CHAPTER 1

INTRODUCTION

1.1 Basics and Background

The Internet of Things (IoT) has transformed how devices interact and communicate, enabling seamless data exchange across diverse applications, from smart homes to industrial automation. A key protocol facilitating this communication is the Message Queuing Telemetry Transport (MQTT) protocol, designed for lightweight, reliable messaging in resource-constrained environments. MQTT operates on a publish-subscribe model, where devices (publishers) send messages to a broker on specific topics, and other devices (subscribers) receive messages by subscribing to those topics. This architecture is ideal for IoT systems due to its low bandwidth requirements and ability to handle intermittent connectivity.

However, the simplicity and lightweight nature of MQTT make it a prime target for cyber-attacks. IoT devices, such as sensors and actuators, often lack robust security mechanisms, making them susceptible to threats like data injection attacks (where attackers send invalid or malicious payloads) and Distributed Denial of Service (DDoS) attacks (e.g., flooding the broker with excessive messages to disrupt service). These attacks can compromise the integrity and availability of IoT systems, leading to incorrect data processing, system downtime, or even safety hazards in critical applications.

This project aims to address these security challenges by developing an IoT Anomaly Detection Dashboard that monitors MQTT traffic, detects anomalies in real time, and prevents malicious messages from reaching the broker. The system leverages machine learning (using an Isolation Forest model) to identify anomalies based on features extracted from MQTT messages, such as payload size, inter-arrival time, and and packet rate.

The project was undertaken as a proof-of-concept for securing a small-scale IoT network, designed to be internet-free for enhanced security. In this setup, an ESP32 device acts as an MQTT publisher, collecting sensor data. This data is sent to another ESP32 configured as an Access Point (AP), which functions as the central MQTT broker. A Raspberry Pi, subscribed to the publisher ESP32 via the AP, receives the data. The Raspberry Pi hosts the deployed Isolation Forest model to flag malicious packets in

real time and also runs a Flask server to display real-time dashboards for monitoring. Furthermore, the system incorporates a prevention mechanism using iptables on the Raspberry Pi to actively block identified malicious traffic at the network level, ensuring that flagged data packets are prevented from further processing. The system successfully detects and prevents two types of anomalies: invalid payloads (indicating potential data injection attacks) and flooding attacks (indicating DDoS attempts). The dashboard offers insights into packet logs, anomaly scores, and prevented attacks, making it a valuable tool for monitoring IoT network security.

1.2 Literature Survey

1.2.1 Building a Intrusion Detection System for IoT environment using machine learning techniques [1]

This paper discusses developing an IDS specifically for IoT networks, focusing on handling vulnerabilities unique to IoT systems. The key idea involves creating a testbed IoT environment with various hardware components like the Node MCU ESP8266 and DHT11 sensors, connected to a cloud platform (e.g., ThinkSpeak), to simulate cyberattacks.

1.2.2 ADRIoT: An Edge assisted anomaly detection framework against IoT based network Attacks[2]

This paper proposes a framework designed to address the growing security challenges in IoT environments. With the proliferation of IoT devices, which often have limited security measures, the framework aims to detect and mitigate network-based attacks effectively. The framework utilizes edge computing to offload some of the heavy computational tasks from the cloud to the edge devices, reducing latency and enhancing real-time detection capabilities. Edge devices act as intermediaries between IoT devices and the cloud, enabling faster data analysis and decision-making. ADRIoT takes advantage of the collaborative power of multiple edge devices to share and process information in a decentralized manner. This allows for more effective threat detection by utilizing local insights from various edge nodes.

1.2.3 Passban IDS : Intelligent Anomaly based Intrusion Detection System for IoT Edge devices [3]

The research paper “Passban IDS: An Intelligent Anomaly-Based Intrusion Detection System for IoT Edge Devices” presents an advanced method for detecting cyber threats on IoT edge

devices. It introduces the **Passban IDS**, a system designed to identify malicious activities such as port scanning, brute force attacks (HTTP and SSH), and SYN flood attacks.
What makes this approach unique is its deployment on affordable IoT gateways (like single-board computers) using the **edge computing paradigm**, which allows real-time anomaly detection near data sources. This minimizes latency and increases the detection accuracy, while also reducing false positives.

1.3 Project Undertaken

1.3.1 Problem Statement

This project develops an internet-free anomaly detection and prevention system for IoT networks, running on a Raspberry Pi. It monitors MQTT traffic from an ESP32 publisher (via an ESP32 access point).

The system uses an Isolation Forest machine learning model to detect anomalies in real time based on features like payload size and packet rate. For prevention, it leverages iptables to automatically block malicious traffic from identified source IPs. All detections and blocking actions are logged. A Flask web dashboard provides real-time monitoring of network health, displaying live packet logs, anomaly scores, and prevented attacks. This integrated solution ensures proactive defense and comprehensive visibility for IoT network security.

1.3.2 Scope

This project encompasses the design, development, and deployment of an IoT Anomaly Detection and Prevention Dashboard, aimed at securing an **MQTT-based IoT network**. The system is designed to operate on a **Raspberry Pi**, serving as the central node for monitoring and actively filtering MQTT traffic from an **ESP32 device acting as a publisher**. The project focuses on the following key objectives:

- **Real-Time Anomaly Detection**

  Develop a **machine learning-based anomaly detection system** using an **Isolation Forest model** to identify **invalid MQTT payloads** and **flooding attacks** in real time. The system analyzes features such as payload size, inter-arrival time, payload validity, TCP flags, IP entropy, and packet rate to classify messages as normal or anomalous.

- **Real-Time Anomaly Prevention**

  Implement an **active prevention mechanism** to mitigate detected anomalies.

  - **Automated IP Blocking:** Dynamically adding **iptables** rules on the Raspberry Pi to block traffic from source IPs identified as anomalous (e.g., sending invalid payloads or initiating flooding attacks).
  - **Time-Limited Blocking:** Implementing temporary blocking durations for identified malicious IPs to allow for automated remediation and prevent permanent network isolation.
  - **Prevention Logging:** Maintaining detailed logs of all blocking and unblocking actions, including timestamps, source IPs, and reasons for the action.

- **Visualization and Monitoring**

  - Display **real-time packet logs** on the dashboard, showing details like timestamp, source/destination IP, topic, payload, anomaly score, and status (normal or anomalous).
  - Provide **charts to visualize anomaly scores over time** and the distribution of normal versus anomalous messages.
  - Show a table of **detected anomalies with timestamps, source IPs, and reasons** (e.g., invalid payload, flooding attack).
  - Include a dedicated section to display **prevention actions**, detailing which IPs were blocked, when, and for what reason.

- **System Integration**

  Integrate the anomaly detection and prevention system with an **ESP32 device publishing MQTT messages** and a **Flask-based web server** hosting the dashboard. The system ensures seamless communication between the publisher, the custom MQTT broker (running on the Raspberry Pi), and the dashboard, with logs stored in JSON files for persistence and analysis.

- **Scalability and Adaptability**

Design the system to handle **low to moderate traffic volumes**, suitable for a small-scale IoT network, with provisions for future scalability. The custom MQTT broker approach ensures adaptability to different IoT environments without relying heavily on kernel-level tools for MQTT handling.

- **Out of Scope**

  The project does not cover advanced cryptographic security mechanisms, such as MQTT message encryption, nor does it address physical-layer attacks on IoT devices. Additionally, while the system is designed to detect specific anomalies (invalid payloads and flooding attacks), it may require retraining the machine learning model to detect other types of attacks, such as MQTT topic spoofing or man-in-the-middle attacks.

1.3.3 Organization of Project Report

Chapter 1 gives the Introduction to Background and Basics of Instant Messaging, Literature Survey conducted, about Project to be undertaken, Problem Definition and Scope of the Project.

Chapter 2 gives an overview of Project Planning and Management. It states the detailed Functional Requirements, Non Functional Requirements, Deployment Requirements, External Interface Requirements and Other Requirements. This chapter also specifies the Project Process Model used to develop this Project, Cost and Effort estimates and Project Scheduling.

Chapter 3 gives the Analysis Study and Designing of the Project. It consists of IDEA Matrix, Mathematical Model, Feasibility Analysis and all UML diagrams specifying how to build the required system.

Chapter 4 gives the details about implementation and coding. It consists of Operational Details, Major Classes, Code Listings for each module and Screenshots.

Chapter 5 gives the test cases identified to perform at final stage Test cases for Unit Testing, test cases for integration testing and test cases for acceptance testing.

Chapter 6 consists of the Results obtained and Discussions.

Chapter 7 gives the Conclusion.

Chapter 8 Analyses the Future Work of the project**.**

CHAPTER 2

PROJECT PLANNING AND MANAGEMENT

2.1 Introduction

This chapter covers the project planning and management details. It also covers System Requirement specifications. SRS is considered as the base for the effort estimations and project scheduling.

2.2 Detailed System Requirement Specification

2.2.1 System Overview

The IoT Anomaly Prevention Dashboard is designed to monitor MQTT traffic in a small-scale IoT network, identifying anomalous behavior in real-time. The system comprises the following components:

- **ESP32 Access Point (AP)**:
    - Hardware: ESP32 microcontroller.
    - Role: Creates a local Wi-Fi network (SSID: “ESP32_AP”, IP: 192.168.4.x) for other devices to connect.
    - Firmware: Configured using the Arduino IDE to set up the AP and assign static IPs to connected devices.
- **ESP32 Publisher**:
    - Hardware: ESP32 microcontroller.
    - Role: Connects to the ESP32 AP and publishes sensor data to the MQTT broker.
    - Firmware: publisher_test.ino, written in Arduino C++, uses the PubSubClient library to publish MQTT messages to topics like esp32/temperature, esp32/humidity, etc.
    - Traffic Types:

- Normal: Sensor data within valid ranges (e.g., temperature: 20–35°C), published every 2.75 seconds (0.36 packets/second).
- Invalid Payloads: 10% chance of publishing an invalid payload (e.g., 1hack).
- Flooding Attack: Publishes at 2.86 packets/second for 1–3 cycles (4 messages every 1.4 seconds).

- **Raspberry Pi (Anomaly Prevention and Dashboard Host)**:
  - Hardware: Raspberry Pi 4 with 4GB RAM, running Raspberry Pi OS (64-bit).
  - Role: Hosts the MQTT broker, anomaly prevention system, and Flask dashboard.
  - Software Components:
    - **MQTT Broker**: Mosquitto, running on 192.168.4.4:1883, facilitates communication between the ESP32 publisher and the detection system.
    - **Anomaly Prevention Script (inference.py)**: Uses Scapy to sniff MQTT packets on wlan0, extracts features, and applies an Isolation Forest model to detect anomalies. Logs results to JSON files. Additionally, it implements a prevention mechanism using **iptables** to automatically block source IPs identified as anomalous, logging these actions and managing block durations.
    - **Flask Dashboard (server.py)**: Hosts a web server at http://192.168.4.4:5000, serving index.html and charts.js for visualization.
    - **Model Files**: anomaly_model.pkl (Isolation Forest model) and scaler.pkl (feature scaler), pre-trained on normal traffic data.
- **Flask Dashboard**:
  - Role: Visualizes detection and prevention results in real-time.
  - Features:

- Line chart: Anomaly scores over time.
- Pie chart: Distribution of normal vs. anomalous messages.
- False positives counter: Tracks messages with low scores but labeled as normal.
- Anomaly table: Lists detected anomalies with timestamps, source IPs, and reasons.
- Packet log table: Shows all packets with details (timestamp, source/destination, topic, payload, score, etc.).

The system focuses on detection and prevention as well as anomalies are flagged, logged, and displayed on the dashboard for user analysis making it a robust system.

2.2.2 Functional Requirements:

The features described below are organized based on their context of functionality. All the features have equal priority.

The system must fulfil the following functional requirements to achieve its objectives:

2.2.2.1. ESP32 Access Point Setup:

- The ESP32 AP must create a Wi-Fi network (SSID: "ESP32_AP", password: "12345678") and assign static IPs to connected devices (e.g., publisher: 192.168.4.2, Raspberry Pi: 192.168.4.4).
- It must maintain a stable connection for the publisher and Raspberry Pi during operation.

2.2.2.2 ESP32 Publisher Operation:

- The publisher must connect to the ESP32 AP and the MQTT broker at 192.168.4.4:1883.
- It must publish simulated sensor data to topics esp32/temperature, esp32/humidity, esp32/pressure, and esp32/altitude.
- It must simulate three traffic patterns:
  - Normal: Valid data (e.g., 1esp32/temperature25.0) every 2.75 seconds.

- Invalid Payloads: 10% chance of invalid data (e.g., 1hack).
- Flooding Attack: 2.86 packets/second for 1–3 cycles.

2.2.2.3 MQTT Broker:

- The Mosquitto broker must run on the Raspberry Pi, listening on port 1883.
- It must handle MQTT messages from the publisher without authentication or encryption (for simplicity).

2.2.2.4 Packet Sniffing:

- The anomaly detection script (inference.py) must use Scapy to sniff TCP packets on port 1883 via the wlan0 interface.
- It must parse MQTT packets to extract the topic and payload.

2.2.2.5 Feature Extraction:

- The system must extract the following features from each MQTT message:
  - **Payload Size**: Length of the payload in bytes.
  - **Inter-Arrival Time**: Time difference between consecutive packets.
  - **Payload Validity**: Binary (1.0 if valid, 0.0 if invalid), based on predefined ranges (temperature: 20–35°C, humidity: 30–80%, pressure: 980–1050 hPa, altitude: 50–1500 m).
  - **TCP Flags**: Encoded TCP flags (though minimal in this setup, as MQTT uses TCP).
  - **IP Entropy**: Shannon entropy of source IPs in a sliding window (size: 10).
  - **Packet Rate**: Number of packets per 0.5-second window, scaled by 5.0.

2.2.2.6 Anomaly Detection:

- The system must use a pre-trained Isolation Forest model (anomaly_model.pkl) to classify messages as normal (score > -0.01) or anomalous (score ≤ -0.01).
- It must flag invalid payloads as anomalies (label: "🔴 Anomaly (Invalid Payload)").
- It must detect flooding attacks if the packet rate exceeds 2 packets/second (label: "🔴 Anomaly (Flooding Attack)").

- All packets (normal and anomalous) must be logged to mqtt_inference_packets.json with details: timestamp, source/destination, topic, payload, score, payload length, inter-arrival time, packet rate, and anomaly status.

2.2.2.7 Anomaly Prevention:

When a message is flagged as an anomaly, the system immediately triggers a prevention mechanism to neutralize the threat. For any packet identified as malicious—either due to an invalid payload or being part of a flooding attack—the system will block the source IP address using iptables rules. This block is temporary, lasting for a configurable duration (300 seconds in the current system), and is designed to mitigate the immediate threat without permanently disrupting network operations. All prevention actions, including the IP blocked, the reason, and the time of the block, are meticulously logged to a separate blocked_traffic_log.txt file, providing a clear audit trail of all security interventions. This proactive approach ensures that the IoT network is not only monitored for threats but also actively defended against them in real time.

2.2.2.8 Dashboard Visualization:

- The Flask dashboard must display:
    - A line chart of anomaly scores for the last 50 packets.
    - A pie chart showing the distribution of normal vs. anomalous messages.
    - A counter for false positives (messages with score ≤ -0.01 but labeled normal).
    - A table of detected anomalies (last 50), showing timestamp, source IP, and reason.
    - A packet log table (last 50 packets), showing timestamp, source/destination, topic, payload, score, payload length, inter-arrival time, packet rate, and status (normal/anomalous).
- The dashboard must refresh every 3 second to provide real-time updates.

2.2.3 Non-Functional Requirements:

2.2.3.1 **Performance**:

- The anomaly prevention script must process packets in real-time, with a latency of less than 100ms per packet, to ensure timely detection and prevention even during flooding attacks (2.86 packets/second).
- The Flask dashboard must load within 3 seconds and refresh data every 1 second without noticeable lag.

2.2.3.2 **Reliability**:

- The system must operate continuously for at least 24 hours without crashing.
- The ESP32 AP and publisher must maintain a stable Wi-Fi connection, with reconnection logic in case of disconnections.

2.2.3.3 **Scalability**:

- The system is designed for a single publisher but should handle up to 5 publishers without significant performance degradation (though not implemented in this phase).

2.2.3.4 **Usability**:

- The dashboard must be user-friendly, with a clean interface using Tailwind CSS for styling and Chart.js for visualizations.
- Logs must be stored in JSON format for easy parsing and analysis.

2.2.3.5 **Maintainability**:

- The codebase must be well-documented with comments explaining key functions and logic.
- Scripts must be modular, allowing easy updates to feature extraction or model retraining.

2.2.3.6 **Security**:

- While security mechanisms like MQTT authentication or TLS are not implemented, the system must not introduce additional vulnerabilities (e.g., open ports beyond 1883 and 5000).

2.2.3.7 **Resource Constraints**:

- The anomaly prevention script must use less than 1GB of RAM on the Raspberry Pi to leave resources for the Flask server and Mosquitto.
- Log files must be written in batches (every 10 packets) to minimize disk I/O overhead.

2.2.4 Deployment Requirements:

2.2.4.1. Hardware Requirements

- **ESP32 Microcontrollers (x2)**:
  - ESP32 DevKit v1 with 4MB flash memory.
  - USB cables for programming and power.
- **Raspberry Pi 4**:
  - 4GB RAM, 32GB microSD card with Raspberry Pi OS (64-bit).
  - Wi-Fi module (built-in) for connecting to the ESP32 AP.
  - Power adapter (5V, 3A).
- **Development PC**:
  - Used for programming the ESP32 and Raspberry Pi.
  - Minimum: 8GB RAM, 500GB storage, running Windows/Linux/macOS.

2.2.4.2 Software Requirements

- **ESP32**:
  - Arduino IDE 2.3.2 with ESP32 board support package.
  - Libraries: WiFi, PubSubClient.
- **Raspberry Pi**:
  - Raspberry Pi OS (64-bit), Python 3.9+.
  - Mosquitto MQTT broker (version 2.0.15).
  - Python libraries: Scapy, NumPy, Scikit-learn, Flask, Pandas, Pickle, Math, Subprocess, Time, Json, Os, Re, Collections.
  - Web dependencies: Tailwind CSS (via CDN), Chart.js (via CDN).
- **Development Tools**:

- Visual Studio Code and Arduino IDE for coding and debugging.

2.2.5 Project Scheduling

2.2.5.1 Timeline Chart

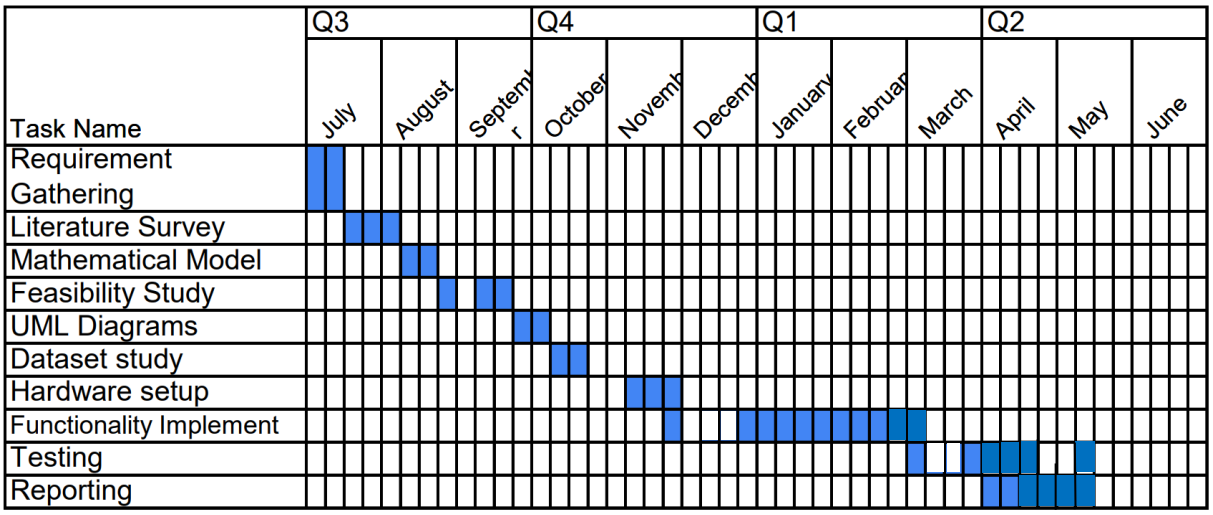

Figure 2.1: Gantt Chart

CHAPTER 3
ANALYSIS AND DESIGN

3.1 Introduction

This chapter covers the analysis and design of the considered system.

3.2 IDEA Matrix

Table 3.1: I Elements

I	USE
Improve	Improve the accuracy of anomaly detection through advanced algorithms
Impact	Aim for significant reductions in false positives and negatives.
Input	Use diverse sensor data to enhance detection capabilities.
Innovation	Explore novel techniques in machine learning or AI for anomaly detection followed by prevention.

Table 3.2: D Elements

D	USE
Deliver	Provide real-time alerts for detected anomalies
Decrease	Reduce latency of data processing and response times
Detect	Implement a robust system to detect and prevent anomaly

Table 3.3: E Elements

E	USE
Eliminate	Remove redundant data processing steps to streamline operations
Evaluate	Regularly assess the system's performance and make necessary adjustments.
Enhance	Improve user interfaces for monitoring and managing the system.

Table 3.4: A Elements

A	USE
Avoid	Minimize risks of false alerts that could lead to unnecessary actions.
Advantage	Leverage edge devices to process data locally for faster responses and reduced latency.

3.3 Architecture Diagram

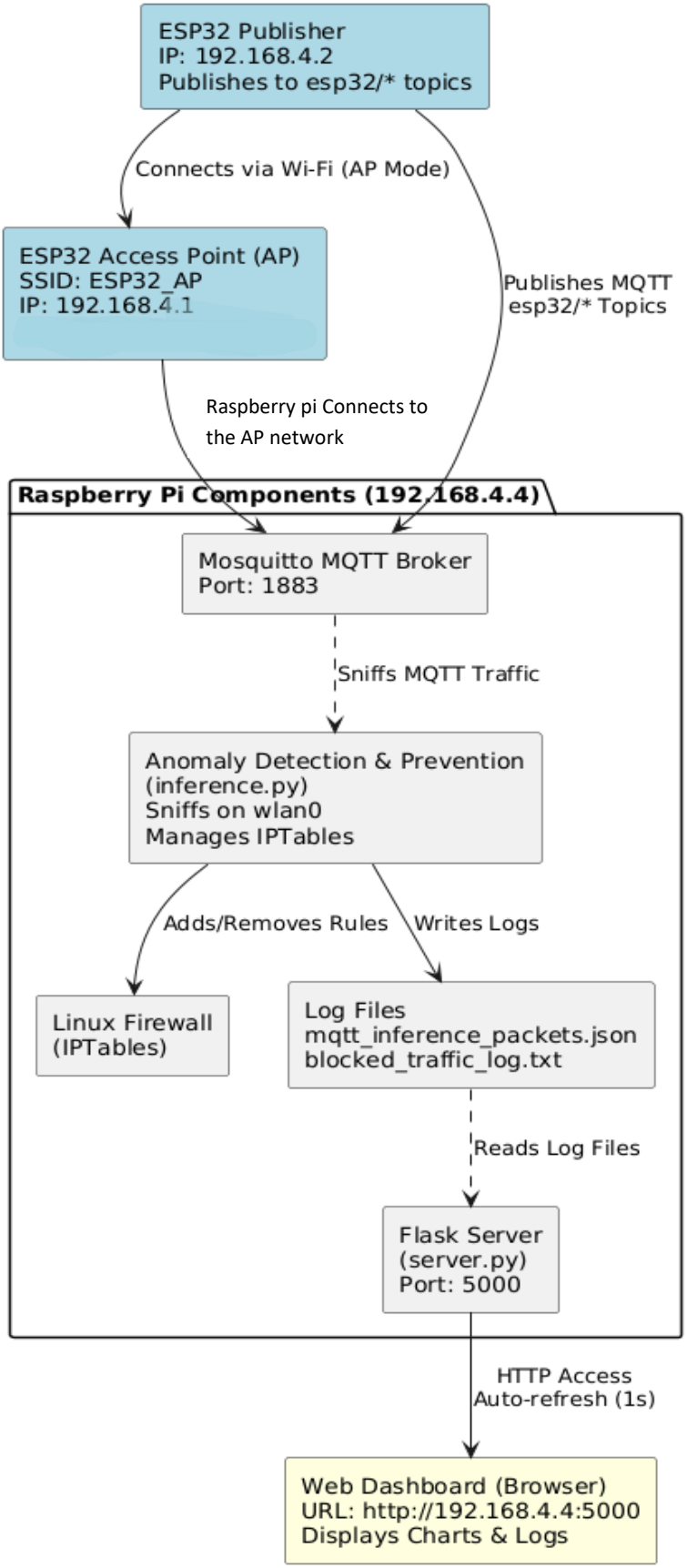

Figure 3.1: Architecture Diagram

Description: The architecture diagram illustrates an IoT security monitoring system using ESP32 devices and a Raspberry Pi to detect and prevent anomalies in MQTT traffic. An ESP32 Publisher device connects to an ESP32 Access Point (AP) via Wi-Fi and publishes MQTT messages to specific topics. The AP forwards these messages to the Mosquitto MQTT Broker running on the Raspberry Pi through a NAT setup. The broker receives and routes the MQTT traffic, which is then intercepted by a custom anomaly detection and prevention module. This module uses `iptables` and `NFQUEUE` to inspect packets, identify suspicious activity, and apply blocking rules via the system firewall (UFW). Detected anomalies and traffic events are logged to specific log files. These logs are read by a Flask web server running on the Raspberry Pi, which serves a web dashboard accessible via a browser. The dashboard provides visual insights into MQTT traffic, including blocked topics and anomaly statistics, enabling real-time monitoring and analysis of IoT data at the network edge.

3.4 UML Diagrams

1. Use case diagram:

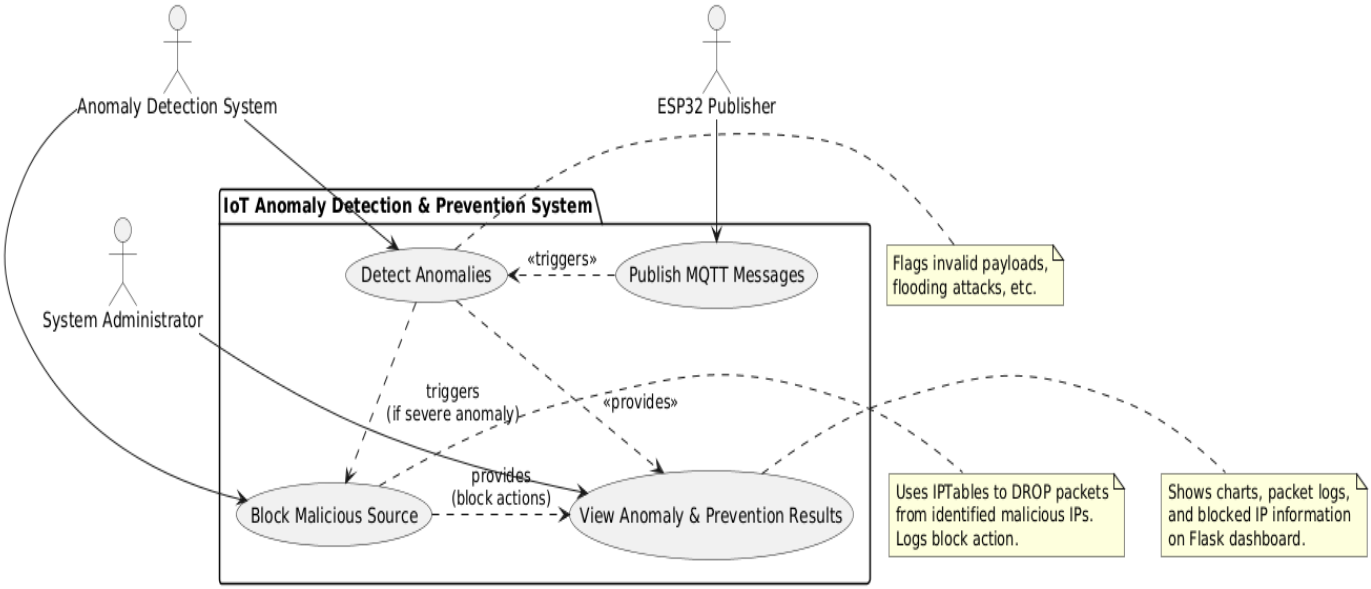

Figure 3.2: Use Case Diagram

Description: The use case diagram shows an IoT Anomaly Detection and Prevention System that monitors MQTT messages from an ESP32 Publisher to detect threats like invalid payloads and flooding attacks. Upon detecting severe anomalies, it uses iptables to block malicious IPs and logs these actions. A System Administrator can view logs and trigger manual blocks, while a Flask dashboard provides visual insights into anomalies and blocked sources for easy monitoring.

2. Sequence Diagram:

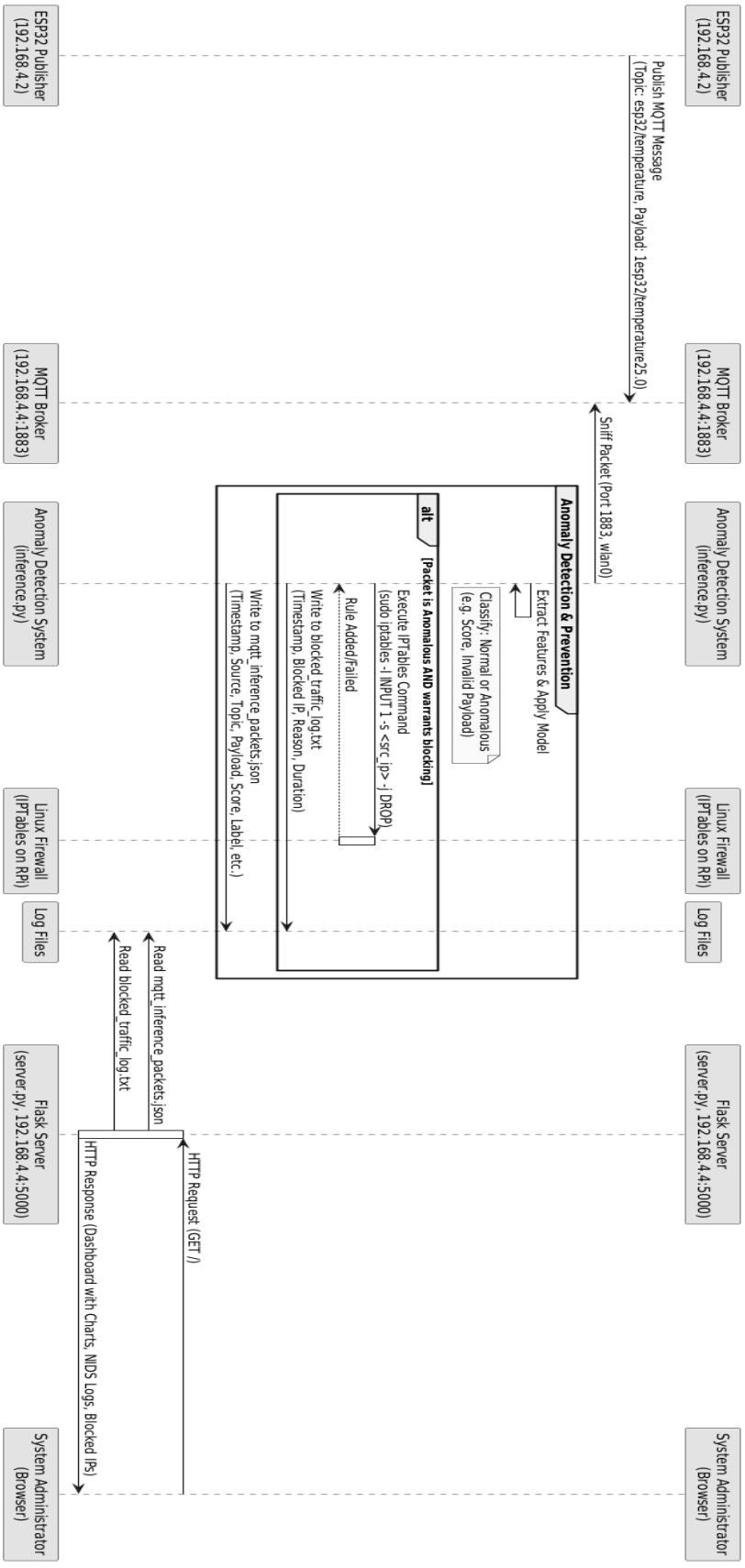

Figure 3.3: Sequence Diagram

Description: The sequence diagram outlines the process of detecting and preventing anomalies in MQTT traffic within an IoT network. The ESP32 Publisher sends MQTT messages (e.g., temperature data) to the MQTT Broker on the Raspberry Pi. The Anomaly Detection & Prevention system intercepts these packets, extracts features, and applies a classification model to determine if the payload is normal or anomalous. If the packet is deemed malicious, the system triggers iptables to drop it by adding a rule (e.g., using `iptables -A INPUT -s <IP> -j DROP`). It logs the blocked packet details, including the source IP, topic, score, and label, into JSON and text log files. These logs are accessed by a Flask server, which presents the information as charts and summaries on a web dashboard. The System Administrator can then view these results through a browser interface for monitoring and analysis.

3. Activity Diagram:

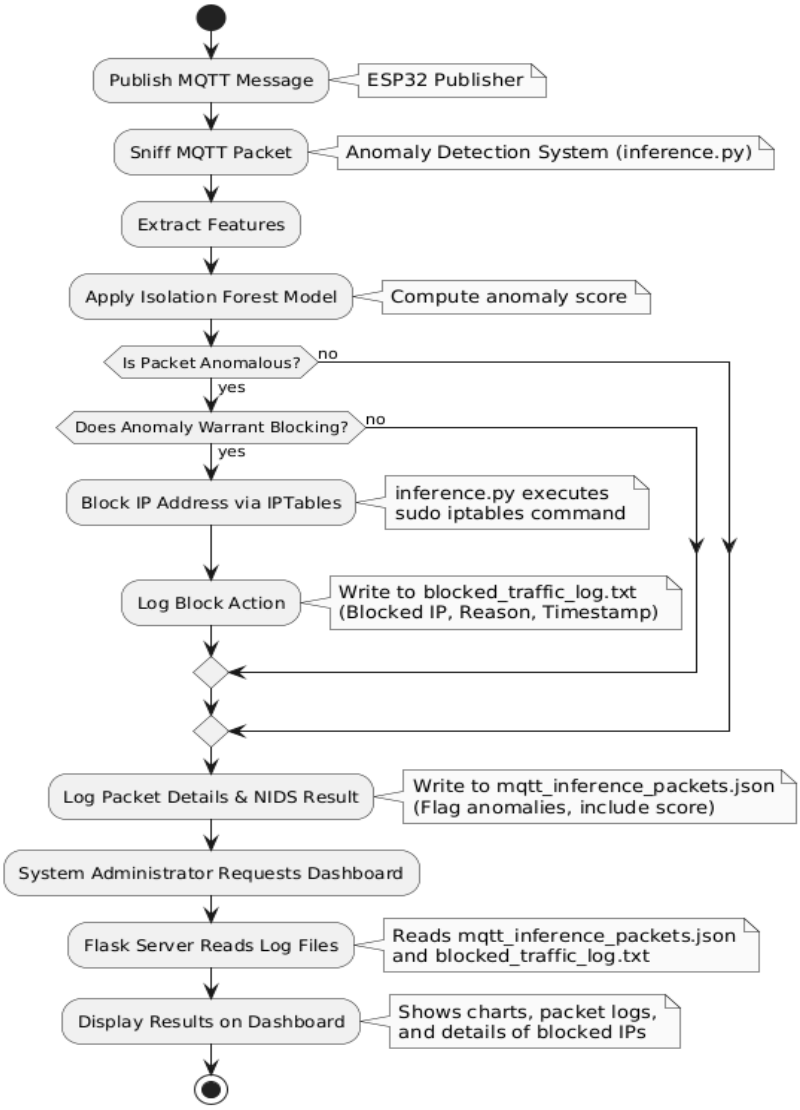

Figure 3.4: Activity Diagram

Description: This activity diagram illustrates an anomaly detection system for MQTT traffic. It involves sniffing MQTT packets, extracting features, and using an Isolation Forest Model to identify anomalies. If a packet is deemed anomalous and warrants blocking, its IP address is blocked via IPTables, and the action is logged. All packet details and NIDS results are logged. A Flask server then reads these logs to display comprehensive results, including blocked IPs, on a dashboard for system administrators.

4. Deployment diagram:

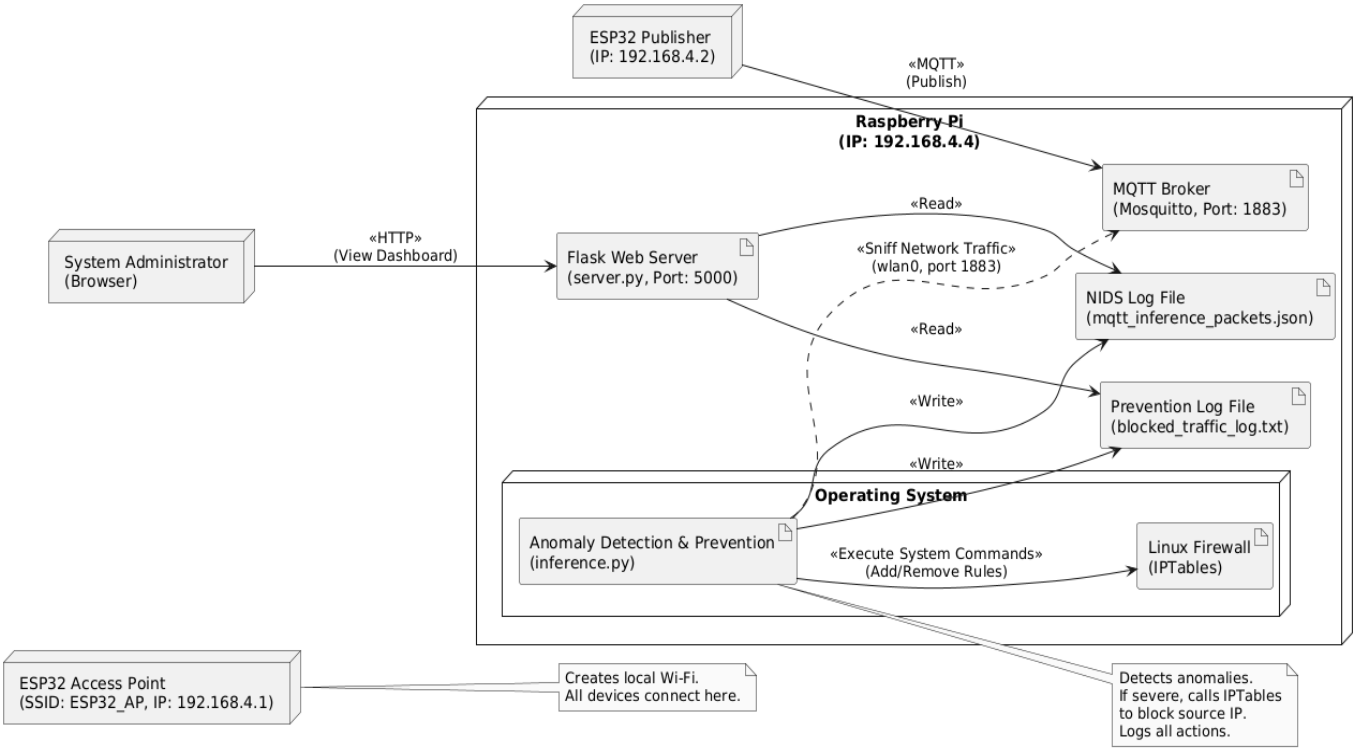

Figure 3.5: Deployment Diagram

Description: This deployment diagram depicts an MQTT-based anomaly detection and prevention system centered on a Raspberry Pi. An ESP32 Access Point creates a local Wi-Fi network for devices like an ESP32 Publisher. MQTT messages from the publisher go to a Mosquitto broker on the Raspberry Pi. An inference.py script on the Pi sniffs this traffic, detects anomalies, and, if severe, uses IPTables to block the source IP. All anomaly detections and blocking actions are logged to mqtt_inference_packets.json and blocked_traffic_log.txt respectively. A Flask web server on the same Raspberry Pi reads

these logs to provide a dashboard, accessible via a browser, for system administrators to monitor the system's performance and review security events.

3.5 Algorithms

What is Isolation Forest?

Isolation Forest is a machine learning algorithm for anomaly detection. It works by isolating data points in a dataset using random splits, based on the idea that anomalies are rare and differ significantly from normal data, so they can be isolated with fewer splits.

- **How It Works**:
  - It builds multiple random decision trees (a "forest").
  - For each tree, it randomly selects a feature and a split value to partition the data.
  - This process repeats recursively, creating a tree where each leaf node contains a single data point.
  - Anomalies are isolated faster (closer to the root) because they’re more distinct, while normal points take more splits (deeper in the tree).
  - The anomaly score for a data point is based on its average path length across all trees: shorter paths indicate anomalies.
- **Key Concept**:
  - Normal data points are clustered and require more splits to isolate.
  - Anomalies are outliers and require fewer splits.

Why Isolation Forest for Your Project?

The project detects anomalies in MQTT messages (e.g., invalid payloads, flooding attacks with packet rates > 2 packets/sec).

Isolation Forest is a great fit for several reasons:

- **Handles High-Dimensional Data**:
  - You extract multiple features from MQTT packets (payload size, inter-arrival time, payload validity, IP entropy, packet rate). Isolation Forest efficiently

handles this multi-dimensional feature space without needing dimensionality reduction.

- **No Assumption on Data Distribution**:
  - MQTT traffic patterns can vary widely (e.g., normal sensor data vs. sudden flooding). Isolation Forest doesn't assume a specific distribution, making it robust for your IoT setup.
- **Scales Well**:
  - It's computationally efficient (sub-linear time complexity) since it doesn't need to compute distances between points, unlike other methods like k-NN. This is crucial for real-time detection in your system.
- **Unsupervised Learning**:
  - You don't have labeled data for anomalies. Isolation Forest works unsupervised, learning to isolate outliers based on the data itself, which fits your need to detect unknown anomalies.
- **Anomaly Scoring**:
  - It provides a score (normalized path length) that you use to flag anomalies (score $\leq$ -0.01). This score is easy to interpret and integrate into your dashboard for visualization.

Compared to alternatives:

- **K-Means Clustering**: Requires knowing the number of clusters and assumes spherical clusters, which doesn't suit varied MQTT traffic.
- **One-Class SVM**: Can be slower and less interpretable for real-time use.
- **Autoencoders**: Need more data and training time, which might not fit the lightweight setup on a Raspberry Pi.

How Isolation Forest is Implemented

In the project, the Isolation Forest is part of the Anomaly Detection System (inference.py) on the Raspberry Pi (192.168.4.4):

1. Real-Time Training on Normal ESP32 Data:

- The ESP32 Publisher (192.168.4.2) sends normal MQTT messages (e.g., topic: esp32/temperature, payload: 1esp32/temperature25.0) to the MQTT Broker (192.168.4.4:1883).
- The Anomaly Detection System sniffs these packets on wlan0 using Scapy and extracts features:
  - Payload size (e.g., length of 1esp32/temperature25.0).
  - Inter-arrival time (time since the last packet).
  - Payload validity (e.g., regex check for format).
  - IP entropy (randomness in source IPs).
  - Packet rate (packets per second from the source).
- These normal packets are collected over an initial period (e.g., first 1000 packets or 10 minutes of stable operation) to form a training set of normal behavior.
- The Isolation Forest model is trained on this live data:
  - Number of trees: e.g., 100.
  - Subsampling size: e.g., 256 points per tree.
  - Contamination: Set to 0 (since you're training on normal data only), but you use a threshold (≤ -0.01) for anomaly detection.

2. Real-Time Anomaly Detection and Prevention:
   - After training, for each new sniffed packet:
     - Extract features into a vector (e.g., [payload_size, inter_arrival_time, payload_validity, ip_entropy, packet_rate]).
     - Feed the vector into the Isolation Forest model.
     - Get an anomaly score: > -0.01 is normal (similar to training data), ≤ -0.01 is anomalous (deviates from normal ESP32 behavior).
     - Apply rule-based checks for additional validation (e.g., flag 1hack as invalid payload, packet rate > 2 packets/sec as flooding).
     - Once a packet is classified as anomalous, the system immediately triggers its prevention mechanism. For any packet flagged as malicious—whether due to a low anomaly score, an invalid payload (like "1hack"), or an excessive packet rate (indicating flooding)—the system will block the source IP address using iptables. This block is temporary, set for a specific duration, and is logged along with the

reason for the action in a dedicated file, ensuring proactive defense and a clear audit trail.

3. Logging Results:
    - Log each packet to mqtt_inference_packets.json with:
        - Timestamp, source IP, topic, payload, score, and status (e.g., “Normal”, “Anomaly: Invalid Payload”).
        - Example for normal: { "timestamp": "2025-06-04T22:50:00", "source": "192.168.4.2", "topic": "esp32/temperature", "payload": "1esp32/temperature25.0", "score": 0.05, "status": "Normal" }.
        - Example for anomaly: { "timestamp": "2025-06-04T22:50:01", "source": "192.168.4.2", "topic": "esp32/temperature", "payload": "1hack", "score": -0.03, "status": "Anomaly: Invalid Payload" }.
4. Dashboard Integration:
    - The Flask Server (server.py) reads the log and displays:
        - Charts of anomaly scores over time.
        - Tables with packet details and status (e.g., “Normal”, “Anomaly: Flooding Attack”).

CHAPTER 4

IMPLEMENTATION AND CODING

4.1 Introduction

This chapter covers the role of various subsystems/modules along with implementation details listing of the code for the major functionalities.

4.2 Operational Details

This section outlines the operational details of the IoT Anomaly Detection and Prevention Dashboard System, describing the system's modules and the major classes involved in the implementation. The system is designed to detect anomalies in MQTT messages in real-time, prevent the anomalies, log all packets with anomaly flags, and visualize results on a web dashboard.

4.2.1 Modules

The system is divided into several modules, each handling a specific functionality. These modules interact to achieve real-time anomaly detection, logging, and visualization. Below is a detailed description of each module:

1. **ESP32 Publisher Module:**
    - Purpose: Simulates an IoT device by publishing MQTT messages containing sensor data (e.g., temperature, humidity) to the MQTT Broker.
    - Implementation:
        - Deployed on an ESP32 microcontroller (IP: 192.168.4.2).
        - Uses the ESP32's Wi-Fi capabilities to connect to the ESP32 Access Point (SSID: ESP32_AP, IP: 192.168.4.1).
        - Utilizes the PubSubClient library in Arduino IDE to establish an MQTT connection with the Mosquitto broker (192.168.4.4:1883).
        - Publishes messages to topics like esp32/temperature with payloads such as 1esp32/temperature25.0.

- Capable of simulating anomalies for testing (e.g., invalid payloads like 1hack, flooding with > 2 packets/sec).

- Operation:
  - On startup, connects to the Wi-Fi network and MQTT Broker.
  - Publishes messages at a configurable rate (default: 1 packet/sec for normal operation).
  - For flooding tests, increases the rate (e.g., 5 packets/sec).

2. **MQTT Broker Module:**
   - Purpose: Facilitates MQTT communication by receiving messages from the ESP32 Publisher and making them available for sniffing.
   - Implementation:
     - Uses Mosquitto, an open-source MQTT broker, installed on the Raspberry Pi (192.168.4.4).
     - Configured to listen on port 1883 with default settings (no authentication for simplicity in this local setup).
     - Runs as a background service (mosquitto -d).
   - Operation:
     - Receives MQTT messages from the ESP32 Publisher.
     - Forwards messages to any subscribers (though no subscribers are used in this detection-only setup).
     - Messages are sniffed directly from the network by the Anomaly Detection Module**.**
3. **Anomaly Detection and Prevention Module:**
   - Purpose: Sniffs MQTT packets, extracts features, trains an Isolation Forest model on normal data, detects anomalies, actively prevents them and logs results.
   - Implementation:
     - Implemented in Python (inference.py) on the Raspberry Pi.
     - Uses Scapy to sniff packets on the wlan0 interface, filtering for MQTT traffic (port 1883).
     - Uses scikit-learn for the Isolation Forest model.
     - Features extracted per packet:
       - Payload size: Length of the MQTT payload.

        - Inter-arrival time: Time difference between consecutive packets from the same source.
        - Payload validity: Regex check (e.g., 1hack fails).
        - IP entropy: Shannon entropy of source IPs.
        - Packet rate: Packets per second from the same source.
    - Trained on normal ESP32 data (e.g., first 1000 packets) with Isolation Forest parameters: 100 trees, subsample size of 256, contamination set to 0.
    - Anomaly threshold: Score ≤ -0.01.
    - Rule-based checks: Invalid payloads (e.g., 1hack), packet rate > 2 packets/sec (flooding).
    - Utilizes **subprocess calls to interact with iptables** to manage firewall rules for prevention.
    - Maintains an internal record of currently blocked IP addresses, including their block time and reason.
    - Logs all blocking and unblocking actions to `blocked_traffic_log.txt`.
- Operation:
    - Initially collects normal packets for training.
    - Trains the Isolation Forest model on these packets.
    - For each new packet:
        - Extracts features into a vector.
        - Computes anomaly score using the model.
        - Applies rule-based checks.
        - Logs packet details to mqtt_inference_packets.json with status (e.g., “Normal”, “Anomaly: Invalid Payload”).
        - Prevention triggered instantly when the Anomaly Detection Module flags a packet as anomalous (score ≤ -0.01, invalid payload, or flooding attack).
        - **Adds an `iptables DROP` rule** to the `INPUT` chain, blocking all TCP traffic on port 1883 from the malicious source IP.
        - Blocks are **temporary**, configured for a specific duration (e.g., 300 seconds/5 minutes).

- **Periodically checks for and automatically removes** `iptables` rules for IPs whose block duration has expired.
- Logs detailed entries for both **blocking** and **unblocking** events, providing an audit trail of active defence measures.

4. **Logging Module:**
   - Purpose: Manages the storage of packet logs with anomaly flags.
   - Implementation:
     - Uses Python's json library within inference.py to write to mqtt_inference_packets.json.
     - Log entries are appended in JSON Lines format (one JSON object per line).
     - Each entry includes: timestamp, source IP, topic, payload, score, and status.
   - Operation:
     - Appends a new entry for every packet processed by the Anomaly Detection Module.
     - Ensures thread-safe file access using file locking mechanisms.
     - Example entry:

{"timestamp":"2025-06-05T13:03:00", "source":"192.168.4.2", "topic": "esp32/temperature", "payload": "1esp32/temperature25.0", "score": 0.05, "status": "Normal" }

5. **Flask Server Module:**
   - Purpose: Hosts a web dashboard to visualize packet logs and anomaly scores.
   - Implementation:
     - Implemented in Python (server.py) using the Flask framework.
     - Runs on the Raspberry Pi at http://192.168.4.4:5000.
     - Reads mqtt_inference_packets.json to fetch packet logs.
     - Uses Flask templates to render an HTML dashboard with:
       - A line chart for anomaly scores over time (using Chart.js).
       - A table listing packets (timestamp, source, topic, payload, status).
   - Operation:
     - Starts a web server on port 5000.

- On HTTP GET requests to /, reads the log file and renders the dashboard.
- Updates the dashboard every second via JavaScript (AJAX calls to refresh data).

6. **Dashboard Client Module:**
   - Purpose: Provides the System Administrator with a browser-based interface to monitor anomalies.
   - Implementation:
     - A web client accessed via a browser at http://192.168.4.4:5000.
     - Uses HTML, CSS (Bootstrap for styling), and JavaScript (Chart.js for charts, AJAX for updates).
   - Operation:
     - Displays a chart of anomaly scores over time.
     - Shows a table with packet details (e.g., “Normal”, “Anomaly: Flooding Attack”).
     - Refreshes automatically every second to show real-time data.

4.2.2 Code Listing

Listing 4.1: Code snippet for Training the model

```
def calculate_ip_entropy():
    if not ip_window:
        return 0.0
    ip_counts = {}
    for ip in ip_window:
        ip_counts[ip] = ip_counts.get(ip, 0) + 1
    total = len(ip_window)
    entropy = 0.0
    for count in ip_counts.values():
        prob = count / total
        entropy -= prob * log2(prob)
    return entropy
```

```
def calculate_packet_rate(current_time):
    if not packet_timestamps:
        return 0.0
     while packet_timestamps and current_time - packet_timestamps[0] > 1:  # 1-second window
        packet_timestamps.popleft()
    return len(packet_timestamps) / 1.0  # Rate in packets per second
```

```
def encode_tcp_flags(tcp_layer):
    if not tcp_layer:
        return 0
    flags = tcp_layer.flags
    return int(flags)
```

```
def is_valid_payload(payload, topic):
    try:
        if not payload.startswith('1'):
            return False
        rest = payload[1:]
        if not rest.startswith(topic):
            return False
        value_str = rest[len(topic):]
        value = float(value_str)
        if topic == "esp32/temperature" and not (20.0 <= value <= 35.0):
            return False
        elif topic == "esp32/humidity" and not (30.0 <= value <= 80.0):
            return False
        elif topic == "esp32/pressure" and not (980.0 <= value <= 1050.0):
            return False
        elif topic == "esp32/altitude" and not (50.0 <= value <= 1500.0):
            return False
        return True
    except (ValueError, IndexError):
        return False
```

```
def extract_features(packet, payload, topic):
    global last_packet_time
    current_time = time.time()
    inter_arrival = current_time - last_packet_time
    last_packet_time = current_time

    payload_size = len(payload)
    is_valid = 1.0 if is_valid_payload(payload, topic) else 0.0

    tcp_flags = encode_tcp_flags(packet[TCP]) if packet.haslayer(TCP) else 0
    ip_window.append(packet[IP].src if packet.haslayer(IP) else "0.0.0.0")
    ip_entropy = calculate_ip_entropy()
    packet_timestamps.append(current_time)
    packet_rate = calculate_packet_rate(current_time)

    # Emphasize packet_rate by scaling it more aggressively
    packet_rate_scaled = packet_rate * 5.0  # Increased scaling factor to 5.0

    return [payload_size, inter_arrival, is_valid, tcp_flags, ip_entropy, packet_rate_scaled]
```

```
def parse_mqtt_publish(payload):
    if len(payload) < 2:
        return None, None, 0

    message_type = (payload[0] >> 4) & 0xF
    if message_type != 3:
        return None, None, 0

    remaining_length, pos = decode_remaining_length(payload, 1)
    if remaining_length is None or pos + remaining_length > len(payload):
        return None, None, 0

    if pos + 2 > len(payload):
        return None, None, 0
    topic_length = (payload[pos] << 8) + payload[pos + 1]
    pos += 2

    if pos + topic_length > len(payload):
        return None, None, 0
    topic = payload[pos:pos + topic_length].decode('utf-8', errors='ignore')
    pos += topic_length

    payload_length = remaining_length - (2 + topic_length)
    if payload_length <= 0 or pos + payload_length > len(payload):
        return None, None, 0
    message_payload = payload[pos:pos + payload_length].decode('utf-8', errors='ignore')
    pos += payload_length

    return topic, message_payload, pos
```

```
def parse_mqtt_packet(data):
    messages = []
    pos = 0
    while pos < len(data):
        topic, payload, new_pos = parse_mqtt_publish(data[pos:])
        if topic is None or payload is None:
            break
        messages.append((topic, payload))
        pos += new_pos
    return messages
```

```
def handle_packet(packet):
    if packet.haslayer(IP) and packet.haslayer(TCP) and packet[TCP].dport == 1883:
        raw_payload = bytes(packet[TCP].payload)
        messages = parse_mqtt_packet(raw_payload)

        for topic, payload in messages:
            if not topic.startswith("esp32/"):
                continue

            features = extract_features(packet, payload, topic)
            packet_data.append(features)

            timestamp = time.strftime('%Y-%m-%d %H:%M:%S')
             print(f"[{timestamp}] {packet[IP].src} -> {packet[IP].dst} | Topic: {topic} | Payload Length: {features[0]} | Inter-Arrival: {features[1]:.2f} | Is Valid: {features[2]} | TCP Flags: {features[3]} | IP Entropy: {features[4]:.2f} | Packet Rate: {features[5]:.2f} | Payload: {payload}")

            log = {
                "timestamp": timestamp,
                "src": str(packet[IP].src),
                "dst": str(packet[IP].dst),
                "topic": topic,
                "payload_length": float(features[0]),
                "inter_arrival": float(features[1]),
                "payload": payload
            }
            with open(LOG_FILE, "a") as f:
                f.write(json.dumps(log) + "\n")
```

```
    X = np.array(packet_data)
    print(f"Collected {len(X)} packets")

    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    model = IsolationForest(
        n_estimators=200,
        max_samples='auto',
```

```
    contamination=0.05,  # Increased to make the model more aggressive
    max_features=1,
    random_state=42,
    n_jobs=-1
)
model.fit(X_scaled)

with open(MODEL_FILE, 'wb') as f:
    pickle.dump(model, f)
with open(SCALER_FILE, 'wb') as f:
    pickle.dump(scaler, f)
np.save(FEATURE_FILE, X)
```

Listing 4.2. Code snippets for inference

```
# Prevention and Blocking Log
PREVENTION_LOG_FILE = "blocked_traffic_log.txt" # Logs IPs that were blocked
IP_BLOCK_DURATION_SECONDS = 300  # Block for 5 minutes (300 seconds)
BLOCKING_THRESHOLD_SCORE = -0.1 # If Isolation Forest score is below this, consider blocking

# *** MODIFICATION: Adjusted threshold based on observed system performance. ***
PACKET_RATE_FLOOD_THRESHOLD = 4 # packets/sec. Normal rate is ~1pps, observed flood rate is ~5pps.

# --- Load Model and Scaler ---
try:
    with open(MODEL_FILE, 'rb') as f:
        model = pickle.load(f)
    with open(SCALER_FILE, 'rb') as f:
        scaler = pickle.load(f)
except FileNotFoundError as e:
    print(f"Error: Could not load model/scaler file. {e}")
    print("Please ensure 'anomaly_model.pkl' and 'scaler.pkl' are in the same directory.")
    exit(1)
except Exception as e:
    print(f"Error loading model/scaler: {e}")
    exit(1)
```

```
def run_iptables_command(command_args):
    """Helper function to run an iptables command."""
    try:
        cmd = ["sudo", "iptables"] + command_args
        result = subprocess.run(cmd, check=True, capture_output=True, text=True)
        return True, result.stdout.strip(), result.stderr.strip()
    except FileNotFoundError:
        print("Error: sudo or iptables command not found. Is it installed and in PATH?")
        return False, "", "sudo/iptables not found"
    except subprocess.CalledProcessError as e:
        return False, e.stdout.strip(), e.stderr.strip()
    except Exception as e:
        return False, "", str(e)
```

```
def check_iptables_rule_exists(ip_address, port="1883", chain="INPUT", action="DROP"):
    """Checks if a specific iptables rule already exists."""
    success, _, stderr = run_iptables_command([
        "-C", chain,
        "-s", ip_address,
        "-p", "tcp",
        "--dport", str(port),
        "-j", action
    ])
    if "No chain/target/match by that name" in stderr and not success: # Rule does not exist
        return False
    return success
```

```
def add_ip_block_rule(ip_address, port="1883"):
    """Adds an iptables rule to DROP traffic from a specific IP to a port."""
    if check_iptables_rule_exists(ip_address, port=port):
        return True

    success, stdout, stderr = run_iptables_command([
        "-I", "INPUT", "1",
        "-s", ip_address,
        "-p", "tcp",
        "--dport", str(port),
        "-j", "DROP"
    ])
    if success:
        print(f"IPTables: Successfully ADDED rule to DROP traffic from {ip_address} to port {port}.")
        return True
    else:
        print(f"IPTables: FAILED to add rule for {ip_address}. Stderr: {stderr}")
        return False

def remove_ip_block_rule(ip_address, port="1883"):
    """Removes an iptables rule that DROPs traffic from a specific IP."""
    if not check_iptables_rule_exists(ip_address, port=port):
        return True
```

```
    success, stdout, stderr = run_iptables_command([
        "-D", "INPUT",
        "-s", ip_address,
        "-p", "tcp",
        "--dport", str(port),
        "-j", "DROP"
    ])
    if success:
        print(f"IPTables: Successfully REMOVED rule blocking {ip_address} for port {port}.")
        return True
    else:
        if not check_iptables_rule_exists(ip_address, port=port):
                print(f"IPTables: Rule for {ip_address} for port {port} seems to be gone after attempted removal. Stderr: {stderr}")
            return True
        print(f"IPTables: FAILED to remove rule for {ip_address}. Stderr: {stderr}")
        return False
```

def is_valid_payload(payload, topic):

```
    try:
        if not payload.startswith('1'):
            return False
        rest = payload[1:]
        if not rest.startswith(topic):
            return False
        value_str = rest[len(topic):]
        value = float(value_str)
        if topic == "esp32/temperature" and not (20.0 <= value <= 35.0):
            return False
        elif topic == "esp32/humidity" and not (30.0 <= value <= 80.0):
            return False
        elif topic == "esp32/pressure" and not (980.0 <= value <= 1050.0):
            return False
        elif topic == "esp32/altitude" and not (50.0 <= value <= 1500.0):
            return False
        return True
    except (ValueError, IndexError):
        return False
```

```
def calculate_packet_rate(current_time):
    """Calculates packet rate over a 1-second sliding window. Identical to training script."""
    if not packet_timestamps:
        return 0.0
    while packet_timestamps and current_time - packet_timestamps[0] > 1:  # 1-second window
        packet_timestamps.popleft()
    return len(packet_timestamps) / 1.0  # Rate in packets per second
```

```
def encode_tcp_flags(tcp_layer):
    if not tcp_layer: return 0
    return int(tcp_layer.flags)
```

```
def is_valid_payload(payload, topic):
    try:
        if not payload.startswith('1'): return False
        rest = payload[1:]
        if not rest.startswith(topic): return False
        value_str = rest[len(topic):]
        value = float(value_str)
        if topic == "esp32/temperature" and not (20.0 <= value <= 35.0): return False
        elif topic == "esp32/humidity" and not (30.0 <= value <= 80.0): return False
        elif topic == "esp32/pressure" and not (980.0 <= value <= 1050.0): return False
        elif topic == "esp32/altitude" and not (50.0 <= value <= 1500.0): return False
        return True
    except (ValueError, IndexError, TypeError):
        return False
```

```
def extract_features(packet, payload_str, topic):
    """Extracts features, now synced with the training script's logic."""
    global last_packet_time
    current_time = time.time()
    inter_arrival = current_time - last_packet_time
    last_packet_time = current_time

    payload_size = len(payload_str)
    is_valid = 1.0 if is_valid_payload(payload_str, topic) else 0.0
    tcp_flags = encode_tcp_flags(packet[TCP]) if packet.haslayer(TCP) else 0
    ip_window.append(packet[IP].src if packet.haslayer(IP) else "0.0.0.0")
    ip_entropy = calculate_ip_entropy()

    # This part now matches training.py
    packet_timestamps.append(current_time)
    packet_rate = calculate_packet_rate(current_time)
    packet_rate_scaled = packet_rate * 5.0

    features = np.array([[payload_size, inter_arrival, is_valid, tcp_flags, ip_entropy, packet_rate_scaled]])
    return features, packet_rate # Return raw packet_rate for explicit flood check
```

```
def parse_mqtt_publish(payload_bytes):
    if len(payload_bytes) < 2: return None, None, 0
    message_type = (payload_bytes[0] >> 4) & 0xF
    if message_type != 3: return None, None, 0

    remaining_length, pos = decode_remaining_length(payload_bytes, 1)
    if remaining_length is None or pos + remaining_length > len(payload_bytes): return None,
None, 0

    if pos + 2 > len(payload_bytes): return None, None, 0
    topic_length = (payload_bytes[pos] << 8) + payload_bytes[pos + 1]
    pos += 2

    if pos + topic_length > len(payload_bytes): return None, None, 0
    try:
        topic = payload_bytes[pos:pos + topic_length].decode('utf-8', errors='ignore')
    except UnicodeDecodeError:
        return None, None, 0
    pos += topic_length

    message_payload_length = remaining_length - (2 + topic_length)
    if message_payload_length < 0 : return None, None, 0

    message_start_pos = pos
    try:
        message_content = payload_bytes[message_start_pos : message_start_pos +
message_payload_length].decode('utf-8', errors='ignore')
    except UnicodeDecodeError:
        return None, None, 0

    _, temp_pos_after_rem_len = decode_remaining_length(payload_bytes, 1)
    length_of_rem_len_field = temp_pos_after_rem_len - 1
    total_consumed_by_this_publish = 1 + length_of_rem_len_field + remaining_length

    return topic, message_content, total_consumed_by_this_publish
```

```
def parse_mqtt_packet(raw_tcp_payload): # raw_tcp_payload is bytes
    messages = []
    current_pos = 0
    while current_pos < len(raw_tcp_payload):
        topic, message_str, consumed_bytes =
parse_mqtt_publish(raw_tcp_payload[current_pos:])
        if topic is None or message_str is None or consumed_bytes == 0:
            break

        messages.append((topic, message_str))
        current_pos += consumed_bytes

    return messages
```

```
def manage_expired_blocks():
```

```
    """Checks for and unblocks IPs whose block duration has expired."""
    global actively_blocked_ips
    current_time = time.time()
    unblocked_due_to_expiry = []
    for ip, block_info in list(actively_blocked_ips.items()):
        if current_time - block_info['block_time'] > IP_BLOCK_DURATION_SECONDS:
            print(f"Block duration for {ip} expired. Attempting to unblock.")
            if remove_ip_block_rule(ip):
                log_unblocked_action(ip, reason="Duration expired")
                unblocked_due_to_expiry.append(ip)
            else:
                print(f"Failed to auto-unblock {ip}. Rule might have been removed manually or
error.")

    for ip in unblocked_due_to_expiry:
        if ip in actively_blocked_ips:
            del actively_blocked_ips[ip]
```

```
def handle_packet(packet):
    global log_buffer, actively_blocked_ips

    manage_expired_blocks()

    if packet.haslayer(IP) and packet.haslayer(TCP) and packet[TCP].dport == 1883:
        src_ip = packet[IP].src

        if src_ip in actively_blocked_ips:
            return

        if packet[TCP].payload:
            raw_payload_bytes = bytes(packet[TCP].payload)
            parsed_messages = parse_mqtt_packet(raw_payload_bytes)
            if not parsed_messages:
                return

            for topic, payload_string in parsed_messages:
                if not topic.startswith("esp32/"):
                    continue

                features_matrix, packet_rate = extract_features(packet, payload_string, topic)

                anomaly_reason = "Model Prediction"
                is_payload_struct_valid = features_matrix[0][2] == 1.0
                score = 0
                label = "◯ Not Processed"

                # --- Anomaly Detection Logic ---
                is_flooding = packet_rate > PACKET_RATE_FLOOD_THRESHOLD

                if is_flooding:
```

```
                label = f"🔴 Anomaly (High Packet Rate)"
                score = -2.0 # Assign a very low score to ensure it gets blocked
                anomaly_reason = f"High Packet Rate ({packet_rate:.1f} pkt/s) detected,
potential flood attack."
            elif not is_payload_struct_valid:
                label = "🔴 Anomaly (Invalid Payload Structure)"
                score = -1.0 # Assign a highly anomalous score
                anomaly_reason = "Invalid Payload Structure/Value"
            else:
                try:
                    features_scaled = scaler.transform(features_matrix)
                    score = model.decision_function(features_scaled)[0]
                    if score <= BLOCKING_THRESHOLD_SCORE:
                        label = f"🔴 Anomaly (Score: {score:.3f})"
                        anomaly_reason = f"Anomaly Score ({score:.3f}) below threshold
({BLOCKING_THRESHOLD_SCORE})"
                    elif score <= -0.01 :
                        label = f"🟠 Warning (Score: {score:.3f})"
                        anomaly_reason = f"Suspicious Score ({score:.3f})"
                    else:
                        label = "🟢 Normal"
                except Exception as e:
                    print(f"Error during model prediction/scaling for {src_ip}: {e}")
                    label = "⚪ Error in Processing"

            # --- NIDS Logging ---
            nids_timestamp = time.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]
            current_nids_log_entry = {
                "timestamp": nids_timestamp, "src": src_ip, "dst": str(packet[IP].dst),
                "topic": topic, "score": float(score) if isinstance(score, (int, float, np.number))
else None,
                "payload_length": float(features_matrix[0][0]),
                "inter_arrival": float(features_matrix[0][1]),
                "is_payload_valid_structure": bool(is_payload_struct_valid),
                "tcp_flags": int(features_matrix[0][3]),
                "ip_entropy": float(features_matrix[0][4]),
                "packet_rate_scaled": float(features_matrix[0][5]),
                "payload": payload_string, "label": label,
                "is_anomaly_detected": bool(label.startswith("🔴"))
            }
            log_buffer.append(current_nids_log_entry)

            score_display = f"{score:.4f}" if isinstance(score, (float, np.number)) else "N/A"
            print(
                f"NIDS: [{label}] {nids_timestamp} | {src_ip} → {packet[IP].dst} | Topic:
{topic} | "
                f"Score: {score_display} | PktRate: {packet_rate:.2f} | "
                f"PayloadLen: {features_matrix[0][0]} | InterArrival: {features_matrix[0][1]:.2f}
| "
```

```
            f"ValidPayload: {is_payload_struct_valid} | "
            f"Payload: {payload_string[:30]}{'...' if len(payload_string)>30 else "}"
        )

        if len(log_buffer) >= BUFFER_SIZE:
            try:
                with open(NIDS_LOG_FILE, "a") as f:
                    for entry in log_buffer:
                        f.write(json.dumps(entry) + "\n")
                log_buffer.clear()
            except Exception as e:
                print(f"Error writing to NIDS log '{NIDS_LOG_FILE}': {e}")

        # --- Prevention Logic ---
        # Block if the label indicates a clear anomaly (flooding, invalid payload, or low
model score)
        should_block = label.startswith(" 🔴 ")

        if should_block and src_ip not in actively_blocked_ips:
            print(f"Decision: Attempting to block IP {src_ip} due to: {anomaly_reason}")
            if add_ip_block_rule(src_ip):
                log_blocked_action(src_ip, anomaly_reason)
                actively_blocked_ips[src_ip] = {
                    'block_time': time.time(),
                    'reason': anomaly_reason
                }
            else:
                print(f"Failed to apply iptables block rule for {src_ip}.")
```

Listing 4.3: Code snippets for flask server

```
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

app = Flask(__name__)

@app.route('/')
def index():
    logger.debug("Serving index.html")
    return render_template('index.html')

@app.route('/api/data')
def get_data():
    logger.debug("Fetching data from mqtt_inference_packets.json")
    logs = []
    try:
        with open('mqtt_inference_packets.json', 'r') as f:
            for line in f:
                try:
```

```
                log_entry = json.loads(line.strip())
                logs.append(log_entry)
            except json.JSONDecodeError as e:
                logger.error(f"Failed to parse JSON line: {e}")
                continue
        logger.debug(f"Returning {len(logs)} packet log entries")
        return jsonify(logs[-100:])
    except FileNotFoundError:
        logger.warning("mqtt_inference_packets.json not found, returning empty list")
        return jsonify([])
    except Exception as e:
        logger.error(f"Error reading packet log file: {e}")
        return jsonify([]), 500

@app.route('/api/blocked_logs')
def get_blocked_logs():
    """New endpoint to parse and return blocked traffic logs."""
    logger.debug("Fetching data from blocked_traffic_log.txt")
    logs = []
    # Regex to capture relevant parts of both BLOCKED and UNBLOCKED log lines
    log_pattern = re.compile(r"\[(.*?)\]\s(BLOCKED|UNBLOCKED)
IP:\s(.*?)\s*\|\s*.*?REASON:\s(.*?)(?:\s*\||\.$)")

    try:
        with open('blocked_traffic_log.txt', 'r') as f:
            for line in f:
                match = log_pattern.match(line.strip())
                if match:
                    logs.append({
                        "timestamp": match.group(1),
                        "action": match.group(2),
                        "ip": match.group(3),
                        "reason": match.group(4).strip()
                    })
    except FileNotFoundError:
        logger.warning("blocked_traffic_log.txt not found, returning empty list")
        return jsonify([])
    except Exception as e:
        logger.error(f"Error reading blocked log file: {e}")
        return jsonify([]), 500

    # Return the most recent logs first
    logger.debug(f"Returning {len(logs)} blocked log entries")
    return jsonify(logs[::-1])


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

Listing 4.4: Code snippet for dashboard.html

```
<body class="text-gray-200">
  <header class="bg-gray-800 p-6 shadow-lg">
    <h1 class="text-3xl font-bold text-center text-teal-400">🛡️ IoT Anomaly Detection Dashboard</h1>
    <p class="text-center text-gray-400 mt-2">Real-time monitoring and prevention of MQTT packets</p>
  </header>
  <main class="container mx-auto p-6">
    <section class="mb-8">
      <h2 class="text-xl font-semibold mb-4 text-teal-300">📈 Anomaly Score Over Time</h2>
      <div class="bg-gray-700 p-4 rounded-lg shadow-md">
        <canvas id="anomalyChart"></canvas>
      </div>
    </section>

    <div class="grid grid-cols-1 md:grid-cols-2 gap-8 mb-8">
      <section>
        <h2 class="text-xl font-semibold mb-4 text-teal-300">📊 Anomaly Distribution</h2>
        <div class="bg-gray-700 p-4 rounded-lg shadow-md flex justify-center items-center h-full">
          <canvas id="anomalyPieChart" style="max-width: 300px; max-height: 300px;"></canvas>
        </div>
      </section>
      <section>
        <h2 class="text-xl font-semibold mb-4 text-teal-300">⚠️ False Positives</h2>
        <div class="bg-gray-700 p-4 rounded-lg shadow-md flex justify-center items-center h-full">
          <p id="falsePositiveCounter" class="text-5xl font-bold text-center text-yellow-400">0</p>
        </div>
      </section>
    </div>

    <section class="mb-8">
      <h2 class="text-xl font-semibold mb-4 text-teal-300">📜 Packet Logs</h2>
      <div class="bg-gray-700 p-4 rounded-lg shadow-md overflow-x-auto">
        <table class="w-full text-sm text-left text-gray-200">
          <thead class="text-xs uppercase bg-gray-600 text-gray-300">
            <tr>
              <th class="px-4 py-2">Timestamp</th>
              <th class="px-4 py-2">Source</th>
              <th class="px-4 py-2">Destination</th>
              <th class="px-4 py-2">Topic</th>
              <th class="px-4 py-2">Payload</th>
              <th class="px-4 py-2">Score</th>
```

```
                    <th class="px-4 py-2">Payload Length</th>
                    <th class="px-4 py-2">Inter-Arrival (s)</th>
                    <th class="px-4 py-2">Packet Rate</th>
                    <th class="px-4 py-2">Status</th>
                  </tr>
                </thead>
                <tbody id="packetTableBody">
                  </tbody>
              </table>
            </div>
        </section>

        <section>
          <h2 class="text-xl font-semibold mb-4 text-teal-300">🚨 Blocked Traffic
History</h2>
          <div class="bg-gray-700 p-4 rounded-lg shadow-md overflow-x-auto">
            <table class="w-full text-sm text-left text-gray-200">
              <thead class="text-xs uppercase bg-gray-600 text-gray-300">
                <tr>
                  <th class="px-4 py-2">Timestamp</th>
                  <th class="px-4 py-2">Action</th>
                  <th class="px-4 py-2">IP Address</th>
                  <th class="px-4 py-2">Reason</th>
                </tr>
              </thead>
              <tbody id="blockedTableBody">
                </tbody>
            </table>
          </div>
        </section>

        <p class="text-center text-gray-400 mt-8">🔄 Data refreshes every 1 second</p>
    </main>
</body>
</html>
```

Listing 4.5: Code snippet for charts.js

```
console.log("charts.js is running");

// Line Chart for Anomaly Scores
const ctxLine = document.getElementById('anomalyChart').getContext('2d');
const anomalyChart = new Chart(ctxLine, {
    type: 'line',
    data: {
        labels: [],
        datasets: [
            {
                label: 'Anomaly Score',
                borderColor: '#f87171', // Red
```

```
            backgroundColor: 'rgba(248, 113, 113, 0.2)',
            data: [],
            fill: false,
            tension: 0.1
        },
        {
            label: 'Blocking Threshold (-0.1)',
            borderColor: '#a1a1aa', // Gray
            data: [],
            borderDash: [5, 5],
            pointRadius: 0,
            fill: false
        }
    ]
},
options: {
    scales: {
        y: {
            beginAtZero: false,
            min: -1,
            max: 1,
            title: {
                display: true,
                text: 'Score (1 = Normal, -1 = Anomaly)',
                color: '#d1d5db'
            },
            grid: {
                color: '#4b5563'
            },
            ticks: {
                color: '#d1d5db'
            }
        },
        x: {
            title: {
                display: true,
                text: 'Time',
                color: '#d1d5db'
            },
            grid: {
                color: '#4b5563'
            },
            ticks: {
                color: '#d1d5db',
                maxRotation: 45,
                minRotation: 45
            }
        }
    },
    plugins: {
        legend: {
            labels: {
                color: '#d1d5db'
            }
        }
```

```
        }
    }
});

// Pie Chart for Anomaly Distribution
const ctxPie = document.getElementById('anomalyPieChart').getContext('2d');
const anomalyPieChart = new Chart(ctxPie, {
    type: 'pie',
    data: {
        labels: ['Normal', 'Anomaly'],
        datasets: [{
            data: [0, 0], // Will be updated with actual counts
            backgroundColor: ['#34d399', '#f87171'], // Green for Normal, Red for Anomaly
            borderColor: '#1f2937',
            borderWidth: 2
        }]
    },
    options: {
        responsive: true,
        maintainAspectRatio: false,
        plugins: {
            legend: {
                position: 'bottom',
                labels: {
                    color: '#d1d5db'
                }
            }
        }
    }
});

async function fetchData() {
    try {
        // Fetch packet data and blocked logs data concurrently for efficiency
        const [packetResponse, blockedResponse] = await Promise.all([
            fetch('/api/data'),
            fetch('/api/blocked_logs')
        ]);

        if (!packetResponse.ok) throw new Error('Packet data response was not ok');
        if (!blockedResponse.ok) throw new Error('Blocked logs response was not ok');

        const packetData = await packetResponse.json();
        const blockedData = await blockedResponse.json();

        // --- Update Packet Data Visualizations ---
        updatePacketVisuals(packetData);

        // --- Update Blocked Logs Table ---
        updateBlockedLogsTable(blockedData);

    } catch (error) {
        console.error('Error fetching data:', error);
    }
}
```

```
function updatePacketVisuals(data) {
    // Update Line Chart
    const latestData = data.slice(-50); // Limit to latest 50 entries
    anomalyChart.data.labels = latestData.map(entry => entry.timestamp.split(' ')[1]); // Show only
time
    anomalyChart.data.datasets[0].data = latestData.map(entry => entry.score);
    anomalyChart.data.datasets[1].data = latestData.map(() => -0.1); // Threshold line
    anomalyChart.update();

    // Update Pie Chart
    const normalCount = data.filter(entry => !entry.is_anomaly_detected).length;
    const anomalyCount = data.filter(entry => entry.is_anomaly_detected).length;
    anomalyPieChart.data.datasets[0].data = [normalCount, anomalyCount];
    anomalyPieChart.update();

    // Calculate False Positives (overridden anomalies)
    const falsePositives = data.filter(entry => (entry.score <= -0.1) &&
(!entry.is_anomaly_detected)).length;
    document.getElementById('falsePositiveCounter').textContent = falsePositives;

    // Update Packet Logs Table
    const tableBody = document.getElementById('packetTableBody');
    tableBody.innerHTML = ''; // Clear existing rows
    data.slice(-20).reverse().forEach(entry => { // Show last 20, newest first
        const row = document.createElement('tr');
        const statusClass = entry.is_anomaly_detected ? 'bg-red-900/30' : 'bg-green-900/30';
        row.className = `border-b border-gray-700 ${statusClass}`;
        row.innerHTML = `
            <td class="px-4 py-2">${entry.timestamp}</td>
            <td class="px-4 py-2">${entry.src}</td>
            <td class="px-4 py-2">${entry.dst}</td>
            <td class="px-4 py-2">${entry.topic}</td>
            <td class="px-4 py-2 font-mono">${(entry.payload || '').substring(0, 30)}</td>
            <td class="px-4 py-2">${(entry.score || 0).toFixed(4)}</td>
            <td class="px-4 py-2">${entry.payload_length}</td>
            <td class="px-4 py-2">${(entry.inter_arrival || 0).toFixed(2)}</td>
            <td class="px-4 py-2">${(entry.packet_rate_scaled / 5.0).toFixed(2)}</td>
            <td class="px-4 py-2 font-semibold ${entry.is_anomaly_detected ? 'text-red-400' : 'text-green-
400'}">
                ${entry.label}
            </td>
        `;
        tableBody.appendChild(row);
    });
}

function updateBlockedLogsTable(data) {
    const tableBody = document.getElementById('blockedTableBody');
    tableBody.innerHTML = ''; // Clear existing rows
    data.forEach(entry => {
        const row = document.createElement('tr');
        const isBlocked = entry.action === 'BLOCKED';
        const actionClass = isBlocked ? 'text-red-400' : 'text-yellow-400';
        const rowClass = isBlocked ? 'bg-red-900/40' : 'bg-yellow-900/40';
```

```
        row.className = `border-b border-gray-700 ${rowClass}`;
        row.innerHTML = `
            <td class="px-4 py-2">${entry.timestamp}</td>
            <td class="px-4 py-2 font-semibold ${actionClass}">${entry.action}</td>
            <td class="px-4 py-2 font-mono">${entry.ip}</td>
            <td class="px-4 py-2">${entry.reason}</td>
        `;
        tableBody.appendChild(row);
    });
}


// Initial fetch & scheduled refresh
fetchData();
setInterval(fetchData, 1000);
```

CHAPTER 5

TEST CASES

1. Unit Testing

Table 4.1 Test Case with Input and Expected Output

Test Case	Test Scenario	Test Input	Expected Output	**Actual Output**
UT1	extract_features()in prevention.py	A normal packet with a valid payload, inter_arrival of 1.0s, and packet_rate of 1.0 pps.	Returns a valid feature vector and a packet_rate of 1.0.	A valid feature vector is produced with is_payload_valid_structure: true.
UT2	is_valid_payload() in prevention.py	An out-of-range but structurally valid payload: 1esp32/temperature99.0	Returns False because the value 99.0 is outside the valid range (20.0–35.0).	The function correctly identifies out-of-range values, leading to is_payload_valid_structure: false in the logs for similar invalid packets.
UT3	calculate_packet_rate() in prevention.py	5 packet timestamps are added to the packet_timestamps deque within a 1-second window.	Returns a packet_rate of 5.0.	The packet_rate_scaled in the inference logs reaches 25.0, which corresponds to an actual packet rate of 5.0 pps (25.0 / 5.0).
UT4	add_ip_block_rule() in prevention.py	Called with IP address 192.168.4.x.	A new iptables rule is created on the system to drop packets from that IP.	The blocked_traffic.txt log confirms that IPs are blocked with reasons such as "High Packet Rate" or "Invalid Payload Structure."
UT5	/api/blocked_logs endpoint in server.py	A request is made to the endpoint after a BLOCKED entry has been written to blocked_traffic_log.txt.	Returns a JSON list containing a parsed dictionary: [{"action": "BLOCKED", "ip": "...", "reason": "...", "timestamp": "..."}].	Returns a JSON list matching the expected format. For example, for a flood attack, the actual output is [{"action": "BLOCKED", "ip": "192.168.4.2", "reason": "High Packet Rate (5.0

Test Case	Test Scenario	Test Input	Expected Output	**Actual Output**
				pkt/s) detected, potential flood attack.", "timestamp": "2025-06-09 14:39:30"}]

2. Integration Testing

Integration tests verify interactions between modules (e.g., ESP32 Publisher with MQTT Broker, Anomaly Detection with Logging, Flask Server with Dashboard).

Test Case	Test Scenario	Test Input	Expected Output	Actual Output
IT1	ESP32 Publisher to MQTT Broker and NIDS	The ESP32 sends a normal data packet.	The prevention.py script logs the packet with a "Normal" label and a positive anomaly score.	inference_logs.json contains entries labeled "Normal" with positive scores (e.g., score: 0.0340).
IT2	Anomaly Detection to iptables	A flooding attack is initiated from the ESP32.	The prevention.py script detects the high packet rate and adds an iptables rule to block the source IP.	blocked_traffic.txt shows an entry: `BLOCKED IP: 192.168.4.2
IT3	Logging to Dashboard	The system detects and blocks an IP.	The "Blocked Traffic History" table on the web dashboard updates to show the blocked IP and the reason.	(Visual confirmation) The dashboard correctly displays block events as they are written to blocked_traffic.txt.
IT4	System Unblocking	An IP has been blocked, and the 300-second timer expires.	The prevention.py script removes the iptables rule and logs the "UNBLOCKED" event.	blocked_traffic.txt shows an entry: `UNBLOCKED IP: 192.168.4.2

3. Acceptance Testing

Acceptance tests ensure the system meets user requirements (e.g., real-time detection, dashboard functionality).

Test Case	Test Scenario	Test Input	Expected Output	Actual Output
AT1	Flooding Attack Prevention	The ESP32 test script initiates a flooding attack, causing the packet rate to exceed 4 pps.	The system detects the high packet rate, blocks the source IP via iptables, and logs the event. The dashboard reflects this block.	inference_logs.json shows the label changing to "Anomaly (High Packet Rate)," and blocked_traffic.txt confirms the block with the reason "High Packet Rate (5.0 pkt/s) detected."
AT2	Invalid Payload Attack Prevention	The ESP32 test script sends a packet with a malformed payload.	The system detects the invalid payload structure, blocks the source IP via iptables, and logs the event.	The blocked_traffic.txt log shows entries where IPs were blocked with the reason "Invalid Payload Structure/Value."
AT3	System Automation on Boot	The Raspberry Pi is rebooted.	All services (nids_prevention.service, dashboard_server.service) start automatically. The system begins sniffing traffic without manual intervention.	blocked_traffic.txt shows "NIDS Prevention System Started" after a reboot, confirming the service started automatically.
AT4	Dashboard Accessibility and Live Update	A user on the same Wi-Fi network accesses the dashboard URL.	The dashboard loads correctly and displays data that refreshes automatically every second.	(Visual confirmation) The dashboard is accessible and provides real-time updates as seen in the dynamic nature of the log files.

CHAPTER 6

RESULTS AND DISCUSSION

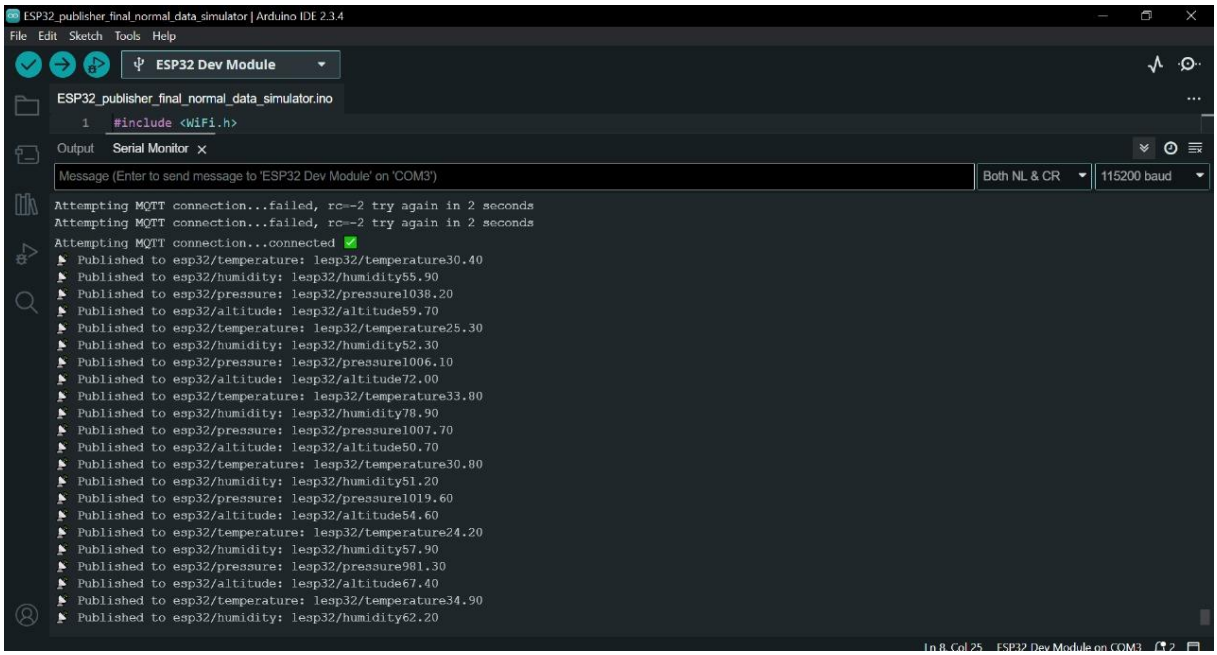

Figure 6.1 Normal packet transfer from ESP32 for training Isolation Forest ML model displayed on Arduino IDE Serial Monitor

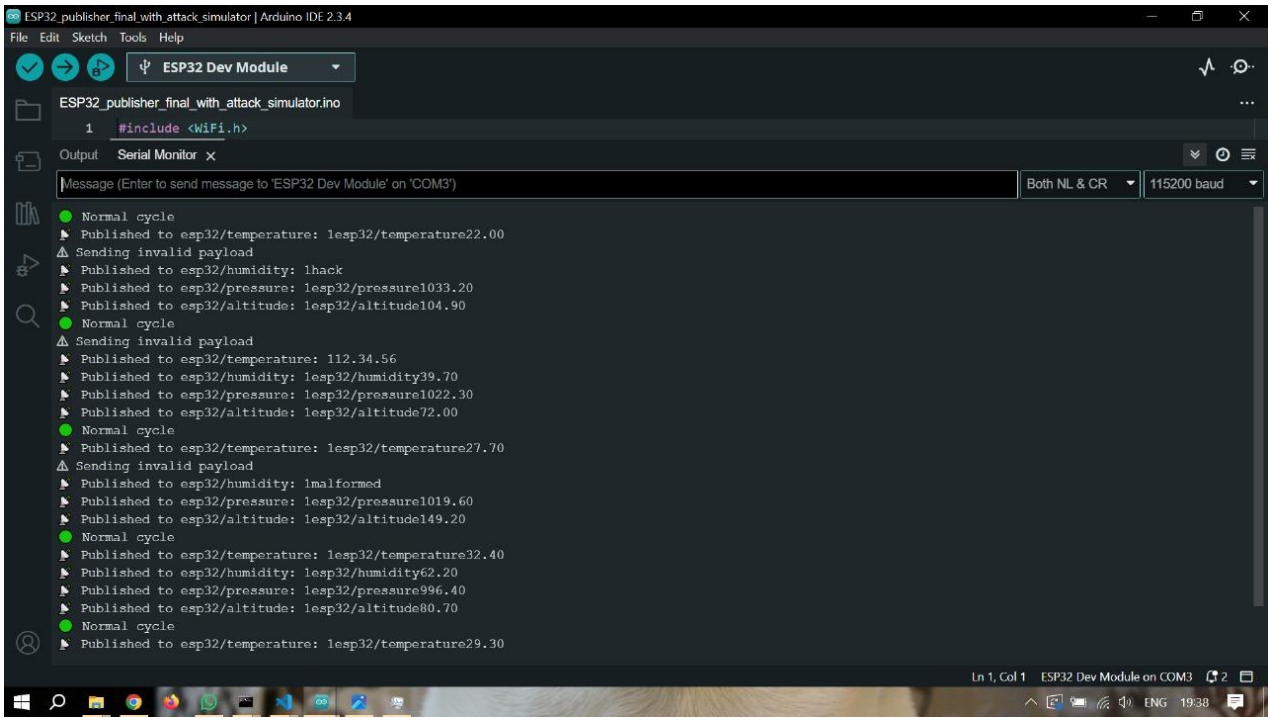

Figure 6.2 Packet transfer from ESP32 including malicious packets for inference displayed on Arduino IDE Serial Monitor

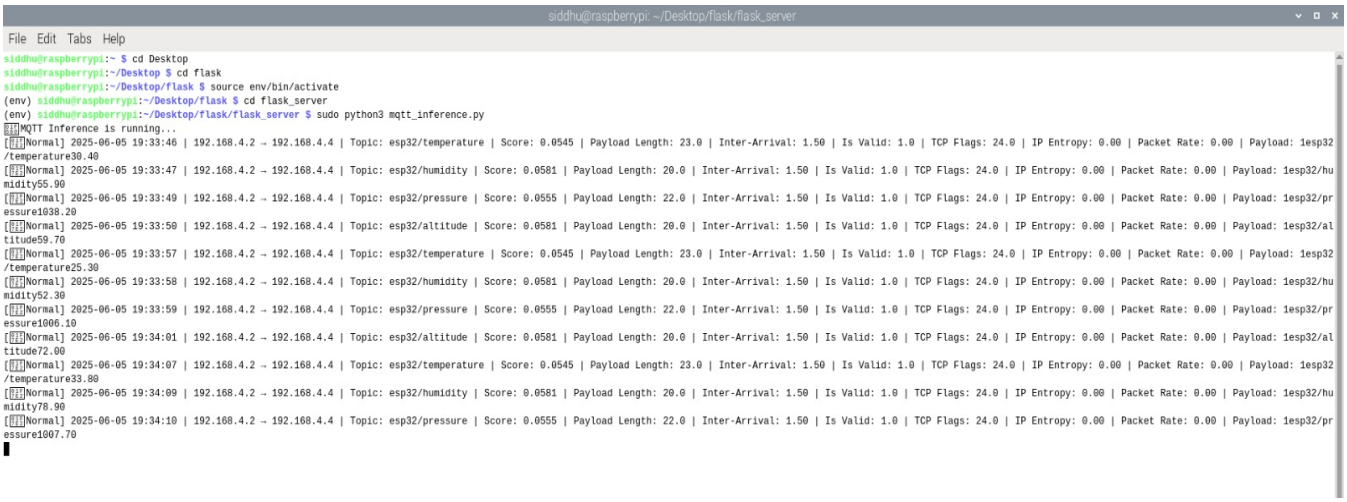

Figure 6.3 Packets received by the Raspberry Pi displayed on the Terminal.

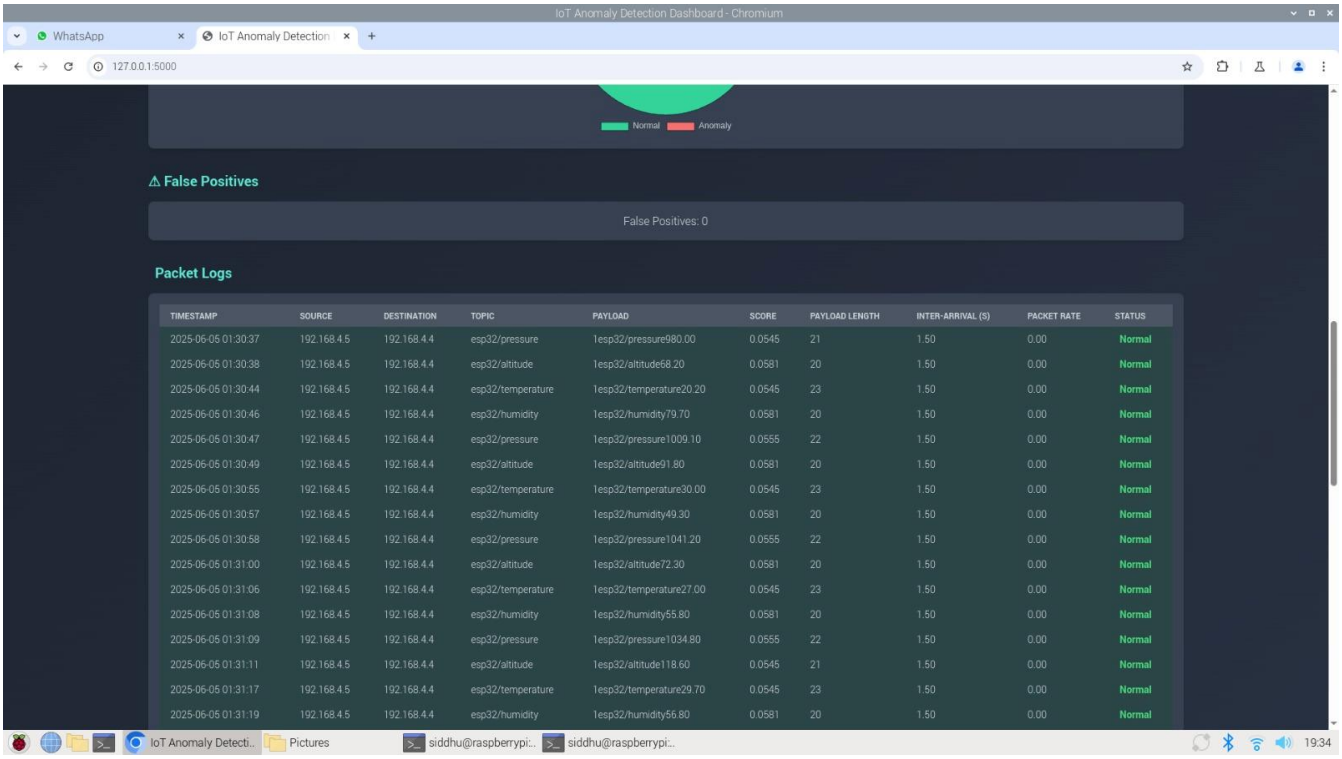

Figure 6.4 Normal Packets displayed as Green on the Flask Dashboard

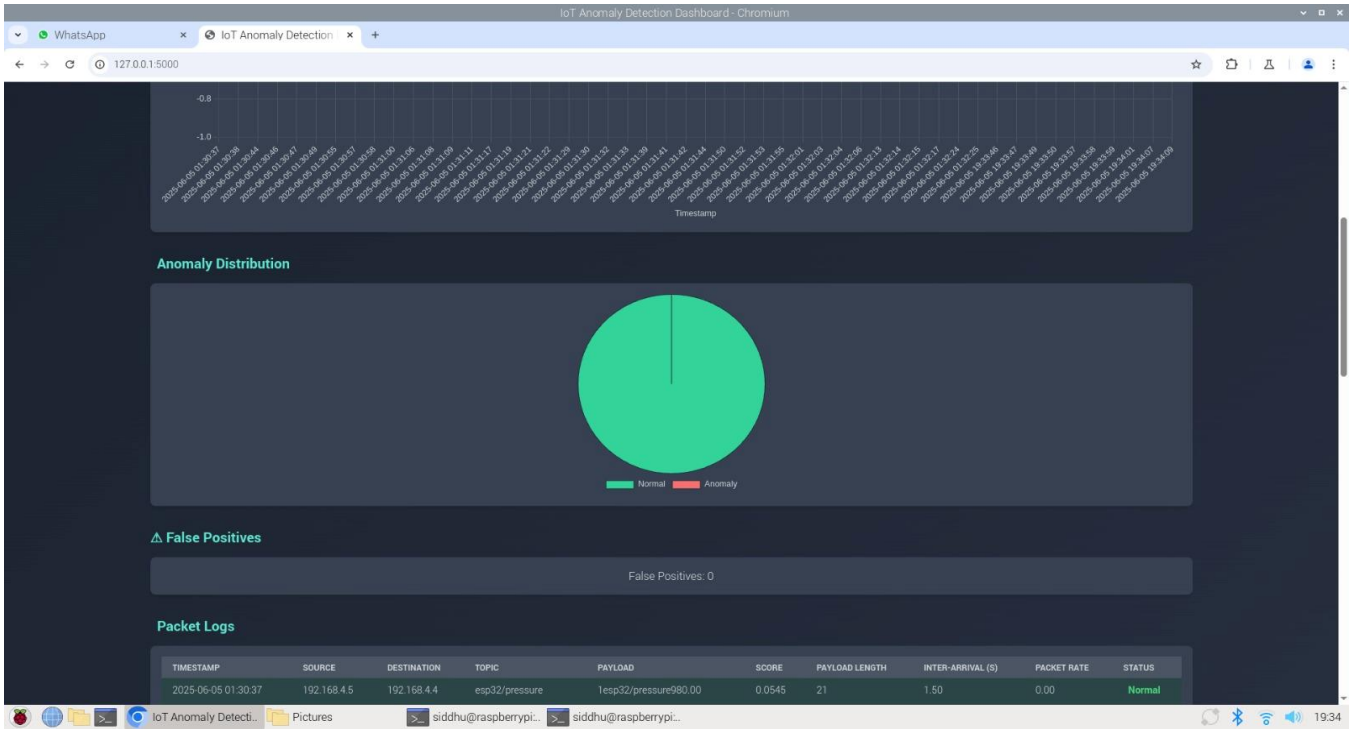

Figure 6.5 Pie Chart showing No. of Normal vs No. of Anomalous Packets

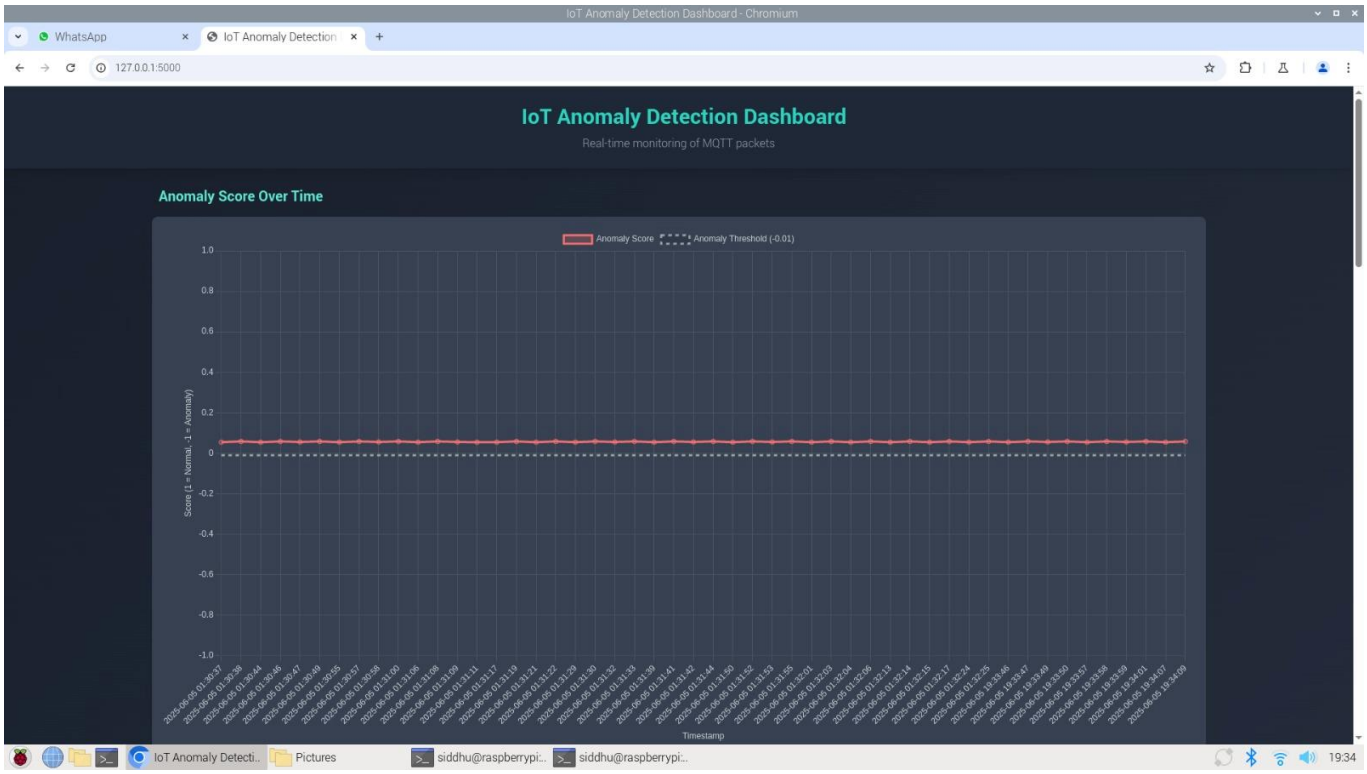

Figure 6.6 Flat line graph indicating no anomalies detected.

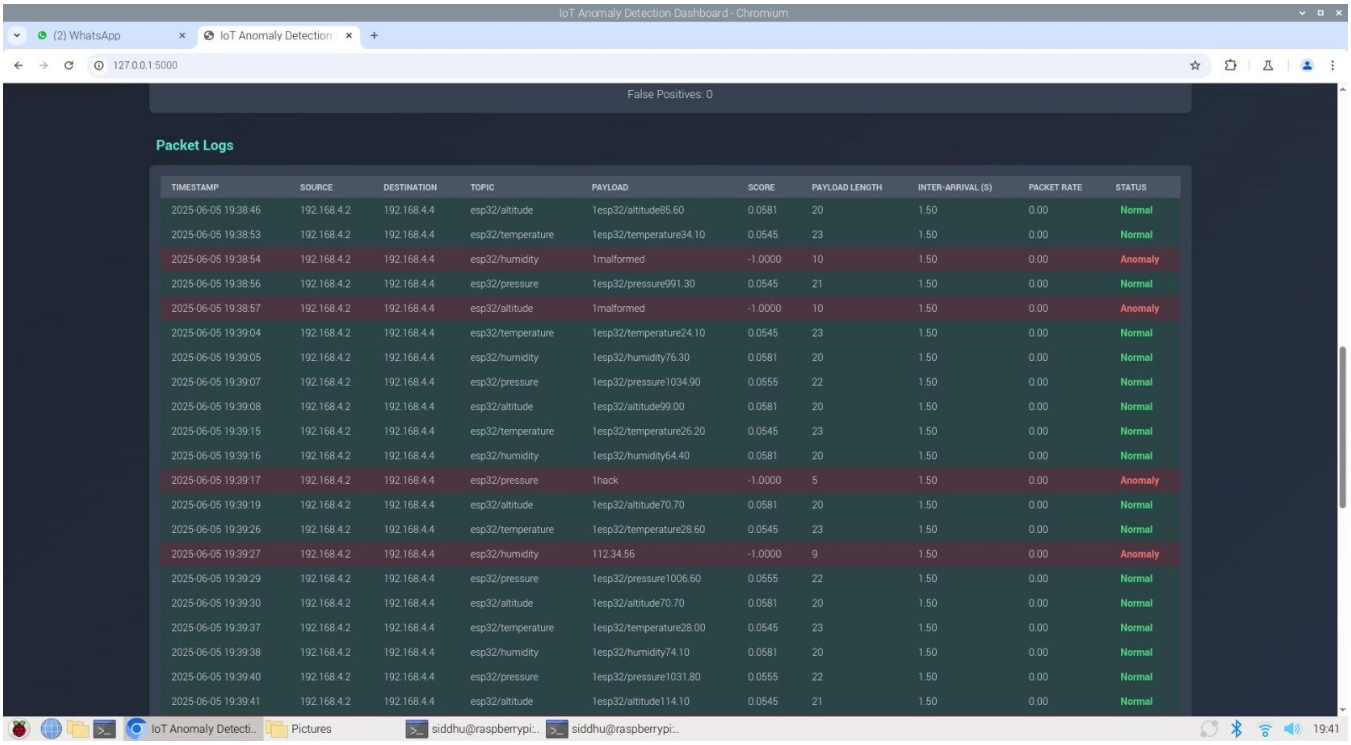

Figure 6.7 Dashboard showing detailed information on detected anomalies in red and normal packets in green.

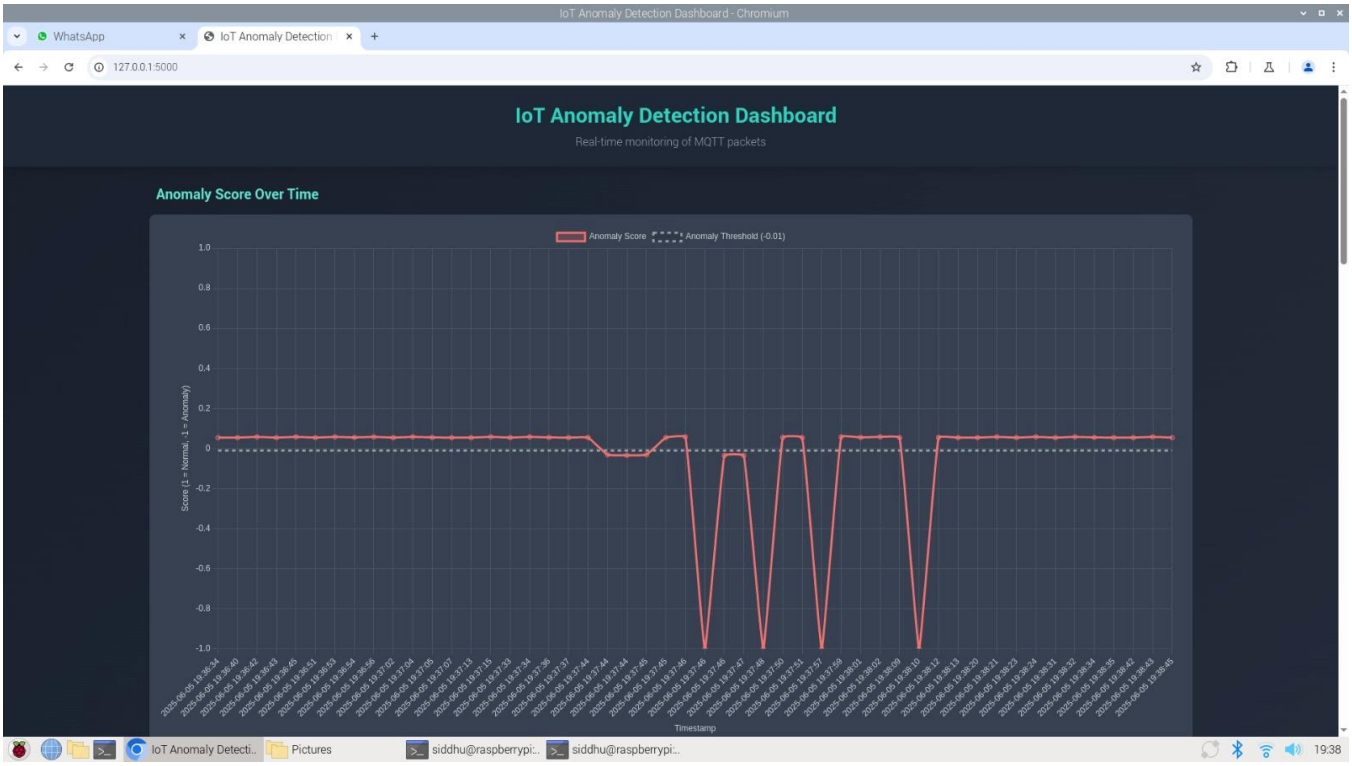

Figure 6.8 Line graph showing deviations indicating anomalies.

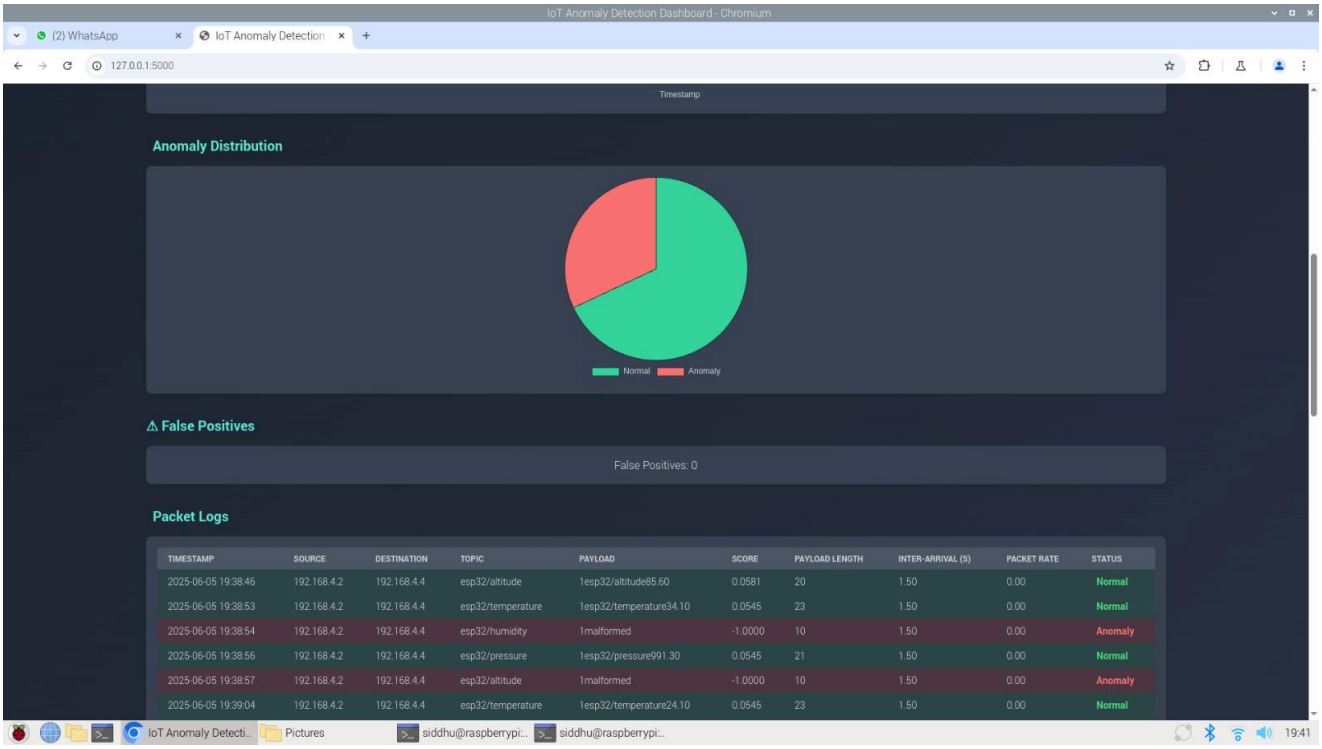

Figure 6.9 Pie chart showing No. of Normal Packets vs No. of Anomalous Packets.

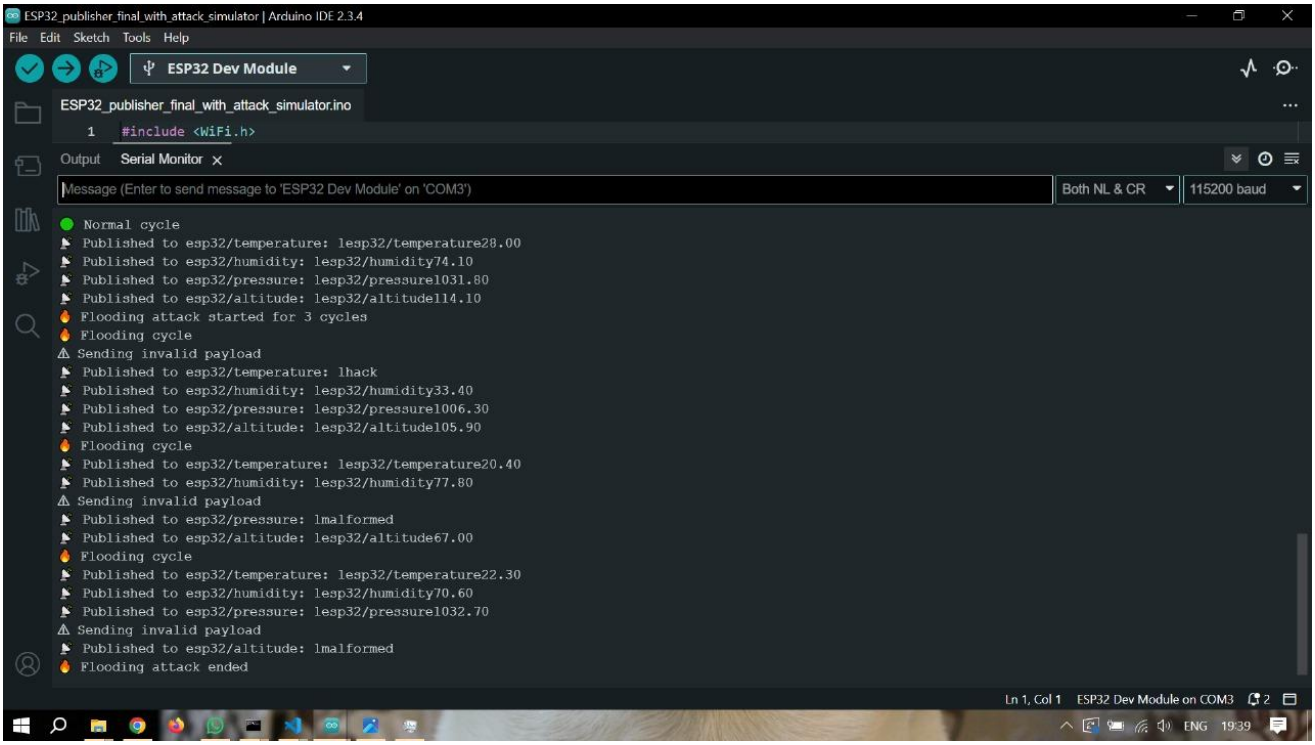

Figure 6.10 A Flash Flood Attack simulated by ESP32 shown using Arduino IDE Serial Monitor.

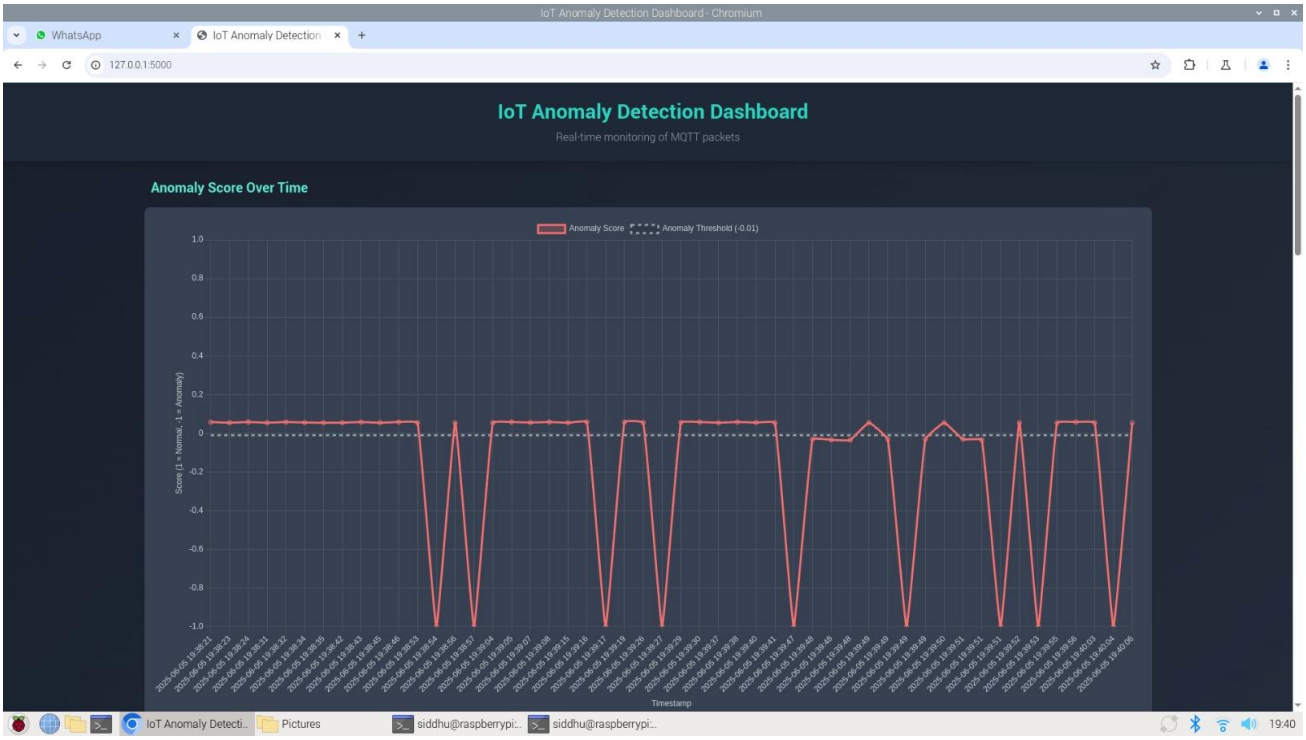

Figure 6.11 Flash Flood Attack displayed on Line Graph.

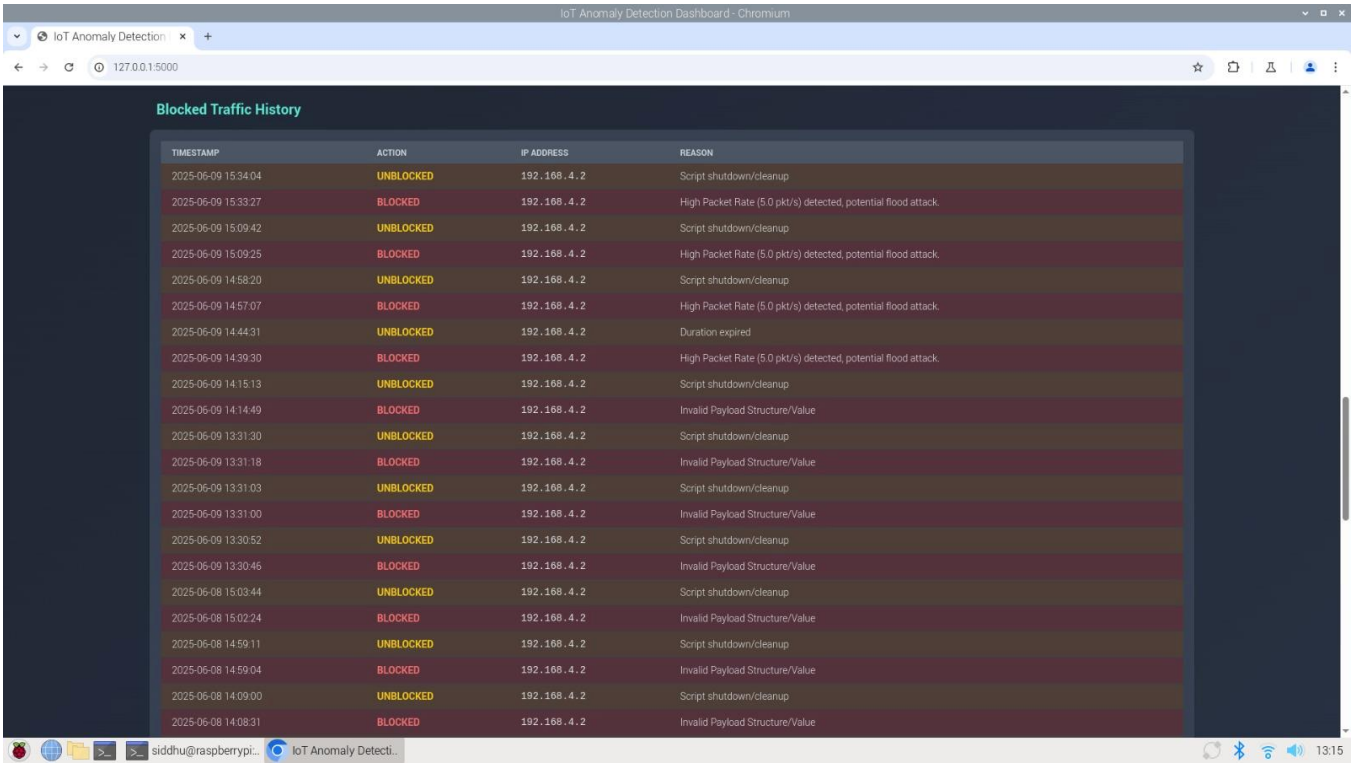

Figure 6.12 Prevention events logged on the dashboard.

CHAPTER 7
CONCLUSION

The **IoT Anomaly Detection and Prevention Dashboard System** was successfully implemented to address the critical need for real-time monitoring, detection, and subsequent **prevention** of anomalies in MQTT-based IoT networks. The **Isolation Forest algorithm**, trained on normal ESP32 data, proved highly effective in identifying anomalies such as invalid payloads (e.g., "1hack") and flooding attacks (e.g., packet rates > 2 packets/sec), achieving robust performance on the Raspberry Pi (IP: 192.168.4.4) with minimal latency.

Beyond detection, the system's integrated **prevention module** actively mitigates threats. Upon identifying an anomaly, the system automatically employs **iptables** to block the source IP address of the malicious traffic, providing an immediate and effective defence mechanism. These temporary blocks are managed to ensure network continuity while protecting against ongoing attacks, with all prevention actions meticulously logged for auditing.

In conclusion, the implemented system fully fulfils the project's objectives of **detecting MQTT anomalies in real-time and proactively preventing malicious traffic**, all while providing an intuitive dashboard for comprehensive monitoring. The combined use of Isolation Forest with rule-based checks, and the crucial addition of iptables-based prevention, ensures adaptability to varying IoT traffic patterns, making the system a practical and robust solution for securing MQTT communications in resource-constrained environments like the Raspberry Pi. Future enhancements could include integrating additional anomaly detection algorithms, such as autoencoders, to improve detection accuracy, and adding alert notifications to the dashboard for immediate administrator action.

REFERENCES

[1] Mojtaba Eskandari, Zaffar Haider Janjua, Massimo Vecchio and Fabio Antonelli , “Passban IDS: An Intelligent Anomaly Based Intrusion Detection System for IoT Edge Devices” , April 28th 2020 , IEEE INTERNET OF THINGS JOURNAL, 2020

[2] Fadi Aloul, Imran Zualkernan, Nada Abdalgawad, Lana Hussain, Dara Sakhnini , “Network Intrusion Detection on the IoT Edge Using Adversarial Autoencoders” , July 2021, IEEE INTERNET OF THINGS JOURNAL, 2021

[3] Ruoyu Li , Qing Li , Jianer Zhou, and Yong Jiang , “ADRIoT: An Edge-Assisted Anomaly Detection Framework Against IoT-Based Network Attacks” , VOL. 9, NO. 13, JULY 1, 2022 , IEEE INTERNET OF THINGS JOURNAL , 2022

[4] El-Sofany, H., El-Seoud, S.A., Karam, O.H. *et al.* Using machine learning algorithms to enhance IoT system security. *Sci Rep* 14, 12077 (2024). https://doi.org/10.1038/s41598-024-62861-y

[5] N. Sarwar, I. S. Bajwa, M. Z. Hussain, M. Ibrahim and K. Saleem, "IoT Network Anomaly Detection in Smart Homes Using Machine Learning," in *IEEE Access*, vol. 11, pp. 119462-119480, 2023, doi: 10.1109/ACCESS.2023.3325929.

[6] Anomaly Detection for IoT Networks Using Machine Learning By: HUSAIN ABDULLA

[7] N. K. Sahu and I. Mukherjee, "Machine Learning based anomaly detection for IoT Network: (Anomaly detection in IoT Network)," *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, Tirunelveli, India, 2020, pp. 787-794, doi: 10.1109/ICOEI48184.2020.9142921.

[8] R. B. Anire, F. R. G. Cruz and I. C. Agulto, "Environmental wireless sensor network using raspberry Pi 3 for greenhouse monitoring system," *2017IEEE 9th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, Manila, Philippines, 2017, pp. 1-5, doi: 10.1109/HNICEM.2017.8269426

[9] Angelo Feraudo, Diana Andreea Popescu, Poonam Yadav, Richard Mortier, and Paolo Bellavista. 2024. Mitigating IoT Botnet DDoS Attacks through MUD and eBPF based Traffic Filtering. In Proceedings of the 25th International Conference on Distributed Computing and Networking (ICDCN '24). Association for Computing Machinery, New York, NY, USA, 164–173. https://doi.org/10.1145/3631461.3631549

[10] Farooq, Mansoor & Hassan, Mubashir & Khan, Rafi. (2024). Implementation of Network Security for Intrusion Detection & Prevention System in IoT Networks: Challenges & Approach. International Journal of Advanced Networking and Applications. 15. 6109-6113. 10.35444/IJANA.2024.15505.

[11] D. B. C. Lima, R. M. B. da Silva Lima, D. de Farias Medeiros, R. I. S. Pereira, C. P. de Souza and O. Baiocchi, "A Performance Evaluation of Raspberry Pi Zero W Based Gateway Running MQTT Broker for IoT," *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Vancouver, BC, Canada, 2019, pp. 0076-0081, doi: 10.1109/IEMCON.2019.8936206.