NETWORKING ANOMALY DETECTION USING TCP/IP PROTOCOLS

A PROJECT REPORT

Submitted by

AARTHI. C	**(422621106001)**
MAGESHWARI. V	**(422621106020)**
PAVITHRA. S	**(422621106023)**
DINESH. A	**(422621106704)**

In partial fulfilment for the award of the degree

of

BACHELOR OF ENGINEERING

in

ELECTRONICS AND COMMUNICATION ENGINEERING

UNIVERSITY COLLEGE OF ENGINEERING, PANRUTI

(A Constituent College of Anna University-Chennai)

ANNA UNIVERSITY :: CHENNAI 600 025

MAY 2025

ANNA UNIVERSITY : CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report **"NETWORKING ANOMALY DETECTION USING TCP/IP PROTOCOLS"** is the Bonafide work of **"Aarthi.C (422621106001), Mageshwari.V (422621106020), Pavithra.S (422621106023), Dinesh.A (422621106704)"** who carried out the project work under my supervision.

SIGNATURE
Dr.A.UMA MAHESWARI M.E.,Ph.D
HEAD OF THE DEPARTMENT(i/c)
Department of Electronics and Communication Engineering
University College of Engineering, Panruti
Panruti - 607106

SIGNATURE
Dr.R.J.KAVITHA M.E.,Ph.D
SUPERVISOR
Department of Electronics and Communication Engineering
University College of Engineering, Panruti
Panruti - 607106

Submitted to the Anna University Project viva voce held on _____ during the year 2024-2025

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DECLARATION

We hereby declare that the project work entitled **"NETWORKING ANOMALY DETECTION USING TCP/IP PROTOCOLS**" is submitted in partial fulfilment of the requirement for the award of the degree in B.E., Anna university, Chennai is a record of our work carried out **C.AARTHI, V.MAGESHWARI, S.PAVITHRA, A.DINESH** during the academic year 2021-2025 under the supervision and guidance of **Dr.R.J.KAVITHA,M.E.,Ph.D., Assistant Professor(Sl.Gr), Department of Electronics and Communication Engineering, University College of Engineering, Panruti.** The extent and source of information are derived from the existing literature and have been indicated through the dissertation at the appropriate places. The matter embodied in this work is original and has not been submitted for the award of any other degree or diploma, either in this or any other University.

AARTHI. C **(422621106001)** _____

MAGESHWARI. V **(422621106020)** _____

PAVITHRA. S **(422621106023)** _____

DINESH. A **(422621106704)** _____

I certify that the declaration made above by the candidate are true.

SIGNATURE OF THE GUIDE,

Dr..R.J.KAVITHA M.E.,Ph.D.,

ACKNOWLEDGEMENT

At the very outset, we wish to express our sincere thanks to all those who involved in this project.

We are extremely thankful to our beloved honourable and respectful Dean(i/c) **Dr.S.MUTHUKUMARAN M.E.,Ph.D.,** for inspiring and motivating us to bring out a perfect and successful project work.

We are so much grateful to our Head of the Department(i/c) **Dr.A.UMA MAHESWARI M.E.,Ph.D.,** for the commendable support and encouragement for the completion of the project with perfection.

It's our immense pleasure to thank our respectful incharge **P.PARIMALAM Scientific Officer G Head, Distributed Digital Control Systems Section, Indira Gandhi Centre for Atomic Research, Kalpakkam** who guided us to make our project success with lot of effort and gratitude.

We thank our project supervisor **Dr.R.J.KAVITHA M.E.,Ph.D.,** for this timely help, valuable suggestions, guidance and moral support and sustained interest in completing the project work successfully.

We thank with genuine conscious to our entire department teaching faculty and non-teaching staff for their valuable suggestion and moral support to build up our career.

Our thanks and appreciations also go to our parents and friends in developing the project and people who have willingly helped us out with their abilities.

ABSTRACT

This project focuses on detecting anomalies in network traffic using a rule-based detection algorithm centered around TCP/IP protocols. Anomalies in network behavior often indicate security threats such as intrusions, scanning, or data exfiltration. To analyze the network data, Wireshark is used to capture live packets and export them as packet capture files. These files are then parsed and examined using Python scripts. The core of the detection system is a set of predefined rules that identify abnormal patterns based on TCP/IP header information such as flags, ports, IP addresses, and packet sizes. For example, repeated SYN packets without ACKs can signal a SYN flood attack, while unexpected port access may indicate scanning attempts. The rules are crafted using common knowledge of network behavior and protocol standards. The system also supports user-defined rules, allowing customization for different network environments. The main goal is to create a lightweight, interpretable, and easy-to-deploy anomaly detection system. Unlike machine learning approaches, rule-based detection provides clear reasoning behind alerts. It is particularly useful for small organizations or academic environments. The project helps in understanding network protocols deeply and applying logical conditions to real-world traffic. Python's scapy and pyshark libraries are used to process packets and apply detection rules. Alert logs are generated and visualized for better interpretation. This approach ensures transparency in detection and quick response to abnormal activity. Overall, the system offers a practical, efficient, and educational tool for network anomaly detection using well-established rule logic.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

TCP	Transmission Control Protocol
IP	Internet Protocol
UDP	User Datagram Protocol
RST	Reset (TCP Flag)
SYN	Synchronize (TCP Flag)
ACKs	Acknowledgements (TCP Flag)
FIN	Finish (TCP Flag)
DDoS	Distributed Denial of Service
MITM	Man-In-The-Middle
PCAP	Packet Capture
Wireshark	Network Protocol Analyzer
Scapy	Python Packet Manipulation Library
PyShark	Python Wrapper for Wireshark
IDE	Integrated Development Environment
MTU	Maximum Transmission Unit
GUI	Graphical User Interface
MAC	Media Access Control
HTTP	Hypertext Transfer Protocol
CSV	Comma-Separated Values

CHAPTER 1

INTRODUCTION

In today's digital world, secure and reliable communication over computer networks is essential. Networks constantly exchange data through various protocols, especially the foundational TCP/IP suite. However, these networks are often vulnerable to anomalies, such as unauthorized access, data breaches, or unusual traffic patterns, which can disrupt services or compromise security. This project focuses on detecting such anomalies within network traffic using a rule-based detection approach. Instead of relying on complex machine learning techniques, it defines specific rules based on known patterns of normal and abnormal behavior. This method ensures transparency and control over detection logic. By using Wireshark, a powerful network protocol analyzer, to capture and inspect live or recorded packet data. Python is used to process this data, extract features, and apply detection rules. By analyzing fields like IP addresses, ports, flags, and packet sizes, it identify suspicious behavior such as SYN floods, port scans, or malformed packets. This system flags anomalies based on deviations from expected protocol behavior. This approach is lightweight, easy to implement, and suitable for small to medium networks. It helps network administrators understand potential threats in real-time. The rule-based method allows for quick customization based on network policies. Overall, this project demonstrates a practical way to enhance network security and monitoring without heavy computational resources. It also provides a learning platform for understanding TCP/IP protocols in depth. By combining Wireshark’s rich capture capabilities with Python’s scripting power, this system creates an efficient detection tool. This project emphasizes clarity, speed, and adaptability for practical anomaly detection. It lays the groundwork for more advanced intrusion detection systems in the future.

1.1 OBJECTIVE

The main objective of this project is to develop a simple and effective system for detecting anomalies in network traffic based on TCP/IP protocols. The system aims to identify unusual or suspicious patterns that may indicate security threats, such as port scans, SYN floods, or unauthorized access attempts. By using a rule-based detection algorithm, it can define clear conditions that represent normal and abnormal behavior in the network. This approach allows for easy customization and better control over the detection process. It will use Wireshark to capture and analyze live or offline network packets in detail. These packets will then be processed using Python to extract relevant information like IP addresses, TCP flags, port numbers, and packet lengths. The goal is to create a Python script that applies predefined rules to this data and flags any traffic that deviates from the expected patterns. This helps in monitoring network health and identifying potential intrusions early. Another objective is to make the system user-friendly and resource-efficient, suitable for educational use or small-scale networks. It also aims to improve the user's understanding of how TCP/IP protocols work and how common attacks can be identified. In this project want to provide a hands-on learning experience with real packet data. Additionally, this project serves as a base for extending the system into more complex intrusion detection tools in the future. The final outcome should be a working anomaly detection system that enhances network security through rule-based monitoring. It should also offer insights into traffic behavior and potential risks in real-time. Ultimately, this project promotes practical knowledge of cybersecurity and supports the development of smarter, more secure networks.

1.2 SCOPE OF THE PROJECT

The scope of this project includes designing and implementing a rule-based network anomaly detection system focused on analyzing TCP/IP traffic. It is limited to monitoring and analyzing packets captured through Wireshark and processed using Python scripts. The system will specifically focus on identifying common types of anomalies, such as unusual port activity, SYN flood attacks, and protocol misuse. It will operate at the packet level, inspecting header information and applying predefined rules to detect suspicious behavior. The project is targeted towards small to medium networks, such as in academic, lab, or organizational settings. It is not designed for high-speed enterprise-level monitoring but provides a foundation for future enhancements. The system will support both live capture and offline PCAP file analysis. It will not use machine learning or AI, keeping it simple, explainable, and easy to maintain. Custom rule creation and modification will be supported to adapt to different network environments. The project focuses on TCP, UDP, and IP layer protocols within the TCP/IP model. It does not cover application-layer threat detection such as HTTP or DNS attacks. The system is meant to assist network administrators, students, or cybersecurity beginners in understanding and detecting anomalies manually. This project does not include automatic blocking or alert systems but focuses on detection and reporting. Future work can expand the scope to include real-time dashboards, alerting mechanisms, or integration with intrusion detection systems. The tool will also include basic logging features to record detected anomalies for review. In addition, the system will be modular, allowing users to extend its functionality as needed. Overall, this project offers a practical and educational approach to network anomaly detection through rule-based methods.

1.3 MOTIVATION

The motivation behind this project is the growing need for securing computer networks against unauthorized access and cyber threats. As internet usage increases, so do the risks of attacks such as port scanning, SYN flooding, and unusual traffic patterns. Many networks, especially small organizations and educational institutions, lack the resources to implement complex security systems. This inspired us to develop a lightweight, understandable, and affordable solution. Rule-based detection offers a clear and simple way to identify anomalies without the need for advanced machine learning knowledge. It allows users to define exactly what kind of traffic should be considered abnormal. Using Wireshark helps us capture real network data, making the project practical and realistic. Python, being a powerful and beginner-friendly programming language, makes it easier to automate the analysis process. This project also motivated by the desire to better understand how TCP/IP protocols function and how attackers exploit them. It provides a hands-on learning experience in networking and cybersecurity. It also aims to bridge the gap between theoretical knowledge and real-world network monitoring. By creating a tool that is easy to modify and expand, this system encourage continuous learning and experimentation. The ability to detect threats early can help prevent damage and data loss. Ultimately, this project is driven by the goal of building a simple, effective, and educational system to improve network safety. it promotes open-source and accessible security solutions that can benefit a wider community. It encourages students and researchers to explore packet-level data in an interactive way. The system also lays a strong foundation for those who wish to later incorporate more advanced techniques. With minimal setup and cost, it demonstrates how security awareness and monitoring can be achieved effectively.

1.4 PROBLEM IDENTIFICATION

In today's digital environment, computer networks are constantly exposed to various types of threats and suspicious activities. One major problem is the difficulty in detecting unusual or malicious traffic in real-time, especially in small networks that lack advanced security systems. Many organizations rely on manual monitoring or expensive intrusion detection tools that are often complex and resource-heavy. This makes it hard for students, small businesses, or system administrators to identify early signs of attacks such as SYN floods, port scanning, or abnormal IP packet behavior. Most existing systems use machine learning, which requires large datasets and high computational power, making them less practical for beginners or lightweight environments. Another issue is the lack of understanding of how network protocols work at a low level, leading to poor analysis of traffic patterns. Without a proper detection mechanism, even small anomalies can lead to bigger network compromises. Therefore, there is a need for a simple, rule-based system that can analyze TCP/IP traffic and highlight suspicious patterns. Capturing raw packet data using tools like Wireshark is possible, but analyzing it manually is time-consuming and error-prone. A rule-based Python solution can automate this process and make it efficient. This project aims to solve these problems by building a user-friendly, transparent, and customizable anomaly detection system. It identifies real-time threats using basic network rules without needing machine learning. The key problem is how to detect and respond to anomalies in a practical, low-resource, and understandable way.

CHAPTER 2
LITERATURE REVIEW

1. Wawrowski et al. (2023) – "Anomaly Detection Module for Network Traffic Monitoring in Public Institutions"

This study introduces an anomaly detection module tailored for public institutions, emphasizing continuous monitoring of network traffic statistics to identify anomalies. The module operates in both online and offline modes, analyzing features such as packet counts and port flags. It employs statistical methods to detect deviations from normal traffic patterns without relying on predefined rules or machine learning models. The system's design allows for integration with existing security infrastructures, enhancing real-time threat detection capabilities. The authors report that their approach achieves high accuracy in identifying specific attacks, demonstrating its effectiveness in practical scenarios.

2. Ghosh et al. (2023) – "Anomaly Detection for Modbus over TCP in Control Systems Using Entropy and Classification-Based Analysis"

Focusing on industrial control systems, this study employs entropy-based analysis to detect anomalies in Modbus over TCP traffic. The authors analyze statistical deviations in network traffic features to identify potential intrusions, such as Denial of Service (DoS) and Man-in-the-Middle (MITM) attacks. The approach does not rely on machine learning models, instead utilizing statistical measures to establish baselines and detect deviations. Results indicate that while entropy analysis effectively detects certain types of attacks, its efficacy varies depending on the attack type and network conditions.

3. Alam et al. (2023) – "Data-Driven Network Analysis for Anomaly Traffic Detection"

This research presents a data-driven approach to anomaly detection, focusing on analyzing network traffic patterns without relying on predefined rules or machine learning models. By examining statistical properties and deviations in traffic data, the method identifies anomalies indicative of potential security threats. The study emphasizes the importance of understanding baseline network behavior to effectively detect deviations. Results highlight the approach's capability to detect various types of anomalies, including those resulting from DDoS attacks and unauthorized access attempts.

4. Shamim et al. (2023) – "Efficient Approach for Anomaly Detection in IoT Using System Calls"

Addressing the security challenges in IoT environments, this study introduces a host-based anomaly detection approach utilizing system call data. By modelling normal behavior through Markov chains, the method detects deviations indicative of anomalies. The approach is designed to be lightweight, making it suitable for resource-constrained IoT devices. Evaluations on public datasets show high detection accuracy and low false-positive rates, demonstrating its effectiveness in real-world IoT scenarios.

5. Anonymous (2023) – "An Approach for Anomaly Detection in Network Communications Using k-Path Analysis"

This paper introduces a novel method for anomaly detection based on k-path analysis in network communications. By modeling network interactions using a three-state Markov model, the approach identifies unusual of events that may signify malicious activity. The method focuses on detecting anomalies in communication paths, providing a more granular view of network behavior.

6. Duan et al. (2022) – "Network Traffic Anomaly Detection Method Based on Multi-Scale Residual Feature"

This paper proposes an anomaly detection method that captures multi-scale residual features in network traffic. By decomposing traffic data into different time spans using wavelet transform techniques, the method identifies anomalies based on residual patterns. The approach emphasizes the importance of analyzing long-term dependencies in traffic data to uncover potential anomalies. Experimental results demonstrate that this method outperforms traditional techniques in detecting anomalous network traffic, highlighting the value of multi-scale analysis in anomaly detection.

7. Irofti et al. (2022) – "Unsupervised Abnormal Traffic Detection through Topological Flow Analysis"

This research presents a method that leverages the topological structure of network flows to detect anomalies. By modeling network traffic as weighted directed graphs, the approach identifies unusual connectivity patterns indicative of malicious activity. The technique operates without the need for labeled data or predefined rules, making it suitable for unsupervised anomaly detection. Experiments conducted on real network traffic datasets demonstrate improvements over traditional statistical detectors, highlighting the potential of topological features in enhancing detection accuracy.

CHAPTER 3

EXISTING SYSTEM

3.1 INTRODUCTION

Network anomaly detection often uses several traditional, lightweight methods ideal for real-time monitoring with limited resources. Statistical methods analyze patterns in TCP/IP traffic—like packet size, frequency, and flag combinations—to define normal behavior, flagging deviations like a spike in SYN packets as possible attacks. Entropy-based techniques detect randomness in elements like source IPs, where unusually high entropy may signal scanning or distributed attacks. Flow-based systems (e.g., NetFlow) summarize traffic between endpoints to identify volume or structure-based anomalies without inspecting individual packets. These methods are scalable and useful for spotting broad patterns. Protocol behavior monitoring checks if packet sequences follow proper TCP/IP logic, flagging violations like out-of-order flags as suspicious. Topological analysis models network communication as graphs to detect odd connection patterns like sudden spikes to a single node, often indicating scans or botnets. For IoT or small local networks, host-based systems track system calls on devices, spotting internal anomalies through process behavior using models like finite-state machines. These are efficient on low-power devices but hard to scale across many endpoints. Together, these techniques offer different strengths, from catching broad trends to low-level or internal threats.

3.2 BLOCK DIAGRAM

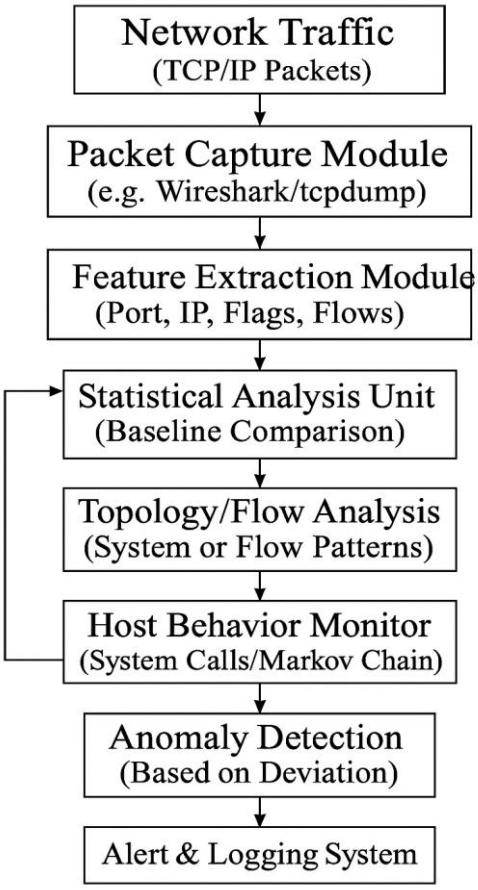


Fig 3.1 Block Diagram of Existing System

1. Network Traffic (TCP/IP Packets)

This block represents the raw network data flowing through a system or organization. It includes all types of packets transmitted over TCP/IP protocols, such as HTTP requests, DNS queries, SSH sessions, and others. The anomaly detection process begins by capturing this traffic to analyze and monitor potential threats or unusual patterns. The data at this stage is unfiltered and includes headers, payloads, IP addresses, port numbers, and control flags (e.g., SYN, ACK).

2. Packet Capture Module (e.g., Wireshark/tcpdump)

The packet capture module is responsible for collecting the actual network packets in real-time or from saved files (e.g., PCAP format). Tools like Wireshark or tcpdump are commonly used for this task. They capture detailed information about each packet, such as timestamp, source/destination IP, protocol, and port numbers. This captured data serves as the foundation for further analysis. It enables monitoring live traffic or reviewing historical network activity for anomalies. Without this step, no traffic data would be available for processing.

3. Feature Extraction Module (Port, IP, Flags, Flows)

After capturing the packets, the next step is to extract meaningful features that can be used for analysis. This includes source and destination IP addresses, TCP/UDP port numbers, packet size, flag combinations (e.g., SYN, ACK, FIN), flow direction, and duration. These features help to quantify and summarize the traffic behavior. Feature extraction simplifies raw packet data into a structured format that is easier to process. It ensures that only the relevant components are analyzed, improving the speed and accuracy of the detections

4. Statistical Analysis Unit (Baseline Comparison)

This module creates and maintains a statistical baseline of "normal" network behavior based on historical data. It monitors things like average packet sizes, connection counts per IP, frequency of flag combinations, and other metrics. Once this baseline is established, any major deviation—like a sudden surge in SYN packets—can be flagged as an anomaly. The statistical model is simple but effective for detecting brute-force attacks or DDoS attempts.

5. Entropy Calculation (Randomness Detection)

Entropy-based detection measures the randomness or unpredictability in certain features of the traffic, such as source IP addresses, destination ports, or payload content. A high entropy value might suggest a distributed scan or randomized attack, whereas low entropy can indicate repeated, targeted activity. Entropy analysis doesn't need prior knowledge or training and works well in detecting subtle irregularities. It is a lightweight and flexible technique that complements statistical analysis by focusing on variability and noise in the data.

6. Topology/Flow Analysis (Graph or Flow Patterns)

This module analyzes the structure of communication between hosts in thenetwork. It models connections as nodes and edges in a graph, helping to identify irregular patterns like sudden spikes in connections to one server (which could indicate scanning or a botnet command). Flow-based tools like NetFlow or IPFIX can be used here. Topological analysis is useful for understanding the broader structure of network activity and identifying attacks that may not be obvious from individual packets but are clear when observing multiple connections over time.

7. Host Behavior Monitor (System Calls/Markov Chain)

Instead of analyzing network packets, this block focuses on behavior inside devices or hosts. It uses system call sequences (like file reads, writes, or network accesses) to model how applications behave. Markov models or state machines can identify when a process performs actions outside of its usual pattern.

8. Anomaly Detection (Based on Deviation)

This is the core decision-making block. It aggregates all the insights from statistical, entropy, topological, and host-based modules to decide if the behavior observed is anomalous. An alert is raised if the behavior significantly deviates from the baseline, entropy thresholds, or flow structure. Since this system doesn't use rules or AI/ML, all decisions are based on numeric deviations and patterns. It ensures simple but effective real-time detection, particularly for known types of threats or volumetric anomalies.

9. Alert & Logging System

Once an anomaly is detected, this final block generates alerts and logs them for review by security teams. The alert may include information such as the timestamp, source/destination IPs, nature of the anomaly, and affected ports. Logs are critical for auditing, forensic analysis, and compliance reporting. This system ensures that suspicious events are not missed and enables human analysts to investigate further. It serves as the bridge between automated detection and human response.

3.3 DISADVANTAGE

1. Limited Accuracy for Stealthy Attacks

Statistical and entropy-based methods often miss stealthy or low-and-slow attacks. These attacks behave like normal traffic and do not cause sudden spikes or unusual patterns, so they may not trigger alerts.

2. High False Positive Rate

Since these methods rely on deviations from statistical norms, legitimate but rare events (e.g., software updates or backups) may be flagged as anomalies. This results in many false alarms that waste analysts' time.

3. Lack of Deep Packet Inspection

These systems typically analyze headers and flow data, not the full content of packets. As a result, they might fail to detect threats hidden inside payloads or application-layer attacks like SQL injection or XSS.

4. Static Thresholds and Baselines

Many systems rely on manually set thresholds or fixed baselines. If network behavior changes over time (like user growth or new services), the baseline becomes outdated, leading to inaccurate detection.

5. Poor Adaptability to New Attack Types

Because these systems don't use learning models, they cannot adapt or evolve to detect new or unknown attacks. They only detect anomalies based on predefined features or known deviations.

6. Resource Intensive for Large Networks

While simple in concept, continuously monitoring traffic and calculating statistical or entropy metrics for large-scale environments can become resource-heavy and slow.

7. Limited Protocol Awareness

Some techniques do not deeply understand protocol behaviors and can overlook violations or misuse in complex protocols (e.g., SSH tunneling or DNS tunneling attacks).

8. No Contextual Understanding

These systems do not understand the context behind traffic such as time of day, user identity, or device type , so they treat all traffic equally, which reduces the effectiveness of detection.

9. Inadequate for Encrypted Traffic

As more data is encrypted (e.g., HTTPS, VPNs), statistical and flow-based methods become less effective because they cannot analyze the packet content, where some attacks may be hidden.

10. Manual Configuration and Maintenance

Maintaining baselines, adjusting thresholds, and tuning the system requires manual effort. This can be time-consuming and prone to error, especially in dynamic or complex networks.

CHAPTER 4

NETWORKING ANOMALY DETECTION USING TCP/IP PROTOCOLS

4.1 INTRODUCTION

The proposed system aims to enhance anomaly detection in TCP/IP networks by implementing a **rule-based detection algorithm** using **Wireshark** for packet capture and **Python** for traffic analysis. Unlike traditional methods that rely only on statistical patterns or require complex machine learning models, this system focuses on predefined rules derived from known protocol behaviors and attack signatures. These rules can be crafted to detect specific anomalies such as SYN flood attacks, port scans, and protocol violations by inspecting packet fields like IP addresses, TCP flags, and port numbers. Wireshark will be used to capture real-time network traffic or load packet data from .pcap files, which will then be processed in Python. The Python script will apply a set of clearly defined rules to each packet or flow to identify suspicious behavior. This rule-based approach offers better transparency, ease of implementation, and lower computational overhead, making it suitable for small to medium-sized networks. Moreover, the system is customizable new rules can be added or existing ones modified based on emerging threats. Alerts will be generated when packets match suspicious criteria, and detailed logs will help network administrators investigate further. This setup ensures quick detection of common attacks without the need for training datasets or high-end hardware. By combining the power of Wireshark's detailed traffic visibility with the flexibility of Python scripting, the proposed system offers a practical, efficient, and maintainable solution for real-time network anomaly detection.

4.2 BLOCK DIAGRAM :

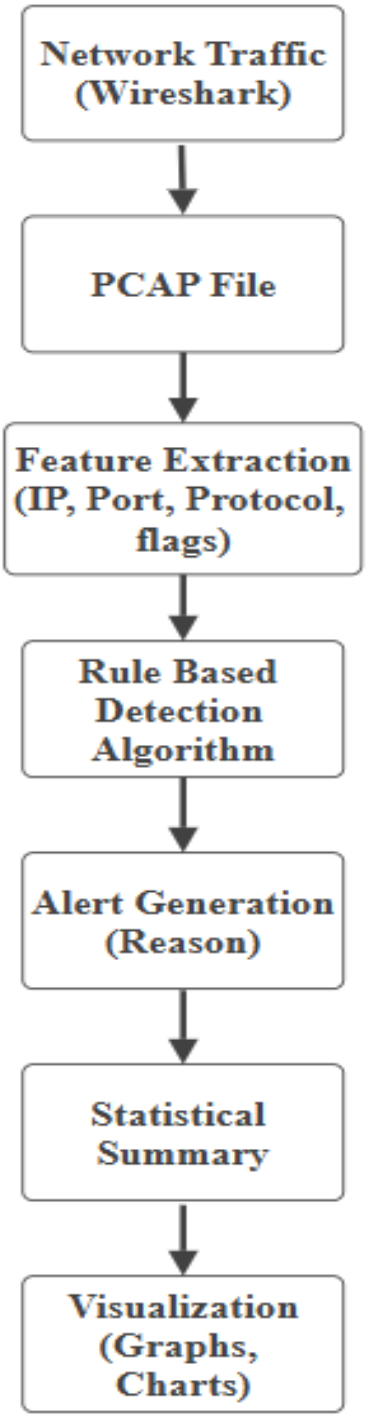


Fig 4.1 Block Diagram of Networking Anomaly Detection using TCP/IP

4.3 METHODOLOGY

1. Network Traffic (TCP/IP)

This step involves capturing all data packets traveling through a network. It includes both incoming and outgoing traffic using the TCP/IP protocol suite. The traffic may consist of web browsing, file transfers, remote logins, or other network activities. The goal is to analyze this traffic to detect suspicious or abnormal patterns. TCP/IP headers carry critical information used in later stages for anomaly detection.

2. Packet Capture (Wireshark)

Wireshark is used to sniff or capture live network traffic from the interface. It collects data in the form of packets and stores them in a .pcap (Packet Capture) file. It captures both payload and header information for each packet. The captured traffic includes details such as source and destination IPs, ports, protocols, flags, and timestamps. This step ensures all raw network activity is recorded for further analysis.

3. PCAP File / Live Feed

The captured traffic is either saved into a PCAP file or directly processed via a live feed. This acts as the input to the anomaly detection system. In offline mode, PCAP files allow analyzing past traffic; in online mode, live capture helps in real-time monitoring. Both modes offer flexibility for testing and deployment depending on use cases. This step passes data to the Python engine for analysis.

4. Packet Parsing & Preprocessing (Python with Scapy/pyshark)

Scapy or pyshark libraries in Python are used to read and parse the PCAP data. Preprocessing includes filtering unnecessary packets, converting raw binary fields into human-readable formats, and timestamp normalization. It helps clean and structure the data for easy analysis.

5. Feature Extraction (IP, Port, Protocol, TCP Flags, Size, Time)

From each packet, important features are extracted such as source/destination IPs, ports, protocols used (TCP, UDP), TCP flags (SYN, ACK, RST), packet size, and timestamps. These features are essential for detecting anomalies. For example, RST flags help detect connection resets, and port numbers help spot scans. Structured data from this step feeds into the rule-based engine.

6. Rule-Based Detection Engine (If-Else Rules for RST, SYN Floods, Duplicates,etc.)

This engine applies predefined rules to the extracted features. If-else logic is used to detect known anomalies like repeated RST packets, SYN floods (many SYNs without ACKs), or duplicate ACKs. For example, if SYNs exceed a threshold within a short time frame, it flags a potential SYN flood. This lightweight engine is efficient and doesn't require training data like ML models.

7. Anomaly Classification (RST, Duplicate ACK, SYN Scans, etc.)

Once an anomaly is detected, it is classified into types such as "Connection Reset," "Duplicate ACK," or "SYN Scan." This classification helps in understanding the nature and severity of the anomaly. Each anomaly type has different security implications—RSTs may indicate abnormal terminations, while SYN floods may signal denial-of-service attacks. Proper labeling supports detailed analysis and visualization.

8. Is Anomaly Detected?

This is the decision-making node. If any rule is triggered, the packet is marked as anomalous; otherwise, it is considered normal. This conditional check directs the system toward either alert generation or simply labeling the traffic as begin. It helps to filter out noise and reduce false positives in large datasets. Only anomalies proceed to logging and alerting stages.

9. Alert Generation & Logging

If an anomaly is detected, the system logs detailed information such as the packet line number, anomaly type, and source-destination IPs. Optionally, alerts can be sent to a dashboard or file for real-time monitoring. This ensures suspicious activity is recorded for forensic and audit purposes. These logs help network administrators take action quickly.

10. Mark as Normal Traffic

Packets that do not trigger any rule are marked as normal. These are not logged as anomalies and can be ignored or stored for baseline comparison. This step helps keep the system lightweight and focused only on abnormal behavior. It also contributes to improving future detection by maintaining a profile of legitimate traffic.

11. Statistical Summary (Packet Count, Type Counts, Protocol Breakdown)

The system generates overall statistics summarizing total packets, TCP-specific packets, and counts for each anomaly type. It may also break down traffic by protocol, size, and frequency. This helps in quantifying the network health and anomaly density. These statistics serve as useful insights for administrators to monitor network trends over time.

12. Visualization (Graphs, Charts using Matplotlib/Seaborn)

Finally, the results are visualized using bar charts, pie charts, or timelines. Tools like Matplotlib and Seaborn in Python are used to plot anomalies over time, protocol usage, and type distributions. Visual representations make it easier to interpret data patterns, identify peak anomaly times, and present reports to stakeholders. This improves overall situational awareness.

4.4 ADVANTAGES

1. Lightweight and Fast Detection

Rule-based systems are computationally efficient and don’t require heavy resources. They can quickly process network packets and detect known patterns of anomalies with minimal delay, making them ideal for real-time or near real-time detection.

2. No Need for Training Data

Unlike machine learning models, this system doesn't require large datasets or training. It operates using predefined rules and logic, which reduces complexity and makes the system easier to deploy and maintain.

3. Transparent and Explainable Logic

Each detection rule is based on clear conditions (e.g., multiple SYNs without ACKs = possible SYN flood), which makes the system fully explainable. This transparency helps administrators understand why an alert was generated.

4. Easy Integration with Wireshark and Python Tools

The use of widely supported tools like Wireshark, Scapy, and pyshark ensures easy packet capture and parsing. Python offers rich libraries for parsing, logic, and visualization, making development smooth and modular.

5. Customizable and Flexible Rule Design

You can easily modify or add new rules based on emerging threats or specific network behaviors. This flexibility allows the system to be tailored to different environments, such as enterprise networks, IOT, or educational setups.

6. Useful for Forensics and Reporting

The system includes logging, classification, and visualization, which helps in documenting suspicious activities for future analysis. The statistical summaries and graphs make it easier to present results to non-technical stakeholders.

7. Cost-Effective Solution

Since it uses open-source tools and doesn't require proprietary software or machine learning infrastructure, it's a low-cost solution suitable for academic, small business, or research projects.

8. Real-Time and Offline Analysis Capabilities

You can analyze both live traffic and offline PCAP files, giving flexibility in use case such as live monitoring in production or forensic analysis in lab settings.

9. Protocol-Specific Detection

By analyzing TCP/IP flags and packet behavior, it offers precise anomaly detection at the transport layer, which is critical for identifying low-level attacks such as scans, resets, and floods.

10. Educational and Practical Value

This project offers great learning opportunities in network protocols, cybersecurity, traffic analysis, and Python programming. It also demonstrates how real-world network threats are detected practically.

CHAPTER 5

RULE BASED DETECTION ALGORITHM

5.1 INTRODUCTION

The algorithm used in this project is a **rule-based detection algorithm**, specifically designed to identify anomalies in TCP/IP network traffic by applying a set of predefined rules. This approach relies on analyzing captured packets, using tools like **Wireshark**, and checking for suspicious patterns, behaviors, or protocol violations based on known network threats. The algorithm inspects various TCP/IP fields such as source/destination IP addresses, port numbers, TCP flags (SYN, ACK, FIN, RST), sequence numbers, and packet frequency. For example, a rule may flag a SYN flood if a high number of SYN packets are sent without corresponding ACK responses. Another rule might detect port scanning by identifying rapid connection attempts to multiple ports on a single host. These rules are coded in Python and applied to either live traffic or stored PCAP files. The strength of rule-based algorithms lies in their clarity, ease of implementation, and ability to catch well-known attack patterns with minimal computing resources. This makes them ideal for smaller networks or educational environments where real-time detection is needed without complex machine learning models. However, they require frequent updates to stay effective against evolving threats. Overall, the algorithm provides a structured and efficient way to monitor and detect network anomalies based on logical conditions and packet behaviors.

5.2 RULE BASED DETECTION ALGORITHM

The rule-based detection algorithm works by defining a set of logical rules that describe abnormal or suspicious network behaviors. These rules are applied to TCP/IP packet data captured from the network using tools like **Wireshark** or **Python libraries such as Scapy or PyShark**. Each rule checks specific packet attributes such as TCP flags, IP addresses, port numbers, and packet frequency. For example, if a large number of SYN packets are received without corresponding ACK responses, it could indicate a SYN flood attack. These rules are predefined and based on known attack patterns .The algorithm runs in real-time or on saved PCAP files to analyze and detect threats without needing any training or AI models. It is simple, fast, and highly interpretable, making it suitable for small to medium network environments.

Rule-Based Detection Algorithm – Predefined Rules

1. Duplicate ACKs

- **Rule:** Detect 3 or more identical ACKs in sequence from the same source.
- **Wireshark Filter:** tcp.analysis.duplicate_ack
- **Reason:** Frequent duplicate ACKs often signal network congestion.
- **Purpose:** Alerts you to know possible packet loss or attempted manipulation like MITM

2. Retransmissions

- **Rule:** Same TCP segment resent multiple times.
- **Wireshark Filter:** tcp.analysis.retransmission
- **Reason:** Retransmissions suggest poor network quality, packet drops.
- **Purpose:** Helps detect connection instability or spoofing attempts.

3. Connection Resets (RST Attacks)

- **Rule:** Spike in TCP RST packets from one IP.
- **Wireshark Filter:** tcp.flags.reset == 1
- **Reason:** Abrupt connection resets may indicate scanning, crashing attempts, or reset floods.
- **Purpose:** Identifies attackers trying to break or hijack sessions.

4. SYN Floods / Scans

- **Rule:** More than 100 SYNs with no ACKs in a short window (10 seconds).
- **Wireshark Filter:** tcp.flags.syn == 1 && tcp.flags.ack == 0
- **Reason:** Flooding servers with half-open TCP connections is a classic DoS tactic.
- **Purpose:** Detect denial-of-service or stealth scanning behavior.

5. Large Packets

- **Rule:** TCP/IP packets larger than 1500 bytes.
- **Wireshark Filter:** frame.len > 1500
- **Reason:** Oversized packets may attempt to overflow buffers or carry hidden payloads.
- **Purpose:** Protects systems from large payload injection or buffer overflow attacks.

6. Malformed Packets

- **Rule:** Invalid TCP flag combinations (e.g., SYN+FIN), bad checksums.
- **Wireshark Filter:** tcp.flags.syn == 1 && tcp.flags.fin == 1 or ip.checksum.bad == 1
- **Reason:** Malformed packets may be crafted for evasion or crashing target systems.
- **Purpose:** Detects suspicious or non-standard traffic likely caused by an attacker.

7. Protocol Violations

- **Rule:** Improper TCP flag sequences (e.g., data before SYN).
- **Wireshark Filter:** Use TCP stream analysis or sequence logic.
- **Reason:** Violations imply that protocols are being misused intentionally.
- **Purpose:** Identifies rogue clients or crafted packets violating TCP/IP specs.

8. High Frequency Flows

- **Rule:** Over 100 flows initiated per second from one source IP.
- **Wireshark Technique:** Use conversation statistics (→ Statistics > Conversations).
- **Reason:** Excessive flow rates suggest botnets, brute force, or port scans.
- **Purpose:** Identifies IPs generating suspicious volumes of connections.

CHAPTER 6

ARCHITECTURE DIAGRAM

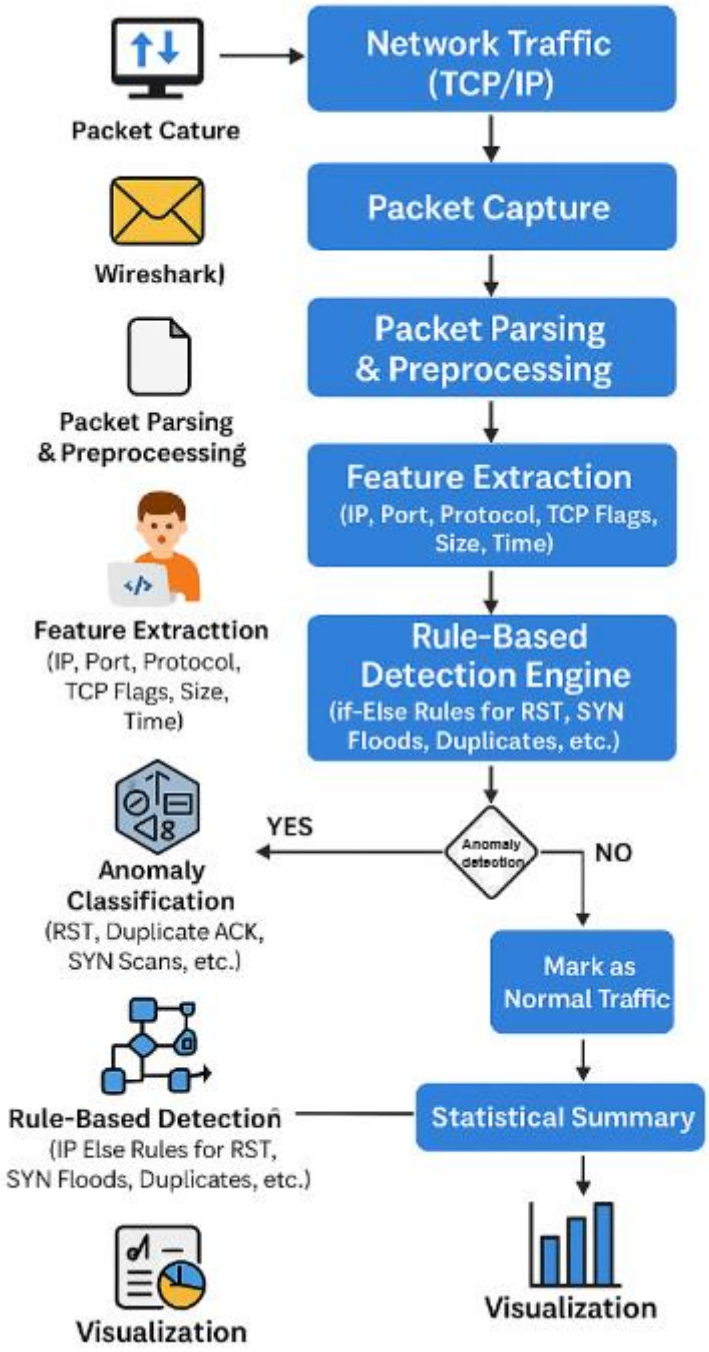


Fig 6.1 Architecture diagram of Networking Anomaly Detection

1. Network Traffic (TCP/IP)

This is the raw internet or internal network data flowing between connected devices using the TCP/IP protocol. It includes all types of packets such as HTTP, FTP, DNS, and more, representing real-time communications. The data can include legitimate traffic as well as potentially harmful anomalies like SYN floods, port scans, or data exfiltration attempts. Capturing this traffic is the first step in any network anomaly detection system. The quality and scope of this captured data directly influence the effectiveness of the later analysis. Monitoring this layer is crucial for identifying both usage patterns and threats.

2. Packet Capture

Packet capture tools like Wireshark, tcpdump, or tshark intercept and record the data packets moving across the network. These tools enable the creation of PCAP files, which store detailed information about each packet, including headers, payloads, timestamps, and protocol types.It must be performed efficiently and accurately to ensure no data is missed. High-quality packet capture ensures that anomalies, even if rare or subtle, are preserved for inspection. It forms the backbone of network forensics and intrusion detection.

3. PCAP File / Live Feed

This component represents the storage or streaming of captured packets either in saved PCAP files or directly through a live data feed. PCAP files are useful for retrospective analysis, while live feeds are critical for real-time monitoring and alerting. This modularity allows flexibility enabling both offline investigation and online prevention. This data is passed forward for deeper inspection and preprocessing. Proper parsing and extraction at this stage help avoid inefficiencies downstream.

4. Preprocessing (Python)

In this step, the raw packet data is cleaned, parsed, and organized into a structured format using Python libraries like Scapy or PyShark. This involves removing unnecessary fields, normalizing timestamp formats, extracting relevant features (such as IP addresses, ports, protocol types, and TCP flags), and potentially anonymizing sensitive data. The goal is to convert raw packet logs into a usable form for detection engines and visualizations. Efficient preprocessing directly impacts the accuracy and speed of anomaly detection.

5. Rule-Based Detection Engine

This engine uses predefined rules and logic (typically "if-else" conditions) to identify suspicious behavior patterns such as SYN flood attacks, duplicate ACKs, or abnormal RST flags. These rules are often based on well-known network attack signatures or RFC protocol standards. It is a deterministic and transparent approach that is easy to understand and customize. The engine acts as the core security mechanism in rule-based anomaly detection systems.

6. Anomaly Classification

Once potential anomalies are detected, this module classifies them into types such as SYN scans, duplicate packets, RST floods, etc. This helps in prioritizing alerts and deciding on further actions, such as blocking traffic or alerting admins. Classification adds context and specificity, helping stakeholders understand the severity and nature of a threat.

7. Statistical Summary

This module compiles and summarizes data from the analyzed traffic, including total packet counts, counts per protocol, attack frequencies, source/destination trends, and other useful metrics. It provides a high-level overview of network activity and can reveal patterns or anomalies over time.

8. Visualization

The final output is converted into graphical representations using tools like Matplotlib or Seaborn. These visualizations may include bar graphs, line charts, heatmaps, and pie charts to display trends, distributions, and anomalies clearly. Visualization aids in better understanding, faster analysis, and more effective communication with non-technical stakeholders. It transforms numeric data into intuitive formats that reveal insights at a glance. This step enhances reporting, debugging, and education in cybersecurity. Visualization is critical for continuous monitoring dashboards and executive summaries.

CHAPTER 7

SYSTEM REQUIREMENTS

7.1 SOFTWARE REQUIREMENTS

1. Operating System

- **Alternatives**: Windows 10/11 (with admin/root access for packet capture)

2. Python

- **Version**: Python 3.8 or higher
- **Reason**: Used for packet parsing, feature extraction, and implementing rule-based detection logic

3. Python Libraries

- scapy – For packet sniffing and parsing PCAP files
- pyshark – Wrapper for wireshark for PCAP parsing (alternative to Scapy)
- pandas – For feature structuring and tabular data analysis
- numpy – For numerical processing (if needed in summarization)
- matplotlib / seaborn – For data visualization and graphing

4. Packet Capture Tools

- **Wireshark** – For GUI-based packet capturing and protocol analysis

5. Development Environment

- **IDE/Text Editor**: Jupyter Notebook

6. Documentation Tools

- Jupyter Notebook – For combining code, output, and documentation

7.2 INTRODUCTION

In any software development project, especially one focused on networking anomaly detection, defining clear software requirements is essential to ensure smooth development, testing, and deployment. The software requirements for this project are carefully selected to support packet capture, traffic analysis, feature extraction, rule-based anomaly detection, and visualization of network behavior. The operating system forms the base, with Windows 10/11 or macOS being suitable platforms, provided administrative access is available for capturing packets. Python, being a versatile and widely-used programming language, is chosen for scripting the detection logic and handling packet data. The project makes extensive use of Python libraries such as Scapy and Pyshark for packet parsing, while Pandas and NumPy support data processing and analysis tasks.

For visualization and understanding traffic patterns, libraries like Matplotlib and Seaborn are employed to generate graphs and charts. Logging, an essential part of the system, is handled using Python's built-in logging module to record anomalies for review. Packet capture tools like Wireshark are used to obtain network traffic either in real-time or from stored PCAP files. The development environment is set up using Jupyter Notebook, which allows interactive coding along with inline outputs and documentation, making analysis more intuitive. A virtual environment tool such as venv or virtualenv is also recommended to isolate project dependencies and avoid conflicts. These software components work together to provide a lightweight, flexible, and efficient environment to implement rule-based anomaly detection without relying on heavy or complex frameworks. The combination of these tools ensures that the system remains easy to maintain, expand, and adapt for future use.

7.3 KEY FEATURES

1. Operating System

The operating system serves as the foundational platform where the entire project runs. Windows 10/11 or macOS are recommended because they support the tools needed for network analysis and have stable driver support. These systems also allow user-friendly interfaces and easy installation of software packages. Admin or root access is required to capture network packets, as packet sniffing tools need elevated privileges. The OS also handles memory, process, and I/O management, which is crucial when running data-intensive tasks like live packet capture. Compatibility with Python and network libraries is well-tested on these platforms. Users can utilize built-in terminal or command prompt utilities to run scripts. These systems support tools like Wireshark and virtual environments effectively. They also enable GUI-based and CLI-based network monitoring tools

2. Python (Version 3.8 or Higher)

Python is the core programming language used for developing the detection engine and processing packets. Its simplicity and vast ecosystem make it ideal for both scripting and data analysis tasks. Version 3.8 or above is preferred due to compatibility with newer libraries and features like assignment expressions and improved performance. Python enables writing compact yet powerful logic to detect anomalies using if-else rules. It supports both object-oriented and functional programming, giving flexibility in structuring code. Python has extensive community support and documentation, which helps troubleshoot issues quickly. The Python runtime is lightweight and cross platform, making it portable. Overall, Python streamlines data parsing, rule checking, and visualization tasks in this project.

3. Python Libraries

a. Scapy

Scapy is a powerful packet manipulation tool in Python that lets you sniff, parse, analyze, and build packets. It supports many protocols, especially TCP/IP, which is vital for this project. You can easily extract fields like IP, port, and flags from captured traffic. Scapy allows custom filtering and lets you define how packets are processed in real-time or from PCAP files. It helps detect anomalies like SYN floods by inspecting flag patterns. The syntax is simple and flexible for crafting rule-based logic. It's well-documented and supports advanced use cases like fuzzing or traceroute. Scapy is ideal for building low-level detection mechanisms without external dependencies. It's also fast and scriptable, making it great for automation. In this project, Scapy handles both live capture and offline PCAP parsing efficiently.

b. Pyshark

Pyshark is a Python wrapper for the Wireshark packet analysis engine (tshark). It provides a high-level interface to work with PCAP files or live traffic. Unlike Scapy, which reads raw bytes, Pyshark interprets packets using Wireshark's decoding, offering detailed protocol-level fields. It’s easier to use for high-level applications and doesn’t require manual parsing. Pyshark is suitable when you need to extract deep protocol insights (like DNS queries, HTTP payloads). It integrates well with pandas for structured analysis. Though slower than Scapy, its accuracy in decoding complex protocols is a key strength. Pyshark also handles malformed packets better in some cases. It’s a good alternative or supplement to Scapy when Wireshark-level parsing is needed.

c. Pandas

Pandas is a powerful data analysis library used for working with structured data like tables. In this project, it organizes packet features (IP, port, flags) into DataFrames for easy analysis. It supports operations like filtering, grouping, and sorting traffic data, helping identify abnormal patterns. With pandas, you can quickly generate summaries such as packet counts or protocol distributions. It integrates well with visualization tools and statistical libraries. Pandas helps transform raw data into meaningful features for detection. Its flexible API simplifies handling large datasets from PCAP files. For logging and reporting, pandas allows exporting results to CSV or Excel.

d. NumPy

NumPy is used for efficient numerical operations in Python. Though not always required, it's helpful for statistical analysis, such as computing averages or deviations in traffic size. It enables fast computation on arrays, which can improve performance when dealing with large packet traces. NumPy integrates well with pandas and is often used for summarizing numeric traffic features. It also offers random number generation and linear algebra functions, useful in simulation or testing scenarios. In this project, it supports backend data processing without slowing down the application. It's lightweight and compatiblewith most Python data tools. NumPy makes numerical data handling more efficient and readable.

e. Matplotlib / Seaborn

Matplotlib and Seaborn are Python libraries for visualizing data through graphs and charts. They help users understand network behavior trends, such as spikes in SYN packets or protocol distribution. Matplotlib is versatile and allows full customization of graphs, while Seaborn provides better aesthetics and plots

default. Together, they help build dashboards for anomaly reporting. These tools are essential for presenting findings in a clear and visual format. They support bar graphs, pie charts, heatmaps, and timelines. This visual analysis helps quickly spot irregular behavior in the network. It also supports saving figures as images or PDFs for reporting.

f. Logging

The built-in logging module in Python is used to record anomalies and system events. It helps track when and what kind of attack was detected, useful for auditing and reviewing incidents. Logs can be written to files with timestamps, severity levels, and messages. This adds transparency and traceability to the detection engine. It supports different log levels (INFO, WARNING, ERROR), helping organize alerts by importance. Logging can also be configured to send alerts to external systems or tools. It's lightweight, easy to set up, and essential for maintaining operational awareness in real-time.

4. Packet Capture Tool – Wireshark

Wireshark is a GUI-based tool used to capture and analyze live network traffic. It decodes packets at all layers of the TCP/IP stack, showing headers, flags, and payloads. For this project, it's used to generate PCAP files that are later processed by Python. Wireshark helps in validating captured traffic and understanding network flow visually. It supports filtering packets using expressions, making it easy to isolate anomalies. Tshark (its CLI version) can be used with Pyshark for scripting. Wireshark is highly reliable and supports hundreds of protocols. It also provides statistics and summaries built into its interface. It's an essential tool for both development and validation.

5. Development Environment – Jupyter Notebook

Jupyter Notebook provides an interactive coding environment where code, outputs, graphs, and documentation can coexist. It is especially helpful for data analysis and visualization tasks. Users can run code in small chunks (cells), which makes testing and debugging easier. It supports real-time feedback, letting you analyze packet behavior step-by-step. Jupyter supports Markdown for writing notes and explanations alongside code. It also integrates seamlessly with libraries like pandas, matplotlib, and seaborn. It's browser-based, meaning it works across platforms and requires no special IDE.

6. Virtual Environment Tools – venv / virtualenv

venv or virtualenv allows developers to create isolated Python environments for the project. This means dependencies and library versions won't interfere with other Python projects on the same system. It ensures consistent behavior across machines by freezing the required package versions. Developers can safely test and install new libraries without breaking the global Python setup. Virtual environments also support project-specific configurations.Setting up a virtual environment is simple and integrates well with IDEs like Jupyter.

7. Documentation Tool – Jupyter Notebook

Beyond development, Jupyter Notebook also serves as a documentation tool. It allows developers to include detailed explanations, charts, and results in the same file as the code. This makes the project more understandable for others, including future maintainers or evaluators. It's especially useful for presenting results in academic or professional settings. The combination of code, visualizations, and text helps in storytelling through data. For this anomaly detection project, Jupyter provides a central space for documenting detection logic, observations, and findings in a clean format.

CHAPTER 8

MODULES LIST

- Packet Capture Module
- Packet Parsing & Preprocessing Module
- Feature Extraction Module
- Rule-Based Detection Engine Module
- Anomaly Classification Module
- Alert Generation & Logging Module
- Statistical Summary Module
- Visualization Module
- User Interface / Execution Environment Module

8.1 MODULES DESCRIPTION

1. Packet Capture Module

This module is responsible for capturing real-time network traffic or loading previously recorded PCAP files. Tools like Wireshark, tshark, or tcpdump are used to collect data packets from the TCP/IP stack. It enables monitoring of both incoming and outgoing packets on specified network interfaces. Packet capturing ensures a raw data source for further analysis. The user can either capture live traffic or work with stored PCAP files. It must run with administrator/root privileges to access low-level network data. Proper filtering (like port/protocol filtering) can be applied at this stage to limit the data scope. This module forms the foundation for all subsequent analysis steps.

2. Packet Parsing & Preprocessing Module

Once the packets are captured, this module parses the raw data into structured form using Python libraries like Scapy or Pyshark. Parsing includes breaking down each packet into header fields and payload. The module filters out irrelevant or malformed packets to ensure clean input. Preprocessing also involves removing duplicates and handling missing or corrupted fields. The result is a clean dataset of network activity ready for feature extraction. This stage ensures accuracy and reduces noise. Efficient parsing improves performance of later modules. It's the bridge between raw network data and usable input for detection logic.

3. Feature Extraction Module

In this module, relevant attributes are extracted from the parsed packets. These include source and destination IPs, source and destination ports, protocol types (TCP/UDP), TCP flags (SYN, ACK, RST), packet size, and timestamp. These features are essential for analyzing connection behavior. The extracted data is organized using pandas into a tabular format. This structure makes it easier to apply rule-based logic later. Effective feature extraction is crucial for accuracy in detection. The module ensures all relevant behavioral indicators are captured. It simplifies complex packet data into meaningful attributes.

4. Rule-Based Detection Engine Module

This is the core of the anomaly detection system. It applies a set of if-else conditions or rule logic to detect suspicious patterns. For example, repeated SYN packets without corresponding ACKs may indicate a SYN flood. RST storms, duplicate ACKs, and abnormal flag combinations are also detected here. Rules can be written and modified based on network policy or threat types. This module does not require training data or models, making it simple and fast. It enables deterministic anomaly detection.

5. Anomaly Classification Module

After detection, anomalies are grouped into meaningful categories. These include SYN scans, RST storms, duplicate packets, and port scans. Classification helps in understanding the type and severity of threats. It makes the output more readable and actionable for users or analysts. This module ensures the detection results are not just binary alerts but informative labels. It also supports multiple anomaly types simultaneously. Well-labeled outputs are useful for further security response or automated actions. This module improves clarity and usability of the detection engine.

6. Alert Generation & Logging Module

When an anomaly is detected, this module creates alerts for the user. It logs each event with relevant information such as timestamp, anomaly type, involved IPs and ports. Logging is done to a file or console, which helps in auditing and tracing past incidents. This module ensures real-time visibility and permanent records. It may also trigger notifications or warnings based on severity. Logs can be analyzed later for trend analysis or forensic investigation. Clear, consistent alerting improves trust in the system. This module provides actionable insight from detection.

7. Statistical Summary Module

This module compiles general statistics about the analyzed traffic. It counts total packets, anomalies, and traffic breakdown by protocols or ports. It helps understand overall network behavior beyond just anomalies. These summaries offer context to the alerts. For instance, a high number of SYN packets can be suspicious only if unusual in volume. The statistics can be used for tuning rule thresholds. Regular summaries assist in monitoring and capacity planning. This module enhances the analytical value of the system.

8. Visualization Module

Here, data from the detection and statistics modules are transformed into graphs and charts. Libraries like matplotlib and seaborn are used to plot bar graphs, time series, pie charts, etc. Visualizations make it easy to spot trends, spikes, and anomalies. They help users interpret complex network behavior at a glance. Charts can be generated for anomalies by type, volume over time, or protocol distribution. The module improves user experience and understanding. It turns numeric data into actionable visual formats.

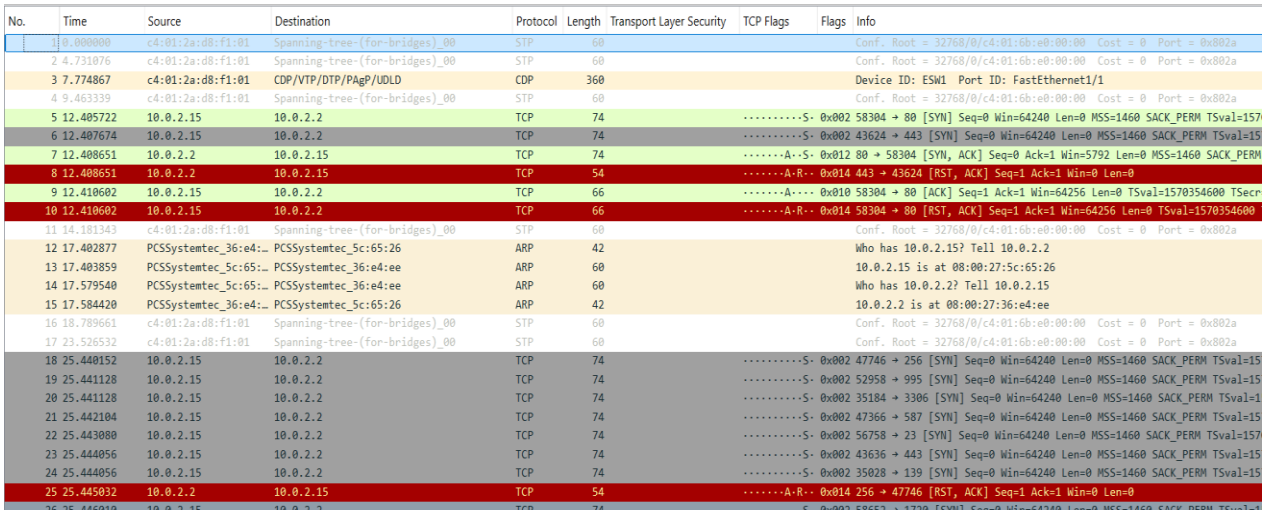
9. User Interface / Execution Environment Module

This module provides a working interface for running and interacting with the system. It can be a Jupyter Notebook or command-line tool that executes all other modules. The interface includes input options (PCAP file path, filter criteria) and displays outputs like logs and graphs. It supports step-by-step execution for learning and debugging. Jupyter Notebook also allows combining documentation with code, improving maintainability. This module is key for testing and user interaction. It wraps the backend logic into a usable frontend.

CHAPTER 9

RESULTS & DISCUSSIONS

9.1 RESULTS



No.	Time	Source	Destination	Protocol	Length	Transport Layer Security	TCP Flags	Flags	Info
1	0.000000	c4:01:2a:d8:f1:01	Spanning-tree-(for-bridges)_00	STP	60				Conf. Root = 32768/0/c4:01:6b:e0:00:00 Cost = 0 Port = 0x802a
2	4.731076	c4:01:2a:d8:f1:01	Spanning-tree-(for-bridges)_00	STP	60				Conf. Root = 32768/0/c4:01:6b:e0:00:00 Cost = 0 Port = 0x802a
3	7.774867	c4:01:2a:d8:f1:01	CDP/VTP/DTP/PAgP/UDLD	CDP	360				Device ID: ESW1 Port ID: FastEthernet1/1
4	9.463339	c4:01:2a:d8:f1:01	Spanning-tree-(for-bridges)_00	STP	60				Conf. Root = 32768/0/c4:01:6b:e0:00:00 Cost = 0 Port = 0x802a
5	12.405722	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	58304 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=157
6	12.407674	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	43624 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=15
7	12.408651	10.0.2.2	10.0.2.15	TCP	74		·····A··S·	0x012	80 → 58304 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 SACK_PERM
8	12.408651	10.0.2.2	10.0.2.15	TCP	54		·····A·R··	0x014	443 → 43624 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
9	12.410602	10.0.2.15	10.0.2.2	TCP	66		·····A····	0x010	58304 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1570354600 TSecr
10	12.410602	10.0.2.15	10.0.2.2	TCP	66		·····A·R··	0x014	58304 → 80 [RST, ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1570354600
11	14.181343	c4:01:2a:d8:f1:01	Spanning-tree-(for-bridges)_00	STP	60				Conf. Root = 32768/0/c4:01:6b:e0:00:00 Cost = 0 Port = 0x802a
12	17.402877	PCSSystemtec_36:e4:…	PCSSystemtec_5c:65:26	ARP	42				Who has 10.0.2.15? Tell 10.0.2.2
13	17.403859	PCSSystemtec_5c:65:…	PCSSystemtec_36:e4:ee	ARP	60				10.0.2.15 is at 08:00:27:5c:65:26
14	17.579540	PCSSystemtec_5c:65:…	PCSSystemtec_36:e4:ee	ARP	60				Who has 10.0.2.2? Tell 10.0.2.15
15	17.584420	PCSSystemtec_36:e4:…	PCSSystemtec_5c:65:26	ARP	42				10.0.2.2 is at 08:00:27:36:e4:ee
16	18.789661	c4:01:2a:d8:f1:01	Spanning-tree-(for-bridges)_00	STP	60				Conf. Root = 32768/0/c4:01:6b:e0:00:00 Cost = 0 Port = 0x802a
17	23.526532	c4:01:2a:d8:f1:01	Spanning-tree-(for-bridges)_00	STP	60				Conf. Root = 32768/0/c4:01:6b:e0:00:00 Cost = 0 Port = 0x802a
18	25.440152	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	47746 → 256 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=15
19	25.441128	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	52958 → 995 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=15
20	25.441128	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	35184 → 3306 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1
21	25.442104	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	47366 → 587 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=15
22	25.443080	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	56758 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=157
23	25.444056	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	43636 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=15
24	25.444056	10.0.2.15	10.0.2.2	TCP	74		··········S·	0x002	35028 → 139 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=15
25	25.445032	10.0.2.2	10.0.2.15	TCP	54		·····A·R··	0x014	256 → 47746 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Fig 9.1 Wireshark Dataset

```
📊 COMPREHENSIVE ANOMALY STATISTICS:
• Total Packets        : 11629
• TCP Packets          : 11345
• Duplicate ACKs       : 392
• Retransmissions      : 0
• Connection Resets    : 1258
• SYN Floods/Scans     : 2
• Large Packets        : 163
• Malformed Packets    : 0
• Protocol Violations  : 17
• High Frequency Flows: 0
```

Fig 9.2 Anomaly Statistics

```
🚨 DETAILED ANOMALY CLASSIFICATION:

▸ Connection Reset (RST) (1258 instances):
Abnormal Packet at Line 8:
  Type: Connection Reset (RST)
  Protocol: TCP
  Source: 10.0.2.2:443
  Destination: 10.0.2.15:43624
  Reason: Abrupt connection termination.
```

(a) Connection Reset

```
▸ Duplicate ACK (392 instances):
Abnormal Packet at Line 2152:
  Type: Duplicate ACK
  Protocol: TCP
  Source: 10.0.2.2:21
  Destination: 10.0.2.15:54696
  Count: 3
  Reason: Multiple ACKs (3) for same sequence number.
```

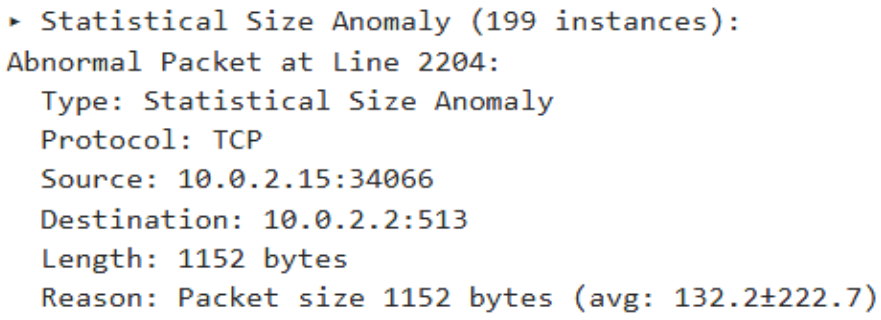
(b) Duplicate ACK

```
▸ Large Packet (163 instances):
Abnormal Packet at Line 2526:
  Type: Large Packet
  Protocol: TCP
  Source: 10.0.2.2:8180
  Destination: 10.0.2.15:35886
  Length: 1514 bytes
  Reason: Oversized packet (1514 > 1500 bytes).
```

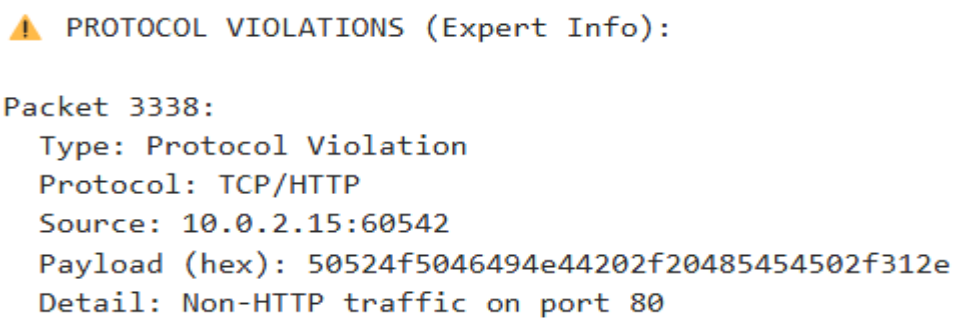
(c) Large Packet

```
▸ SYN Flood/Port Scan (2 instances):
Abnormal Packet at Line 20:
  Type: SYN Flood/Port Scan
  Protocol: TCP
  Source: 10.0.2.15
  Destination: 10.0.2.2:3306
  Count: 5
  Reason: Excessive SYNs (5) from single source.
```

(d) SYN Flood/Port



(e) Statistical Size Anomaly



(f) Protocol Violation

Fig 9.3 (a),(b),(c),(d),(e),(f) Detailed Anomalies

The output images present a detailed classification of various anomalies detected in TCP/IP traffic using a rule-based detection system. The system flagged 1,258 instances of Connection Resets (RST), which indicate abrupt terminations of TCP sessions, possibly due to scanning or forced disconnections. It also detected 392 cases of Duplicate ACKs, typically resulting from packet loss or retransmission attempts. There were 163 Large Packets with sizes exceeding 1500 bytes, which may indicate fragmentation or improper configurations. Two instances of SYN Flood or Port Scan activity were identified, where excessive SYN packets were sent from a single source, pointing to potential probing or DoS attempts. The system also noted 199 Statistical Size Anomalies, where packets significantly deviated from the average size, suggesting irregular payload patterns. Additionally, a Protocol Violation was flagged where non-HTTP data was sent over port 80, which is expected to carry only HTTP traffic .

1. Anomaly Type Distribution

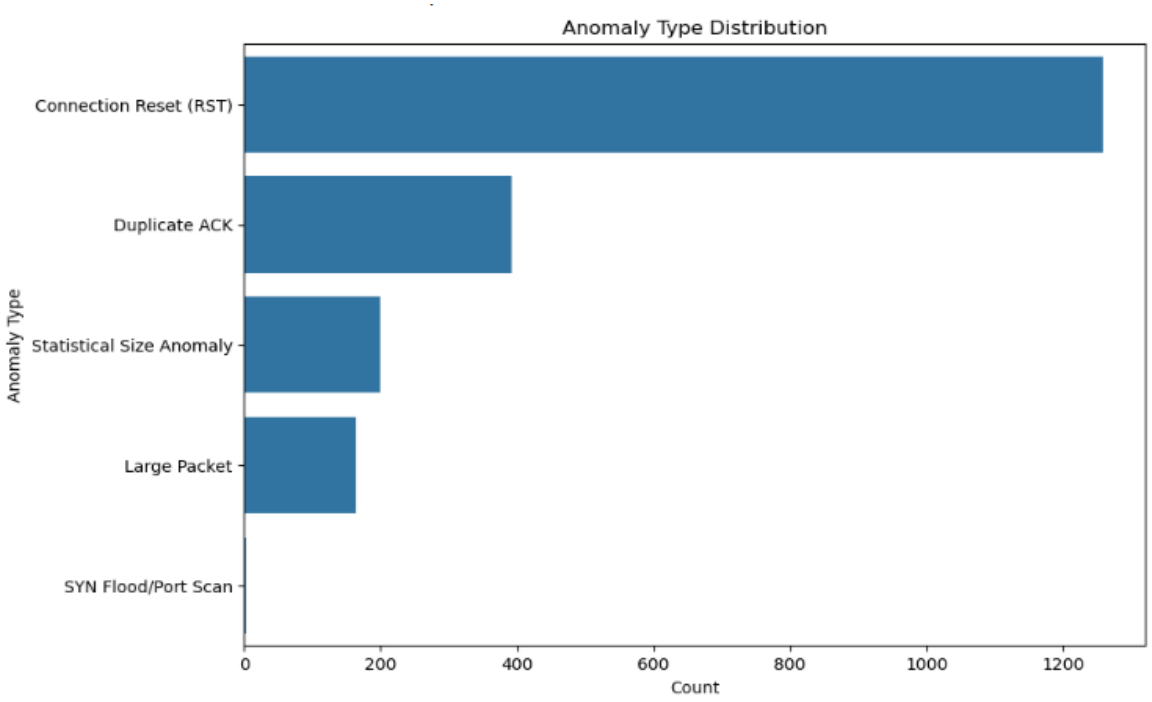


Fig 9.4.1 Anomaly Type Distribution

2. Protocol Distribution

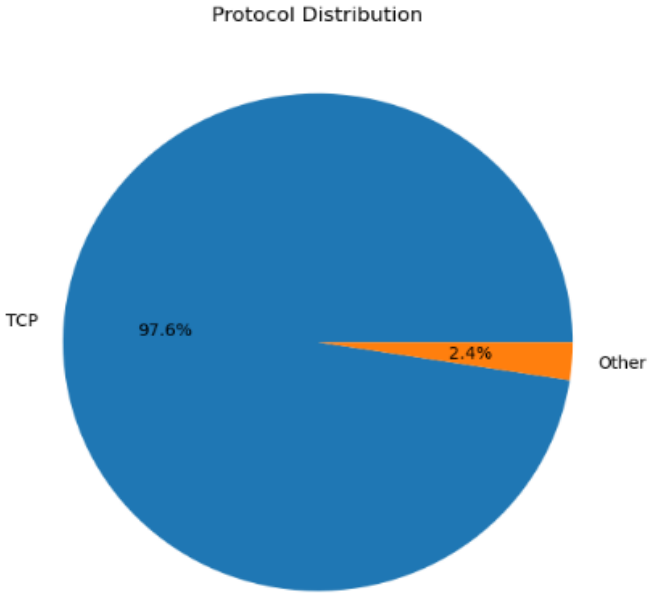


Fig 9.4.2 Protocol Distribution

3. Packet Size Distribution

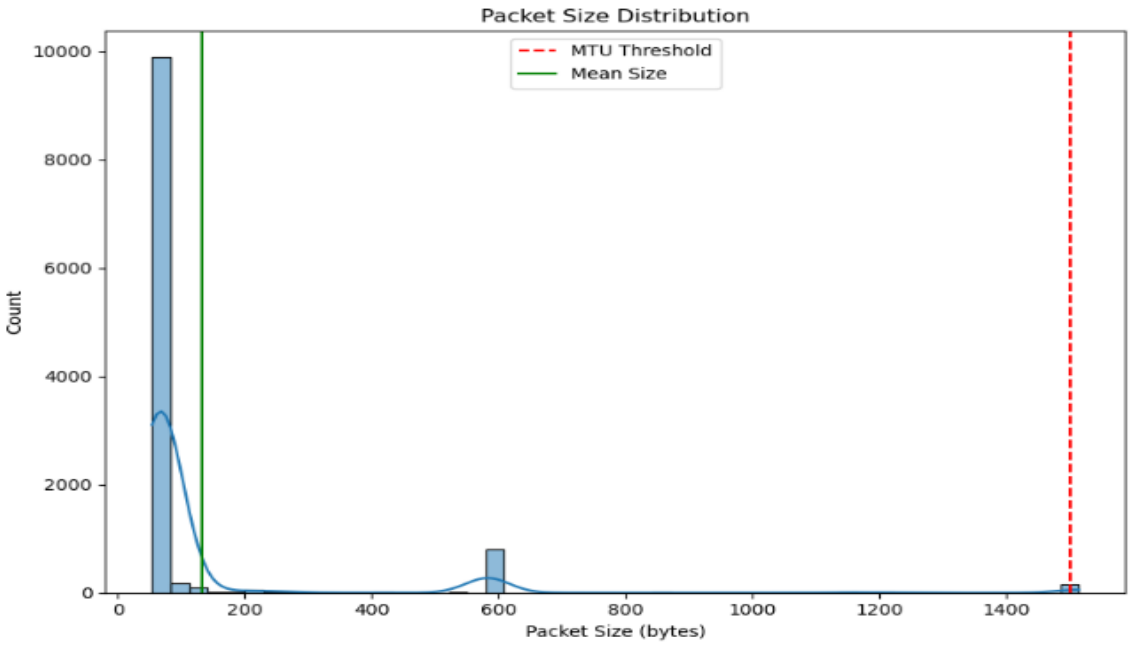


Fig 9.4.3 Packet Size Distribution

4. Anomaly Timeline

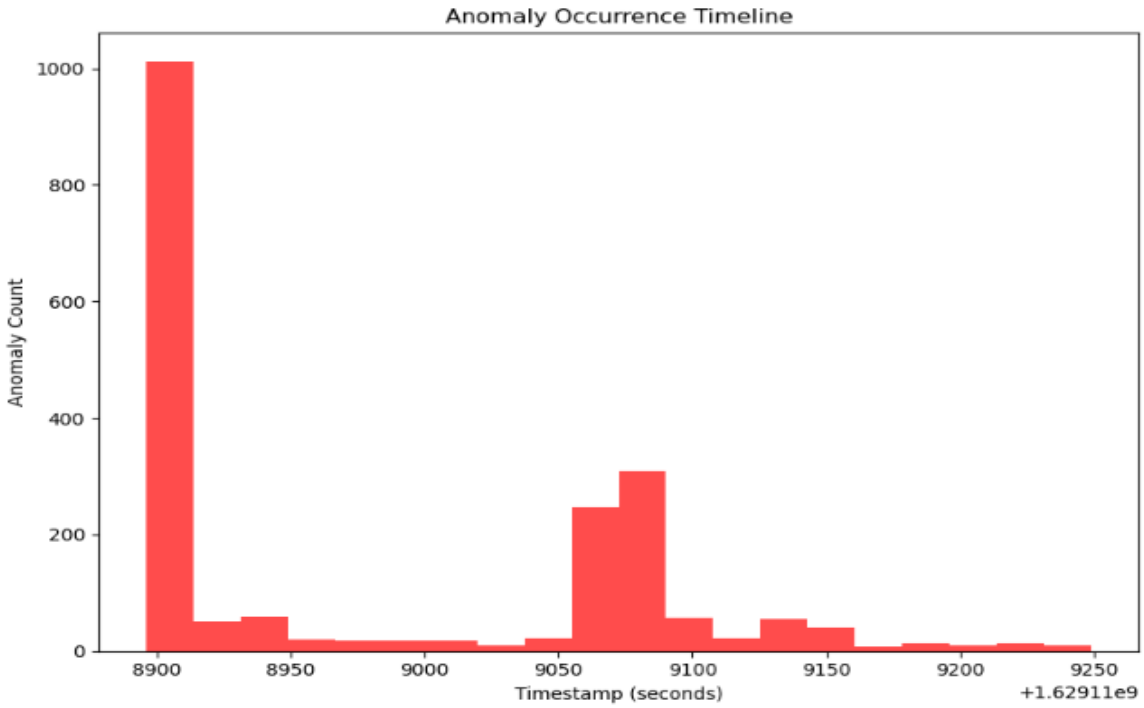


Fig 9.4.4 Anomaly Timeline

These four graphs explains different network anomalies. Figure 9.4.1 is a bar chart showing the number of each anomaly type. Connection Reset (RST) is the most common, followed by Duplicate ACKs and Large Packets. Figure 9.4.2 is a pie chart showing that 97.6% of the traffic is TCP, and only 2.4% is other protocols. Figure 9.4.3 is a histogram that shows most packets are small in size. Some packets are larger than the MTU limit of 1500 bytes, which may be abnormal. The red dashed line shows the MTU limit, and the green line shows the average size. Figure 9.4.4 is a timeline that shows when anomalies happened during the traffic capture. Most anomalies occurred in the beginning, then decreased over time These graphs help us understand how often and when the network faced issues. They also show which protocol and packet sizes were most common. This helps in finding unusual behavior and keeping the network secure.

9.2 DISSCUSSION

The analysis shows that most of the network traffic was TCP, making up 97.6% of all packets. The most common anomaly was Connection Resets, which may point to scanning or unstable connections. Duplicate ACKs were also frequent, suggesting possible packet loss. A smaller number of Large Packets and SYN Flood attempts were found, indicating potential fragmentation or probing attacks. Protocol Violations, like non-HTTP data on port 80, hint at misuse. Graphs showed that most packets were small and that anomalies mostly happened early in the capture. The timeline helps track when issues occurred. Overall, the system helped identify abnormal traffic patterns clearly.

CHAPTER 10

CONCLUSION & FUTURE ENHANCEMENT

10.1 CONCLUSION

This project successfully demonstrates a practical approach to detecting anomalies in network traffic using TCP/IP protocols through a rule-based detection system. By leveraging Wireshark for packet capture and Python for analysis, the system is able to identify various anomalies such as SYN floods, duplicate ACKs, and abrupt connection resets based on predefined rules. The use of Scapy or pyshark enables efficient packet parsing, while custom Python scripts automate the detection and classification of suspicious traffic. This method provides transparency, simplicity, and ease of implementation, especially for small to medium-sized networks. It does not rely on complex machine learning algorithms, making it resource-efficient and suitable for environments with limited computational power. The generated statistical summaries and visualizations also help network administrators understand network health and security trends. Overall, the rule-based approach offers an accurate, low-cost, and flexible solution for monitoring and detecting anomalies. It is also beneficial from an educational perspective, helping students and beginners grasp the fundamentals of network security and protocol analysis. The project proves that effective anomaly detection is achievable without relying on AI or big data frameworks, making it a good fit for academic and learning use cases.

10.2 FUTURE ENHANCEMENT

While the current system is efficient in detecting known anomalies using static rules, future improvements can make it even more powerful and adaptive. One key enhancement could be the integration of machine learning techniques to detect unknown or evolving threats that do not match predefined rules. This would allow the system to learn from new traffic patterns and adapt over time. Another improvement would be adding real-time alerts via email or dashboard notifications to allow faster incident response. The current detection can also be extended to include more complex protocol behaviors such as DNS tunneling or HTTP-based attacks. Additionally, integrating a dynamic rule-updating system based on threat intelligence feeds can enhance detection capabilities. Automation of packet capture and scheduled scans could also streamline operations for real-time monitoring. In terms of performance, the system could be optimized to handle high-speed traffic in larger networks. Adding a user-friendly GUI for rule management and visualization would improve accessibility. Lastly, incorporating log export features (e.g., CSV, JSON, PDF) and integration with security information and event management (SIEM) tools would make the system enterprise-ready. These enhancements would greatly expand the practical use cases of this project and move it toward a robust intrusion detection solution.

APPENDIX

```
# IMPORT SECTION
import pandas as pd                    # For data manipulation and analysis
from scapy.all import *                # For packet parsing and analysis
import matplotlib.pyplot as plt        # For data visualization
import seaborn as sns                  # For enhanced visualizations
from collections import defaultdict    # For dictionary with default values
import textwrap                        # For text formatting
import numpy as np                     # For numerical operations

# MAIN ANALYSIS FUNCTION
def analyze_pcap(pcap_file):
    print(f"\n🔍 Analyzing {pcap_file}...")
    # INITIALIZATION SECTION
    # Read PCAP file and initialize data structures
    packets = rdpcap(pcap_file)        # Read all packets from PCAP
    anomalies = []                     # Store detected anomalies
    expert_info = []                   # Store protocol-specific findings
    # Initialize statistics dictionary
    stats = {
        'total_packets': len(packets),  # Total packets in file
        'tcp_packets': 0,              # Count of TCP packets
        'duplicate_acks': 0,           # Duplicate ACKs detected
        'retransmissions': 0,          # Packet retransmissions
        'resets': 0,                   # TCP connection resets
        'syn_floods': defaultdict(int),  # SYN flood attempts
```

```
    'large_packets': 0,             # Packets exceeding MTU
    'malformed': 0,                 # Malformed packets
    'protocol_violations': 0,       # Protocol violations
    'unusual_frequency': 0          # High frequency flows
}
# Initialize tracking data structures
seq_tracker = defaultdict(list)     # Track sequence numbers
ack_tracker = defaultdict(list)     # Track ACK numbers
flow_tracker = defaultdict(list)    # Track network flows
packet_sizes = []                   # Store packet sizes for analysis
timestamps = []                     # Store packet timestamps
# Time window analysis variables
time_window = 1.0                   # 1-second analysis window
current_window_start = packets[0].time if packets else 0  # First packet time
window_counts = defaultdict(int)    # Count packets per flow per window

# PACKET PROCESSING LOOP
for i, pkt in enumerate(packets):
    pkt_num = i + 1                 # Packet number (1-based index)
    timestamps.append(pkt.time)     # Record timestamp
    # Skip non-IP/TCP packets
    if not (pkt.haslayer(IP) and pkt.haslayer(TCP)):
        continue
    # Extract packet layers and basic info
    ip = pkt[IP]
    tcp = pkt[TCP]
```

```
        stats['tcp_packets'] += 1
        length = len(pkt)
        packet_sizes.append(length)
        flow_key = (ip.src, ip.dst, tcp.sport, tcp.dport)  # Unique flow identifier
        # TIME WINDOW ANALYSIS
        if pkt.time - current_window_start > time_window:
            current_window_start = pkt.time
            window_counts.clear()
        window_counts[flow_key] += 1

        # DUPLICATE ACK DETECTION
        ack_key = (ip.src, ip.dst, tcp.ack)
        ack_tracker[ack_key].append(pkt_num)
        if 'A' in tcp.flags and len(ack_tracker[ack_key]) > 2:
            stats['duplicate_acks'] += 1
            anomalies.append({
                'packet': pkt_num,
                'type': 'Duplicate ACK',
                'protocol': 'TCP',
                'src': f"{ip.src}:{tcp.sport}",
                'dst': f"{ip.dst}:{tcp.dport}",
                'ack': tcp.ack,
                'count': len(ack_tracker[ack_key]),
                'reason': f"Multiple ACKs ({len(ack_tracker[ack_key])}) for same
sequence number."
            })
```

```
# RETRANSMISSION DETECTION
seq_key = (ip.src, tcp.sport, tcp.seq)
if 'PA' in tcp.flags and seq_key in seq_tracker:
    stats['retransmissions'] += 1
    anomalies.append({
        'packet': pkt_num,
        'type': 'Retransmission',
        'protocol': 'TCP',
        'src': f"{ip.src}:{tcp.sport}",
        'dst': f"{ip.dst}:{tcp.dport}",
        'seq': tcp.seq,
        'reason': f"Possible retransmission of sequence {tcp.seq}"
    })
seq_tracker[seq_key] = pkt_num
# CONNECTION RESET DETECTION
if 'R' in tcp.flags:
    stats['resets'] += 1
    anomalies.append({
        'packet': pkt_num,
        'type': 'Connection Reset (RST)',
        'protocol': 'TCP',
        'src': f"{ip.src}:{tcp.sport}",
        'dst': f"{ip.dst}:{tcp.dport}",
        'reason': "Abrupt connection termination."
    })
```

```
        # SYN FLOOD/PORT SCAN DETECTION
        if tcp.flags == 'S':
            stats['syn_floods'][ip.src] += 1
            if stats['syn_floods'][ip.src] == 5:
                anomalies.append({
                    'packet': pkt_num,
                    'type': 'SYN Flood/Port Scan',
                    'protocol': 'TCP',
                    'src': ip.src,
                    'dst': f"{ip.dst}:{tcp.dport}",
                    'count': stats['syn_floods'][ip.src],
                    'reason': f"Excessive SYNs ({stats['syn_floods'][ip.src]}) from
single source."
                })

        # LARGE PACKET DETECTION
        if length > 1500:
            stats['large_packets'] += 1
            anomalies.append({
                'packet': pkt_num,
                'type': 'Large Packet',
                'protocol': 'TCP',
                'length': length,
                'src': f"{ip.src}:{tcp.sport}",
                'dst': f"{ip.dst}:{tcp.dport}",
                'reason': f"Oversized packet ({length} > 1500 bytes)."
            })
```

```
# MALFORMED PACKET DETECTION
if len(pkt) < 20 or not pkt.haslayer(TCP) or not pkt.haslayer(IP):
    stats['malformed'] += 1
    anomalies.append({
        'packet': pkt_num,
        'type': 'Malformed Packet',
        'protocol': 'RAW',
        'length': len(pkt),
        'reason': "Invalid packet structure or size."
    })

# HTTP PROTOCOL VIOLATION DETECTION
if TCP in pkt and pkt[TCP].dport == 80 and Raw in pkt:
    if not (pkt[Raw].load.startswith(b'GET') or
            pkt[Raw].load.startswith(b'POST') or
            pkt[Raw].load.startswith(b'HTTP')):
        stats['protocol_violations'] += 1
        expert_info.append({
            'packet': pkt_num,
            'type': 'Protocol Violation',
            'protocol': 'TCP/HTTP',
            'src': f"{ip.src}:{tcp.sport}",
            'detail': "Non-HTTP traffic on port 80",
            'payload': pkt[Raw].load[:50].hex()
        })
```

```
        # HIGH FREQUENCY TRAFFIC DETECTION
        if window_counts[flow_key] > 100:
            stats['unusual_frequency'] += 1
            anomalies.append({
                'packet': pkt_num,
                'type': 'High Frequency Traffic',
                'protocol': 'TCP',
                'src': f"{ip.src}:{tcp.sport}",
                'dst': f"{ip.dst}:{tcp.dport}",
                'count': window_counts[flow_key],
                'reason': f"High packet rate ({window_counts[flow_key]}/sec) detected."
            })


    # STATISTICAL SIZE ANALYSIS
    if packet_sizes:
        avg_size = np.mean(packet_sizes)
        std_size = np.std(packet_sizes)
        for i, size in enumerate(packet_sizes):
            if abs(size - avg_size) > 3 * std_size:
                pkt_num = i + 1
                if pkt_num <= len(packets) and packets[pkt_num-1].haslayer(IP) and packets[pkt_num-1].haslayer(TCP):
                    ip = packets[pkt_num-1][IP]
                    tcp = packets[pkt_num-1][TCP]
                    anomalies.append({
                        'packet': pkt_num,
```

```
                'type': 'Statistical Size Anomaly',
                'protocol': 'TCP',
                'length': size,
                'src': f"{ip.src}:{tcp.sport}",
                'dst': f"{ip.dst}:{tcp.dport}",
                'reason': f"Packet size {size} bytes (avg: {avg_size:.1f}±{std_size:.1f})"
            })

    # HELPER FUNCTION FOR TEXT WRAPPING
    def print_wrapped(text, width=70):
        """Print text with wrapping to specified width."""
        print(textwrap.fill(text, width=width))
    # REPORT GENERATION SECTION
    print("\n📊 COMPREHENSIVE ANOMALY STATISTICS:")
    stat_items = [
        ('Total Packets', stats['total_packets']),
        ('TCP Packets', stats['tcp_packets']),
        ('Duplicate ACKs', stats['duplicate_acks']),
        ('Retransmissions', stats['retransmissions']),
        ('Connection Resets', stats['resets']),
        ('SYN Floods/Scans', len([a for a in anomalies if a['type'] == 'SYN Flood/Port Scan'])),
        ('Large Packets', stats['large_packets']),
        ('Malformed Packets', stats['malformed']),
        ('Protocol Violations', stats['protocol_violations']),
        ('High Frequency Flows', stats['unusual_frequency']) ]
```

```
for name, value in stat_items:
    print(f"• {name:<20}: {value}")
# ANOMALY DETAILS REPORT
print("\n🚨 DETAILED ANOMALY CLASSIFICATION:")
anomaly_types = set(a['type'] for a in anomalies)
for anomaly_type in sorted(anomaly_types):
    count = len([a for a in anomalies if a['type'] == anomaly_type])
    examples = [a for a in anomalies if a['type'] == anomaly_type][:3]
    print(f"\n▸ {anomaly_type} ({count} instances):")
    for ex in examples:
        print(f"Abnormal Packet at Line {ex['packet']}:")
        print(f"  Type: {ex['type']}")
        print(f"  Protocol: {ex['protocol']}")
        if 'src' in ex:
            print(f"  Source: {ex['src']}")
        if 'dst' in ex:
            print(f"  Destination: {ex['dst']}")
        if 'length' in ex:
            print(f"  Length: {ex['length']} bytes")
        if 'count' in ex:
            print(f"  Count: {ex['count']}")
        print_wrapped(f"  Reason: {ex['reason']}", 60)
# EXPERT PROTOCOL ANALYSIS REPORT
if expert_info:
    print("\n⚠️ PROTOCOL VIOLATIONS (Expert Info):")
    for info in expert_info[:5]:
```

```
            print(f"\nPacket {info['packet']}:")
            print(f"  Type: {info['type']}")
            print(f"  Protocol: {info['protocol']}")
            print(f"  Source: {info['src']}")
            print(f"  Payload (hex): {info['payload']}")
            print_wrapped(f"  Detail: {info['detail']}", 60)
    # VISUALIZATION SECTION
    plt.figure(figsize=(18, 12))
    # Subplot 1: Anomaly Type Distribution
    plt.subplot(2, 2, 1)
    anomaly_df = pd.DataFrame(anomalies)
    if not anomaly_df.empty:
        sns.countplot(y='type',
data=anomaly_df,
order=anomaly_df['type'].value_counts().index)
    plt.title('Anomaly Type Distribution')
    plt.xlabel('Count')
    plt.ylabel('Anomaly Type')
    # Subplot 2: Protocol Distribution
    plt.subplot(2, 2, 2)
    protocol_counts = {
        'TCP': stats['tcp_packets'],
        'Other': stats['total_packets'] - stats['tcp_packets']
    }
    plt.pie(protocol_counts.values(),
labels=protocol_counts.keys(), autopct='%1.1f%%')
```

```
    plt.title('Protocol Distribution')
    # Subplot 3: Packet Size Distribution
    plt.subplot(2, 2, 3)
    if packet_sizes:
        sns.histplot(packet_sizes, bins=50, kde=True)
        plt.axvline(1500, color='r', linestyle='--', label='MTU Threshold')
        plt.axvline(np.mean(packet_sizes), color='g', linestyle='-', label='Mean Size')
        plt.title('Packet Size Distribution')
        plt.xlabel('Packet Size (bytes)')
        plt.ylabel('Count')
        plt.legend()
    # Subplot 4: Anomaly Timeline
    plt.subplot(2, 2, 4)
if timestamps and anomalies:
        anomaly_times = [packets[a['packet']-1].time for a in anomalies if a['packet']-1 < len(packets)]
        plt.hist(anomaly_times, bins=20, color='red', alpha=0.7)
        plt.title('Anomaly Occurrence Timeline')
        plt.xlabel('Timestamp (seconds)')
        plt.ylabel('Anomaly Count')
    plt.tight_layout()
    plt.show()
# MAIN EXECUTION BLOCK
if __name__ == "__main__":
    analyze_pcap(r'C:\Users\mages\OneDrive\Documents\dataset2.pcap')
 # Update with your path
```

REFERENCES

1. PyShark Documentation. (2023). "PyShark: Python Wrapper for Tshark," https://github.com/KimiNewt/pyshark.
2. Wireshark Foundation. (2023). "Wireshark User Guide," https://www.wireshark.org/docs/.
3. Python Software Foundation. (2023). Python 3.10 Documentation, https://docs.python.org/3/.
4. Biondi, P. (2023). Scapy Documentation: Python-based Interactive Packet Manipulation Tool, https://scapy.readthedocs.io.
5. Jupyter Project. (2023). Jupyter Notebook Documentation, https://jupyter.org/documentation.
6. Waskom, M. L. (2021). "Seaborn: Statistical Data Visualization," Journal of Open Source Software, Vol. 6, No. 60, pp. 3021.
7. Hwang, K., and Chow, Y. (2012). Distributed and Cloud Computing, Morgan Kaufmann, Chapter 7: Network and Cloud Security.
8. Cichonski, P., Millar, T., Grance, T., and Scarfone, K. (2012). "Computer Security Incident Handling Guide," NIST SP 800-61 Revision 2.
9. Bhuyan, M. H., Bhattacharyya, D. K., and Kalita, J. K. (2014). "Network Anomaly Detection: Methods, Systems and Tools," IEEE Communications Surveys & Tutorials, Vol. 16, No. 1, pp. 303–336.
10. McKinney, W. (2010). "Data Structures for Statistical Computing in Python," Proceedings of the 9th Python in Science Conference.
11. Scarfone, K., and Mell, P. (2007). "Guide to Intrusion Detection and Prevention Systems (IDPS)," NIST SP 800-94.
12. Hunter, J. D. (2007). "Matplotlib: A 2D Graphics Environment," Computing in Science & Engineering, Vol. 9, No. 3, pp. 90–95.

13. Beale, J., and Caswell, B. (2004). Snort 2.1 Intrusion Detection, Syngress Publishing.
14. Northcutt, S., and Novak, J. (2002). Network Intrusion Detection, New Riders Publishing.
15. Zwicky, E., Cooper, S., and Chapman, D. (2000). Building Internet Firewalls, O'Reilly Media.