GUI BASED NETWORK PACKET SNIFFER

A Project report submitted

in partial fulfillment of requirement for the award of degree

BACHELOR OF TECHNOLOGY

in

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE

by

POTHANA NAGAVISHNU	**(2203A51189)**
LOKULA NARESH	**(2203A51367)**
KANKALA AKSHAY KUMAR	**(2203A51112)**
BHEEMALLA NIKHILESHWAR	**(2203A51545)**

Under the guidance of

Dr.Balajee Maram

Professor, School of CS&AI.

SR University, Ananthsagar,Warangal,Telagnana-506371

SR University

Ananthasagar, Warangal.

CERTIFICATE

This is to certify that this project entitled **"GUI BASED NETWORK PACKET SNIFFER**" is the bonafied work carried out by **P.NAGAVISHNU, L.NARESH, K.AKSHAY KUMAR, B.NIKHILESHWAR as a Project for the partial fulfillment to award the degree BACHELOR OF TECHNOLOGY in School of Computer Science and Artificial Intelligence during the academic year 2024-2025 under our guidance and Supervision.**

Dr.Balajee Maram
Professor
SR University
Anathasagar,Warangal

Dr. M.Sheshikala
Professor & Head,
School of CS&AI,
SR University
Ananthasagar, Warangal.

Reviewer-1
Name:
Designation:
Signature:

Reviewer-2
Name:
Designation:
Signature:

ACKNOWLEDGEMENT

We owe an enormous debt of gratitude to our Project guide **Dr.Balajee Maram, Professor** of the School of CS&AI , **Dr. M.Sheshikala, Professor** and Dean of the School of CS&AI, **Dr.Indrajeet Gupta Professor** for guiding us from the beginning through the end of the Capstone Project with their intellectual advices and insightful suggestions. We truly value their consistent feedback on our progress, which was always constructive and encouraging and ultimately drove us to the right direction.

We express our thanks to project co-ordinators **Mr. Sallauddin Md, Asst. Prof., and Dr.D.Ramesh Asst. Prof.** for their encouragement and support.

Finally, we express our thanks to all the teaching and non-teaching staff of the department for their suggestions and timely support.

CONTENTS

LIST OF FIGURES

ABSTRACT

The GUI-Based Network Packet Sniffer is a Python-driven application engineered to capture, inspect, and analyze live network packets through an intuitive graphical user interface. Utilizing powerful libraries such as Scapy for packet-level capture, Tkinter for GUI development, and threading for non-blocking performance, the tool offers users real-time visibility into network activity while maintaining smooth and responsive operation. This project blends core concepts of network security, system programming, and user-centered design, providing a valuable platform for hands-on learning and lightweight traffic monitoring.

The application allows users to select network interfaces, apply basic protocol filters (e.g., TCP, UDP, ICMP), and observe captured packets with key metadata such as source/destination IPs, protocol type, and port numbers. Each packet is dynamically displayed in a structured, scrollable table, making it easy to monitor and interpret traffic patterns. The interface supports automatic refresh and provides visual feedback on the capture state, improving usability for beginners and advanced users alike.

Designed with modularity and simplicity in mind, the tool captures packets without storing them, minimizing memory overhead and enhancing performance during continuous capture. Multithreading ensures that the GUI remains operational during packet sniffing, and administrative privilege checks help guide the user in configuring system-level access for proper functionality. During development, we explored low-level networking, privilege escalation mechanisms, and platform compatibility, especially on Windows-based systems.

This packet sniffer serves as an educational and practical utility for those learning about network layers, protocol behavior, and traffic inspection. While it does not perform deep packet analysis or intrusion detection, it provides a reliable foundation for extending features such as data export, advanced filtering, or integration with threat databases. The project exemplifies how open-source libraries and event-driven programming can be used to create efficient, platform-aware, and user-friendly security tools in the field of network monitoring.

CHAPTER 1

INTRODUCTION

In today's interconnected world, monitoring network traffic is a fundamental practice in securing digital communication and ensuring the healthy operation of computer networks. Packet sniffing—capturing and analyzing data packets traversing a network—plays a pivotal role in understanding network behavior, diagnosing issues, and identifying suspicious activities. This project, titled "GUI-Based Network Packet Sniffer," introduces a user-friendly, real-time packet capture tool that enables users to monitor and interpret live network traffic through an intuitive graphical interface.

The application is developed using Python and integrates multiple powerful libraries to provide a complete network sniffing experience. Scapy is used for low-level packet capture, extracting essential information such as IP addresses, ports, and protocols directly from raw traffic. Tkinter is employed to create a clean, interactive GUI that allows users to select network interfaces, apply protocol filters (TCP, UDP, ICMP), and view captured packet metadata in a structured, scrollable format. The system also leverages multithreading to ensure seamless performance during live capture without freezing the interface.

This packet sniffer is designed with accessibility in mind, offering a hands-on approach to learning about data transmission, protocol behavior, and the internal workings of network communication. It provides a practical tool for students, educators, and entry-level security enthusiasts who wish to explore the basics of network monitoring without the complexity of enterprise-level analyzers. By emphasizing modularity, clarity, and ease of use, this project underscores the potential of open-source tools in demystifying network protocols and encouraging deeper engagement with cybersecurity fundamentals.

PROBLEM IDENTIFICATION

In the modern digital landscape, every device connected to a network is constantly sending and receiving data packets. Monitoring these packets is essential for diagnosing connectivity issues, understanding data flow, and identifying unusual activity. However, most existing packet sniffing tools are either command-line based or built for advanced users, making them difficult to use for students, beginners, or non-technical users. This project addresses the lack of a simple, user-friendly tool that can capture and display live packet information in a clear and accessible format.

Popular tools like Wireshark, while powerful, can overwhelm users with vast amounts of low-level packet details and complex filters. These tools often require a deep understanding of networking protocols, making it hard for learners to extract meaningful insights without prior experience. Additionally, such tools may not provide real-time responsiveness in a lightweight form, especially when running on standard hardware or in educational environments.

There is a clear need for a basic packet sniffing application that prioritizes usability and clarity. Users should be able to select a network interface, apply simple protocol filters, and view key packet attributes such as source/destination IPs, ports, and protocol types—all in real time. The absence of beginner-friendly tools with graphical interfaces, minimal setup requirements, and real-time packet visibility has inspired the development of this GUI-based packet sniffer.

This project is designed to make network traffic monitoring accessible to anyone with basic computer skills. It simplifies the process of packet inspection and provides a visual, interactive way to learn about networking concepts, making it ideal for educational purposes, home labs, and lightweight network diagnostics.

CHAPTER 2

Requirement Analysis, Risk Analysis, Feasibility Analysis

To develop the “GUI Based Packet Sniffer”, a set of functional and non-functional requirements were defined:

Functional Requirements:

- Detect and list all available network interfaces on the system (both Windows and Linux).
- Allow the user to select an interface for packet capture via a graphical dropdown.
- Provide a text input for specifying custom packet filters (e.g., tcp, udp, icmp)
- Capture live network traffic based on the selected interface and filter..
- Extract and display (Source IP address , Destination IP address, Protocol Type, Port No)
- Enable real-time monitoring with a Start/Stop sniffing control

Non-Functional Requirements:

- User-friendly graphical interface using Tkinter with dropdowns, buttons, and live data tables
- Ensure platform compatibility, particularly for Windows and Linux systems.
- Maintain low resource (CPU/RAM) consumption to run efficiently on personal computers..
- Prevent permanent storage or logging of sensitive packet data for privacy.

Tools and Libraries:

- Programming Language: Python 3.x
- GUI Development: Tkinter
- Packet Sniffing: Scapy
- Interface Discovery: psutil (fallbacks via subprocess and OS utilities)
- Multithreading: Python’s threading module for concurrent sniffing and UI updates
- Networking Support : socket, ctypes, platform, subprocess (for system compatibility)

These requirements ensure the tool meets usability, performance, and security expectations for educational and light diagnostic purposes.

CHAPTER 3
PROPOSED SOLUTION

The proposed system is a Python-based "GUI Based Network Packet Sniffer" designed to capture and analyze live network traffic through a graphical user interface. It simplifies the process of network monitoring by providing real-time insights into incoming and outgoing packets on a selected network interface. Built with Python's Tkinter for GUI and Scapy for packet sniffing, the tool caters to cybersecurity learners, educators, and IT professionals seeking an intuitive traffic inspection solution without the complexity of full-scale enterprise tools like Wireshark.

This system adopts a modular, thread-based architecture. The application begins by allowing the user to select a network interface and apply optional protocol-level filters (e.g., TCP, UDP, ICMP). Upon clicking "Start Sniffing," the tool begins capturing packets in real-time, extracting vital metadata such as source IP, destination IP, protocol type, and service port numbers. Each captured packet is dynamically displayed in a scrollable table within the GUI, with live updates handled by a background thread to maintain responsiveness.

A distinguishing feature of this tool is its built-in traffic mode indicator and live rate monitoring. The system measures the number of packets received per second (PPS) and classifies traffic intensity into Low (🟢), Medium (🟠), and High (🔴) levels. If a high volume of packets (over 50 PPS) is detected, the application triggers a warning popup to alert the user of potentially suspicious or intensive network activity, making it effective for basic threat awareness.

Captured data is stored temporarily in memory and not written to disk, reinforcing a privacy-conscious design that ensures user confidentiality. The sniffer also includes administrative privilege checks—required on some systems for packet capture—ensuring correct execution in secure environments.

Key Features of the Proposed System:

1. Real-Time GUI-Based Sniffing

   Utilizes Scapy to monitor and display live packets as they arrive, showing protocol information and IP-level metadata through a real-time updating Treeview widget in the GUI.

2. Interface Selection and Filter Control

   Users can choose from a list of active network interfaces and optionally apply protocol filters (e.g., "tcp", "udp") to refine captured data, allowing targeted traffic analysis.

3. Dynamic Traffic Mode Detection

   An integrated packet-per-second counter calculates traffic rates and visually indicates traffic levels (Low, Medium, High) through emoji-based status updates. Anomalous spikes trigger alert popups for user awareness.

4. ⚠️ High Traffic Alerting

   If packet flow exceeds predefined thresholds (e.g., 50 PPS), the application issues a popup warning, helping users detect high-traffic scenarios like potential DoS attacks or broadcast storms.

5. Modular and Threaded Design

   Packet sniffing and rate monitoring are run in separate threads, ensuring the GUI remains responsive and scalable. This design also makes future expansion easier (e.g., logging, protocol parsing).

6. Privacy-First Architecture

   All captured packets remain in-memory and are not written to disk or transmitted externally, ensuring that the tool can be used safely on personal and academic systems without privacy concerns.

7. Cross-Platform Compatibility

   Built using standard Python libraries, the tool works across Windows and Linux systems. On Windows, it automatically checks for and prompts elevation to administrator mode, as required for raw socket access.

8. Educational Utility

   Designed with simplicity and readability in mind, this project is ideal for students and beginners exploring networking and cybersecurity. The intuitive layout and live feedback provide immediate insights into how data flows through a system.

In summary, this "GUI Based Network Packet Sniffer" offers a lightweight, extensible, and educational solution for real-time packet capture and traffic inspection. It provides users with immediate visibility into their network environment while highlighting suspicious traffic patterns through alerting and status indicators. This makes it an ideal platform for learning, testing, and building more advanced network security tools in the future.

MODEL TRAINING

Currently, the GUI-Based Network Packet Sniffer you built uses basic, rule-based logic to detect traffic patterns. For example, it classifies traffic as "Low", "Medium", or "High" based on how many packets are captured per second. This works well for basic monitoring, but it may not be smart enough to detect more advanced or hidden cyber threats.

To make the sniffer smarter, we can add Machine Learning (ML) in the future. ML allows the system to learn from past network traffic and detect threats automatically—even those that don't follow common rules.

Here's how that would work:

1. Data Collection: First, we need to collect a large amount of network traffic data, including both normal behavior and different types of cyberattacks. You can either capture this data using your own sniffer or use public datasets like CICIDS2017 or UNSW-NB15, which contain labeled examples of both safe and malicious traffic.

2. Feature Extraction: From each captured packet or network flow, we extract useful features (patterns). These might include:

    - Packet size
    - Source and destination IP address and ports
    - Time between packets (inter-arrival time)
    - Protocol used (TCP, UDP, ICMP, etc.)
    - Number of packets in a flow
    - Flags used in the packets (like SYN, ACK)

3. Training the Model: After extracting features, the data is split into two parts: training and testing. The training part teaches the ML model how to recognize good and bad traffic. Common models include:

    - Decision Trees
    - Support Vector Machines (SVM)
    - Neural Networks

4. Model Evaluation: The testing data is used to see how well the model performs.

    - Accuracy (how often it's right)

- Precision (how well it avoids false positives)
- Recall (how well it catches real attacks)
- F1 Score (balance between precision and recall)

5. Integration into the Sniffer: Once the model performs well, we can integrate it into the real-time GUI Packet Sniffer. Instead of just showing traffic speed (pps), the model will analyze live packets and classify them as

    - Normal (benign)
    - Suspicious (possibly malicious

    The GUI can then highlight or alert the user when malicious traffic is detected, making the tool smarter and more effective against modern cyber threats.

Summary: While your current GUI-based sniffer is rule-based and good for monitoring traffic volume, adding machine learning in the future would allow it to detect threats in real-time—without relying on fixed thresholds. This would help spot hidden, evolving, or zero-day attacks and make the tool more powerful and intelligent.

ARCHITECTURE DIAGRAM

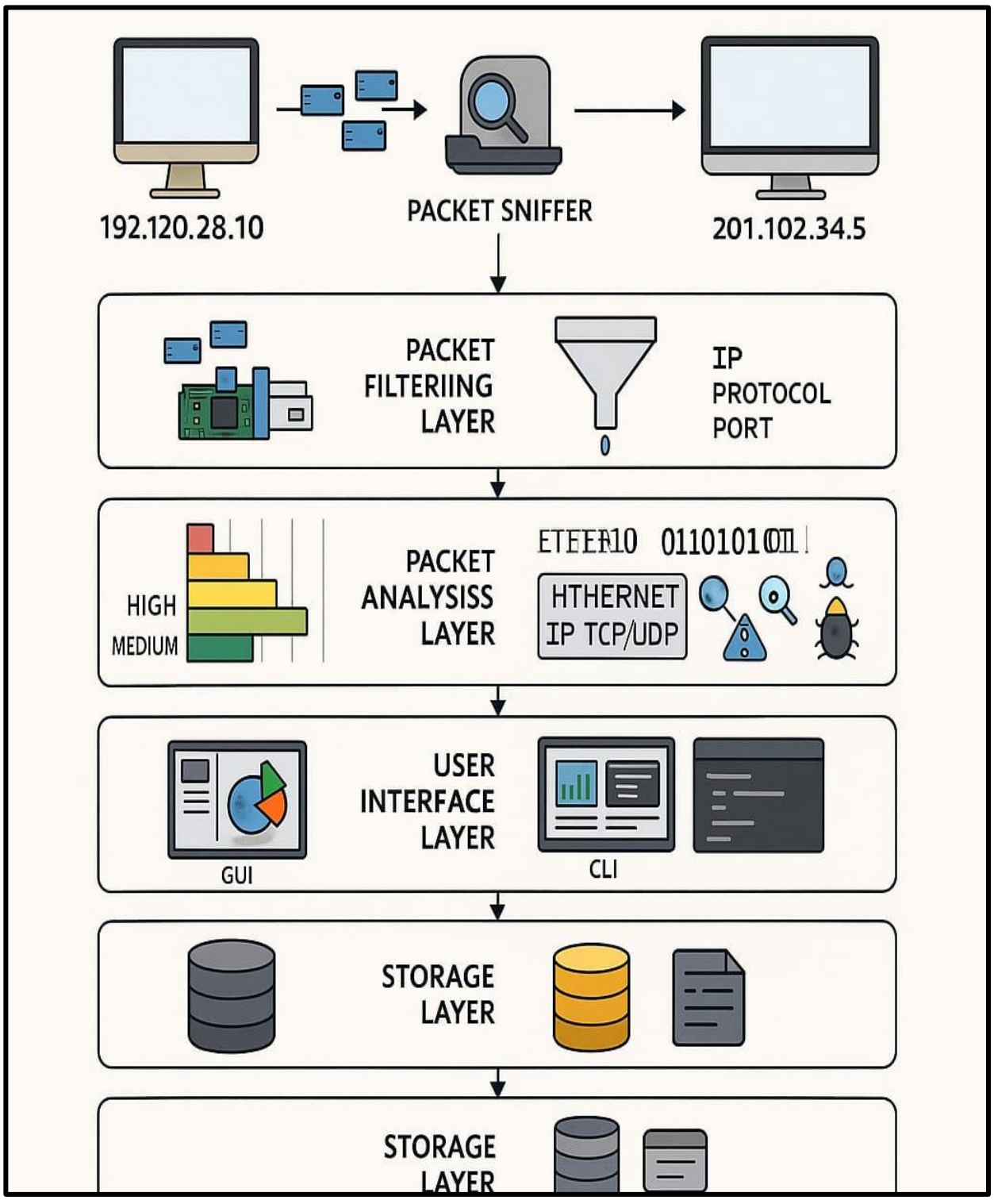


Fig: 1 Architecture Diagram of GUI based Network Packet Sniffer

FLOW CHART

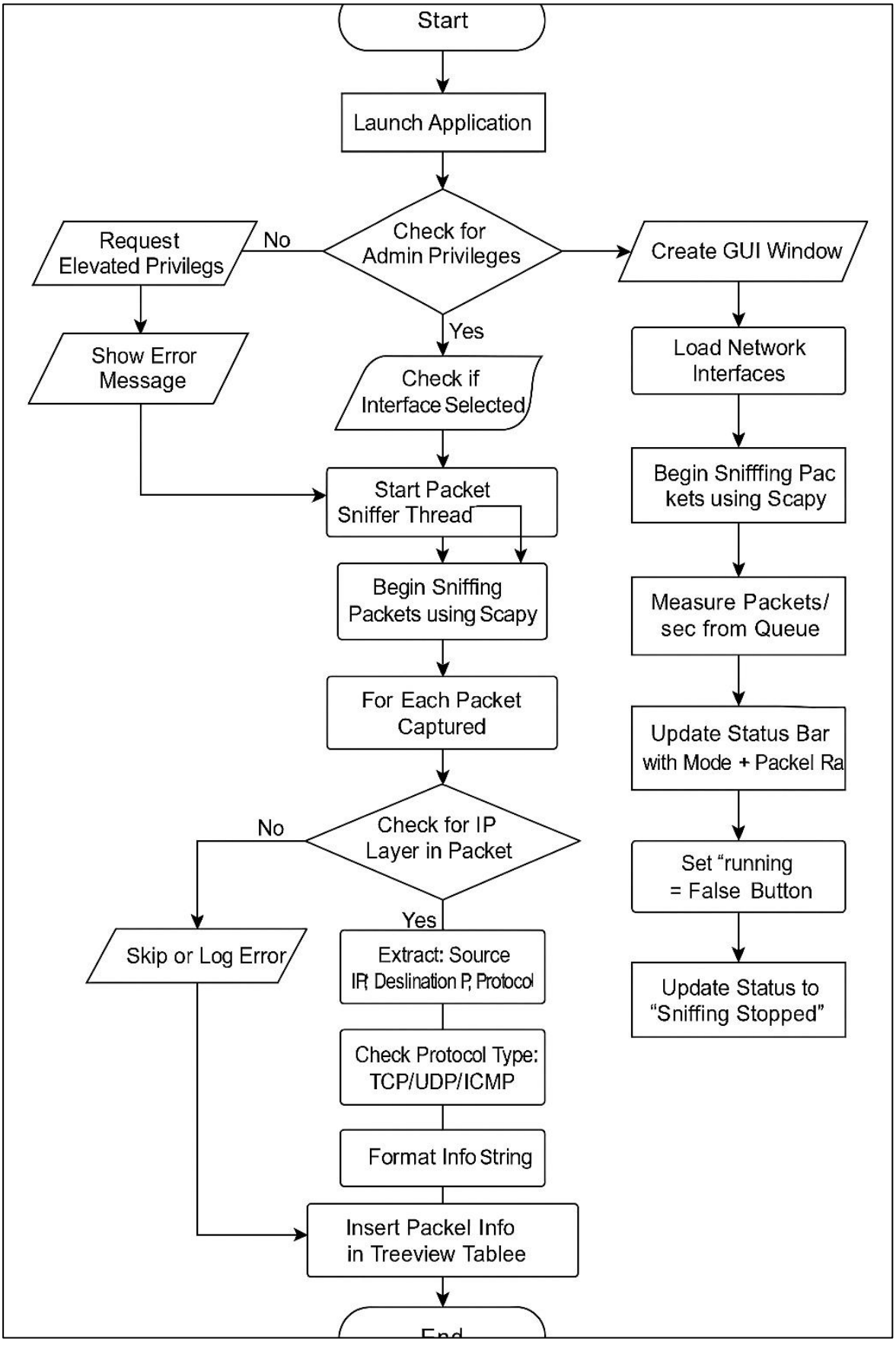


Fig 2 : Flow chart Diagram for GUI Based Network Packet Sniffer

DATA FLOW

Understanding the internal data flow of the *GUI Based Network Packet Sniffer* provides insights into how live network packets are captured, filtered, analyzed, and displayed through a user-friendly graphical interface in real time. The system integrates Scapy for low-level packet handling and Tkinter for building the interactive GUI. Here's a step-by-step breakdown of the data pipeline:

1. Packet Capture from Network Interface

The data flow initiates with live packet capture directly from the system's network interface card (NIC). This is achieved through Scapy's sniff() function, which operates in promiscuous mode, allowing the application to intercept all inbound and outbound packets on the selected interface regardless of their destination.

The user selects the desired interface through a drop-down in the GUI, which is populated using psutil and Scapy's get_if_list() functions. Once selected, a background thread is launched to continuously sniff packets using the applied filter (e.g., tcp, udp, or icmp).

Each captured packet contains protocol-specific details embedded in layers like Ethernet, IP, TCP, UDP, or ICMP. These layers are parsed using Scapy to extract meaningful fields such as:

- Source and Destination IP addresses
- Protocol type (TCP, UDP, ICMP)
- Source and Destination ports (for TCP/UDP
- Basic info message (e.g., "TCP 443 → 587")

2. Real-Time Packet Filtering and Structuring

As packets are intercepted, a callback function packet_callback() processes each one. It filters only those packets that contain the IP layer, ensuring relevance and reducing noise (e.g., ignoring broadcast or malformed packets).

Each valid packet is parsed and structured into a tuple containing:

- Source IP
- Destination IP
- Protocol Name

- Additional Info

These structured values are then inserted into a TreeView table in the GUI, allowing real-time visualization and logging of live traffic. This immediate structuring enables efficient memory use and easy access to essential data fields.

3. Traffic Rate Monitoring and Alert Generation

A separate background thread monitors the packet flow rate per second using a thread-safe queue. Every processed packet is pushed into this queue, and a 1-second window counts the number of packets processed.

Based on the count, the system categorizes traffic load into three dynamic traffic modes:

- 🟢 **Low** (< 20 packets/sec)
- 🟠 **Medium** (20–50 packets/sec)
- 🔴 **High** (> 50 packets/sec)

If high traffic is detected, a warning alert (messagebox.showwarning) is triggered to notify the user of possible congestion or abnormal activity—potentially hinting at threats such as DoS attacks or network scanning.

4. Visualization and Status Update:

While the current implementation does not use advanced plotting libraries, it still presents structured, real-time data through the GUI. The dynamic status bar updates with:

- Traffic load mode
- Packets per second (PPS)
- Real-time feedback like "🔵 Sniffing started" or "🟡 Sniffing stopped"

5. Graphical User Interface and Threaded Interaction:

The application's frontend is built with Tkinter. Users interact via:

- Interface selection dropdown
- Protocol filter entry
- Start/Stop buttons for packet capture

The GUI runs on the main thread while both sniffing and traffic monitoring operate on separate background threads. This design ensures non-blocking UI updates and real-time responsiveness, with safe synchronization using queues and root.after() calls for updating widgets from worker threads.

Captured data is displayed in the TreeView widget, allowing scrolling and live updating. Alerts and status messages help guide users during different network states.

6. Summary of Data Flow:

The system follows a structured and modular data flow that ensures accurate, fast, and user-centric monitoring of network traffic:

1. **Interface Detection** – Lists available network interfaces with IP addresses.
2. **Packet Capture** – Scapy sniffs live packets from the selected interface.
3. **Filtering & Parsing** – Valid packets are structured with relevant metadata.
4. **Traffic Monitoring** – Background thread calculates PPS and detects anomalies.
5. **Visualization & Alerts** – GUI displays packets and alerts in real-time.

CHAPTER 4

IMPLEMENTATION

The **GUI Based Network Packet Sniffer** is developed using Python and leverages several powerful libraries to provide a real-time, interactive environment for monitoring and analyzing network traffic. The architecture of the application is modular, consisting of components for packet capturing, traffic rate monitoring, dynamic protocol detection, and a responsive graphical interface. The integration of Scapy, Tkinter, and multithreading enables the system to deliver accurate and timely insights while remaining lightweight and user-friendly.

4.1 Environment Setup

To begin with, Python 3.x is required along with the following libraries:

- Scapy – for packet sniffing and network-level access
- Psutil – for enumerating and selecting active network interfaces
- Tkinter – for GUI development and event-driven interface
- Threading & Queue – for concurrent background sniffing and traffic mode monitoring

The environment can be set up using pip:

pip install scapy psutil

4.2 GUI Design (Tkinter)

The **Graphical User Interface** (GUI) is developed using Tkinter, which offers a simple and effective way to interact with the network sniffer. Key features of the GUI include:

- A dropdown to select the desired network interface from active adapters.
- An input field for entering BPF-compatible capture filters (e.g., tcp, udp, icmp).
- Start/Stop buttons to control packet sniffing
- A live table (TreeView) displaying captured packets in real-time, including Source, Destination, Protocol, and Info columns.
- A status label dynamically indicating current traffic conditions (Low, Medium, High).
- Alert pop-ups for high traffic detection.

4.3 Packet Sniffing (Scapy)

When the user clicks **"Start Sniffing"**, the selected interface and filter are passed to Scapy's sniff() function. Packet capturing occurs in a separate thread to avoid blocking the GUI. Each captured packet is analyzed in

real-time, extracting details such as:

- Source IP
- Destination IP
- Transport Layer Protocol (TCP, UDP, ICMP)
- Ports and other descriptive info

sniff(iface=iface, prn=self.packet_callback, filter=filter_str, store=False)

4.4 Real-Time Traffic Rate Monitoring:

A dedicated background thread continuously monitors the rate of incoming packets. Packets are counted per second using a thread-safe queue mechanism. Based on the packet-per-second (pps) rate, the application categorizes traffic into three dynamic modes:

- 🟢 **Low**: 0–20 packets/sec
- 🟠 **Medium**: 21–50 packets/sec
- 🔴 **High**: >50 packets/sec

When high traffic is detected, the application raises a warning popup, alerting the user of potential anomalies such as network scans or DoS attempts.

4.5 Protocol Classification and Display:

Each packet is checked for the presence of standard transport protocols:

if packet.haslayer(TCP): protocol = 'TCP'

elif packet.haslayer(UDP): protocol = 'UDP'

elif packet.haslayer(ICMP): protocol = 'ICMP'

These values are displayed live in the packet table alongside source and destination IPs and port-based info (e.g., TCP 443 → 51514). This provides real-time visibility into the types of traffic traversing the selected interface.

4.6 Multithreading and Responsiveness:

To ensure smooth user experience, the sniffer and traffic monitor run in separate threads from the main GUI thread. Python's threading.Thread is used along with queue.Queue to facilitate communication between the threads and the main UI loop. This keeps the interface fully operational and responsive, even under heavy traffic loads.

threading.Thread(target=self.run_sniffer, args=(iface, filter_str), daemon=True).start()

The GUI updates (packet table and status bar) are queued safely using root.after(), preserving Tkinter's single-threaded nature.

4.7 Interface Selection and Admin Privileges:

The application uses the psutil module to list all active network interfaces along with their IP addresses. This is presented to the user via a dropdown, ensuring proper selection. As raw packet capturing requires elevated privileges, the tool includes platform-specific checks for admin/root access and auto-prompts the user on Windows systems.

if os.name == 'nt' and not ctypes.windll.shell32.IsUserAnAdmin():

ctypes.windll.shell32.ShellExecuteW(None, "runas", sys.executable, " ".join(sys.argv), None, 1)

4.8 Privacy Considerations:

The application is designed for local, temporary analysis and does not store or log packets to disk. All captured data exists only in memory and is discarded once sniffing stops. This ensures privacy and makes the tool suitable for educational and ethical use case

PROGRAM

```
import tkinter as tk
from tkinter import ttk, messagebox
from scapy.all import sniff, IP, TCP, UDP, ICMP, get_if_list
import platform
import os
import ctypes
import sys
import psutil
import socket
import threading
import time
import queue

class PacketSnifferGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Network Packet Sniffer with Traffic Modes")
        self.root.geometry("850x600")
        self.running = False
        self.packet_queue = queue.Queue()
        self.create_widgets()

    def create_widgets(self):
        main_frame = ttk.Frame(self.root, padding="10")
        main_frame.pack(fill=tk.BOTH, expand=True)

        ttk.Label(main_frame, text="Select Network Interface:").grid(row=0, column=0, sticky=tk.W)
        self.interface_combo = ttk.Combobox(main_frame, state="readonly")
```

```
        self.interface_combo.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=5)

        self.refresh_button = ttk.Button(main_frame, text="Refresh", command=self.load_interfaces)
        self.refresh_button.grid(row=0, column=2, padx=5)

        ttk.Label(main_frame, text="Filter (e.g., tcp, udp, icmp):").grid(row=1, column=0, sticky=tk.W)
        self.filter_entry = ttk.Entry(main_frame)
        self.filter_entry.grid(row=1, column=1, sticky=tk.EW, padx=5, pady=5)
        self.filter_entry.insert(0, "tcp")

        button_frame = ttk.Frame(main_frame)
        button_frame.grid(row=2, column=0, columnspan=3, pady=10)

        self.start_button = ttk.Button(button_frame, text="Start Sniffing", command=self.start_sniffing)
        self.start_button.pack(side=tk.LEFT, padx=5)

        self.stop_button = ttk.Button(button_frame, text="Stop Sniffing", command=self.stop_sniffing,
state=tk.DISABLED)
        self.stop_button.pack(side=tk.LEFT, padx=5)

        ttk.Label(main_frame, text="Captured Packets:").grid(row=3, column=0, sticky=tk.W)

        self.packet_tree = ttk.Treeview(main_frame, columns=("src", "dst", "protocol", "info"),
show="headings")
        self.packet_tree.grid(row=4, column=0, columnspan=3, sticky=tk.NSEW)

        self.packet_tree.heading("src", text="Source")
        self.packet_tree.heading("dst", text="Destination")
        self.packet_tree.heading("protocol", text="Protocol")
        self.packet_tree.heading("info", text="Info")
```

```
        scrollbar = ttk.Scrollbar(main_frame, orient=tk.VERTICAL, command=self.packet_tree.yview)
        scrollbar.grid(row=4, column=3, sticky=tk.NS)
        self.packet_tree.configure(yscrollcommand=scrollbar.set)

        self.status_var = tk.StringVar()
        self.status_var.set("🟡 Ready")
        ttk.Label(main_frame, textvariable=self.status_var).grid(row=5, column=0, columnspan=3,
sticky=tk.W)

        main_frame.columnconfigure(1, weight=1)
        main_frame.rowconfigure(4, weight=1)

        self.load_interfaces()

    def get_network_interfaces(self):
        interfaces = []
        try:
            net_if_addrs = psutil.net_if_addrs()
            for iface_name, iface_addrs in net_if_addrs.items():
                ip_address = ""
                for addr in iface_addrs:
                    if addr.family == socket.AF_INET:
                        ip_address = addr.address
                        break
                interfaces.append((iface_name, ip_address))
        except:
            pass
        return interfaces or [(iface, '') for iface in get_if_list()]

    def load_interfaces(self):
```

```
        interfaces = self.get_network_interfaces()
        if not interfaces:
            messagebox.showwarning("Warning", "No network interfaces found.")
            self.status_var.set("❌ No interfaces found.")
            return
        interface_names = [f"{name} ({desc})" if desc else name for name, desc in interfaces]
        self.interface_combo['values'] = interface_names
        self.interface_combo.current(0)
        self.status_var.set(f"✅ {len(interfaces)} interfaces loaded.")

    def packet_callback(self, packet):
        self.packet_queue.put(1)
        try:
            if packet.haslayer(IP):
                src_ip = packet[IP].src
                dst_ip = packet[IP].dst
                proto = packet[IP].proto
                proto_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
                proto_name = proto_map.get(proto, str(proto))
                info = ""

                if packet.haslayer(TCP):
                    info = f"TCP {packet[TCP].sport} → {packet[TCP].dport}"
                elif packet.haslayer(UDP):
                    info = f"UDP {packet[UDP].sport} → {packet[UDP].dport}"
                elif packet.haslayer(ICMP):
                    info = "ICMP Packet"

                self.root.after(0, lambda: self.packet_tree.insert("", tk.END, values=(src_ip, dst_ip,
proto_name, info)))
```

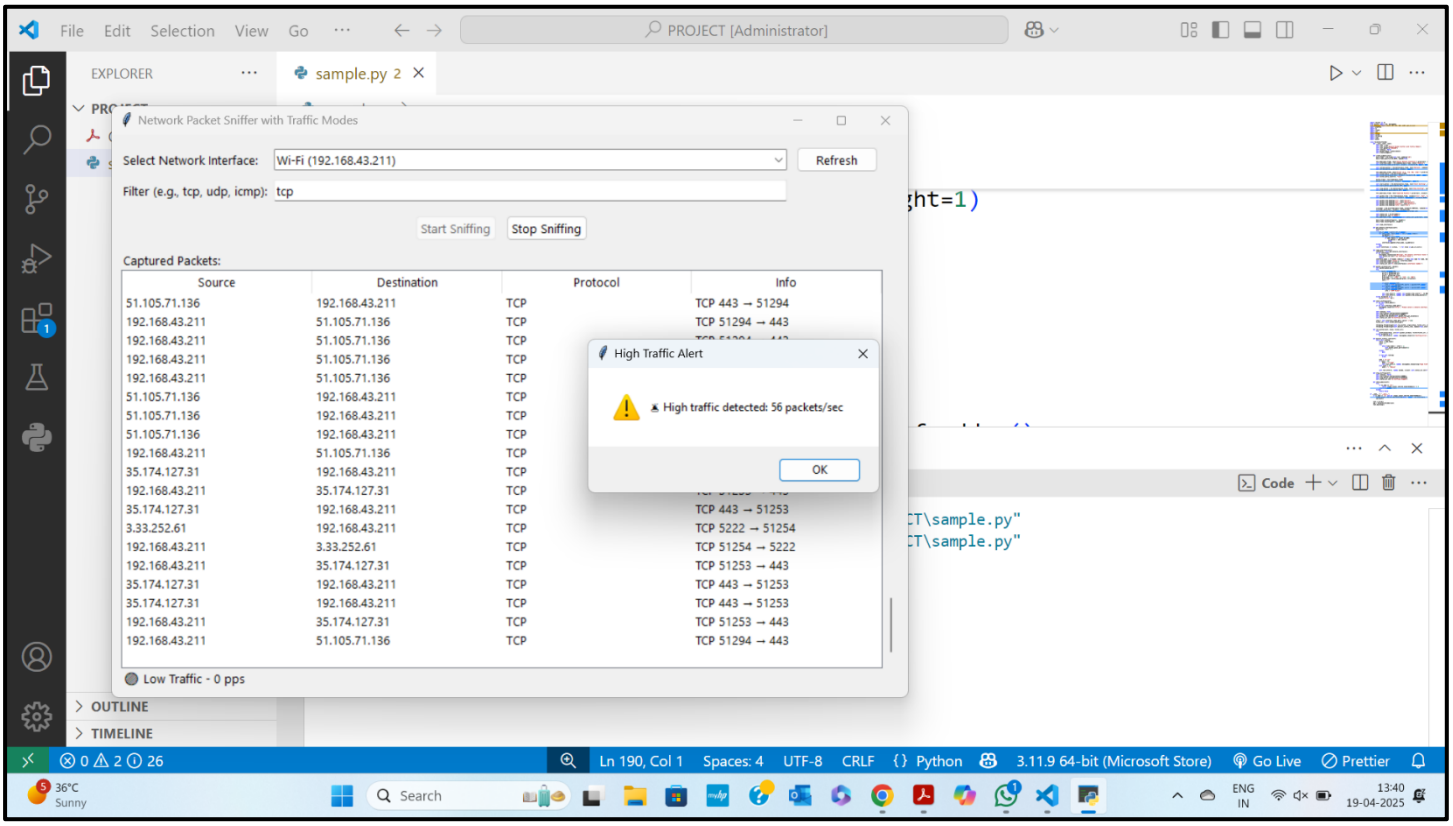
```
            self.root.after(0, lambda: self.packet_tree.yview_moveto(1))
        except Exception as e:
            print(f"Error: {e}")

    def start_sniffing(self):
        if not self.check_admin():
            return
        if not self.interface_combo.get():
            messagebox.showerror("Error", "Please select a network interface.")
            return

        self.running = True
        self.start_button.config(state=tk.DISABLED)
        self.stop_button.config(state=tk.NORMAL)
        self.packet_tree.delete(*self.packet_tree.get_children())
        self.status_var.set("🔵 Sniffing started...")

        iface = self.interface_combo.get().split(" (")[0]
        filter_str = self.filter_entry.get()

        threading.Thread(target=self.run_sniffer, args=(iface, filter_str), daemon=True).start()
        threading.Thread(target=self.monitor_traffic_rate, daemon=True).start()

    def run_sniffer(self, iface, filter_str):
        try:
            sniff(iface=iface, prn=self.packet_callback, filter=filter_str, store=False)
        except Exception as e:
            self.root.after(0, lambda: messagebox.showerror("Sniffing Error", str(e)))

    def monitor_traffic_rate(self):
```

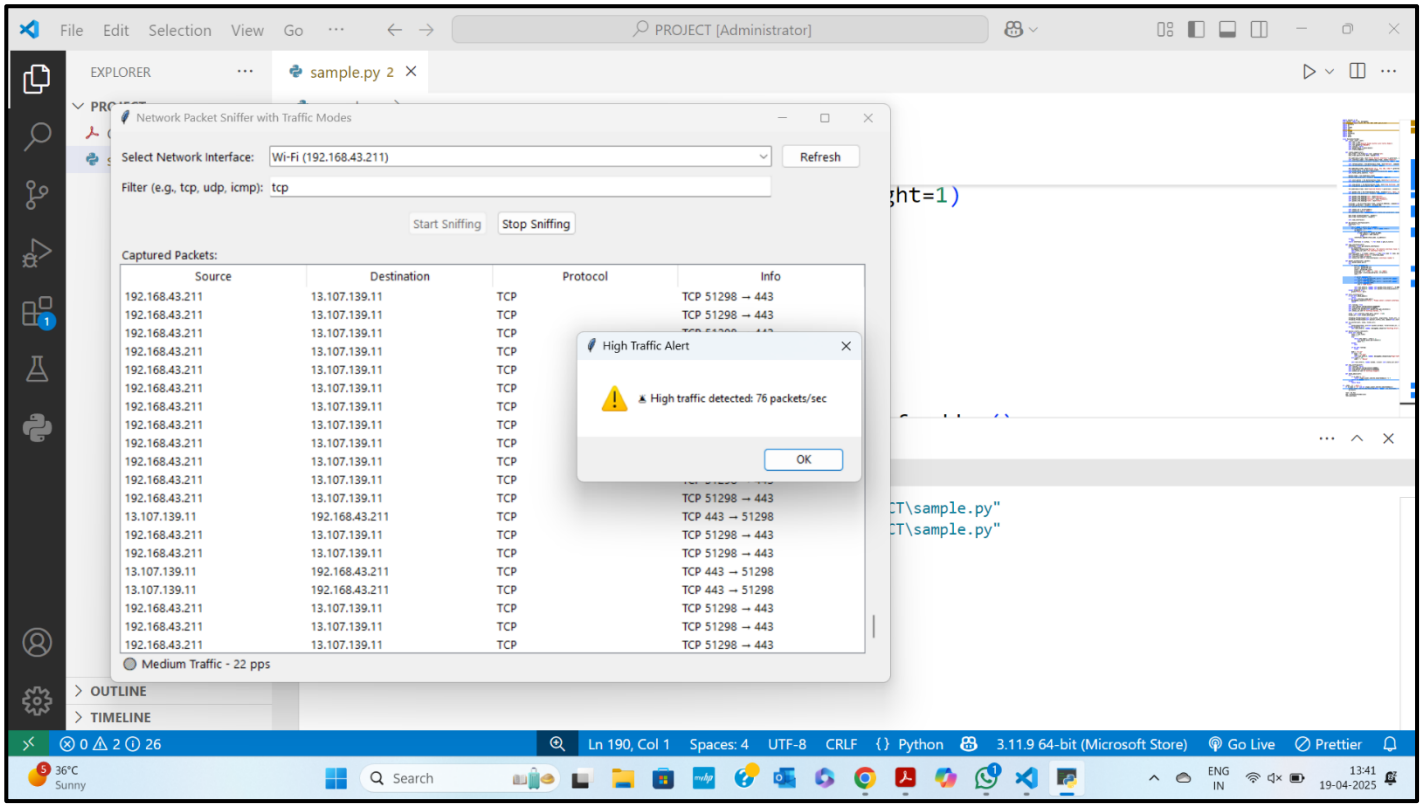
```
        while self.running:
            start = time.time()
            count = 0
            try:
                while time.time() - start < 1:
                    self.packet_queue.get(timeout=1)
                    count += 1
            except:
                pass

            if not self.running:
                break

            mode = "🟢 Low"
            if count > 50:
                mode = "🔴 High"
                self.root.after(0, lambda: messagebox.showwarning("High Traffic Alert", f"🚨 High traffic detected: {count} packets/sec"))
            elif count > 20:
                mode = "🟠 Medium"

            self.root.after(0, lambda m=mode, c=count: self.status_var.set(f"{m} Traffic - {c} pps"))

    def stop_sniffing(self):
        self.running = False
        self.start_button.config(state=tk.NORMAL)
        self.stop_button.config(state=tk.DISABLED)
        self.status_var.set("🟡 Sniffing stopped")

    def check_admin(self):
```

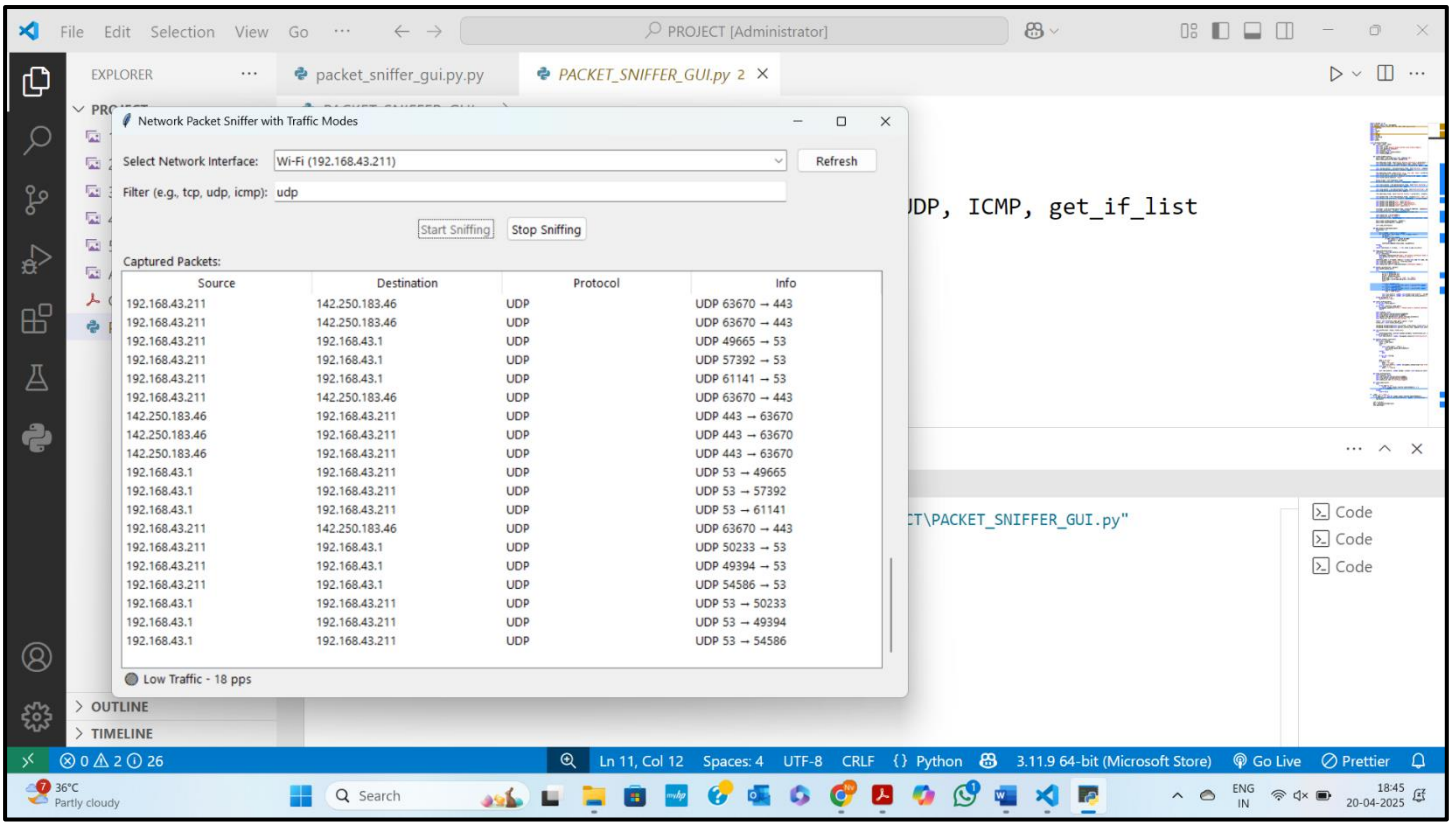
```
        try:
            if os.name == 'nt':
                return ctypes.windll.shell32.IsUserAnAdmin() != 0
            return os.getuid() == 0
        except:
            return False

if __name__ == "__main__":
    if os.name == 'nt' and not ctypes.windll.shell32.IsUserAnAdmin():
        ctypes.windll.shell32.ShellExecuteW(None, "runas", sys.executable, " ".join(sys.argv), None, 1)
        sys.exit()

    root = tk.Tk()
    app = PacketSnifferGUI(root)
    root.mainloop()
```

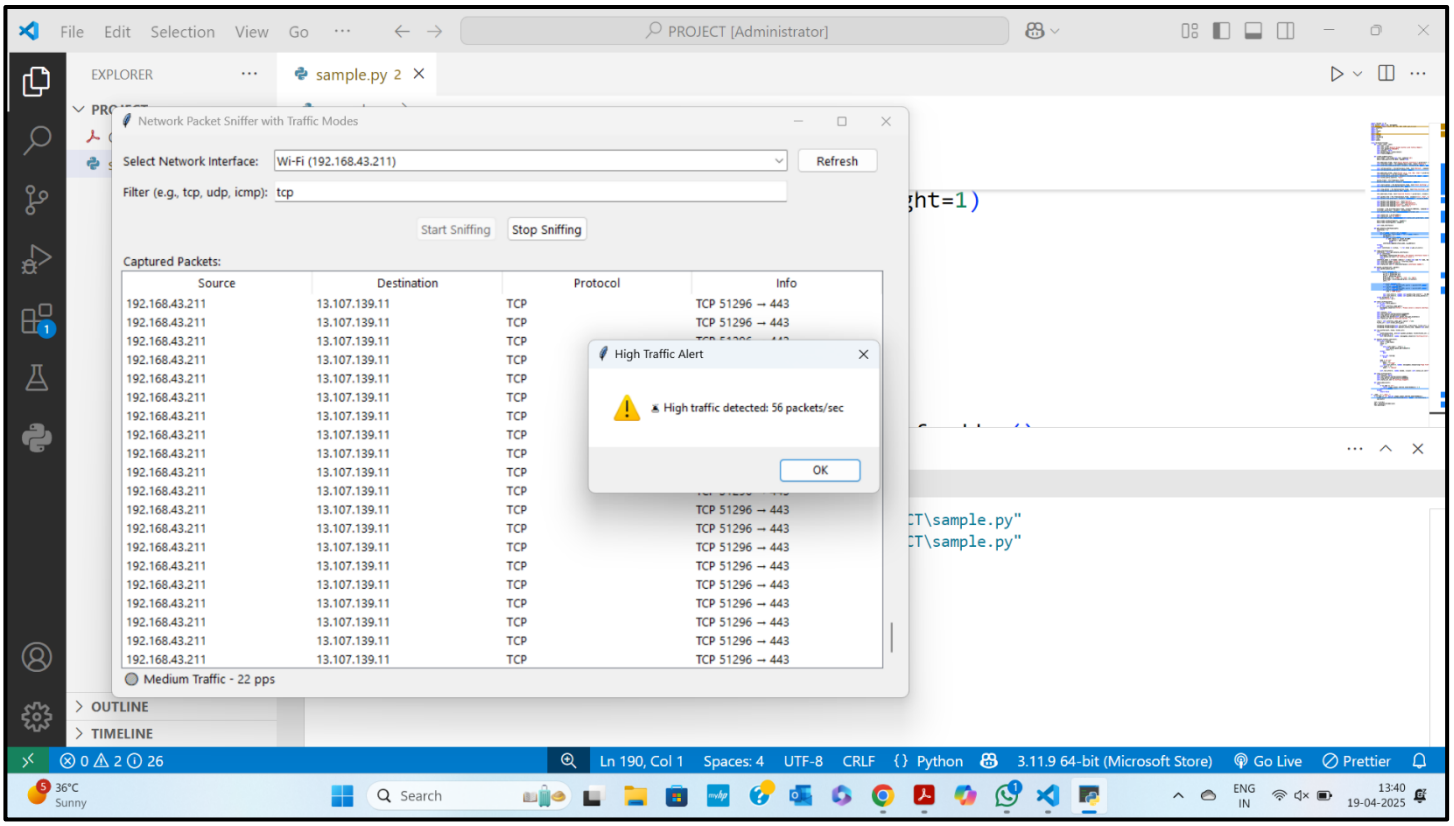
CHAPTER 5

RESULTS






Network Packet Sniffer with Traffic Modes
Select Network Interface:
Wi-Fi (192.168.43.211)
Refresh
Filter (e.g., tcp, udp, icmp):
udp
Start Sniffing
Stop Sniffing
Captured Packets:
Source
Destination
Protocol
Info
192.168.43.211 142.250.183.46 UDP UDP 63670 → 443
192.168.43.211 142.250.183.46 UDP UDP 63670 → 443
192.168.43.211 192.168.43.1 UDP UDP 49665 → 53
192.168.43.211 192.168.43.1 UDP UDP 57392 → 53
192.168.43.211 192.168.43.1 UDP UDP 61141 → 53
192.168.43.211 142.250.183.46 UDP UDP 63670 → 443
142.250.183.46 192.168.43.211 UDP UDP 443 → 63670
142.250.183.46 192.168.43.211 UDP UDP 443 → 63670
142.250.183.46 192.168.43.211 UDP UDP 443 → 63670
192.168.43.1 192.168.43.211 UDP UDP 53 → 49665
192.168.43.1 192.168.43.211 UDP UDP 53 → 57392
192.168.43.1 192.168.43.211 UDP UDP 53 → 61141
192.168.43.211 142.250.183.46 UDP UDP 63670 → 443
192.168.43.211 192.168.43.1 UDP UDP 50233 → 53
192.168.43.211 192.168.43.1 UDP UDP 49394 → 53
192.168.43.211 192.168.43.1 UDP UDP 54586 → 53
192.168.43.1 192.168.43.211 UDP UDP 53 → 50233
192.168.43.1 192.168.43.211 UDP UDP 53 → 49394
192.168.43.1 192.168.43.211 UDP UDP 53 → 54586
Low Traffic - 18 pps
UDP, ICMP, get_if_list
T\PACKET_SNIFFER_GUI.py"
Code
Code
Code


Network Packet Sniffer with Traffic Modes
Select Network Interface:
Wi-Fi (192.168.43.211)
Refresh
Filter (e.g., tcp, udp, icmp):
tcp
Start Sniffing
Stop Sniffing
Captured Packets:
Source
Destination
Protocol
Info
192.168.43.211 13.107.139.11 TCP TCP 51296 → 443
192.168.43.211 13.107.139.11 TCP TCP 51296 → 443
Medium Traffic - 22 pps
High Traffic Alert
High traffic detected: 56 packets/sec
OK
ght=1)
T\sample.py"
T\sample.py"

CHAPTER 6

LEARNING OUTCOME

The development and deployment of the GUI-Based Network Packet Sniffer provided a practical and in-depth learning experience in the field of computer networking, cybersecurity, and software development. This project integrated multiple technologies and theoretical concepts into a real-time tool that captures, processes, and displays live network traffic. The following learning outcomes summarize the core competencies and knowledge gained throughout the implementation process.

6.1 Real-Time Network Protocol Understanding

Through live packet capture and protocol analysis, we deepened our understanding of essential networking protocols such as TCP, UDP, ICMP, and IP. Observing traffic in real time allowed us to visualize the actual behavior of packets as they traverse the network, solidifying our knowledge of packet structures, headers, port communication, and the TCP/IP stack.

6.2 Low-Level Packet Capture with Scapy

Scapy enabled direct interaction with network interfaces for packet sniffing. We gained valuable insights into crafting and analyzing packets at various layers, understanding how different protocols are layered and how to extract information like source/destination IPs, ports, and payload types. This low-level access helped us grasp the fundamentals of network surveillance and diagnostic tools.

6.3 Dynamic GUI Development with Tkinter

Designing the graphical interface using Tkinter provided us with experience in building responsive desktop applications. We learned how to integrate dropdowns for interface selection, Treeview tables for packet display, and labels for dynamic status updates. This improved our understanding of user interface design and how to create applications that are both interactive and user-friendly.

6.4 Thread Management and Real-Time Responsiveness

Maintaining real-time responsiveness while capturing packets was a significant challenge. By implementing multithreading, we ensured the GUI remained active and responsive during long-running background tasks. This helped us learn about Python's threading model, thread safety, and synchronization between the logic and UI layers.

6.5 Real-Time Traffic Monitoring and Alerting

We implemented a simple yet effective traffic rate monitoring system that categorizes traffic as Low, Medium, or High based on packets per second. Real-time alerts for high traffic activity exposed us to the concepts of basic anomaly detection and early-stage threat indication mechanisms used in intrusion detection systems (IDS).

6.6 Practical Exposure to Filtering and Interface Handling

We learned how to handle multiple network interfaces dynamically using psutil and scapy.get_if_list(). The use of custom BPF (Berkeley Packet Filter) filters enhanced our ability to capture specific packet types such as TCP or UDP, improving both performance and relevancy of captured data.

6.7 Debugging and Problem-Solving in Networking Contexts

During development, we encountered challenges such as permissions on packet sniffing, cross-platform compatibility, filter errors, and UI glitches. Overcoming these issues taught us how to debug networking applications, understand Windows/Linux privilege models, and resolve real-world programming errors effectively.

6.8 Understanding Traffic Behavior through Live Data

Analyzing the flow and frequency of packets in different network conditions gave us practical exposure to traffic patterns, allowing us to identify burst activity, idle times, and unusual behaviors. This hands-on experience is crucial for roles in cybersecurity and network analysis.

6.9 Modular Code Structuring and Integration

The project brought together components from various domains — packet sniffing, UI rendering, threading, queue management, and conditional alerting — into a single cohesive application. This reinforced the importance of writing modular, readable, and maintainable code while managing multiple external libraries and dependencies.

6.10 Ethical and Privacy Considerations

While working with real-time network data, we remained aware of ethical concerns and ensured that no captured data was stored or transmitted. This emphasized the importance of privacy in cybersecurity tools and responsible data handling.

PROJECT IMPACT AND FUTURE SCOPE

The development of the **GUI Based Network Packet Sniffer** has significantly contributed to enhancing practical knowledge in the fields of network security, packet analysis, and user-centric software design. By implementing a real-time packet capturing tool with an intuitive graphical interface, the project provides a powerful yet accessible way to monitor and analyze network traffic. This empowers users—especially beginners and cybersecurity learners—to visualize network activity without relying solely on command-line tools.

The tool facilitates:

- **Better network visibility**, aiding in the detection of suspicious activities like unauthorized access or high traffic spikes.
- **Educational insights** for students and professionals by offering hands-on experience in understanding protocols like IP, TCP, UDP, and ICMP.
- **Troubleshooting support** for network administrators to diagnose connectivity issues and performance bottlenecks.

While the current version of the GUI Based Network Packet Sniffer offers foundational packet analysis capabilities, there are several avenues for future enhancement:

1. **Advanced Threat Detection**
   Incorporate machine learning-based anomaly detection algorithms to identify more complex patterns such as botnet traffic or zero-day exploits.
2. **Protocol-Specific Filters and Analysis**
   Add deep packet inspection (DPI) features to analyze specific application-layer protocols like HTTP, DNS, or FTP in greater detail.
3. **Log Export and Report Generation**
   Enable automatic export of traffic logs and the generation of structured reports for audit and documentation purposes.
4. **Remote Monitoring Capabilities**
   Extend the tool’s functionality to allow monitoring of network traffic across multiple systems from a central dashboard.

CONCLUSION

The GUI Based Network Packet Sniffer project has been a rewarding experience that blended theoretical concepts of network communication with practical implementation using Python. Through the development of this tool, we gained in-depth knowledge of how network packets are transmitted, captured, and analyzed at different layers of the OSI and TCP/IP models.

Utilizing Scapy for low-level packet sniffing gave us exposure to the inner workings of network interfaces and protocols, while the integration of Tkinter enabled the creation of a user-friendly and responsive graphical interface. The combination of backend packet processing with frontend GUI elements not only enhanced the usability of the application but also provided insight into the principles of software modularity and real-time interaction.

The use of multithreading played a crucial role in maintaining interface responsiveness, especially during continuous packet capture. This introduced us to concurrency handling in Python, an essential concept in developing interactive and efficient real-time systems.

By observing live traffic and analyzing patterns such as protocol distribution, IP flow, and unusual traffic surges, we developed a practical understanding of network monitoring and its importance in cybersecurity. Although the current implementation focuses on fundamental packet sniffing, it serves as a foundational platform for more advanced security features like intrusion detection or traffic filtering.

This project not only strengthened our technical skills in Python programming, networking, and GUI design but also highlighted the importance of accessible cybersecurity tools in both educational and administrative settings. The journey from conceptualization to deployment offered numerous problem-solving opportunities and encouraged us to think critically about performance, design, and user experience

In conclusion, the **GUI Based Network Packet Sniffer** stands as a functional and extensible application, demonstrating how open-source libraries and a thoughtful design approach can create effective solutions for network visibility and analysis. It marks a significant step in our learning path and opens up opportunities for further innovation in the fields of cybersecurity and real-time data systems.

REFERENCES

1. Garg, S., & Nia, M. M. (2018).

   Scapy-based packet sniffing and analysis for cybersecurity. International Journal of Computer Science and Information Security (IJCSIS), **16**(5), 138–144.

   . Relevant for implementing packet sniffing using Scapy.

2. Shinde, P., & Ghotekar, V. (2017).

   Visualization techniques for network traffic data. Proceedings of the International Conference on Computer Science and Engineering (ICCSE), 57–63.

   . Useful for integrating Matplotlib to visualize captured packet data.

3. Zhang, X., Zhao, L., & Li, H. (2015).

   Real-time network traffic monitoring and anomaly detection using machine learning. Journal of Information Security and Applications, **23**, 34–40.

   . Explains techniques for real-time monitoring that can be extended to GUI-based tools

4. Practical Packet Analysis: Using Wireshark to Solve Real-World Network Problems"

   By Chris Sanders

   – Though it's Wireshark-focused, the theory and practice of packet inspection apply directly to understanding sniffing.

5. "Python GUI Programming with Tkinter"

   By Alan D. Moore

   – A comprehensive guide for building modern GUIs using Tkinter, perfectly suited for your GUI-based application.

6. "Computer Networking: A Top-Down Approach"

   By James F. Kurose & Keith W. Ross

   – A widely used textbook that gives foundational and advanced knowledge on networking protoco