VS Code.**
- **DB SQLite Browser.**
- **Snort.**
- **Suricata.**
- **Git/GitHub.**
- **Lucid**

Chapter 5
Results and Discussion

5.1 Findings

- The development and testing of the **Hybrid Signature–Anomaly Intrusion Detection System (IDS)** produced several significant results related to its performance, behavior, and implementation. One of the key achievements was the successful integration of both signature-based and anomaly-based detection techniques into a single hybrid system.
- In terms of detection performance, the system achieved high accuracy in identifying both known and unknown attacks. The signature-based component reliably detected threats that matched known patterns, including various forms of malware, denial-of-service (DoS) attacks, and brute-force login attempts. The anomaly-based module, which was powered by the Isolation Forest machine learning algorithm, effectively identified deviations from normal traffic behavior, allowing the system to detect previously unseen or modified attacks, such as zero-day threats.
- The hybrid IDS achieved an overall detection accuracy of **99.72%**.
- false positive rate of **3.2%**.
- precision of **99.73%**.
- recall of **99.72%**.
- These metrics were validated using the **CICIDS2017** dataset, which contains labeled data representing both normal and malicious traffic across a variety of attack types.
- An unexpected finding during testing was the tendency of the anomaly detector to raise false alarms on encrypted traffic, such as HTTPS, due to unfamiliar traffic patterns. Additionally, when both detection modules ran simultaneously, there was a slight increase in processing latency under high network loads, which suggests the need for performance tuning in future versions.

5.2 Goals Achieved

- At the beginning of the project, several key goals were defined to guide the development of the Hybrid Intrusion Detection System (IDS). By the end of the project, most of these goals were successfully achieved. Here's an overview of how each objective was met:
- **Design and Implementation of the Hybrid IDS**
  - The main goal was to build a system that combines both signature-based and anomaly-based detection methods. This was fully achieved. We developed a complete system where both detection components work together in a single pipeline. The system processes captured traffic from .**Pcap** files and generate alerts for both known and unknown threats. Signature detection identifies attacks using predefined rules, while the anomaly detection module uses a machine learning model to detect unusual behavior.
- **Improved Accuracy and Reduced False Positives**
  - Another important goal was to increase the overall accuracy of detection while keeping the false positive rate low. The hybrid approach helped us achieve this. By combining two detection techniques, the system was able to detect a wider range of threats and give more reliable results. Although the false positives weren't eliminated (especially with encrypted

traffic), the rate was significantly lower than what we typically see with a standalone-based system.

- **Testing Using Real-World Data**
  - We aimed to test the system using realistic and diverse network traffic, and this was achieved using the CICIDS2017 dataset. This dataset includes a variety of attack types along with normal traffic, making it ideal for both training the anomaly detection model and testing the overall system performance. It helped prove the system's effectiveness under real-world scenarios.
- **User Interface for Monitoring and Alerts**
  - One of the final goals was to create a user-friendly interface for monitoring system behavior and viewing alerts in real time. We successfully built a basic dashboard that displays alerts and provides useful information to system administrators. This makes it easier to track suspicious activity and understand what's happening inside the network.

5.3 Further Work

- Although the Hybrid Intrusion Detection System (IDS) built in this project showed strong performance, there are several areas where it could be improved and expanded in the future. These enhancements would make the system more accurate, scalable, and suitable for use in real-world environments.
- One important area for future development is the ability to detect threats hidden in encrypted traffic. Currently, the system doesn't analyze the actual contents of encrypted packets (like HTTPS or SSL/TLS traffic), which means that certain attacks could go unnoticed. In future versions, we could explore behavioral analysis techniques that rely on traffic patterns (like timing, flow size, or frequency) to detect anomalies in encrypted data—without decrypting it. This would help protect user privacy while still identifying potential threats. In cases where it's legally and ethically acceptable, secure SSL/TLS inspection methods could also be considered.
- Another enhancement is to improve how anomalies are detected. Instead of using just one detection method, future versions could combine several lightweight models—like Isolation Forests, One-Class SVMs, and k-Means clustering—in what's called an ensemble approach. Each model has its own way of looking at traffic, and combining their outputs can give more balanced and accurate results. This would also help reduce false alarms.
- Additionally, introducing context-aware anomaly detection can make the system smarter. This means analyzing not just the traffic itself, but also the context behind it—like what time it happened, what device it came from, or whether it's part of a user's regular behavior. This helps the system understand what's normal in each situation and only raises alerts when something truly unusual happens.
- Another major challenge we faced was related to computational limitations. Deep learning models, which often offer higher accuracy, were not practical to train on our local hardware due to the lack of processing power. That's why we relied on simpler models like Random Forests. In the future, we could overcome this limitation by using cloud platforms like Google Cloud, AWS, or Microsoft Azure. These services offer access to powerful GPUs and TPUs, which can handle heavy training workloads. With this upgrade, we could train more complex models on larger datasets, potentially improving detection accuracy and generalization.

- Some features we planned for the system were left out due to time and resource constraints. For example, the system currently runs on a single machine, but in the future.
- It could be deployed in a distributed setup. This would allow it to monitor large-scale networks in real time, which is especially important for organizations with high traffic. The user interface also has room for improvement. Right now, the dashboard provides basic alerts and traffic monitoring for PCAP files. Future versions could offer more interactive visualizations like real-live traffic graphs, threat summaries, alert levels, and log export options, giving administrators deeper insight and control.
- Lastly, one of the most useful future upgrades would be to implement adaptive or online learning. This means the IDS would be able to keep learning from new traffic patterns and update itself automatically, without manual retraining. This kind of feature would make the system more responsive and up to date as new types of attacks emerge.
- In summary, while the current system works well and proves the value of a hybrid approach, there's a lot of potential to make it even better. With improvements in detection techniques, smarter context handling, more computing power, better interfaces, and continuous learning, the Hybrid IDS could become a much more advanced and reliable tool for protecting modern networks.

5.4 Ethical, Legal, and Social Issues

- This project raised several ethical, legal, and social considerations, particularly due to its nature as a security system that inspects network traffic.
- From an ethical standpoint, it is essential that such a system is used strictly for defensive and protective purposes. The IDS must not be repurposed for unauthorized surveillance, data harvesting, or monitoring of users without consent. Ethical deployment requires transparency, user awareness, and strict access control policies.
- Legally, the project was developed using only publicly available, open-source datasets, which helped ensure compliance with data privacy regulations such as the General Data Protection Regulation (GDPR). No personal or sensitive data from real users was involved in the training or evaluation of the system.
- On a social level, the implementation of a hybrid IDS contributes positively to the security of digital infrastructure. As cyber threats grow in scale and complexity, systems like this can play a vital role in protecting sensitive information and maintaining public trust in digital services. However, misuse of such technology can also raise privacy concerns, particularly in contexts where monitoring is done without proper oversight or legal frameworks. It is therefore important that these systems are governed by clear ethical guidelines and legal boundaries.

Chapter 6
Conclusions and Future Work

6.1 Conclusions

- ✓ Based on the test cases outlined in **Chapter 4: Signature IDS & Anomaly IDS**, both have successfully achieved its objectives and addressed the problem statement defined at the beginning of this documentation. The following key accomplishments demonstrate the system's effectiveness.
- ✓ The primary objective of this project was to design and implement a **hybrid Intrusion Detection System (IDS)** that effectively combines both **signature-based** and **machine learning-based anomaly detection** techniques. The final system can detect a wide range of cyberattacks — both known and unknown — through real-time traffic analysis. This chapter summarizes the main accomplishments, key findings, and insights gained during the project development.
- ✓ **Summary of Achievements:**
  - ➢ Throughout the development lifecycle, the system was modularly built using Python and the **Scapy** library for packet parsing. The core achievements are summarized as follows:
  - ➢ **Packet Parsing Module**: The Packet class was implemented to extract vital information from live or captured network traffic. This included fields like source/destination IP and ports, protocol type, TCP flags, payload content, window size, and header lengths.
  - ➢ **Signature-Based Detection**: A customizable rule engine was built to match packets against user-defined rules stored in an SQLite database. These rules support advanced options such as:
    - ▪ Protocol, IP, and port matching
    - ▪ Payload string or PCRE (Perl-Compatible Regular Expression) matching
    - ▪ TCP flag inspection
    - ▪ Threshold-based detection logic (e.g., source IP frequency)
    - ▪ ICMP type checking and payload length (**dsize**)
  - ➢ **Feature Extraction and Flow Analysis**: A Flow class was developed to group packets into bidirectional flows. From these flows, the system extracted 20+ handcrafted statistical features, such as packet lengths, inter-arrival times (IAT), segment sizes, and flow durations which served as inputs to the machine learning model.
  - ➢ **Anomaly-Based Detection Using ML**:
    - ▪ The machine learning component was designed as a hybrid system composed of:
    - ▪ A **Stacking Classifier** that combines multiple base classifiers such as **Decision Tree**, **Logistic Regression**, **K-Nearest Neighbors**, and **CatBoost** to detect known attack patterns.

- An **Isolation Forest** for **unsupervised anomaly detection**, capable of identifying unknown or zero-day attacks that deviate from normal traffic behavior.

- This combination allowed the system to be robust in recognizing both labeled (known) and unlabeled (unknown) attack instances.
- Real-**Time Detection & Logging**: The system was integrated with a lightweight Flask web interface to accept .**pcap** files or live traffic and display real-time alerts. Matched rules and ML-predicted anomalies were logged into a local SQLite database for later analysis.

✓ **Key Findings and Observations**

- **Stacking Classifier**: By combining the strengths of multiple learners, the stacking model produced **high accuracy, precision, and recall** across all known attack classes in the CIC-IDS2017 dataset. This ensemble technique outperformed individual base classifiers.
- **Isolation Forest Effectiveness**: Isolation Forest proved to be lightweight, fast, and effective for anomaly detection. It successfully identified suspicious behaviors not explicitly present in the training data, filling the gap left by supervised models.
- **Rule Engine Flexibility**: The rule matching logic was powerful enough to replicate basic Snort-like detection behavior, including support for threshold, flags, content, pcre, and dsize fields — allowing deep inspection of packet characteristics.
- **Real-Time Efficiency**: The system was optimized for real-time operation. Packet parsing, rule matching, and flow-based ML prediction could all run concurrently without significant performance issues for moderate traffic volumes.

✓ **Project Impact and Usefulness**

- This hybrid IDS can be deployed in educational environments, lab networks, or extended into production systems with minimal enhancements. Its modular structure, custom rule support, and machine learning core make it suitable for:
  - **Small enterprise networks**
  - **Research in anomaly detection**
  - **Cybersecurity training and simulation**
  - **Baseline development for future commercial-grade IDS tools**
- It bridges the gap between traditional static detection and dynamic behavioral modeling.

✓ **Challenges Faced**

- During the project, several challenges were encountered:
  - Balancing false positives and negatives in ML prediction, especially from the anomaly detector.

- Ensuring consistency in feature scaling and input alignment across stacking and isolation models.
- Optimizing the speed of rule-matching for high-frequency traffic.
- Hardware Resources to train the **ML Model** because it takes a lot of time and a lot ot Resources from the Laptop such as the **CPU, GPU and RAM.**

➢ Despite these challenges, the final implementation remained stable and functional.

✓ **Future Work**

➢ To extend and improve the project further, the following enhancements are recommended:

- **Encrypted Traffic Analysis:** Develop techniques for identifying anomalies in encrypted traffic using metadata features.
- **Online Learning Models:** Implement continuous learning for models to adapt over time.
- **Visualization Dashboard:** Integrate real-time dashboards using Streamlit or Grafana for monitoring.
- **Automatic Rule Generation:** Use ML to suggest new rules from frequently detected anomalies.
- **Enhancing the Known Attacks Prediction:** by Using Deep Learning Algorithms Such as: **DNN**
- **Parallel Processing:** Enhance scalability using multiprocessing or **GPU**-based model inference.

References

References

[1] Fortinet, "What is the CIA Triad and Why is it important?" Fortinet, 2023. https://www.fortinet.com/resources/cyberglossary/cia-triad

[2] P. Rawat, "Common Security Attacks in the OSI Layer Model," InfoSec Train, Jan. 25, 2023.

https://www.infosectrain.com/blog/common-security-attacks-in-the-osi-layer-model/

[3] D. Robinette, "What are the Two Main Types of Intrusion Detection Systems?" Stamus networks.com, 2023.

https://www.stamus-networks.com/blog/what-are-the-two-main-types

[4] GeeksforGeeks, "Intrusion Detection System (IDS) - GeeksforGeeks," GeeksforGeeks, Apr. 08, 2019.

https://www.geeksforgeeks.org/intrusion-detection-system-ids/

[5] S. Jelen, "Security Trails | Intrusion Detection Systems: Types, Detection Methods and Challenges," securitytrails.com, Jul. 20, 2021.

https://securitytrails.com/blog/intrusion detection-systems

[6] "Types of Intrusion Detection System (IDS)," www.paubox.com. https://www.paubox.com/blog/types-of-intrusion-detection-system-ids

[7] Wikipedia Contributors, "Man-in-the-middle attack," Wikipedia, May 17, 2019.

https://en.wikipedia.org/wiki/Man-in-the-middle_attack

[8] M. Andriani, "HTTPS Flood | Knowledge Bases | MazeBolt," Knowledge Base | MazeBolt |, Mar. 29, 2017.

https://kb.mazebolt.com/knowledgebase/https-flood/

[9] Wikipedia Contributors, "Transport Layer Security," Wikipedia, Mar. 26, 2019. https://en.wikipedia.org/wiki/Transport_Layer_Security

[10] "What Is HTTPS Spoofing? How It Works & Examples | Twingate," Twingate.com, 2024.

https://www.twingate.com/blog/glossary/https%20spoofing

[11] F5, "What is SSL/TLS Encryption?" www.f5.com, 2024. https://www.f5.com/glossary/ssl-tls-encryption

[12] Guest Blogger, "Open-Source IDS Tools: Comparing Suricata, Snort, Bro (Zeek), Linux," Levelblue.com, Jun. 2024.

https://levelblue.com/blogs/security-essentials/open source-intrusion-detection-tools-a-quick-overview

[13] "SNS Insider | Strategy and Stats," Snsinsider.com, 2024. https://www.snsinsider.com/reports/intrusion-detection-system-market-4660

[14] "Intrusion Detection and Prevention Systems Market by Solutions & Services - 2024|MarketsandMarkets," Marketsandmarkets.com, 2019. https://www.marketsandmarkets.com/Market-Reports/intrusion-detection-prevention system-market-199381457.html

[15] F. Hock and P. Kortis, "Commercial and open source-based Intrusion Detection System and Intrusion Prevention System (IDS/IPS) design for an IP networks," 2015 13th International Conference on Emerging eLearning Technologies and Applications (ICETA), Nov. 2015, doi: https://doi.org/10.1109/iceta.2015.7558466.

[16] "Usage — Scapy 2.4.4. documentation," scapy.readthedocs.io. https://scapy.readthedocs.io/en/latest/usage.html

[17] C. to, "Anomaly-based intrusion detection system," Wikipedia.org, Nov. 23, 2004. https://en.m.wikipedia.org/wiki/Anomaly-based_intrusion_detection_system

[18] "The History of Anomaly Detection - Techniques, Tools and Use Cases," ChaosGenius - Open-Source Business Observability Platform, Aug. 16, 2021.

https://www.chaosgenius.io/blog/a-brief-history-of-anomaly-detection/

[19] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly based network intrusion detection: Techniques, systems and challenges," Computers & Security, vol. 28, no. 1–2, pp. 18–28, Feb. 2009, doi: https://doi.org/10.1016/j.cose.2008.08.003.

[20] M. Bhavsar, K. Roy, J. Kelly, and Odeyomi Olusola, "Anomaly-based intrusion detection system for IoT application," Discover Internet of things, vol. 3, no. 1, May 2023, doi: https://doi.org/10.1007/s43926-023-00034-5.

[21] M. Aljanabi, M. A. Ismail, and A. H. Ali, "Intrusion Detection Systems, Issues, Challenges, and Needs," International Journal of Computational Intelligence Systems, vol. 14, no. 1, 2021, doi: https://doi.org/10.2991/ijcis.d.210105.001.

[22] S. Parhizkari, "Anomaly Detection in Intrusion Detection Systems," www.intechopen.com, Oct. 09, 2023. https://www.intechopen.com/chapters/88343

[23] A. Shaikh and P. Gupta, "Advanced Signature-Based Intrusion Detection System," *Springer*, 2023. https://www.researchgate.net/publication/362136869_Advanced_Signature-Based_Intrusion_Detection_System

[24] J. Díaz-Verdejo, et al., "On the Detection Capabilities of Signature-Based Intrusion Detection Systems," *Applied Sciences*, vol. 12, no. 2, 2022. https://www.mdpi.com/2076-3417/12/2/852

[25] N. Agarwal and S. Z. Hussain, "Identification of Flaws in the Design of Signatures for IDS," *arXiv preprint*, 2018. https://arxiv.org/abs/1805.10848

[26] Z. K. Maseer, et al., "A Meta-analysis and Systematic Review for Anomaly-Based Network Intrusion Detection Systems," *arXiv*, Aug. 2023. https://arxiv.org/abs/2308.02805

[27] A. Khraisat and A. Alazab, "A Critical Review of Intrusion Detection Systems in the Internet of Things," *Cybersecurity*, vol. 4, 2021.
https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00077-7

[28] M. Liu, et al., "RCLNet: Residual-CNN-LSTM with Attention for Anomaly Detection in IoMT," *Frontiers in Digital Health*, vol. 6, 2024.
https://www.frontiersin.org/articles/10.3389/fdgth.2024.1467241/full

[29] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)*, 2018.
https://www.unb.ca/cic/datasets/ids-2017.html